POLITECNICO DI TORINO

Collegio di Ingegneria Informatica, del Cinema e Meccatronica

Corso di Laurea Magistrale
in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

In-house Game Engine for 2D Games

Relatori

Prof. Andrea Giuseppe Bottino

Prof. Francesco Strada

Candidato

Vincenzo Castro

Anno accademico 2020/2021

Indice dei Contenuti

Indice delle figure

1. Prefazione

Questa tesi è frutto di una profonda passione per il videogioco, visto sia come puro mezzo di intrattenimento che come massima espressione artistica e scientifica. In alcuni contesti, questi ultimi due concetti vengono spesso separati ed anteposti, in altri contesti trovano invece la giusta armonia, come nel videogioco, dove entrambi si sposano trovando un connubio perfetto che porta l'arte a trasformarsi in scienza e viceversa.

Durante il mio percorso accademico da ingegnere informatico mi sono imbattuto in diverse discipline dalle quali sono rimasto piacevolmente affascinato. Tra queste mi sento di citare matematica, fisica, geometria, programmazione avanzata in svariati linguaggi ed a diversi livelli, codifica audio e video, computer grafica, animazione 3D e game development. Giunto al momento di scegliere la tesi della mia laurea magistrale ho voluto cimentarmi in un progetto che racchiudesse quanto più possibile le conoscenze acquisite, cercando di trasformare in qualcosa di concreto la teoria e la pratica esercitate in ogni singola disciplina, unitamente alla passione per il videogioco che mi accompagna fin dalla tenera età. Così nasce la mia tesi su un in-house game engine per applicazioni 2D, dove ogni componente è realizzata from scratch a partire dalle conoscenze apprese dai miei studi accademici e seguendo quelle che sono le practice attualmente utilizzate in questo campo.

L'obiettivo di questa tesi è pertanto fortemente didattico, ed ha come scopo quello di acquisire conoscenze profonde sulla ipotetica struttura dei moderni game engine ed i vari sub-engine di cui si compongono, oltre che acquisire conoscenze su come vengono effettivamente implementate le principali feature che tali engine mettono a disposizione del game developer.

Durante la progettazione e lo sviluppo si è cercato di applicare quanto più possibile le proprie conoscenze accademiche integrando ed attingendo laddove necessario a risorse esterne che il world wide web mette pubblicamente a disposizione. Le tecnologie utilizzate alla base dell'engine sono:

- Java come linguaggio di programmazione.
- LightWeight Java Game Library (LWJGL) [1], ovvero una libreria Java che fa da interfaccia verso librerie native C e C++ come OpenGL [44], OpenAL [45], GLFW [46], ed altre.

La scelta del linguaggio di programmazione è ricaduta su Java per i seguenti motivi:

- Si è scelto di esularsi da aspetti tipici dei linguaggi della famiglia C come la gestione della memoria ed una forte dipendenza dall'architettura della piattaforma, in modo da focalizzarsi prevalentemente sull'architettura dell'engine, sull'implementazione dei vari moduli e sugli algoritmi che stanno dietro alle feature offerte.
- Si è deciso di favorire la portabilità. Essendo un linguaggio JVM-based, il bytecode ottenuto in fase di compilazione può essere facilmente eseguito su pressoché qualsiasi piattaforma desktop sulla quale sia installata una Java Virtual Machine.
- Esistono relativamente pochi engine basati su questo linguaggio.
- L'esistenza della libreria LWJGL [1] semplifica sotto certi aspetti la vita dello sviluppatore al quale viene fornito un ambiente gamedev-ready, ovvero comprendente le principali librerie e pacchetti utilizzati in ambito game dev.

Come Proof of Concept dell'engine sono stati realizzati tre mini-giochi nei quali si è cercato di sfruttare quante più feature possibile che l'engine mette a disposizione. I tre giochi in questione sono:

- Un clone di Super Mario Bros del 1985, del quale sono stati riprodotti i primi due livelli di gioco.
- Un clone del famoso gioco per mobile Flappy Bird.
- Una rivisitazione personale del famoso gioco da tavolo Forza Quattro, meglio conosciuto fuori dall'Italia con il nome di Connect Four.

2. Game and Physics Engines

Un game engine è un framework che fornisce allo sviluppatore una serie di strumenti e funzionalità che permettono di realizzare applicazioni grafiche real-time senza la necessità di dover sviluppare da capo componenti che sono di uso comune nella maggior parte di tali applicazioni, in questo modo lo sviluppatore può concentrarsi sulla logica e le caratteristiche peculiari della propria applicazione.

I game engine quindi mettono a fattor comune quelle che sono le esigenze di sviluppo che si hanno nella produzione di applicazioni grafiche real-time, fornendo inoltre un'astrazione dal middleware che viene effettivamente utilizzato per realizzare operazioni complesse come il rendering o la gestione della fisica. Di fatto la maggior parte degli utilizzatori di un engine non conosce e non si chiede mai quale libreria grafica o motore di fisica sta utilizzando l'engine o quali algoritmi utilizza quest'ultimo per offrire determinate feature.

2.1 A brief glimpse of history about Game Engines

Intorno agli anni 70, ancor prima che il termine game industry significasse effettivamente qualcosa, i videogiochi erano realizzati interamente dalle fondamenta ed erano strettamente legati all'hardware sul quale erano destinati a girare. Questo costringeva gli sviluppatori a scrivere l'applicazione tante volte quante erano le piattaforme sulla quale sarebbe stata eseguita.

Fu agli inizi degli anni 80 che il concetto di game engine iniziò a prendere forma, grazie a diversi prodotti in ambito consumer che permettevano all'utente di costruire da se il proprio mondo di gioco attraverso l'uso di building blocks offerti dall'applicativo. Un esempio è possibile trovarlo in Pinball Construction Set del 1983, il quale permetteva all'utente di realizzare il proprio flipper.

Intorno alla metà degli anni 90 la casa di sviluppo idSoftware [2] diede vita a due pietre miliari della storia del videogame, Doom e Quake rivoluzionarono il genere first-person-shooter portando novità mai viste prima ed acquisendo una popolarità senza precedenti. Così molte società dell'epoca vollero mettersi in ballo, tuttavia anziché costruire i loro giochi da zero decisero di usare sotto licenza la struttura di gioco già prodotta da altre compagnie, come idSoftware [2], e customizzarla aggiungendo le proprie grafiche, progettando nuovi livelli, scrivendo la propria storia ed implementando la propria logica, ma l'architettura sottostante dell'applicazione sarebbe rimasta invariata.

Nel 1998 Epic Games [3] rilasciò la prima versione dell'oggi conosciutissimo Unreal Engine [4]. Ciò segnò un'altra pietra miliare nell'industria del videogioco, che vide un sempre crescente numero di sviluppatori e designers utilizzare engine di terze parti per sviluppare i propri giochi.

Nel momento in cui questa tesi sta venendo scritta, Unreal Engine 5 è stato rilasciato in early access per chi vuole avere un'anteprima delle feature offerte. Durante questi anni il mercato ha visto la nascita di decine di engine, molti cercano di essere quanto più possibile general purpose, altri mirano alla realizzazione di giochi appartenenti ad una specifica categoria. Alcuni sono a pagamento, altri open source, altri sono liberi ma chiedono delle royalties sulle entrate. Alcuni mirano ad una grafica quanto più possibile foto realistica, altri meno. Alcuni engine nascono per piattaforme ben specifiche, altri favoriscono la portabilità su diverse

piattaforme. Anche la tipologia di mercato dei game engine è andata diversificandosi col passare del tempo, oggi questi non vengono più usati soltanto in ambito videoludico, ma anche in ambito cinematografico [5][6], militare [7][8], architettonico [8][9], medicale [11] e corporate. Qualunque applicazione grafica che abbia la necessità di essere real-time, oggi viene spesso realizzata ricorrendo ad un game engine.

2.2 Industry Standard Game Engines

Un engine raggiunge lo status di industry standard quando la sua adozione all'interno del mercato raggiunge una quota significativa. Gli engine appartenenti a questa categoria sono quindi tra i più utilizzati al mondo e vengono usati per realizzare opere tra le migliori oggi in circolazione. Superfluo dire che tali engine offrono una moltitudine di feature e sono continuamente in evoluzione per offrire ai loro utilizzatori la massima esperienza possibile.

2.2.1 Unreal Engine

Come detto in precedenza, Unreal Engine [4] fu uno dei primi veri game engine che iniziarono ad essere utilizzati in ambito videoludico. La sua prima versione iniziò ad essere realizzata nel 1995 con Tim Sweeney alla scrittura di gran parte del codice del progetto, e vide la luce nel 1998 con il lancio del primo titolo della serie Unreal, un fps che sfruttava diverse feature dell'engine quali la collision detection, il colored lighting ed un insieme di variegati VFX.

Il 2002 fu il tempo di Unreal Engine 2 e di titoli come Killing Floor ed Unreal Tournament. Da allora un numero spropositato di titoli vide l'impiego di Unreal Engine, tra i più moderni troviamo la serie Borderlands, la serie Gears of War, la serie Batman: Arkham, Octopath Traveler, Ark: Survival Evolved, Final Fantasy VII Remake, Dragon Ball FighterZ, Street Fighter V, ed il capostipite dei battle-royal Fortnite. Scostandoci dal gaming troviamo anche grosse produzioni televisive come Disney's The Mandalorian [5], nel quale Unreal Engine è stato utilizzato per ricreare scenari e ambientazioni.

Da Unreal Engine 4 è stato abbandonato l'UnrealScript, ovvero il linguaggio di scripting nativo utilizzato nelle versioni precedenti di Unreal, in favore del sistema di scripting visuale chiamato Blueprints. Tale sistema è un sistema di programmazione nodale e favorisce tutte quelle figure che non hanno conoscenze di programmazione a livello di codice come i designers in quanto possono facilmente prototipare le loro idee senza mai dover lasciare l'Unreal Editor, senza dover quindi avere a che fare con i tool e gli strumenti utilizzati dagli sviluppatori come IDE ed altri software. Nel caso di sviluppatori, questi hanno a disposizione la possibilità di scrivere il loro codice in C++, avendo accesso all'intera api di Unreal Engine, che nel caso del sistema Blueprints risulta essere una sottoporzione.

E' possibile utilizzare Unreal Engine gratuitamente, l'unico onere risiede nel dover pagare il 5% dei propri introiti ad Epic Games nel caso il lavoro prodotto venga pubblicato. Ciò non costituisce affatto un deterrente per gli utilizzatori, anzi, sembra un compromesso più che onesto data la qualità e le performance che le applicazioni prodotte con Unreal Engine riescono ad ottenere, inoltre la vasta mole di documentazione e tutorial sparsi per la rete rende questo engine ancora più appetibile.

2.2.2 CryEngine

Quando fu presa la decisione di costruire ed offrire sul mercato un proprio engine, Crytek [12] scelse un approccio senza compromessi puntando ad ottenere una grafica quanto più realistica possibile. Il 2002 vide quindi l'uscita nel mercato del CryEngine [13], ciò nonostante il primo titolo basato su questo engine uscì due anni dopo sotto il nome di Far Cry. Questo fu il primo titolo della serie Far Cry, oggi molto conosciuta ed amata da molti videogiocatori del genere fps. Tuttavia, il vero successo del CryEngine fu dovuto all'uscita del fps sci-fi Crysis, divenuto una serie apprezzatissima dagli amanti del genere e composta da 3 capitoli principali. Tale gioco venne così tanto acclamato per la sua grafica mozzafiato e i requisiti hardware richiesti che nel corso degli anni a venire divenne una sorta di benchmark per stimare le performance di un pc da gioco, da cui nacque il meme "Can it run Crysis?" [91] utilizzato dai videogiocatori per stabilire se un pc fosse degno di essere usato per videogiocare o meno. Tale meme venne poi utilizzato come nome del settaggio grafico massimo nell'edizione remastered di Crysis uscita nel settembre 2020. Il retaggio del CryEngine è invecchiato bene negli anni e ad oggi esso è ancora uno dei più popolari engine sul mercato.

Come Unreal Engine [4], è possibile scaricare ed utilizzare gratuitamente il CryEngine [13] dal relativo sito web, nel quale è possibile trovare anche una vasta mole di documentazione, tutorial ed una considerevole community, pertanto il numero di risorse per imparare è abbastanza ampio. Nonostante ciò, il CryEngine è definito come uno strumento difficile da padroneggiare rispetto alla concorrenza. Esso gode anche di un esteso marketplace all'interno del quale è possibile trovare asset veramente unici, molti dei quali sono completamente gratuiti. L'utilizzo dell'engine, sebbene gratuito, prevede delle royalties sui guadagni.

L'engine è scritto in C++, Lua e C# e supporta uno scripting tramite Lua.

2.2.3 Unity

Rilasciato nel 2005 esclusivamente come game engine per Mac OS X, Unity [14] adesso supporta più di 25 piattaforme tra le più disparate tra computer, console e smartphone. Unity viene spesso discusso in modo abbastanza controverso dalla community, chi dice sia solo un ottimo engine per imparare, chi dice che esistono molti altri engine migliori di esso, chi lo reputa il migliore engine in assoluto, la verità risiede nel fatto che Unity è un engine molto versatile e capace di adattarsi a veramente molti contesti, lo ritroviamo in banali titoli mobile ma anche in titoli che potrebbero considerarsi tripla A. Unity è supportato da una delle più grandi community di sviluppatori che si possano trovare sul web, complice la vastità di piattaforme supportate dall'engine, inoltre è ricco di documentazione e tutorial su praticamente ogni aspetto, e pertanto risulta molto appetibile anche agli sviluppatori che si sono da poco approcciati al game dev. E' possibile scaricare ed utilizzare questo engine gratuitamente fin quando il proprio business rimane inferiore ai 100K $ l'anno.

Nelle sue versioni più datate, Unity supportava due linguaggi di scripting, C# e UnityScript, le versioni più recenti hanno visto la fine di UnityScript in quanto la community ha preferito largamente C#. Unity gode di un physics engine molto avanzato, inoltre supporta molto bene l'integrazione con tool come 3Ds Max e Maya. Esso è anche un engine abbastanza elastico rispetto ad altri competitor come Unreal Engine [4] e CryEngine [13].

Alcuni dei titoli più conosciuti sviluppati con Unity sono Subnautica, Hearhstone e Cuphead. Inoltre, la stragrande maggioranza dei titoli mobile vengono oggi realizzati con Unity.

2.2.4 Godot

Si tratta di un engine open-source rilasciato originariamente nel 2014 sotto licenza MIT che negli ultimi anni ha conquistato una community non indifferente. Oggi arrivato alla sua terza versione, la quarta è attualmente in lavorazione. Spesso comparato a Unity [14] per il set di feature offerte, viene scelto soprattutto per la gratuità d'uso ed i vari scripting languages offerti che ad oggi risultano essere C++, C# e GDScript. Ciò lo rende appetibile anche agli sviluppatori che sono soliti utilizzare engine più conosciuti basati prevalentemente su C++. In rete è possibile trovare molta documentazione insieme a numerosi tutorial ed una forte community. Tra i vari titoli sviluppati con Godot [15] risulta difficile trovarne qualcuno che sia veramente conosciuto, in quanto le compagnie che sono dietro agli altri engine più conosciuti hanno sicuramente più vistosità per via dei fondi spesi in campagne di marketing con le quali Godot purtroppo non riesce a competere.

2.2.5 Amazon Lumberyard

Pochi sanno che il colosso Amazon da qualche anno offre anche un game engine. Lumberyard [16] è stato rilasciato nel 2016 in versione beta e, come ci si poteva immaginare, offre una stretta integrazione con i servizi Amazon quali Twitch e AWS. Oltre a fornire il codice sorgente per intero esso è anche completamente gratuito, fatta eccezione per i servizi AWS che si intendono utilizzare per il proprio prodotto. Amazon non ha costruito Lumberyard dalle fondamenta, bensì esso è basato su una licenza del CryEngine [13] costata circa 60 milioni di dollari ed apporta modifiche minimali sul comparto rendering, fisica ed illuminazione, il focus di Amazon è stato, come detto precedentemente, l'integrazione con i vari servizi offerti dal colosso. La difficoltà di utilizzo di Lumberyard è pressochè la stessa del CryEngine, con la differenza che Amazon si sta impegnando nel fornire ai suoi utenti quanta più documentazione, tutorial, forum ed esempi possibile, al fine di attirare quanto più possibile la community di sviluppatori. Così come nel CryEngine, il linguaggio di scripting usato in Lumberyard è C++. Al momento non troviamo molti titoli sviluppati con tale engine, unica menzione va fatta per Star Citizen, titolo molto promettente della casa si sviluppo Cloud Imperium Games ed ancora in fase di sviluppo.

2.3 Proprietary Game Engines

2.3.1 Frostbite

Engine originariamente creato da DICE [17] nel 2008 per lo sviluppo di titoli fps, oggi giunto alla sua terza versione e capace di supportare giochi di diverso genere. I titoli prodotti con il Frostbite [18] sono spesso dei capolavori sotto molti aspetti, sia graficamente che dal punto di vista delle performance, inoltre godono di feature peculiari offerte dall'engine come la distruttibilità ambientale ed un AI avanzata. Esso è l'engine dietro a grandi serie come quella di Battlefield, e titoli come FIFA 20 e 21, Madden NFL, Army of Two: The Devils Cartel, alcunni Need for Speed e Medal of Honor, Plant vs. Zombies, Star Wars: Battlefront e Anthem. L'unico modo di mettere le mani su questo engine è lavorare come sviluppatore presso EA o avere quest'ultima come publisher in modo da poter usare il Frostbite per realizzare il proprio titolo.

2.3.2 Source

Engine sviluppato da Valve [19] e rilasciato nel lontano 2004 assieme a Counter-Strike: Source. Nel 2015 venne annunciata la seconda versione dell'engine ed il primo titolo che lo utilizzava, Dota 2. Negli ultimi anni il Source engine [20] ha perso terreno rispetto ai competitor, a tal punto che un titolo realizzato con esso sarebbe abbastanza distinguibile rispetto ad altri titoli realizzati con engine più all'avanguardia. Nonostante ciò rimane inalterato il fatto che i titoli realizzati con esso resteranno nella storia del videogioco, tra questi troviamo Half-Life, Counter-Strike, Portal, e parecchi altri molto conosciuti dalla maggior parte dei videogiocatori. Il Source engine è un engine molto flessibile e per questo preso parecchio di mira dalle community di modder, anche perché realizzare un gioco con esso risulterebbe abbastanza costoso e ciò ha spinto ancor di più lo sviluppo di mod.

2.3.3 id Tech

La casa di sviluppo idSoftware [2] è stata a lungo pioniere nell'ambito dei game engine, debuttando nel 1993 con la prima versione di id Tech ed il rilascio di Doom, fino ad arrivare ai nostri giorni con id Tech versione 7 [21]. Nonostante questo engine non sia molto utilizzato, i pochi titoli realizzati con esso usano solitamente tecniche innovative e sperimentali, caratteristiche indistinguibili della casa di sviluppo che sta dietro l'engine. Attualmente l'unico modo per utilizzare tale engine è entrare a far parte di un team di sviluppo di uno dei titoli per i quali idSoftware ha concesso l'utilizzo del loro engine.

2.3.4 IW Engine

Qualunque gamer che si rispetti collegherebbe le parole 'Infinity Ward' alla celeberrima serie Call of Duty. L'Infinity Ward Engine [22], nelle sue diverse varianti, è l'engine che sta dietro a praticamente tutti i titoli di tale serie. Il suo primo debutto avvenne nel 2003 con il rilascio di Call of Duty, l'engine di allora si basava su una versione modificata di id Tech 3, il 2005 vide invece il rilascio di Call of Duty 2 basato su un engine proprietario di Infinity Ward [23], ovvero IW Engine. Nel corso degli anni l'engine di Infinity Ward si è molto evoluto e l'anno 2019 vide il rilascio di Call of Duty: Modern Warfare, che sfruttava una versione dell'engine realizzata quasi ad-hoc per venire incontro alle esigenze della serie Modern Warfare, fornendo una grafica molto realistica. Come detto IW Engine è proprietà di Infinity Ward, pertanto l'unico modo per utilizzarlo è ottenere un lavoro presso Infinity Ward, Treyarch o una delle altre compagnie che realizza i giochi della serie CoD.

2.3.5 Anvil

Anche Ubisoft [24], un altro grande colosso della game industry, possiede il proprio engine. Si tratta di Anvil [25], engine con il quale nel 2007 venne pubblicato il primo titolo della serie Assassin's Creed. Da allora tutti i titoli di tale serie sono stati realizzati per mezzo dell'Anvil engine, che è stato ovviamente soggetto a continui aggiornamenti durante il corso di questi anni [25]. Anche numerosi titoli della serie Tom Clancy e Prince of Persia sono stati realizzati su Anvil. Come tutti gli altri engine proprietari, anche nel caso di Anvil l'unica possibilità di utilizzarlo risiede nell'entrare a far parte di uno dei team di sviluppo, in questo caso di uno dei titoli targati Ubisoft.

2.4 Thesis-work like Game Engines

In questa sezione verranno presi in considerazione engine che offrono caratteristiche simili all'engine sviluppato per questa tesi. Si tratta quindi di engine molto più piccoli rispetto a quelli citati nella sezione relativa agli standard industriali, pertanto sarà possibile elencare più o meno nel dettaglio le feature offerte da ognuno di questi.

2.4.1 Cocos 2D-X

Si tratta di un framework open-source multi piattaforma per la realizzazione di giochi 2D, libri interattivi, demo ed altre applicazioni grafiche e fa parte della suite di prodotti Cocos [26], ovvero prodotti mirati allo sviluppo di videogiochi. Tra i suoi punti di forza vengono citati la velocità, la semplicità d'uso, il supporto della community e la gratuità d'utilizzo. L'engine è scritto in C++ e tra le più importanti feature troviamo:

- Javascript and Typescript scripting languages
- 2D and 3D game development with features that meet the specific needs of various game types
- Supports building for Web, iOS, Android, Windows and Mac
- Tiles, Tilemaps, Sprites, Particles, Animations, ScheneGraph, Lighting, 2D/3D rendering, Audio, out of the box gui elements, Physics and Collision with Box2D
- Support for 2D animation tools: Spine and DragonBones

2.4.2 GameMaker Studio

E' un ambiente di sviluppo per videogiochi prevalentemente 2D. Ideato originariamente nel 1999 dal professor Mark Overmars con l'obiettivo di fornire uno strumento per lo sviluppo di videogiochi che non necessitasse conoscenze di programmazione, nel 2007 lo sviluppo continuò nelle mani di YoYo Games che ancora oggi continua a supportare GameMaker [27] con continui aggiornamenti e migliorie. Anche questo framework è scritto in C++ e le sue principali caratteristiche sono:

- Scripting with GameMakerLanguage (C like) and drag-and-drop visual programming language
- Supports building for Microsoft Windows, macOS, Ubuntu, HTML5, Android, iOS, Amazon Fire TV, Android TV, Microsoft UWP, PlayStation 4, Xbox Oneand Nintendo Switch
- Uses Direct3D on Windows, UWP, and Xbox One; OpenGL on macOS and Linux; OpenGL ES on Android and iOS, WebGL or 2D canvas on HTML5, and proprietary APIs on consoles
- Focus on 2D games, mostly tiles and sprites based, allowing out-of-box use of raster graphics and vector graphics (via SWF)
- Sprites, tileset animations, and skeletal 2D animations with Spine
- Box2D physics engine end Google's LiquidFun particle physics engine
- Audio and networking
- Shaders support with GLSL or HLSL

- Limited use of 3D graphics, in form of vertex buffer and matrix functions
- Other editor features in order to simplify the production pipeline

2.4.3 jMonkeyEngine

Questo engine si propone come un moderno e developer-friendly game engine scritto principalmente in Java, dal design minimalista e creato per gli sviluppatori che vogliono nelle loro applicazioni il supporto di un game engine mantenendo tuttavia il completo controllo sul loro codice con la possibilità di estendere ed adattare l'engine al loro flusso di lavoro [28]. Tra le principali feature troviamo:

- Made especially for modern 3D development
- Supports building for Microsoft Windows, macOS, Linux, Android and Raspberry Pi
- Support for WAV, MP3 and OGG formats and Global, directional and positional sounds
- Inpu with Mouse, Keyboard, Gamepad and touchscreen
- LOD, Light Culling and Single Pass Lighting
- Animation blending and interpolation with support for spatial, bone and morph animations
- Java, Kotlin or Groovy scripting languages
- Physics powered by Bullet physics engine
- Integrated profiler
- Different built-it post process effects
- Phong Lighting Materials and PBR Materials with support for Vertex, Fragment and Geometry shader
- Texture atlas and particle system support
- Use LWJGL as default renderer, so it can exploit up to OpenGL 4

2.4.4 FXGL

FXGL [29] è un game development framework basato su OpenJFX (Java FX) che può essere usato per la realizzazione di giochi 2D di diverso genere, applicazioni business con animazioni e controlli UI complessi, applicazioni 3D sperimentali, progetti accademici e commerciali, insegnamento ed apprendimento di tecniche di game development, ecc. Tra le sue feature occorre citare:

- Multi-Layer rendering
- Canvas Particle System
- Dynamic Texture Manipulation and Parallax Background
- Animation with different interpolators
- Different built-in in-game menu and ui elements
- Voronoi Tessesllation and built-in postprocessing effects

- Saving/Loading System with game settings support
- Physics powered by JBox2D
- gdxAI integration, A* pathfinding, GOAP and FSM support
- Key & mouse input management with full input mocking

2.4.5 LITIengine

Gratuito ed open-source, LITIENGINE [30] è un semplice game engine 2D scritto in Java da due fratelli bavaresi. Esso consta di due componenti: una leggera libreria Java che fornisce l'infrastruttura principale di gioco, ed un map-editor dedicato per la creazione di giochi 2D tile-based. Lo stato di rilascio è ancora in fase beta, tra le sue feature troviamo:

- Supports building for Windows, Linux and MacOS
- 2D Only Game Engine providing built in features to create tile based 2D games with plain java, be it a platformer or a top-down adventure.
- Basic Game Infrastructure (GameLoop, Configuration, Resource Management, Logging)
- 2D Render Engine (GUI Components, Sprites, Shapes, Spritesheet Animations, Text, Ambient Lighting, Particle System) based on plain Java AWT Graphics, thus no Open GL
- 2D Sound Engine (support for .wav, .mp3 and .ogg)
- Simple 2D Physics Engine with basic Collision Detection and Resolving, and possibility to apply forces to entities.
- Player Input via Gamepad/Keyboard/Mouse
- Message Based Networking Framework

2.4.6 libGDX

Si tratta di un game development framework che fornisce una api unica per tutte le piattaforme supportate. Gode di una buona community e di una altrettanto buona documentazione [31]. Alcune delle feature supportare sono:

- General purpose 2D/3D Java game development framework
- Written in Java with some JNI components exploiting C and C++. It supports programming in different JVM based languages such as Java, Scala, Clojure, Kotlin ecc.
- Builds on Windows, Mac, Linux, Android, iOS, BlackBerry and HTML5.
- Exploits LWJGL as rendering pipeline on desktop, WebGL on html5 and OpenGL ES on mobile.
- Sound effect playback for WAV, MP3 and OGG
- AI for pathfinding, decision making and movement
- Exploits Box2D for 2D physics and Bullet for 3D physics
- Freetype text renering

- Supports mouse, keyboard, touchscreen, controllers, accelerometer, gyroscope and compass as well as touch gesture detection
- Integration with different game services such as Google Play Games and more as well as a cross-platform api for in-app purchases
- Lots of built-in UI widgets
- Particle system, texture atlases and tile map support

2.4.7 LionEngine

Il LionEngine [32] è un game engine scritto in Java e progettato esclusivamente per giochi 2D. E' utilizzabile sotto forma di libreria .jar, include un semplice level editor ed al suo stato attuale semplifica lo sviluppo di titoli appartenenti al genere platform, strategia e shoot'em up. Tra le sue feature troviamo:

- Builds for Desktop and Android platforms
- Designed for 2D games: Platform, Strategy and Shoot'em Up
- Based on plain Java AWT Graphics, thus no Open GL
- Advanced image usage (images, sprites, animations, tiles, fonts, parallax ...)
- Easy keys retrieval, mouse movement
- Management of music file are also available (Wav, Midi, and more using plug-ins)
- Windowed, full-screen and applet formats fully supported
- Advanced image filtering capability (Bilinear, HQ2X, HQ3X)
- Sequence control (intro, menu, game part, credits...)
- Network layer Client-Server system, Customizable network message, Integrated chat system
- Tile based map package
- Compatibility with raster bar effect
- Pathfinding through A* algorithm
- Objects handling system (creating them, updating them, rendering, and retrieving)
- Objects transformation (size, translation), gravity handling, collision handling, raycast collision system

2.4.8 SilenceEngine

Engine per applicazioni 2D/3D scritto in Java [33]. Tra le sue caratteristiche di punta troviamo la semplicità d'utilizzo, multipiattaforma, molto modulare e completamente customizzabile nei suoi vari moduli. Qui alcune delle feature:

- Builds for Windows, Mac OS X, Linux, HTML5 (through GWT) and Android
- Discrete collision detection based on SAT algorithm
- Basic 2D/3D game engine
- Supports loading of 2D tile maps in .xml format

- Uses OpenGL 3.3 and OpenAL 1.1 as respectively video and audio renderer by means of LWJGL 3 for Desktop platforms, WebGL4J, GWT-AL and GwtOpentype libraries regarding to Web environment, and AndroidOpenAL for Android platforms.

2.4.9 Golden T Game Engine

Spessio abbreviato in GTGE [34], è una game engine library completamente gratuita e scritta in Java che permette la realizzazione di giochi 2D multipiattaforma di vario genere, ed è attivamente supportata. Tra le sue feature di rilievo troviamo:

- Builds on Windows, Linux, and Mac OS X
- Supports fullscreen, windowed, or applet mode (embedded in webpage)
- OpenGL renderer via JOGL or LWJGL
- Audio support, wave, midi, mp3, and ogg (.wav, .mid, .mp3, .ogg) audio playback
- Complete bitmapped font routines for drawing left, right, centered, and justified alignment text
- Sprite management, support for animated and directional sprite
- Built-in collision check, with basic, enhanced, and pixel perfect collision detection

2.4.10 AndEngine

Si tratta di un game engine per piattaforma Android scritto in Java da Nicolas Gramlich atto alla creazione di semplici giochi 2D [35]. E' completamente gratuito, ma come suggerisce l'autore le donazioni sono ben accettate. Supporta le seguenti feature:

- Exploits OpenGL ES to use hardware accelerated graphics
- Supports sprites, tiled sprites and sprite animations
- Scene management and HUD components
- Supports music and sound effects
- Physics based on Box2D physics engine
- Fonts renderer
- Supports vector-based graphics with AndEngine's SVG extension

2.4.11 FIFE Game Engine

Si tratta di un game engine gratuito, open-source e multipiattaforma per la creazione di giochi 2D con visuale top-down o isometrica [36]. Il core dell'engine è scritto in C++. I linguaggi di scripting supportati sono C++ e Python. Di seguito una lista delle sue feature:

- Builds for Windows, Linux and Mac
- Audio looping and OggVorbis format support
- TrueType and bitmap fonts
- Multiple definable key frame animations per object
- Support for different tile and object grids

- Custom XML-based map file format
- Multiple cameras and multiple layers per map
- Window mode (fullscreen & windowed)
- OpenGL renderer and transparency for tiles & objects

2.4.12 Castle Game Engine

Engine open-source e multi piattaforma per la creazione di titoli 2D/3D di diverso genere [37]. Scritto in Object Pascal, supporta diverse feature grafiche e permette l'importazione di asset di gioco prodotti sui maggiori programmi di modellazione oggi esistenti, inoltre è dotato anche di un'interfaccia grafica. Tra le varie feature troviamo:

- Builds on Windows, Linux, macOS, Android, iOS, Nintendo Switch, Raspberry Pi
- OpenGL powered rendering
- Physics, creatures with AI and navmesh, and more.
- Cross-platform user-interface with anchors and automatic scaling.
- Many 2D/3D model formats are supported: glTF 2.0, Spine JSON, VRML, Collada, 3DS, Wavefront OBJ, MD3, STL and others
- Interactive animation interpolated at runtime and baked animations support
- Supports different image formats (PNG, JPG, PPM, BMP, and much more), included is also support for Khronos KTX and DDS (textures with compression, mipmaps, 3d, cube maps) and RGBE format (Radiance HDR format)
- FreeType fonts rendering support
- Shadow maps and shadow volumes (full implementation, with z-fail / z-pass switching, silhouette detection etc. limited only to a single light)
- Bump mapping (normal maps), specular maps, shininess maps
- Bump mapping algorithms like the classic bump mapping (take normal from the texture), up to the steep parallax bump mapping with self-shadowing.
- Shaders support with GLSL
- Multi-texturing, cube map texturing, 3D textures, compressed textures, compressed and/or downscaled texture, anisotropic filtering, hardware occlusion query, Anti-aliasing (by OpenGL multi-sampling)
- Kraft Physics Engine and Octrees based collision detection
- 3D sound engine powered by OpenAL. WAV and OggVorbis formats are supported
- 2D Tilemaps support

2.4.13 TileEngine

Si tratta ti un engine scritto in Portable C (C99) completamente gratuito, open-source e multipiattaforma per la realizzazione di titoli 2D dall'aspetto vintage/retro [38]. Il suo rendering scanline-based, fa dei raster effects una delle sue core feature. Tra le altre feature troviamo:

- Builds for Windows (32/64), Linux PC(32/64), Mac OS X and Raspberry Pi
- True raster effects: modify render parameters between scanlines
- Pixel accurate sprite vs sprite and sprite vs layer collision detection
- Scale sprites, rotate and scale any layer
- Animate sprites and tilesets
- Many built-in retro effects: mosaic for SNES-like pixelation, per-column tile offset, CRT emulation

2.5 Industry Standard Physics Engines

Il physics engine è un modulo fondamentale di ogni game engine che si rispetti, anche per quelli più basilari, in quanto qualsiasi gioco, in modo implicito o esplicito, in modo trasparente o meno trasparente per il game developer, avrà la necessità di utilizzare una qualche funzionalità inerente la fisica, sia questa strettamente connessa alla dinamica ed alle interazioni di un qualche corpo rigido o collegata in qualche modo alla parte di collision detection.

I physics engine utilizzati come standard nella game industry sono relativamente pochi, e permettono la simulazione di corpi rigidi e soft bodies, oltre che la possibilità di definire constraint di un qualche tipo tra di essi. Solitamente offrono anche diversi tipi di simulazione della fisica, ovvero a tempo continuo o discreto. Alcuni di questi engine sono integrati direttamente all'interno dei game engine più conosciuti. Vediamo adesso una panoramica dei physics engine più utilizzati nella game industry e delle funzionalità che offrono.

2.5.1 PhysX

La soluzione per la physics simulation di NVIDIA è chiamata PhysX SDK [39], scalabile, open-source e multipiattaforma, supporta un ampia gamma di dispositivi. PhysX è già integrato in alcuni dei più popolari game engine tra cui Unreal Engine e Unity 3D. Tra le varie feature troviamo:

- Multithreaded simulation
- Memory usage management
- Support for different measurement units and scales
- Multiple broad-phase collision detection algorithms
- Convex-mesh, triangle-mesh and primitive shape collision detection
- Mesh instancing and scaling
- Discrete and continuous collision detection
- Advanced scene query system
- Finite Element Model, an industry-standard simulation technique for deformable bodies
- Rigid body dynamics, soft body dynamics (like cloth simulation, including tearing and pressurized cloth), ragdolls and character controllers, vehicle dynamics, particles and volumetric fluid simulation

- Arbitrary collision meshes

2.5.2 Bullet

Physics engine originariamente ideato da Erwin Coumans [40], è stato molto utilizzato sia nel mondo dei video giochi che nel mondo del cinema per la creazione di VFX, e viene utilizzato tutt'oggi anche se minormente per via di una più grande concorrenza con i competitor. SI tratta di un engine open-source e gratuito, avente le seguenti feature:

- Rigid body and soft body simulation with discrete and continuous collision detection
- Different collision shapes: sphere, box, cylinder, cone, convex hull using GJK, non-convex and triangle mesh
- Soft body support: cloth, rope and deformable objects
- A rich set of rigid body and soft body constraints with constraint limits and motors
- Plugins for Maya, Softimage, integrated into Houdini, Cinema 4D, LightWave 3D, Blender, Godot, and Poser

2.5.3 Havok

A qualunque videogiocatore sarà capitato di trovare nei titoli iniziali di un gioco la scritta 'Havok' e chiedersi cosa fosse. Ebbene, Havok [41] è un physics engine molto popolare che possiamo trovare in svariati titoli tripla A, ed è progettato per gestire al meglio applicazioni complesse soggette a svariate interazioni fisiche. Tra le sue feature principali troviamo:

- Stable and robust, highly optimized contact and constraint solving, even for large game worlds
- Supports a large variety of different constraint types, including industry-leading solutions for ragdolls and vehicles
- Includes constraint motors, customizable constraints, and breakable constraints
- Lightweight "Physics Particle" feature supports super-fast debris and rigid body effects
- Most robust, fully featured, and battle–tested collision detection technology available
- Optimized and flexible collision detection pipeline
- Continuous detection for environments, characters, and vehicles
- A visual debugger which displays and records in–depth profiling and debugging data across all supported platforms

2.5.4 Box2D

Quasi tutti i game engine e framework atti alla creazione di titoli 2D integrano al loro interno Box2D [42] come physics engine. Si tratta di un motore fisico 2D open-source, sviluppato da Erin Catto e pubblicato per la prima volta nell'anno 2007. Il linguaggio di programmazione utilizzato è C++, quasi in versione portable, in quanto all'interno del codice non è utilizzato praticamente nessun container della STL, pertanto è possibile compilare il sorgente su qualsiasi sistema dotato di un compiler C++. Negli anni, l'engine è stato portato su molti altri linguaggi, ma anche diverse piattaforme delle quali vale la pena menzionare Nintendo DS,

Wii, Android, BlackBerry 10, iOS, ed altri popolari sistemi operativi. Di seguito le varie feature offerte dall'engine:

- Continuous collision detection
- Contact callbacks: begin, end, pre-solve, post-solve
- Convex polygons and circles
- Multiple shapes per body
- One-shot contact manifolds
- Dynamic tree broadphase
- Efficient pair management
- Fast broadphase AABB queries
- Collision groups and categories
- Continuous physics with time of impact solver
- Persistent body-joint-contact graph
- Island solution and sleep management
- Contact, friction, and restitution
- Stable stacking with a linear-time solver
- Revolute, prismatic, distance, pulley, gear, mouse joint, and other joint types
- Joint limits, motors, and friction
- Momentum decoupled position correction
- Fairly accurate reaction forces/impulses

3. In-House Game Engine

3.1 Main Architecture and Features

L'engine si propone come motore di gioco 2D, pertanto tutte le tecniche utilizzate per offrire le varie feature sfruttano la bidimensionalità per semplificare laddove possibile.

Come la gran parte dei moderni game engine, anche questo è basato sul pattern architetturale conosciuto come Entity-Component-System, spesso abbreviato in ECS [43].

Dal punto di vista informatico, nel mondo dei videogame viene naturale pensare ad ogni elemento come una sorta di entità, entità che avrà un qualche tipo di funzionamento e quasi sicuramente una collocazione all'interno dell'ambiente 3D. Da questa visione viene quindi naturale adottare il pattern ECS, dove ogni elemento della scena è una entità (Entity) alla quale possono essere agganciati diversi componenti (Component) che andranno a caratterizzare quello che sarà il comportamento di quella entità all'interno della scena sotto una molteplicità di fattori, ad esempio come essa interagirà con il resto dell'ambiente 3D e le altre entità, se ed in che modo dovrà essere renderizzata a schermo, se e come l'utente potrà interagire con essa, e così via. All'interno dell'engine esisteranno poi dei sotto sistemi (System) aventi il compito di gestire una determinata categoria di componenti.

Come intuibile, l'adozione del pattern ECS [43] implica pertanto la creazione di diversi componenti, in modo da poter dare alle entità determinati comportamenti. Per una migliore manutenibilità dell'architettura e per mettere a fattor comune funzionalità simili, risulta naturale organizzare tali componenti in modo gerarchico sfruttando anche una suddivisione in categorie, ad esempio ci saranno quelli facenti parte della categoria dei MeshRenderer che risultano dedicati a definire come una entity verrà renderizzata, quelli appartenenti alla categoria dei RigidBody dedicati a definire come una entity dovrà interagire con la scena, e così via. Tra i components built-in più comuni che sono stati realizzati troviamo il Transform, il MeshRenderer, RigidBody, Animator, Camera, Canvas, Script e AudioSource. Risulta chiaro quindi che il pattern ECS incoraggia l'adozione di diverse tecniche di programmazione ad oggetti come l'ereditarietà ed il polimorfismo.

Come detto, i componenti possono essere suddivisi per categorie, pertanto, a seguito di una categorizzazione oculata, viene ancora una volta naturale creare dei sotto moduli dell'engine atti ad occuparsi esclusivamente di una sottocategoria, favorendo nuovamente la manutenibilità per mezzo di una suddivisione logica del framework. Tali sotto moduli si ritroveranno quindi a gestire componenti che condividono funzionalità simili, ma la cui implementazione potrebbe differire in base allo specifico tipo. Ciò rende evidente l'utilizzo del concetto di interfaccia, proprio dei linguaggi di programmazione ad oggetti e necessario affinché i sotto moduli dell'engine possano operare su collezioni di oggetti aventi una base comune ma tipo differente. Tali sotto moduli implementano anche un altro pattern tipico del game development, ovvero il singleton, che identifica una classe della quale all'interno dell'applicazione vive al più una sola istanza. Vediamo adesso una breve overview su quelli che sono i principali sotto moduli dell'engine e la famiglia di component sui quali operano.

La parte di rendering è gestita dal RenderingEngine, che tiene traccia dei vari MeshRenderer component presenti in scena e sfrutta le loro informazioni per determinare chi dovrà essere renderizzato ed in che ordine. Il MeshRenderer component invece contiene informazioni su come renderizzare una mesh. Il TileMap component ad esempio è un particolare tipo di

MeshRenderer ed ha lo scopo di renderizzare un insieme di tile 2D. Il rendering sfrutta la libreria grafica OpenGL [44] insieme allo shading language GLSL [47].

La parte di animazione è gestita dall'AnimationEngine. Esso tiene traccia dei vari Animator component presenti in scena e determina chi dovrà essere aggiornato. Ogni Animator contiene informazioni su quali animazioni sono state definite, così come informazioni di playback, callback da eseguire, ecc. Compito dell'AnimationEngine è anche quello di determinare tra quanto tempo accadrà il prossimo evento di un'animazione settata per essere tracciata a tempo continuo, come ad esempio potrebbe essere una callback che l'utente ha settato per essere eseguita al termine di una transform animation.

La parte di fisica è gestita dal PhysicsEngine. Esso tiene traccia dei vari RigidBody presenti in scena e si occupa di gestire le interazioni tra essi, pertanto implementa tutta la logica di collision detection e resolution. I collider supportati sono di tipo AABB e la fisica viene gestita a tempo continuo. Esso dispone anche di una CollisionMatrix che specifica quali categorie di oggetti possono collidere o meno. Il PhysicsEngine si appoggia ad un'altra classe, Physics, che implementa gli algoritmi di collision detection, raycasting, ed altre funzioni di utilità.

Infine, il modulo GameEngine implementa il game loop ed ha il compito di orchestrare i sub-engine di rendering, animation e physics, oltre che implementare altre funzionalità minori.

Le principali feature del game engine possono essere così elencate:

- Entities custom scripting with different callbacks related to collision detection events and animation events
- Mouse and keyboard key events
- Continuous collision detection
- Raycasting and other rigidbody collision utilities
- Axis Aligned Bounding Boxes (AABB) colliders and triggers
- Collision matrix
- Animation system providing animation player in order to control and check the status of animations playback and set user-defined animation events
- Support to both transform and material animations. Other components can be made animable too.
- Possibility to create custom animations by defining keyframes and interpolation type between them
- Audio sources supporting both PCM and MP3, playback controls and looping
- Canvas with different primitives in order to make simple HUDs by drawing 2D geometrical shapes and images
- Canvas raycasting
- Possibility to define custom meshes supporting vertex coordinates, vertex colors and UV coordinates
- Support for custom shaders in GLSL shading language
- Built-in shader for tiles and tile animations
- Tile-sheet based particle system

- Texture filtering and mip-maps
- Scenes creation and management
- Tilemaps loading and tiled world generator

3.2 GameObject and Transform Component

La classe dell'engine che implementa il concetto di entity è GameObject. Essa rappresenta letteralmente un oggetto di gioco, ed in quanto tale avrà una collocazione all'interno della scena 3D, motivo per cui ogni GameObject possiederà automaticamente un Transform component. Quest'ultimo contiene la matrice di trasformazione 4x4 dell'oggetto al quale è attaccato e supporta tutte le funzionalità per traslare, ruotare e scalare l'oggetto nel suo sistema di riferimento locale o globale.

La convenzione usata per gli assi di riferimento è la seguente: UP = Y+, FORWARD = Z+, RIGHT = X+.

I GameObject supportano il concetto di parenting, pertanto è possibile rendere un GameObject child di un altro. In tal modo le trasformazioni applicate al parent si ripercuoteranno anche su tutti i relativi figli. Il concetto di parenting è molto utilizzato sia in ambito game development sia in ambito computer grafica in generale.

Un GameObject può essere abilitato o disabilitato. Quando abilitato, lui e tutti i suoi eventuali figli faranno parte della scena 3D ed avranno dei comportamenti dettati dagli eventuali component ad essi collegati. Quando disabilitato, un GameObject e tutti i suoi eventuali figli continueranno ad esistere all'interno della scena 3D ma perderanno qualsiasi comportamento derivante dai component a loro attaccati, come se questi ultimi vengano effettivamente disabilitati.

Anche i singoli component di un GameObject possono essere abilitati o disabilitati. Nel primo caso, il GameObject cui sono attaccati godrà del comportamento che questi specificano, nel secondo caso invece il GameObject perderà qualsiasi comportamento derivante dallo specifico component disattivato.

Il modo più semplice per creare un GameObject avente come matrice di trasformazione la matrice identità è il seguente: `GameObject aGameObj = new GameObject("aGameObj");`

3.3 Scenes and Scene Management

Si è parlato spesso di scena 3D senza finora darne una definizione. Una scena è banalmente una collezione di GameObject eventualmente imparentati che l'utente avrà collocato in giro per l'ambiente 3D e sui quali avrà probabilmente attaccato dei component per ottenere un determinato comportamento. All'interno dell'engine, la classe che implementa il concetto di scena è Scene. La classe Scenes implementa invece il concetto di scene manager, ovvero quel sistema che tiene traccia delle scene create dall'utente e della scena attualmente in esecuzione, esso permette inoltre di richiedere il caricamento di una nuova scena che andrà quindi a rimpiazzare quella attualmente caricata.

La definizione di una scena avviene via codice, l'utente dovrà indicare allo scene manager il nome della scena insieme al metodo statico che si occuperà di istanziare i vari GameObject di cui la scena sarà composta: `Scenes.addScene("MainMenuScene", MyScenes::MainMenu);`

3.4 Game Loop

Il game loop rappresenta l'insieme delle operazioni svolte dall'engine per ogni singolo frame. In figura 3.1 e figura 3.2 possiamo vedere un game loop flow semplificato dell'engine realizzato.

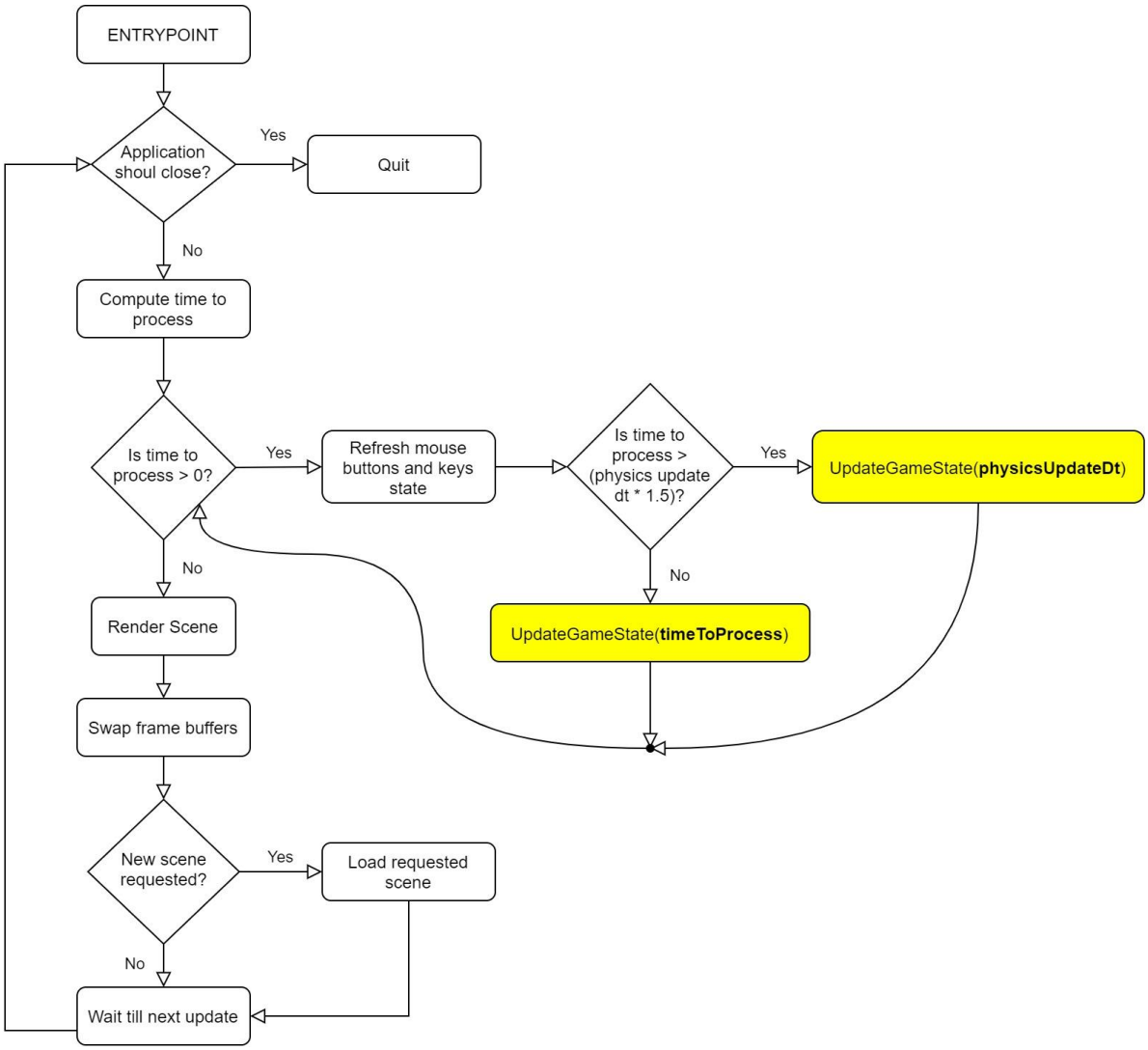

Figura 3.1: Flow del ciclo di esecuzione del game loop.

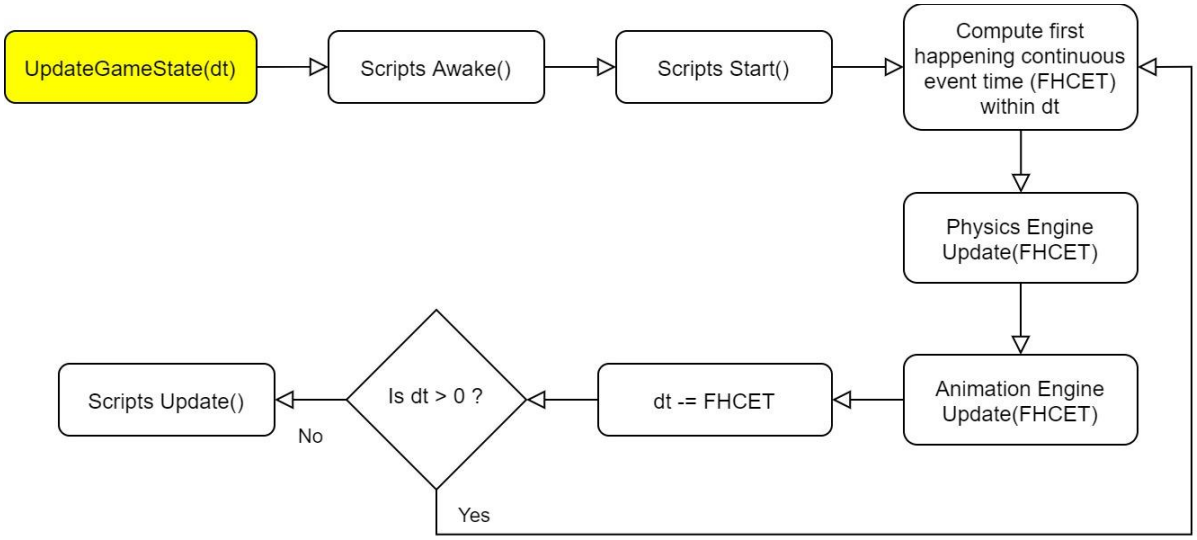

Figura 3.2: Funzione UpdateGameState(dt) del game loop.

In ogni game loop la fisica è solitamente processata ad intervalli fissi di tempo, da cui il termine fixed deltatime, anche se il delta time trascorso dall'ultimo update non è esattamente un multiplo del fixed deltatime scelto. Ciò permette di calcolare in modo abbastanza esatto quanto la simulazione fisica sarà vicina a quella reale, ed inoltre dà allo sviluppatore la possibilità di formulare diverse assunzioni come ad esempio quelle che permettono di calibrare a dovere le velocità massime degli oggetti in scena al fine di poter rilevare delle collisioni a tempo discreto. Tuttavia il deltatime dal precedente frame non sarà mai un multiplo del fixed deltatime, in questo caso l'engine ha solitamente due possibilità:

1) Verranno eseguiti tanti cicli di update della fisica pari al risultato della divisione intera dt/fixedDt, più un ultimo ciclo per processare il resto della divisione.
2) Verranno eseguiti tanti cicli di update della fisica pari al risultato della divisione intera dt/fixedDt, il resto viene scartato o conservato e quindi aggiunto al tempo di processing del prossimo frame.

Il secondo caso porta chiaramente ad una simulazione incoerente, dove il frame renderizzato manca di alcuni istanti di tempo che sono stati scartati o lasciati per il frame successivo. Il primo mantiene la coerenza della simulazione ma porta ad uno spreco di risorse, in quanto l'ultimo ciclo di update verrà utilizzato per processare il resto della divisione intera che potrebbe corrispondere ad alcuni millisecondi così come ad una frazione di millisecondo.

La scelta adottata nell'engine realizzato non corrisponde a nessuna delle due soluzioni sopra discusse, adotta invece una semplice euristica: se il resto della divisione intera dt/fixedDt è > fixedDt*0.5 allora varrà la pena usare le risorse per processare il dt restante tramite un ciclo aggiuntivo di update, altrimenti l'ultimo ciclo di update processerà oltre che il fixed deltatime anche la frazione di deltatime restante, risparmiando così risorse. In tal modo il ciclo aggiuntivo di update viene eseguito solamente se ne vale la pena, ovvero se il tempo residuo è comunque considerevole, altrimenti l'ultimo ciclo avrà un deltatime che andrà oltre il fixed deltatime in modo da coprire anche il residuo. Anche se può sembrare che in tal modo si perde l'assunzione di avere un deltatime fisso per la fisica, con tutto ciò che ne deriva, non è così, in quanto il deltatime della fisica sarà sempre il fixed update tranne che per l'ultimo ciclo nel quale sarà comunque costretto ad un upper bound pari a fixedDt*1.5, pertanto risulta ancora possibile fare le formulazioni ed assunzioni di cui sopra.

3.5 Physics

La fisica è un elemento di cui oggi ogni game engine non può fare a meno. Che venga usata per dare vita a delle simulazioni di corpi complessi e deformabili, o per avere dei corpi rigidi che interagiscono tra loro seguendo le leggi della fisica, o per fare dei semplici test di intersezione, la fisica così come tutto ciò ad essa correlabile è un elemento chiave dei moderni videogiochi così come dei game engine con i quali vengono realizzati.

Anche ai game developer in erba sarà successo di imbattersi nel raycasting e nel dover assegnare dei collider a delle entità in scena, seppur queste funzionalità non trovano diretta connessione con la fisica dei corpi rigidi esse devono comunque essere supportate dal physics engine che si sta utilizzando. Troviamo l'uso di queste funzionalità anche dove non ce lo si aspetterebbe, come nel caso del rendering della scena, dove è spesso utile ricavare quali geometrie dovranno essere effettivamente renderizzate e pertanto vengono sfruttate funzionalità come la segmentazione dello spazio ed intersection test per ricavare velocemente le geometrie che intersecano il view frustum della camera. Pertanto è chiaro come il physics engine e le varie funzionalità che offre siano uno strumento indispensabile sia all'interno del game engine sia nelle applicazioni prodotte con quest'ultimo.

Relativamente alla fisica dei corpi rigidi, molti physics engine, così come anche il lavoro di questa tesi, offrono una simulazione basata su quella branchia della fisica conosciuta come meccanica classica, della quale fanno parte le leggi di Newton.

3.5.1 Colliders

Qualunque game developer ha sicuramente sentito questo termine durante il suo lavoro. Un collider identifica una porzione di spazio rappresentata da un solido geometricamente semplice come un box o una sfera. Se abbinato ad un corpo rigido, il collider rappresenterà la forma di quel corpo e pertanto anche il modo come questo interagirà con gli altri corpi rigidi aventi anch'essi un collider. La semplicità geometrica della forma di un collider è necessaria, in tal modo i calcoli matematici per determinare eventuali intersezioni con altri collider risulteranno computazionalmente semplici e quindi si prestano bene ad elaborazioni real-time. I collider trovano largo utilizzo in una molteplicità di contesti, dai semplici test di intersezione durante la collision detection al calcolo di volumi di movimento e raycast volumetrici al culling durante la fase di rendering.

Tra i tipi di collider più utilizzati troviamo:

- Sphere: una semplice sfera, identificabile tramite le coordinate del centro e la dimensione del raggio.
- Axis Aligned Bounding Box: spesso abbreviati in AABB, si tratta di box esattamente allineati al sistema di coordinate globale, pertanto non possono ruotare in alcun modo. Vengono identificati dalla posizione del centro e l'estensione lungo i tre assi x, y e z.
- Oriented Boundig Box: spesso abbreviati in OBB, si tratta di box per i quali è possibile definire anche un orientamento, a differenza degli AABB. Sono tra i collider più utilizzati.
- Capsule: si tratta di un collider composto da un cilindro con delle semisfere ai poli. Viene spesso utilizzato per definire il volume di collisione di creature umanoidi in modo che queste possano scivolare agevolmente all'interno della scena 3D senza incastrarsi.

- Mesh: è il tipo di collider più complesso. Come suggerisce il nome, l'intera mesh, e quindi ogni triangolo che la compone, diventa collidabile, restituendo così la massima accuratezza della simulazione. Ovviamente una tale accuratezza si paga in termini di complessità computazionale degli algoritmi di collision detection che sarà necessario utilizzare.

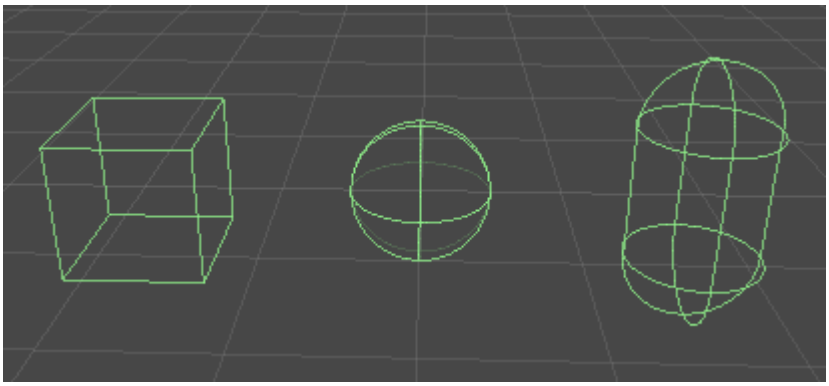

Figura 3.3: Da sinistra a destra rispettivamente un box, uno sphere ed un capsule collider.

Quando abbinato ad un corpo rigido, un collider può essere settato in modalità trigger, in questo caso non gli sarà più possibile interagire fisicamente con gli altri collider della scena e potrà essere liberamente compenetrato da questi, tuttavia il physics engine sarà ancora in grado di determinare quando un altro corpo rigido sta attraversando il trigger e pertanto potrà richiamare correttamente le eventuali callback settate dal game developer. Questa modalità è spesso utilizzata per definire delle zone di scena che una volta attraversate daranno il via ad un qualche tipo di evento/interazione, da cui il nome trigger.

L'engine realizzato si basa su collider di tipo AABB e supporta anche la modalità trigger.

3.5.2 RigidBody Component

Il RigidBody component permette di associare il comportamento della fisica dei corpi rigidi ad una data entity e di specificarne proprietà fisiche come la massa ed il coefficiente di restituzione. Una volta assegnato tale component, la entity entrerà a far parte della simulazione fisica in veste di corpo rigido e sarà quindi gestita dal physics engine, pertanto potrà essere mossa solamente tramite interazioni con il suo RigidBody, sia per mezzo di interazioni con altri RigidBody facenti parte della simulazione, sia applicando forze ed impulsi via script.

Un corpo rigido non sarebbe tale senza una forma, pertanto tra i parametri del RigidBody component sarà necessario specificare anche il collider che si intende utilizzare come forma del corpo rigido.

E' risaputo che le simulazioni fisiche sono computazionalmente onerose, soprattutto quelle real-time a tempo continuo, pertanto è necessario ottimizzare laddove possibile. A tal proposito, una prima ottimizzazione potrebbe essere quella di suddividere i RigidBody in statici e dinamici, differenziando quindi i corpi rigidi che vengono utilizzati per realizzare la geometria fisica del mondo di gioco, che solitamente è statica, dai corpi rigidi dinamici che sono invece quelli che solitamente hanno la possibilità di muoversi all'interno del mondo. Solo questa differenziazione porta già grossi vantaggi, in quanto le proprietà fisiche di un

corpo statico non cambiano, ciò permette quindi al physics engine di risparmiare numerose operazioni soprattutto riguardanti la parte di collision detection come vedremo in seguito.

Inoltre, ci si potrebbe trovare nell'esigenza di avere delle entità con delle animazioni sul Transform che devono comunque interagire con altri corpi rigidi, e pertanto dovranno a loro volta possedere un RigidBody component. In tal caso il physics engine dovrà stimare le proprietà fisiche del corpo rigido in base all'animazione riprodotta sulla entity, e non avrà altresì controllo sulle proprietà cinematiche del RigidBody.

Per sopperire a tutte queste eventualità, l'engine realizzato mette a disposizione 4 tipi di RigidBody:

- STATIC: Inteso per tutti i RigidBody che dovranno restare fermi nella loro posizione durante la simulazione.
- DYNAMIC: Inteso per quei RigidBody che devono muoversi seguendo le leggi della fisica. La fisica dei movimenti e risposte alle collisioni sono gestite automaticamente dal Physics Engine in base ai parametri settati nel RigidBody component.
- DYNAMIC_CUSTOM: Serve per quei RigidBody per i quali si vuole un controllo totale sulla gestione della fisica e della risposta alle collisioni, nel senso che queste saranno completamente a carico dell'utente che dovrà quindi prendersi l'onere di settare le opportune callback via script.
- DYNAMIC_ANIMATED: E' inteso per RigidBody che devono muoversi tramite un movimento dettato dalle Transform animation attive sulla loro entity. Il physics engine deriverà le proprietà fisiche dell'oggetto a partire dalle transform animation attive su di esso. Inoltre nessun corpo rigido sarà in grado di deviare il RigidBody dalla sua animazione.

Il RigidBody component realizzato espone metodi per applicare forze, impulsi ed accelerazioni tramite script. Sempre tramite script, l'utente ha la facoltà di programmare delle callback che verranno chiamate a seguito di una collisione con un altro corpo rigido. Esistono versioni di tali callback anche per RigidBody che possiedono un collider di tipo trigger.

3.5.3 The Physics Engine

Il physics engine è stato creato con lo scopo di avere un modulo a se stante che si occupasse di aggiornare in modo coerente la fisica di gioco. Esso tiene traccia di tutti i RigidBody attivi presenti nella scena ed è in grado di determinare quale sarà la prossima collisione entro un determinato lasso temporale. Quanto detto si riflette esattamente nella sua interfaccia lato codice, in quanto tra i metodi esposti troviamo rispettivamente quelli per aggiungere/rimuovere un RigidBody dalla simulazione, un altro che permette di ricavare il lasso di tempo della prossima eventuale collisione ed infine un metodo update(dt) che aggiorna in modo coerente la simulazione per un dato deltatime. Il physics engine viene quindi coordinato dal game engine proprio tramite tale interfaccia.

Come già anticipato nel paragrafo relativo alle feature dell'engine, il physics engine realizzato simula una fisica a tempo continuo, questo si traduce nel fatto che esso sarà in grado di determinare con precisione l'istante di collisione tra oggetti aventi dimensioni e velocità virtualmente arbitrarie. In linea generale, l'approccio utilizzato dall'engine per aggiornare la simulazione per una quantità di tempo pari a 'dt' è quello descritto in figura 3.4.

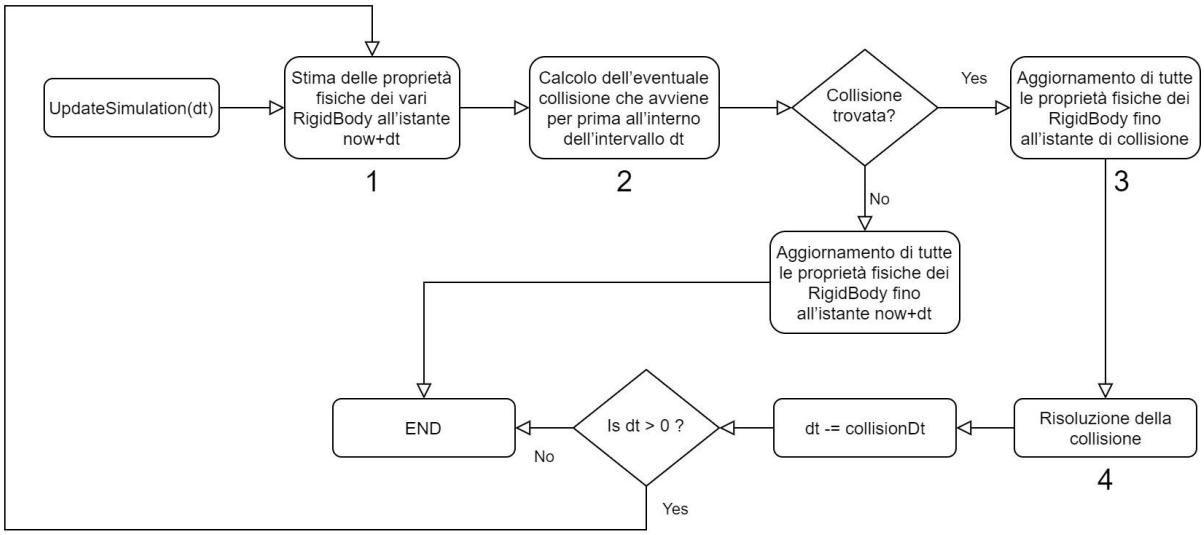

Figura 3.4: Diagramma di flusso della funzione di aggiornamento della simulazione fisica.

Sebbene il flow possa apparire discretamente semplice ed intuitivo dal punto di vista logico, dal punto di vista implementativo la situazione è ben diversa, ed ogni punto tra i quattro rappresentati in figura implica l'utilizzo di diverse funzionalità inerenti la fisica, oltre che l'impiego di diversi algoritmi, approcci, strutture di partizionamento spaziale ed approssimazioni.

3.5.4 Physics Properties Estimation and Numerical Integration

Il moto di un oggetto è descritto rispettivamente da 3 grandezze fisiche, ovvero accelerazione, velocità e posizione in funzione del tempo, dove la velocità indica il rateo di cambiamento della posizione in funzione del tempo, e l'accelerazione indica il rateo di cambiamento della velocità in funzione del tempo.

In un contesto come un game engine, la simulazione viene valutata frame per frame, con ogni corpo rigido avente una determinata accelerazione, velocità e posizione ad inizio frame. Pertanto, non si ha una legge oraria che determina esattamente l'accelerazione di un corpo al variare del tempo, ragion per cui al fine di stimare la posizione che avrà un oggetto dopo un lasso di tempo dt è necessario integrare l'accelerazione nell'intervallo [now, now+dt], e successivamente integrare il risultato nuovamente nello stesso intervallo. Chiaramente, fare una stima ad intervalli discreti frame per frame conduce ad un'inaccuratezza della simulazione, inaccuratezza che è possibile mitigare riducendo il dt dell'update della fisica, in tal modo più tale lasso di tempo è piccolo migliore sarà l'accuratezza con cui il physics engine modellerà i cambiamenti in accelerazione e velocità, portando a risultati che più si avvicinano al vero.

I metodi che vengono utilizzati per realizzare tali integrazioni prendono il nome di metodi di integrazione numerica [48], e tra i più usati in ambito physics engine per ricavare la posizione dei corpi rigidi frame per frame troviamo:

- Explicit Euler: è il più semplice tra i metodi discussi. Ad ogni update della simulazione fisica velocità e posizione del corpo vengono così stimate:
    - $v_{now+dt} = v_{now} + a_{now}dt$ (3.1)
    - $p_{now+dt} = p_{now} + v_n dt$

La semplicità di tale metodo si ripercuote nella sua imperfezione. Infatti, nel caso di un'accelerazione non nulla, la velocità utilizzata nel calcolo della posizione dovrebbe essere soggetta a dei cambiamenti durante il deltatime, questo metodo invece la considera costante durante tutto l'intervallo, generando quindi una imprecisione della simulazione che aumenterà nel tempo.

- Implicit Euler: conosciuto anche con il nome di 'Backward Euler', questo metodo risulta più completo del precedente, in quanto la velocità e la posizione al tempo now+dt vengono calcolate tenendo in considerazione rispettivamente l'accelerazione e la velocità al tempo now+dt:
    - $v_{now+dt} = v_{now} + a_{now+dt}dt$ (3.2)
    - $p_{now+dt} = p_{now} + v_{n+dt}dt$

  La pecca di questo metodo è la necessità di conoscere in anticipo l'accelerazione al prossimo frame, in modo da avere i dati necessari ad effettuare l'integrazione, tuttavia questo requisito non è sempre così scontato.
- Semi-Implicit Euler: chiamato anche 'Symplectic Euler', questo metodo è un compromesso tra i primi due, in quanto l'accelerazione durante tutto l'intervallo viene considerata costante ed avente il valore al tempo now, mentre la posizione viene calcolata tenendo conto della velocità aggiornata al tempo now+dt:
    - $v_{now+dt} = v_{now} + a_{now}dt$ (3.3)
    - $p_{now+dt} = p_{now} + v_{now+dt}dt$

  Questo metodo riesce ad essere veloce quanto Explicit Euler, ma molto più accurato, assottigliando quindi di gran lunga il problema dell'accumulo dell'errore al trascorrere del tempo.
- Verlet: Il metodo Verlet si basa sul calcolare direttamente la velocità a partire dalla posizione finale, la posizione precedente, ed il deltatime tra le due. In tal modo la formula per la stima della posizione diventa la sequente:
    - $p_{now+dt} = p_{now} + (p_{now} - P_{prev}) + a_{now}dt^2$ (3.4)

  Questo metodo è solitamente utilizzato per il calcolo di sistemi particellari sulla GPU, in quanto il costo del calcolo della velocità delle particelle è spesso inferiore al costo della lettura da un buffer contenente le velocità che potrebbe risultare in un cache miss con una conseguente perdita di tempo.
- Runge-Kutta: Come accennato in precedenza, usare un dt inferiore produce maggiore accuratezza. La famiglia di metodi Runge-Kutta sfrutta proprio questo principio, suddividendo il deltatime più volte cercando di ottenere una media che possa avvicinarsi quanto più possibile al valore reale. In questo modo si avrà una maggiore accuratezza della simulazione al trascorrere del tempo, tuttavia il costo di una tale integrazione si moltiplica per il numero di suddivisioni del deltatime, in quanto ogni suddivisione implica una computazione aggiuntiva. Pertanto, in contesti come i videogame dove una totale accuratezza non è necessaria, si preferisce avere una precisione inferiore in vantaggio di un numero maggiore di fps e fluidità generale della simulazione.

Riflettendo sul modo in cui l'utente, o anche il physics engine stesso, può interagire con i RigidBody, ci accorgiamo che non è possibile applicare su di essi delle forze variabili in funzione del tempo, l'unico modo per muovere un RigidBody consiste nell'applicare su di esso una forza costante o un impulso. E' quindi ragionevole pensare che all'interno di ogni

intervallo di update i RigidBody saranno soggetti a delle forze costanti, ovvero che non cambieranno fino al prossimo update o collisione. Una forza costante implica un'accelerazione costante, ed un moto con accelerazione costante, detto anche moto uniformemente accelerato, è descritto analiticamente dalle leggi cinematiche della fisica come segue:

- $v_{now+dt} = v_{now} + a_{now}dt$ (3.5)
- $p_{now+dt} = p_{now} + v_{now}dt + 0.5a_{now}dt^2$

Come si evince dalle equazioni soprastanti, esiste quindi un metodo esatto per stimare la posizione e velocità che avrà un dato RigidBody assumendo che questo si muova con una accelerazione costante all'interno del time step. E' evidente come la soluzione analitica sia quadratica, a differenza dei metodi numerici esaminati in precedenza.

L'engine realizzato usa la soluzione analitica per stimare con esattezza le proprietà cinematiche di un RigidBody dopo un intervallo di tempo dt.

3.5.5 Collision Detection

La collision detection, nel caso di una fisica a tempo discreto, ha il compito di determinare se due o più RigidBody stanno compenetrando tra loro a seguito dell'update delle proprietà cinematiche dei corpi rigidi, nel caso continuo invece, ha il compito di determinare se e quando due RigidBody collideranno in base alla stima effettuata sulle loro proprietà cinematiche. Come vedremo, gli algoritmi di collision detection differiscono non poco in base al tipo di detection effettuata, ovvero se a tempo continuo o discreto, inoltre il physics engine può utilizzare delle strutture di partizionamento spaziale per velocizzare operazioni come quelle relative al collision test.

3.5.5.1 Collision Layer Matrix

La matrice dei layer di collisione è una matrice binaria quadrata simmetrica 32x32 che indica se il collision layer alla riga x può collidere con il collision layer alla colonna y, pertanto righe e colonne rappresenteranno i vari layer di collisione, per un totale di 32. Righe e colonne aventi stesso indice rappresenteranno lo stesso layer, pertanto ogni cella della diagonale indicherà se un layer può collidere con se stesso o meno. Data la simmetria di tale matrice, il programmatore potrà settare tranquillamente la parte destra della matrice lasciando a 0 la parte sinistra, fatta eccezione ovviamente per la diagonale. La matrice dei layer di collisione viene sfruttata dal physics engine come test di collisione preliminare atto a stabilire se due RigidBody potranno collidere o meno in base al loro collision layer, a prescindere pertanto dal tipo di fisica discreta o continua. Ogni RigidBody apparterrà al layer di collisione settato sul suo GameObject alla voce 'Layer'. L'utente farà riferimento ai vari layer tramite nomi simbolici, in modo da poter dare un significato semantico ad ogni layer in base al relativo nome. Di seguito viene riportato un esempio di creazione di una collision layer matrix insieme ai nomi dei vari layer:

```
PhysicsEngine.instance.collisionLayerMatrix.setCollisionLayersNames(
        "None", "All", "Terrain", "Coin", "BoardCellTrigger"
);
```

```
PhysicsEngine.instance.collisionLayerMatrix.setCollisionLayerMatrix(
        new int[][]{
                {0,0,0,0,0},
                {0,1,1,1,0},
                {0,0,0,1,0},
                {0,0,0,1,0},
                {0,0,0,0,0}
        }
);
```

3.5.5.2 Discrete AABB Collision Detection Algorithms & Collision Data

In questo paragrafo verrà discusso il metodo più utilizzato per stabilire se due AABB stanno collidendo, e quindi se c'è un qualche tipo di compenetrazione tra i due. Tale metodo è basato sul Separating Axis Theorem [49][56].

Separating Axis Theorem

Il separating axis theorem enuncia che due figure convesse non stanno intersecando in alcun modo se esiste almeno una linea (o un piano, in un contesto 3D) che può essere interposta tra i due senza intersecarli. Ciò si traduce nel trovare un asse sul quale la proiezione delle due figure non si sovrappone. In tal caso, si potrà quindi assumere che le due figure non stanno compenetrando in alcun modo.

Nel caso di poligoni convessi, gli assi da testare saranno quelli corrispondenti alle normali di ogni lato (o faccia, in un contesto 3D) dei due poligoni, omettendo il test per quegli assi paralleli ad un asse già testato.

Nel caso di AABB tridimensionali, questo si riduce a testare i tre assi fondamentali x, y e z. Pertanto due AABB intersecheranno se, per ogni asse, la somma degli half extent dei due AABB lungo quell'asse è inferiore alla distanza tra gli AABB lungo quell'asse.

A livello computazionale il test di intersezione tra AABB basato sul SAT è molto leggero, pertanto esso è stato spesso utilizzato in diverse funzionalità dell'engine.

Collision Data

Per dati di collisione si intende quell'insieme di informazioni inerenti la collisione necessarie al physics engine per calcolare la corretta risposta alla collisione. Come mostrato in figura 3.5, tra queste informazioni troviamo il vettore normale alla superfice collisa, il vettore di overlap contenente per ogni asse l'ammontare di intersezione tra i due oggetti, ed i punti di contatto tra i due oggetti, spesso denominati 'collision manifold'. Nel caso di collisioni tra AABB, l'overlap ed i contact points sono abbastanza semplici da calcolare, la collision normal viene invece decisa in base ad una euristica che tipicamente sceglie come vettore normale l'asse lungo il quale si ha un'intersezione minore, il cosiddetto least intersection axis, tale vettore sarà in direzione opposta al vettore spostamento A->B, dove A e B sono rispettivamente i due AABB

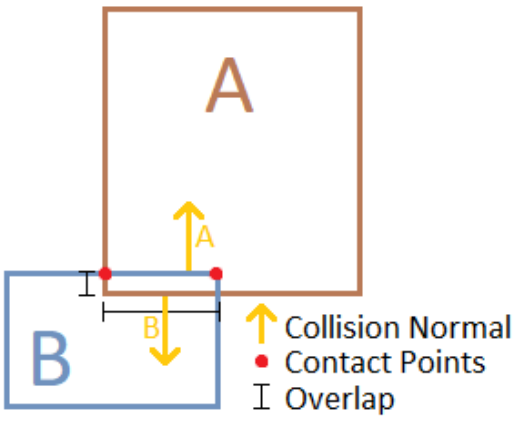

Figura 3.5: Dati di collisione relativi ad una compenetrazione tra collider di tipo AABB.

coinvolti nella collisione. Come detto, la collision normal viene scelta in base ad una euristica, pertanto in alcuni casi potrebbe non essere la scelta migliore. Per mitigare questo inconveniente, il physics engine potrebbe utilizzare altre informazioni nel calcolo della collision normal, come le proprietà fisiche del corpo rigido quali il vettore velocità, un'altra euristica potrebbe quindi essere quella di scegliere la collision normal in direzione dell'asse nel quale si ha una maggiore velocità. Va specificato infine che i dati di una collisione vengono calcolati appositamente in riferimento ad un dato collider, non saranno pertanto identici per i collider A e B, o meglio, alcuni dati saranno identici come l'overlap ed i punti di intersezione, altri saranno opposti come la collision normal.

3.5.5.3 Continuous AABB Collision Detection Algorithms & Collision Data

In questo paragrafo verrà discusso il metodo utilizzato dall'engine per stabilire se e quando due AABB collideranno lungo il loro moto, basandosi sulla loro posizione attuale e quella stimata al tempo *now+dt*. Il vettore differenza tra la posizione stimata e quella attuale verrà chiamato vettore di movimento.

Swept AABB Collision Detection Algorithm

Tale approccio si basa sul concetto del volume casting, ovvero i due AABB verranno estrusi in linea retta lungo il loro vettore di movimento fino a quando non collideranno [50][55] come mostrato in figura 3.6. Il risultato se trovato sarà un numero nell'intervallo]0, 1] che indicherà la percentuale del vettore di movimento che ognuno degli AABB dovrà percorrere per arrivare alla posizione di collisione.

Matematicamente, bisognerà ricavare per ogni asse la percentuale della relativa componente del vettore di movimento da percorrere affinché:

- Il collider A entri in contatto con il collider B. Tale tempo prende solitamente il nome di near o entry.
- Il collider A esca dal contatto con il collider B. Tale tempo prende solitamente il nome di far o exit.

Si ricaveranno quindi 6 tempi, ovvero un tempo di near ed uno di far per ogni asse. A partire da tali tempi si dovrà altresì ricavare il massimo tra i tempi di near chiamato maxNearTime, ed il minimo tra i tempi di far chiamato minFarTime, i quali corrisponderanno rispettivamente al vero istante di collisione ed al vero istante di uscita dalla collisione. A questo punto, se il maxNearTime sarà > del minFarTime oppure il maxNearTime sarà > 1 oppure il minFarTime sarà <= 0 vuol dire che non ci sarà nessuna collisione, altrimenti il maxNearTime sarà il risultato cercato. Per semplificare la logica di calcolo, solitamente viene considerato il vettore di movimento di A relativamente al movimento di B, in modo da rendere A l'unico collider in movimento come mostrato in figura 3.6, in tal caso al vettore movimento di A andrà sottratto il vettore movimento di B, portando quindi ad una variazione della destinazione di A.

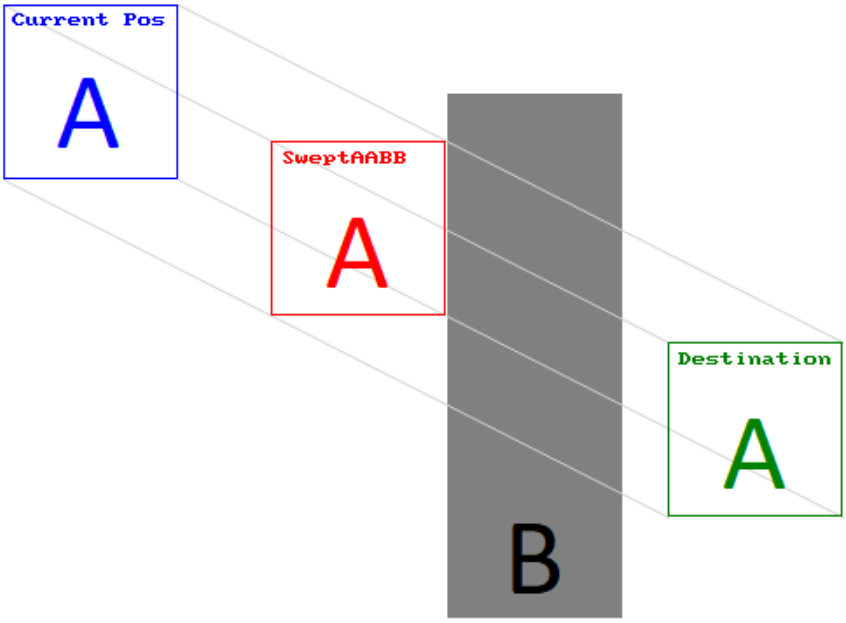

Figura 3.6: Algoritmo di continuous collision detection SweptAABB.

Collision Data

A differenza dei dati di collisione calcolati a tempo discreto, nel continuo non ci sarà nessun overlap, ne tantomeno dei punti di intersezione, in quanto i due corpi rigidi si fermeranno esattamente al momento dell'impatto. Sarà presente invece il vettore normale alla superfice di collisione, che nel caso di AABB avrà direzione pari all'asse dal quale è stato ricavato il maxNearTime, e verso opposto al verso del vettore di movimento del corpo rigido lungo quello stesso asse.

NB: L'algoritmo SweptAABB [50] come detto precedentemente assume che i corpi si stiano muovendo verso il punto destinazione in linea retta e con velocità costante, ma come sappiamo questa assunzione non sarà sempre vera, in quanto il moto dei corpi rigidi simulato dall'engine è un moto ad accelerazione costante tra i deltatime regolato dalle leggi della cinematica che descrivono il moto uniformemente accelerato, e perciò in linea generale sarà curvilineo e con velocità variabile all'interno del deltatime. Pertanto, l'uso dello SweptAABB al fine di calcolare l'istante di collisione a tempo continuo tra due AABB costituisce la prima approssimazione che è stato necessario adottare all'interno dell'engine. Ciò semplifica sicuramente la complessità computazionale del calcolo dell'istante di collisione, e costituisce comunque una buona approssimazione della realtà in quanto il physics engine verrà aggiornato ad intervalli di tempo molto fini, segmentando quindi il moto dei corpi in micro movimenti curvilinei perfettamente approssimabili con dei movimenti su linea retta a velocità costante. Inoltre l'eventuale errore introdotto da una tale approssimazione lo si avrà esclusivamente nel momento in cui viene rilevata una collisione, se un corpo rigido non collide con nessun'altro durante il deltatime allora esso si troverà esattamente nella posizione stimata.

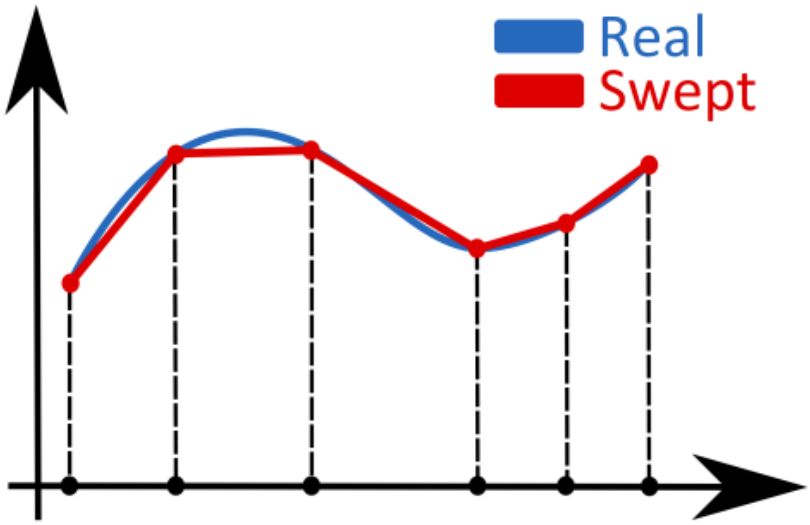

Figura 3.7: L'algoritmo di continuous collision detection basato su Swept assume che i RigidBody si muovino a velocità costante durante il time step.

3.5.5.4 Space Partitioning Structures

Finora sono stati esaminati dei metodi per determinare se due AABB stanno intersecando, o se e quando intersecheranno durante il loro moto. Tuttavia, il physics engine deve essere in grado di determinare tutti i corpi rigidi con i quali un RigidBody sta intersecando, o intersecherà all'interno di un intervallo di tempo, e questo vale per ogni RigidBody facente parte della simulazione.

Un primo approccio potrebbe essere quello di testare ogni corpo rigido contro tutti gli altri, ottenendo così una complessità computazionale quadratica. Nel caso di una collision detection a tempo discreto, ciò vorrebbe dire applicare l'algoritmo SAT [49] a tutte le possibili coppie di RigidBody facenti parte della simulazione, nel caso continuo dovremmo invece applicare l'algoritmo SweptAABB [50] su tutte le possibili coppie. Per quanto tali algoritmi siano computazionalmente leggeri, soprattutto il SAT, al crescere del numero di corpi rigidi presenti nella simulazione la quantità di calcoli che il physics engine deve sostenere ad ogni ciclo di update inizia a diventare consistente a causa della quadraticità dell'approccio scelto. Tra le feature del physics engine già discusse, la LayerCollisionMatrix è sicuramente utile ad eliminare dalla computazione un certo numero di collision test, tuttavia essa dipende molto dalla logica dell'applicazione e non è quindi da prendere in considerazione. La possibilità di avere dei RigidBody statici e dinamici è sicuramente d'aiuto in quanto solamente quelli dinamici provocheranno delle collisioni, è quindi possibile eliminare dalla computazione i collision test STATICvsALL, lasciando solamente i DYNAMICvsALL. Pertanto, nel caso di una scena con solo RigidBody statici, la parte di collision detection sarà praticamente nulla. L'aver distinto tra corpi rigidi statici e dinamici purtroppo non basta, in quanto al crescere del numero dei RigidBody la complessità computazionale continua a tendere verso n2. Uno dei modi per ridurre drasticamente il numero di collision test da effettuare è suddividere i RigidBody in piccoli gruppi in base al volume che occupano all'interno dello spazio, ed eseguire poi il collision test DYNAMICvsALL relativamente al gruppo dove risiede un RigidBody, in tal modo più gruppi si avranno e minore in teoria sarà il numero di intersection test da fare. L'approccio di clustering descritto richiede l'utilizzo di strutture di partizionamento ed indicizzazione spaziale [51]. Tra queste, le più note sono le griglie 2D/3D ed i quadtree [52][53], chiamati octree nella loro versione 3D. Questi ultimi esistono in diverse varianti, in base alla tipologia del contesto e necessità dell'applicazione.

Grids 2D/3D

Facenti parte della famiglia di metodi basati su hashing spaziale, le griglie sono probabilmente le strutture più semplici da realizzare, ma anche tra le più performanti se usate nel giusto contesto. In una griglia, lo spazio viene suddiviso in celle di ugual dimensione, ed ogni cella può contenere un numero indefinito di oggetti, inoltre, la griglia viene spesso allineata al sistema di riferimento globale, ciò vuol dire che le celle possono considerarsi come degli AABB. Un oggetto verrà inserito in più celle se esso interseca entrambe. Conoscendo il numero di righe e colonne, e la dimensione della cella, è chiaramente possibile determinare tramite un semplice calcolo le celle che un oggetto di tipo AABB andrà ad occupare, pertanto le operazioni di inserimento e rimozione sono computazionalmente banali. Relativamente al calcolo dei test di collisione, bisognerà iterare su ogni cella effettuando il DYNAMICvsALL su tutti i RigidBody contenuti al suo interno. Per dimensionare correttamente la size della cella, massimizzando così i benefici della griglia, bisognerà far si che ogni cella contenga il minor numero di oggetti possibile (minimizzando così i test DYNAMICvsALL per cella), ed ogni oggetto venga mappato nel minor numero di celle possibile (avere un oggetto che ricade su più celle vuol dire eseguire un DYNAMICvsALL aggiuntivo su ogni cella nel quale è stato inserito).

Come sempre, esistono diverse implementazioni delle griglie 2D/3D [52][53], ma in linea generale esse presentano i seguenti pro e contro:

+ Semplice da implementare e molto efficiente in termini di memoria
+ Essendo predeterminato il numero di righe e colonne di cui sarà composta la griglia, così come la dimensione della cella, è possibile conoscere a priori quanto la griglia occuperà in memoria
+ Conoscendo la posizione della griglia nel mondo, così come il numero di righe e colonne della griglia e la dimensione della cella, inserimenti, query e rimozioni risultano computazionalmente semplici
+ La semplicità computazionale di inserimenti e rimozioni si traduce in un buon supporto agli oggetti in movimento, in quanto questi saranno soggetti ad un continuo reindexing all'interno della struttura
- Se la distribuzione degli oggetti nel mondo non è uniforme molte celle saranno vuote, e pertanto sprecate
- Se la dimensione degli oggetti varia di molto, si potrebbero avere casi dove una cella conterrà una moltitudine di oggetti piccoli oppure un oggetto grande ricadrà su una moltitudine di celle, perdendo così il vantaggio del partizionamento spaziale con un conseguente aumento drastico dei collision test da eseguire

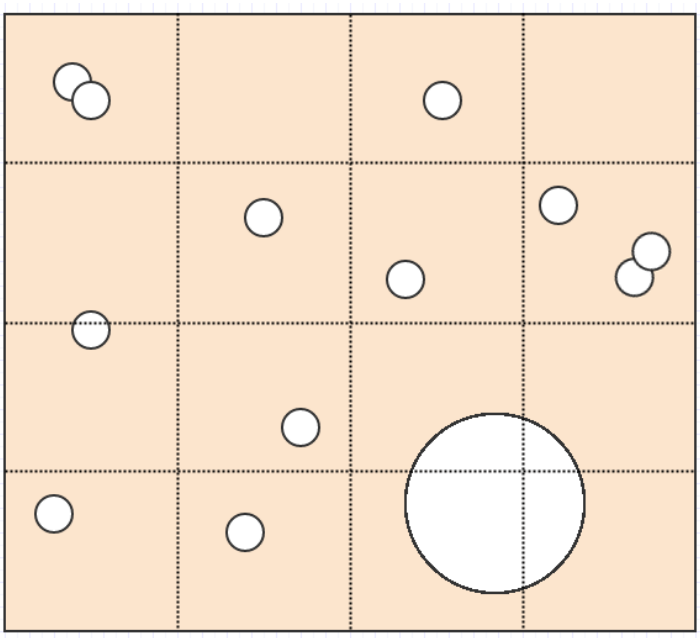

Figura 3.8: Esempio di partizionamento spaziale basato su una griglia 2D.

Quadtree - Octree

Come suggerisce il nome, queste strutture fanno parte della famiglia degli alberi da cui ne ereditano le caratteristiche fondamentali. La struttura è quindi quella di un albero formato da nodi, dove ogni nodo racchiude un volume ben determinato e può contenere degli oggetti oltre che nodi figli, questi ultimi occuperanno una porzione del volume del padre seguendo una logica di suddivisione binaria lungo ogni asse. Un oggetto sarà contenuto in uno o più nodi se questi lo intersecano a livello spaziale. L'intera struttura viene solitamente allineata con il sistema di riferimento globale, in tal modo, ogni nodo può considerarsi una cella di tipo AABB. I quadtree sono strutture in generale più dinamiche rispetto alle griglie, e tentano di sopperire ad alcuni aspetti negativi di queste ultime. Ad esempio, sui quadtree è possibile scegliere il massimo numero di oggetti che saranno contenuti al più in un nodo, e la dimensione delle celle è variabile e si adatta in base al contenuto tramite suddivisione binaria. Il funzionamento di un quadtree è il seguente: il nodo radice rappresenta l'intero volume della struttura, quando un nodo conterrà il numero massimo di oggetti stabilito allora esso darà vita a 4 figli ognuno contenente un quadrante dello spazio occupato dal padre, e gli oggetti contenuti in quest'ultimo verranno trasferiti ai figli in base a chi di questi interseca quale oggetto. In questo modo si avranno sempre delle celle con un numero di oggetti controllato, ed il numero di intersection test per cella rimarrà basso. Oltre al vincolo sul massimo numero di figli di ogni nodo, l'albero dovrà soddisfare anche un altro vincolo, ovvero quello sulla massima profondità, in tal modo, se un nodo si trova alla massima profondità egli non potrà più suddividersi, ed il massimo numero di oggetti contenibili potrà andare fuori limite, così facendo si limitano quei casi in cui un dato oggetto potrà scatenare una suddivisione molto fitta che impatterebbe sia sulle risorse che sui tempi di query e modifica dell'albero. Quando i figli di un nodo contengono complessivamente un numero di oggetti inferiore al limite stabilito, i figli dovranno essere rimossi ed i loro oggetti trasferiti al padre. Relativamente al calcolo dei test di collisione, bisognerà visitare l'albero e per ogni nodo foglia applicare il DYNAMICvsALL su tutti i RigidBody contenuti al suo interno. Ci sono diverse implementazioni dei quadtree [52][53], alcune delle quali non permettono oggetti duplicati su più nodi foglia, in tal caso, un oggetto verrà inserito esclusivamente nel nodo che lo contiene

per intero, sia questo un nodo foglia o meno, così facendo anche la logica per il calcolo dei test di intersezione cambierà. A prescindere dalla specifica implementazione, i quadtree, così come gli octree, hanno i seguenti pro e contro:

+ Non così complessi da implementare ed aventi generalmente un basso memory footprint
+ Supportano molto bene distribuzioni sparse di oggetti nel mondo
+ Query su volumi di spazio implicano la navigazione dell'albero e pertanto rimangono computazionalmente semplici
+ Nonostante siano strutture dinamiche, conoscendo la massima profondità ed il massimo numero di oggetti per figlio, è possibile calcolare un upper-bound di quanto al più la struttura occuperà in memoria
- Inserimenti e rimozioni sono computazionalmente semplici, tuttavia la suddivisione dinamica in figli o il merge di questi in un unico padre si porta dietro un overhead che in alcuni casi potrebbe diventare significativo
- L'overhead dovuto allo split/merge dei nodi rende queste strutture non proprio adatte a supportare oggetti dinamici che hanno bisogno di essere reindicizzati ad ogni ciclo

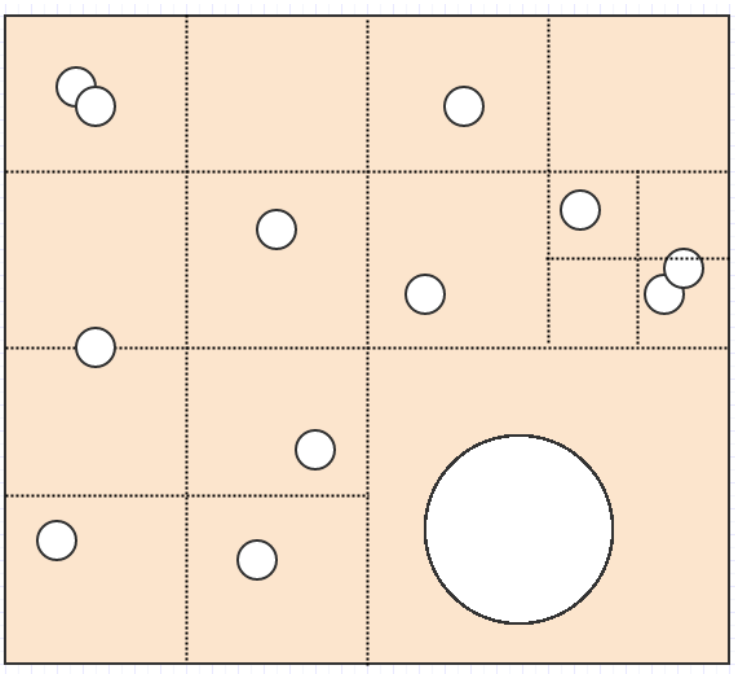

Figura 3.9: Esempio di partizionamento spaziale basato su quadtree.

3.5.5.5 Broad and Narrow Phases

Come osservato nel precedente paragrafo, l'impiego di strutture di indicizzazione spaziale come griglie e quadtree ci permette di ottenere dei bucket di RigidBody che potenzialmente potrebbero collidere, su tutte le possibili coppie DYNAMICvsALL di ogni bucket andrà poi effettuato il test di collisione vero e proprio. Tuttavia, nel caso più generale, alcuni di questi oggetti potrebbero avere dei collider complessi, non per forza degli AABB, ed effettuare il test di collisione direttamente con loro potrebbe essere oneroso. Pertanto, prima di eseguire il test di collisione complesso, si potrebbe eseguire un test preliminare atto a stabilire tramite

una semplice computazione se i due collider potrebbero collidere o meno. Le strutture di indicizzazione spaziale nascono già con lo scopo di creare bucket di collider che potrebbero potenzialmente collidere, fare poi un altro semplice collision test preliminare prima di effettuare il collision test vero e proprio costituisce uno step aggiuntivo che mira ancora una volta a minimizzare per quanto possibile gli sforzi del physics engine.

Broad Phase

Navigare nella struttura di indicizzazione spaziale alla ricerca di tutte quelle coppie di RigidBody DYNAMICvsALL che potrebbero potenzialmente collidere, eseguendo al più dei test di intersezione preliminari, prende il nome di broad phase [54][55]. Tale fase ha quindi il compito di effettuare una scrematura delle coppie di RigidBody da testare, sulle quali andrà poi eseguito un test di intersezione più accurato. Nel caso discreto, il test di intersezione preliminare più utilizzato è il SAT tra AABB, nel caso di collider complessi l'AABB verrà calcolato al volo in quanto risulta un'operazione veramente semplice, ed il SAT verrà applicato su di esso. Anche nel caso continuo viene utilizzato il SAT, tuttavia va sottolineato che nella struttura di indicizzazione spaziale i vari RigidBody sono indicizzati in base alla loro posizione corrente nello spazio, non abbiamo quindi nessuna informazione utile che tenga conto del movimento dei corpi rigidi durante il deltatime, e pertanto sarà impossibile stabilire anche solo preliminarmente se due oggetti potranno potenzialmente collidere all'interno del deltatime o meno. Ragion per cui, solo durante il calcolo delle collisioni a tempo continuo (che ricordiamo essere la sola modalità permessa dall'engine realizzato) i collider dei RigidBody dinamici verranno rimpiazzati con dei 'volumi di movimento', meglio conosciuti come motionSweptCollider o 'swept volume', ovvero dei collider che avvolgono il movimento eseguito dal corpo rigido durante il deltatime (che ricordiamo avvenire in linea retta per via dell'approssimazione usata nell'algoritmo SweptAABB), un esempio di tale collider è visionabile in figura 3.10. L'utilizzo dei motionSweptCollider permette di utilizzare il SAT come collision test preliminare, in quanto l'intersezione di due volumi di movimento di tipo AABB implica una probabilità di collisione. Dato che l'engine realizzato si basa su collider di tipo AABB, anche i motionSweptCollider saranno di tale tipo, anche se chiaramente sarebbe stato preferibile l'uso degli OBB, in quanto come si evince dal corpo rigido A in figura 3.10, il motionSweptCollider di tipo AABB generato avvolge veramente male il moto di A lasciando molto spazio vacante, ciò porterà ad un maggior numero di coppie false positive generate dalla broad phase, un motionSweptCollider di tipo OBB invece avvolgerebbe molto meglio il moto rettilineo degli oggetti aumentando l'efficacia dei test preliminari, e quindi l'efficacia della broad phase. Una volta generati i motionSweptCollider dei corpi rigidi dinamici con aggiornamento a tempo continuo, essi dovranno essere reindicizzati all'interno della struttura di partizionamento spaziale (in quanto il loro collider è stato sostituito dal relativo volume di movimento), a questo punto è possibile applicare la stessa broad phase della collision detection a tempo discreto ottenendo così tutte le coppie di corpi rigidi che potenzialmente collideranno (dopo tale operazione, i motionSweptCollider

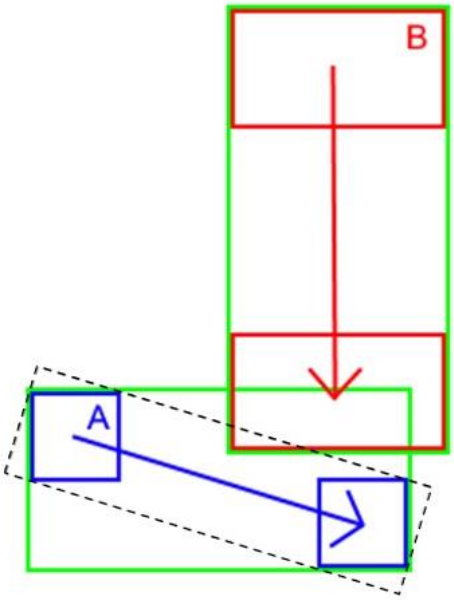

Figura 3.10: Esempio di motion swept colliders.

vanno rimpiazzati con i collider precedenti e la struttura di partizionamento spaziale andrà essere reindicizzata).

Oltre gli approcci citati, ne esistono di altri aventi l'obiettivo di minimizzare ancor di più il numero di intersection test accurati che l'engine dovrà effettuare a valle della broad phase, tra questi citiamo l'algoritmo 'Sweep and Prune', conosciuto anche come 'Sort and Sweep' [53].

Narrow Phase

Una volta stabilito tramite broad phase che una coppia di corpi rigidi può potenzialmente collidere, allora avrà senso applicare il test di collisione più accurato, che nel caso di collision detection a tempo discreto ci dirà se i due collider stanno intersecando, mentre nel caso di collision detection a tempo continuo ci dirà se e quando i due collider entreranno in contatto. Come detto in precedenza, nell'engine realizzato i collider supportati sono esclusivamente di tipo AABB, pertanto applicare la broad phase nel calcolo delle collisioni a tempo discreto vorrà dire eseguire già il test di intersezione più accurato (in quanto nella broad phase il test di collisione preliminare viene eseguito su AABB), e quindi la narrow phase sarà implicita. Nel caso continuo invece, la narrow phase si tradurrà nell'eseguire l'algoritmo SweptAABB [50] sulle varie coppie di corpi rigidi potenzialmente collidenti trovati nella broad phase.

3.5.5.6 Discrete VS Continuous Collision Detection

Tipicamente, come già accennato nei precedenti paragrafi, gli algoritmi di continuous collision detection sono più complessi di quelli per il calcolo delle collisioni a tempo discreto, questo deriva dal fatto che a tempo continuo bisogna tener conto anche del moto dell'oggetto durante tutto il deltatime, moto che nell'algoritmo SweptAABB viene approssimato con un moto rettilineo a velocità costante. In linea generale, non solo gli algoritmi di intersezione, ma tutta la fase di continuous collision detection è notevolmente più complessa della controparte a tempo discreto, questo deriva dal fatto che la continuous collision detection, come visto nel paragrafo riguardante il physics engine, dovrà determinare la collisione che all'interno del deltatime avviene per prima tra tutti i RigidBody facenti parte della simulazione, aggiornare quest'ultima fino all'istante di collisione, risolvere la collisione e ripartire alla ricerca di una nuova collisione che potrebbe verificarsi all'interno del deltatime rimanente, tutto ciò fino a quando il deltatime non sarà stato esaurito. Pertanto, è palese come la collision detection a tempo continuo richieda potenzialmente diverse iterazioni durante il deltatime, oltre a coinvolgere una serie di update parziali della simulazione che invece non sono presenti a tempo discreto, in quest'ultimo caso infatti si tratterà di eseguire una sola iterazione. La maggior complessità che si ha a tempo continuo porta comunque diversi benefici che invece non si hanno a tempo discreto, esaminiamo quindi quali sono i potenziali vantaggi e svantaggi per entrambi i tipi di collision detection.

Continuous:

+ Estrema accuratezza della simulazione.
+ L'utente ha la possibilità di innescare delle azioni esattamente agli istanti di collisione degli oggetti, ed avrà altresì una visione coerente della scena a quell'istante.
+ Il vettore normale alla superfice di collisione è estremamente più accurato e non lasciato ad un'euristica come nel caso discreto.

- Algoritmi di collision test più complessi e potenzialmente diverse iterazioni all'interno del deltatime.

Discrete:

+ L'intera fase di detection è computazionalmente leggera.
+ Algoritmi di collision test computazionalmente semplici rispetto alla controparte continua.
- L'accuratezza della simulazione è strettamente correlata con l'intervallo di update, minore è tale intervallo maggiore sarà l'accuratezza così come sarà maggiore il numero di risorse richieste dall'engine, in quanto si avranno più cicli di update fisico al secondo.
- Per oggetti veloci o molto stretti su certi assi, l'intervallo di update potrebbe non essere sufficiente e generare problemi di collision detection miss come quello del tunneling [53], dove un oggetto viene letteralmente trapassato senza che l'engine sia in grado di rilevare la collisione.

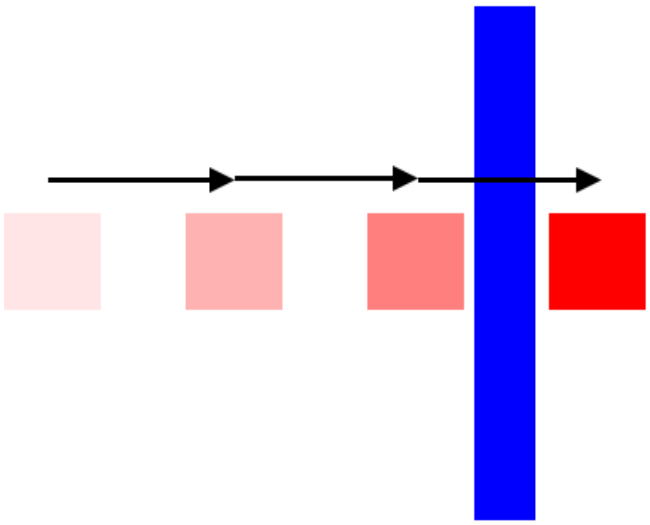

Figura 3.11: Fenomeno di tunnelling causato da collision detection a tempo discreto.

- Il vettore normale alla superfice di collisione è calcolato tramite euristica, la più usata sceglie l'asse con minor compenetrazione come asse del vettore normale. Essendo esso fondamentale nella fase di collision resolution, una cattiva stima può condurre ad effetti indesiderati come il teletrasporto di un corpo rigido da un punto ad un altro, oltre che avere ripercussioni e causare bug nell'applicazione stessa. A tal proposito si vedano i vari glitch di wall jump e wall clipping di serie come Super Mario Bros e titoli affini.

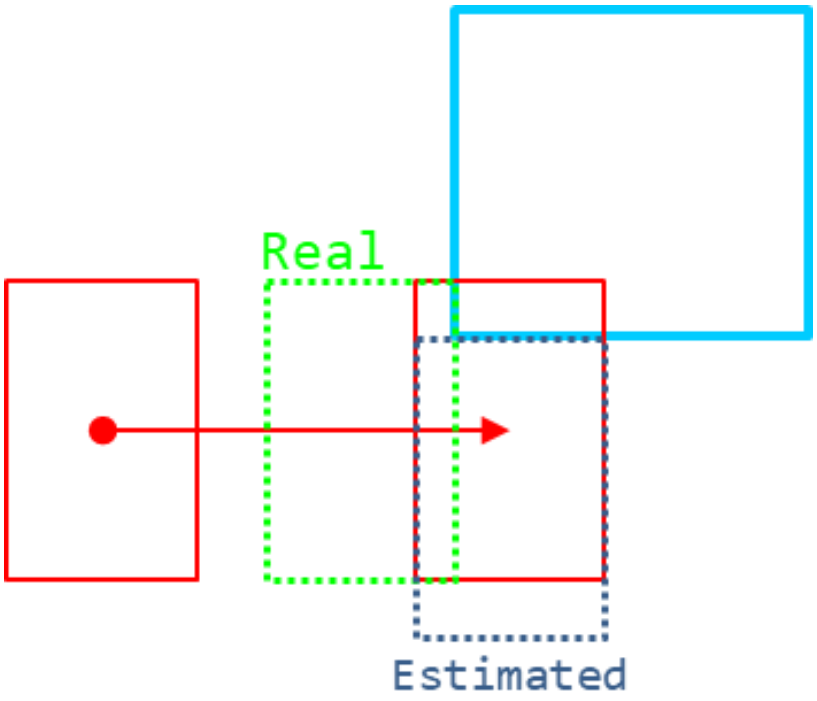

Figura 3.12: Esempio di errato vettore normale alla collisione generato a causa dell'euristica dell'asse con minor compenetrazione.

3.5.5.7 Collision Resolution

Una volta determinata una collisione tra due oggetti e calcolati tutti i dati necessari, essa va risolta. Risolvere una collisione vuol dire fare in modo che gli oggetti coinvolti non collidano più come si andrà avanti con la simulazione. Ciò può essere ottenuto tramite diversi approcci, tra i quali citiamo il metodo della proiezione, il metodo dell'impulso, il metodo della penalità, o una combinazione di questi. Il metodo della proiezione insieme a quello dell'impulso vengono spesso utilizzati per la risoluzione di collisioni a tempo discreto, il metodo della proiezione viene utilizzato prevalentemente per collisioni inerenti soft body, mentre per collisioni a tempo continuo su corpi rigidi si usa tipicamente il solo metodo dell'impulso, che sarà pertanto l'approccio discusso in questa tesi.

Una volta individuata una collisione a tempo continuo, l'engine dovrà aggiornare la simulazione per il solo tempo di collisione tramite l'algoritmo di integrazione scelto per la stima delle proprietà cinematiche degli oggetti, portando così i due corpi esattamente alla posizione d'impatto alla giusta velocità. A questo punto bisognerà risolvere la collisione tramite il metodo dell'impulso. Tale metodo modella la risposta alla collisione tra due corpi rigidi seguendo quelle che sono le leggi fisiche della meccanica classica relative alla conservazione della quantità di moto in un sistema isolato e ad urti elastici ed anelastici [57]. In un tale sistema, i due corpi al momento della collisione ricevono un impulso opposto alla loro direzione di moto ed inversamente proporzionale alla loro massa, impulso che dipenderà anche dall'elasticità dei due corpi. L'elasticità o meno di un corpo dipende fisicamente dalla materia del quale è composto, ciò all'interno dell'engine viene modellato assegnando un cosiddetto 'coefficiente di restituzione' ai RigidBody in fase di creazione, coefficiente che varia da 0 ad 1, questo determina l'elasticità che quel corpo rigido eserciterà in fase di collisione. L'elasticità della collisione sarà quindi data dalla moltiplicazione tra i coefficienti di restituzione dei due corpi coinvolti nella collisione, a 0 si avrà una collisione

completamente anelastica, fino ad arrivare ad 1 con una collisione completamente elastica. Ciò detto, l'impulso che riceverà il corpo A verrà calcolato tramite la seguente formula:

$$J = \frac{|V_{B,A}| * (1 + K_A * K_B)}{{m_A}^{-1} * {m_B}^{-1}} \tag{3.6}$$

$$V_{B,A} = velocità\ di\ B\ rispetto\ ad\ A$$
$$K_A; K_B = coefficiente\ di\ restituzione\ di\ A\ e\ B$$
$$m_A; m_B = massa\ di\ A\ e\ B$$

Ottenuto l'impulso, questo andrà sommato alla velocità dei corpi A e B in modo inversamente proporzionale alla loro massa:

$$V_A \mathrel{+}= {m_A}^{-1} * J\hat{n} \tag{3.7}$$
$$V_B \mathrel{+}= {m_B}^{-1} * J\hat{n}$$

$$\hat{n} = collision\ normal$$

A livello programmatico, l'utente ha la possibilità di poter settare delle callback che verranno invocate al momento della collisione, prima che l'engine abbia effettivamente risolto quest'ultima tramite il metodo illustrato.

3.5.5.8 Raycasting

Si tratta di uno strumento di grande importanza sia a livello applicativo che a livello di engine in quanto può trovarsi alla base di diverse funzionalità di quest'ultimo. Come suggerisce il nome, il raycasting, a livello concettuale, consiste nel tracciare un raggio da un punto ad un altro del mondo, allo scopo di rilevare se durante il tragitto è stato colpito qualcosa, in un contesto come quello di un physics engine, quel qualcosa sarà un corpo rigido avente un certo collider. Le soluzioni per l'implementazione del raycasting spaziano da metodi analitici a metodi numerici ed ovviamente variano in base al tipo di collider presenti nella scena. L'approccio scelto in questo engine consiste nell'usare l'algoritmo SweptAABB [50], già introdotto precedentemente relativamente al calcolo delle collisioni a tempo continuo, in uno dei suoi casi limite, ovvero usando come collider A un collider ad-hoc avente extent nullo ed origine nel punto di partenza del raggio, e come collider B il collider contro il quale testare l'intersezione col raggio. L'algoritmo SweptAABB permette concettualmente di fare un 'casting' del collider A lungo la sua traiettoria di moto per vedere se e quando intersecherà il collider B, ritornando in caso positivo la percentuale del vettore spostamento di A alla quale avviene la collisione, ciò si adatta molto bene ad un'implementazione analitica del raycasting, in quanto azzerando l'extent del collider A allora quest'ultimo si ridurrà ad un punto, che concettualmente verrà proiettato lungo la direzione di moto dando vita ad un raggio. Ancora una volta, l'utilizzo di strutture di partizionamento spaziale discusso nei paragrafi precedenti risulta cruciale al fine di ridurre il numero di intersection test da effettuare per determinare con quali corpi rigidi il raggio intersecherà. Il raycasting come detto ad inizio paragrafo viene spesso utilizzato a livello applicativo, ed è stato anche utilizzato per realizzare uno dei giochi POC di questo engine, tra i casi d'uso abbiamo il ricavare l'oggetto sul quale l'utente ha

cliccato, stabilire se un personaggio si trova per aria o a terra, stabilire se si è colpito un oggetto in un gioco stile fps, e molti altri.

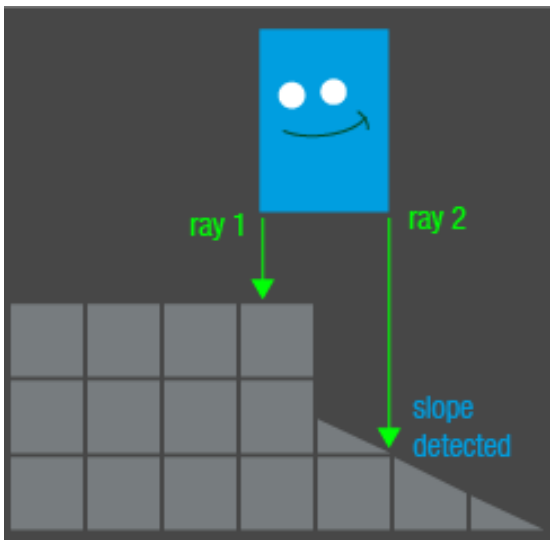

Figura 3.13: Raycasting utilizzato per determinare se un oggetto si trova sopra ad un dislivello.

3.6 Rendering

Quando il physics engine ha concluso la computazione della simulazione, le eventuali animazioni sugli oggetti sono state aggiornate e l'utente ha applicato la sua logica di scripting, allora la scena, che a livello grezzo consiste in un insieme di triangoli ai quali sono associate delle proprietà grafiche, sarà pronta per essere trasformata in una griglia di pixel a partire da un punto di vista ben preciso (tipicamente la camera), producendo quindi un'immagine che in computer grafica prende il nome di render, da cui il nome 'rendering' che rappresenta l'insieme di operazioni necessarie per produrre tale immagine.

3.6.1 An Overview on Modern Computer Graphics Rendering Pipeline

3.6.1.1 Rendering Techniques

Per compiere il processo di rendering esistono diverse tecniche, tra le quali le più conosciute ed utilizzate sono sicuramente il ray tracing e la rasterizzazione [58][59][66].

Ray Tracing

Si tratta di una popolare tecnica di rendering atta a generare immagini molto realistiche in quanto permette di ottenere abbastanza facilmente effetti visuali complessi, dato che questi sono insiti nella tecnica stessa [60]. Come ci si può immaginare dal nome, alla base del ray tracing ci sono dei raggi, questi vengono proiettati all'interno della scena a partire dalla posizione di ogni pixel dell'immagine che si vuole renderizzare, pertanto si avranno tanti raggi quanti sono i pixel dell'immagine. Tali raggi andranno possibilmente a collidere con un oggetto presente in scena, acquisendone le proprietà grafiche, a seconda di tali proprietà potranno essere creati altri raggi a partire dal punto d'impatto, raggi che andranno alla ricerca di altri oggetti dai quali acquisire le proprietà grafiche e combinarle con quelle ottenute fino a

quel momento. Da quanto descritto finora, è palese come tale approccio sfrutti le proprietà fisiche della luce e cerchi di ricalcare il funzionamento di strumenti di cattura quali la macchina fotografica digitale, pertanto anziché focalizzarsi su tutto il possibile flusso luminoso che si avrebbe in una scena, il ray tracing si focalizza solamente su quei raggi che attraverso l'obiettivo andranno a ricadere sul sensore per essere poi campionati su una matrice di pixel. Gli pseudo passi alla base del più semplice algoritmo di ray tracing potrebbero essere i seguenti:

- **per ogni pixel P dell'immagine**
  - crea un raggio **R**
  - **per ogni primitiva O della scena**
    - se **R** interseca **O**
      - trova il colore corretto tenendo conto delle proprietà grafiche di **O** e del punto di intersezione
      - assegna a **P** il colore trovato

La resa grafica raggiunta dal ray tracing tipicamente si paga in termini di tempo, in quanto il numero di raggi creati così come il numero di operazioni effettuate con essi è veramente elevato. Per tali motivi, questa tecnica viene impiegata da decenni, ma anche tuttora, in campo cinematografico, dove ci si possono permettere tempi di rendering della durata di ore, come in Toy Story 3 dove si aveva una media di un frame ogni 7 ore. Tuttavia, questi ultimi anni hanno visto la nascita delle prime tecniche di ray tracing utilizzate in ambiente real-time, e quindi anche nel mondo dei videogiochi. Ciò è stato reso possibile da diversi fattori: 1) a livello algoritmico il ray tracing è implicitamente parallelizzabile, ciò lo si evince anche dallo pseudo codice visto sopra, le operazioni svolte su ogni raggio che parte da un dato pixel possono essere effettuate in modo completamente indipendente dagli altri raggi, ciò rende le attuali schede video l'ambiente ideale per eseguire tali computazioni in parallelo; 2) i maggiori produttori di schede grafiche, ovvero NVIDIA [64] e AMD [65], hanno recentemente iniziato ad includere all'interno delle schede un vero e proprio supporto hardware per operazioni di ray tracing tramite dei core dedicati [62].

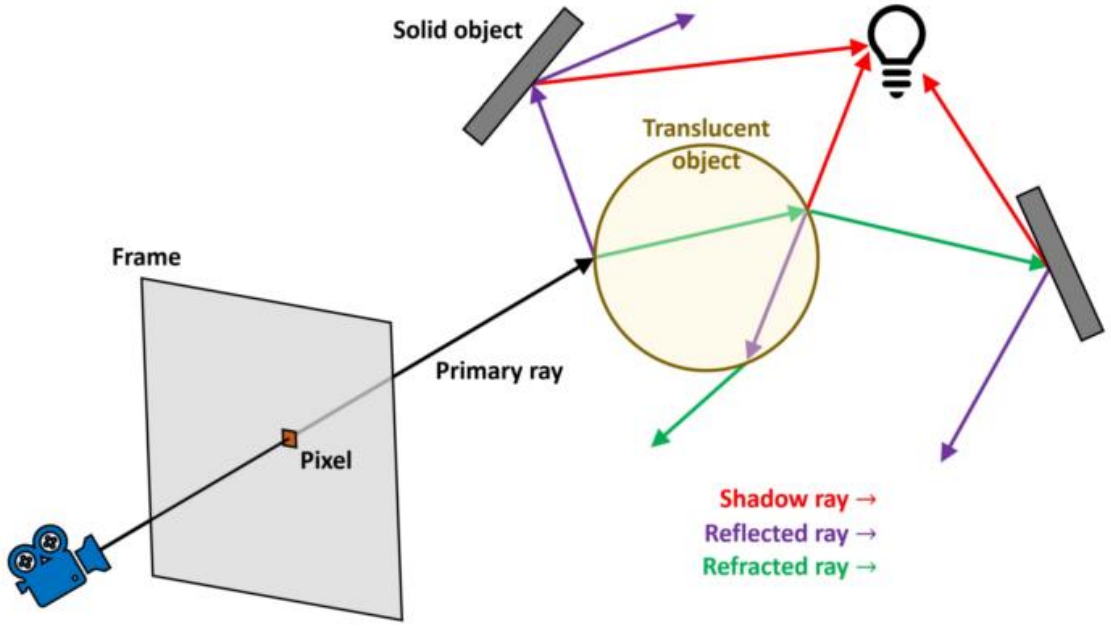

Figura 3.14: Esempio di possibili intersezioni e diramazioni di un raggio durante il ray tracing.

Rasterization

La tecnica della rasterizzazione si contrappone dal punto di vista logico alla tecnica del ray tracing. Se in quest'ultima si partiva dalla posizione dei pixel dell'immagine e si proiettavano dei raggi al fine di esaminare la scena e determinare il colore del pixel, nella rasterization si parte dagli oggetti in scena e si utilizzano delle trasformazioni geometriche per proiettare i triangoli di cui sono composti tali oggetti sulla griglia dei pixel dell'immagine, il colore di un pixel sarà poi ricavato a partire dalle proprietà grafiche dei triangoli che sono stati proiettati su quel pixel [61]. Questa dualità è riscontrabile anche schematizzando gli pseudo passi del più semplice algoritmo di rasterizzazione e confrontandoli con quelli relativi al ray tracing:

- **per ogni primitiva O della scena**
  - proietta **O** sulla griglia dei pixel dell'immagine
  - applica il processo di rasterizzazione sulla proiezione 2D ottenuta
    - **per ogni pixel/frammento P prodotto**
      - trova il colore corretto a partire dalle proprietà grafiche di **O**
      - assegna a **P** il colore trovato

La rasterizzazione in sé non è nient'altro che quel processo che determina quali pixel della griglia ricadono all'interno di un dato triangolo una volta che questo è stato proiettato sulla griglia, nulla di più [67], si tratta quindi di un processo che andrà eseguito per ogni triangolo della scena. Da questa definizione si evince chiaramente che l'operazione di rasterization da sola non è in grado di produrre un render vero e proprio, ma va accompagnata da una serie di operazioni sia a valle che a monte che vanno a caratterizzare una vera e propria pipeline di operazioni che danno vita al processo di rendering basato su rasterization. A differenza del ray tracing dove tutto è basato sull'intersezione dei raggi con gli oggetti di scena, l'approccio basato su rasterizzazione sfrutta prevalentemente trasformazioni geometriche e pertanto è molto veloce, ciò ha fatto si che diventasse negli anni l'approccio più utilizzato in ambito computer grafica real-time e soprattutto l'approccio per il quale le GPU sono state pensate, portando quindi molti step della pipeline di rendering basata su rasterization ad essere supportati via hardware. La semplicità dell'approccio di rasterization si ripercuote comunque sulla sua resa grafica, molto approssimata rispetto a ciò che rappresenta la realtà, pertanto negli anni si sono sviluppati numerosi algoritmi e tecniche per ottenere effetti grafici complessi basati su rasterizzazione.

Si conclude la breve disamina su rasterizzazione e ray tracing con una citazione del vice presidente della divisione graphics research di NVIDIA David Luebke, che dice: "*Rasterization is fast, but needs cleverness to support complex visual effects. Ray tracing supports complex visual effects, but needs cleverness to be fast*" [63].

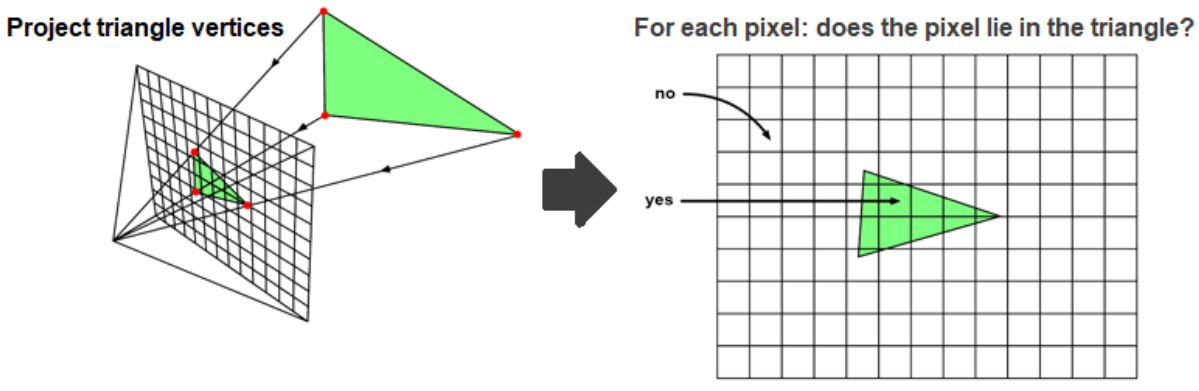

Figura 3.15: Funzionamento della tecnica di rasterization.

3.6.1.2 Fundamental Graphical Data

Alla base dei moderni approcci di rendering stanno dei dati fondamentali, ovvero vertici, shader programs e texture [68]. Questi costituiscono le fondamenta di qualsiasi moderna applicazione di computer grafica. Tali dati sono necessari per il corretto svolgimento degli step della pipeline di rendering, pertanto essi risiederanno generalmente nella memoria della GPU, se così non fosse vorrà dire che per loro è stato previsto un caricamento on-demand, ovvero solo quando necessario, oppure semplicemente la memoria RAM della scheda grafica non è sufficiente e pertanto risiederanno nella memoria RAM del calcolatore in attesa di essere richiamati, in ogni caso essi dovranno comunque essere presenti nella RAM della GPU prima del loro utilizzo.

Vertici

Nella computer grafica, qualsiasi figura geometrica, a prescindere dal suo livello di complessità, sarà composta da un insieme di vertici, quest'ultimo viene comunemente definito mesh. Tuttavia, il concetto di vertice nella computer grafica presenta degli aspetti aggiuntivi rispetto al vertice per come lo si conosce in geometria. Il vertice utilizzato nella computer grafica possiede, come in geometria, delle coordinate generalmente definite nello spazio tridimensionale, ma può anche possedere attributi aggiuntivi come un colore, delle coordinate nello spazio UV, un vettore normale, ed altre informazioni che tipicamente servono per realizzare effetti grafici avanzati.

Shader Programs

Si tratta di piccoli programmi, quasi degli script, eseguiti all'interno della scheda grafica, che trovano posto in diversi step della pipeline di rendering ed assolvono a svariati compiti, da quello di proiettare i vertici di un oggetto 3D sullo schermo allo stabilire il colore finale di un pixel dell'immagine. Per la scrittura del codice degli shader si usano dei linguaggi di alto livello ad-hoc che generalmente tendono ad essere simili al C, come ad esempio GLSL [47] ed HLSL [70]. Gli shader vengono suddivisi in diversi tipi in base allo step della pipeline di rendering dove vengono eseguiti, tra i tipi più comuni abbiamo il Vertex, Geometry e Fragment shader. Ogni tipo di shader program prende degli input e butta fuori degli output, alcuni di questi sono obbligatori in base al tipo di shader, altri sono definibili dall'utente. Più tipi di shader program saranno poi combinati insieme per dar vita a quello che viene chiamato shader executable. Come detto in precedenza, gli shader vengono eseguiti all'interno della scheda grafica, più nel dettaglio essi verranno eseguiti in parallelo dalle centinaia (se non migliaia) di core che le GPU mettono a disposizione, molti di questi core sono dedicati ad eseguire un determinato tipo di shader, altri sono general-purpose permettendo così l'esecuzione di shader di qualsiasi tipo. Vediamo adesso in breve l'utilità dei tre tipi di shader più usati esaminandoli in base al loro ordine all'interno della pipeline di rendering:

- Vertex: i vertex shader prendono un vertice in input ed hanno il compito di trasformare le sue coordinate in clip space. Vedremo in seguito cosa si intende per clip space, per adesso si può immaginare come se fosse la posizione del vertice sullo schermo.
- Geometry: questi shader ricevono in input una primitiva e buttano fuori 0 o più primitive, non per forza dello stesso tipo della primitiva in input. Essi servono solitamente per aumentare il dettaglio geometrico o diminuirlo in base a determinati fattori come potrebbe essere la distanza della primitiva dalla camera, in tal caso tale

processo viene denominato ‘level of detail’, in quanto controlla il livello di dettaglio delle geometrie sulla base di determinati fattori.

- Fragment: giunti a questo step, i fragment shader prendono in input un fragment, ovvero l’output del processo di rasterizzazione, e calcolano il colore del relativo pixel sulla base degli attributi del fragment, ma anche attingendo ad informazioni di illuminazione ed eventuali attributi specificati dall’utente come texture, vettori, scalari, ecc.

Texture

Anche se si può fare a meno di loro all’interno della pipeline di rendering, oggi costituiscono comunque un elemento fondamentale per l’ottenimento di effetti visivi avanzati. Le texture altro non sono che immagini nel vero senso della parola, esse vengono caricate nella memoria della scheda grafica ed usate all’interno di vari tipi di shader per dar vita ad un numero veramente enorme di effetti grafici, dal semplice uv mapping al displacement dei vertici di una mesh e molto altro ancora. Dato il loro utilizzo spropositato da parte degli shader, l’accesso ai pixel di una texture, che nel contesto della computer grafica prendono il nome di texel, deve essere quanto più veloce possibile, pertanto è sensato che le texture risiedano nella memoria della scheda grafica in modo raw, ovvero senza compressione di alcun tipo. Ciò che avviene nella realtà è che le texture possono essere mantenute in modo compresso al fine di risparmiare memoria [71], si pensi ad esempio ai dispositivi mobile dove la memoria è veramente poca. A tal proposito vengono utilizzati degli algoritmi di compressione lossy studiati ad-hoc per ottimizzare la decodifica, e quindi l’accesso randomico, ai vari texel senza impattare sulla velocità di accesso. Tali algoritmi essendo lossy vanno ovviamente a degradare la qualità della texture, ciò può comunque notarsi poco in base al contenuto della stessa. Le GPU inoltre, possono fornire supporto hardware a tali algoritmi di compressione, rendendo accessibili queste funzionalità tramite estensioni specifiche del venditore alle librerie grafiche più conosciute. Tra i vantaggi di avere delle texture compresse, oltre ad un requisito di memoria inferiore, si ha anche un aumento della texel lookup rate che deriva da una riduzione dei dati scambiati sul bus della GPU. Pertanto in scene graficamente complesse, l’uso di texture compresse può comunque essere un vantaggio non da poco. Nell’engine realizzato le texture non vengono compresse, in quanto si vuole una qualità delle texture pari all’originale.

3.6.1.3 Primitives and Winding Order

Il termine primitiva viene spesso utilizzato in ambito computer grafica per riferirsi ad un particolare tipo di mesh, ovvero quelle mesh che sono nativamente supportate dalla libreria grafica che si sta utilizzando. Una primitiva è definita da una lista di vertici e dal modo in cui questi verranno interconnessi. Le primitive costituiscono la base di tutte le mesh più complesse che utilizzerà l’utente all’interno della sua applicazione. Tra le varie primitive esistenti il triangolo è sicuramente quella più conosciuta ed utilizzata, anche perché le GPU sono progettate per renderizzare nel modo più efficiente proprio i triangoli, ma ne esistono anche di altre che andremo adesso ad esaminare:

- Points: un insieme di punti che verranno disegnati per come sono, senza nessuna interconnessione.
- Lines: tale primitiva consente di disegnare una linea tra ogni coppia di vertici che verranno specificati.

- Line Strip: verrà disegnata una linea tra ogni vertice ed il precedente nella lista.
- Line Loop: uguale alla Line Strip, ma verranno congiunti anche l'ultimo vertice con il primo.
- Triangles: disegna un triangolo per ogni tripletta di vertici presente nella lista.
- Triangle Strip: verrà disegnato un triangolo per ogni vertice ed i due precedenti.
- Triangle Fan: simile al Triangle Strip con la differenza che come terzultimo vertice viene sempre utilizzato il primo vertice nella lista.

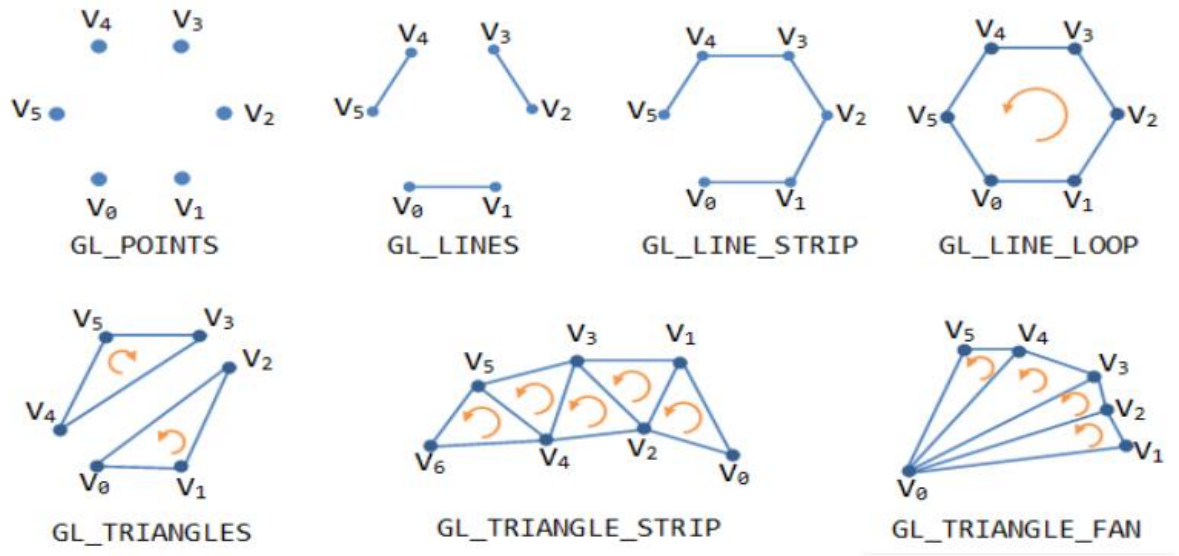

Figura 3.16: Principali primitive di disegno OpenGL.

La figura 3.16, oltre a mostrare le principali primitive della libreria grafica OpenGL, mostra anche delle frecce circolari su tutti quei poligoni chiusi che vanno quindi a comporre una superfice. Queste frecce rappresentano il cosiddetto 'winding order' della superfice poligonale [69], che concettualmente rappresenta il movimento rotatorio che viene generato durante l'interconnessione dei vari vertici della superfice da parte della libreria grafica. Tale movimento genera un vettore normale seguendo la regola della mano destra, un concetto noto in fisica e geometria, tale vettore indicherà la direzione frontale di quella data superfice, pertanto se il winding order è antiorario allora la superfice sarà front-facing, ovvero frontale rispetto all'osservatore, se orario allora la superfice sarà back-facing, ovvero di spalle rispetto l'osservatore. Il winding order viene determinato dall'ordine dei vertici della primitiva, se ad esempio invertissimo V4 con V5 nella primitiva GL_TRIANGLES di figura 3.16, allora il triangolo formato da V3 V4 e V5 avrebbe un winding order antiorario anziché orario. Il winding order è un concetto molto importante, in quanto determina quali facce di una mesh saranno rivolte verso l'osservatore e quali saranno rivolte in direzione opposta, queste ultime come vedremo possono essere soggette al cosiddetto back-face culling, ovvero rimosse dalla pipeline di rendering per risparmiare operazioni e tempo prezioso, in quanto rappresentano facce che con molta probabilità non saranno visibili dall'osservatore in quanto nascoste dalle altre facce dell'oggetto che invece saranno rivolte verso l'osservatore. E' possibile vedere un esempio di back-face culling, così come altri tipi di culling, in figura 3.17. L'associazione tra la tipologia di winding order e l'essere front-facing o back-facing viene generalmente stabilita dalla libreria grafica ed è un comportamento che può essere sovrascritto.

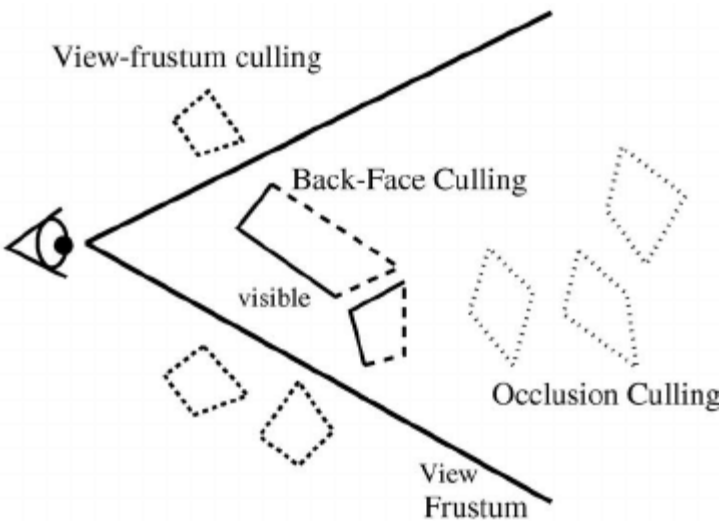

Figura 3.17: L'operazione di View-Frustum Culling scarta dalla pipeline di rendering le facce non visibili dal punto di vista dell'osservatore.

3.6.1.4 A Vertex Journey into Space: From Local to Canvas Space

In questo paragrafo esamineremo le varie trasformazioni cui i vari vertici di ogni singola mesh sono soggetti durante i vari step della pipeline di rendering. Tali trasformazioni avvengono per la stragrande maggioranza tramite moltiplicazioni matriciali, usando le cosiddette matrici di trasformazione. Partiremo quindi dal sistema di riferimento locale fino ad arrivare alla proiezione di un vertice sullo schermo.

Local Space

Questo sistema di riferimento è quello con il quale i vertici vengono definiti all'interno della mesh al momento della creazione della stessa, il centro di tale sistema di riferimento corrisponde quindi con il pivot della mesh, e le coordinate di tutti i vertici di cui essa si compone saranno definite a partire dalle coordinate di tale pivot. E' chiaro che il pivot funge solo da espediente concettuale al fine di determinare correttamente le coordinate dei vertici, esso infatti non fa parte della mesh, in quanto come abbiamo visto nei paragrafi precedenti, una mesh è composta esclusivamente da vertici.

World Space

Quando piazziamo una mesh all'interno della nostra scena, ovvero all'interno del nostro mondo, essa avrà una posizione ben precisa all'interno di quest'ultimo, tuttavia le coordinate locali dei vertici rimarranno tali. Ciò è possibile assegnando delle coordinate a ciò che prima avevamo detto esistere esclusivamente a livello concettuale, ovvero il pivot della mesh. Assegnando quindi della coordinate al pivot della mesh, stiamo sostanzialmente aggiungendo un offset alle coordinate locali dei vari vertici di cui è composta la mesh, in quanto ricordiamo che queste sono definite relativamente alla posizione del pivot. Tale offset matematicamente si traduce in una matrice di trasformazione di tipo traslazionale, che verrà pre-moltiplicata alle coordinate locali di ogni vertice per ottenere quindi la reale posizione del vertice all'interno del mondo. Oltre che ad una posizione, al pivot di una mesh possono anche essere assegnate una rotazione ed una scala, andando a formare una matrice di trasformazione che contiene sia informazioni di traslazione, sia informazioni di rotazione, sia informazioni di scala. Tale

matrice di trasformazione sarà una matrice 4x4 ed è quella contenuta all'interno del Transform component di cui si è discusso nel relativo paragrafo. La trasformazione delle coordinate locali di un vertice alle sue coordinate nel mondo si ottiene pre-moltiplicando la matrice di trasformazione assegnata al pivot per il vettore delle coordinate locali del vertice. La matrice di trasformazione del pivot prende in genere il nome di 'Model Matrix'. Il World Space è conosciuto anche come 'Global Space' o 'Model Space'.

Camera Space

Anche conosciuto come 'View Space' o 'Eye Space', questa trasformazione è il primo vero step per il mapping degli oggetti sullo schermo e consiste nell'ottenere le coordinate dei vertici relativamente alla posizione dell'osservatore all'interno del mondo, come se l'osservatore diventasse quindi l'origine del sistema di riferimento del mondo. Questo ci da una visione geometrica del mondo visto dal punto di vista della camera. Matematicamente, ciò si ottiene semplicemente applicando ai vertici la trasformazione inversa dell'osservatore, ovvero pre-moltiplicando l'inverso della matrice di trasformazione della camera per la posizione in world space di ogni vertice. La matrice di trasformazione dell'osservatore/camera è spesso soprannominata 'View Matrix'.

Clip Space

Il punto di vista dell'osservatore non è caratterizzato solamente da una matrice di trasformazione come visto pocanzi, ma da ben due matrici, dove una è appunto la View Matrix, e la seconda è la matrice che rappresenta la distorsione visiva dell'osservatore, che generalmente può essere di tue tipi, prospettica ed ortografica. Tale matrice in sostanza rappresenta la matrice di trasformazione del view frustum, ovvero quella distorsione da applicare ai vertici della scena affinché essi subiscano la deformazione descritta dal frustum, e prende il nome di 'Projection Matrix'. Tale matrice identifica anche quello spazio nel quale il frustum della camera, prospettico o ortografico che sia, rappresenta un cubo con origine al centro del sistema, ciò vorrà dire che qualsiasi punto contenuto all'interno del frustum verrà mappato all'interno di questo cubo, qualsiasi punto esterno al frustum cadrà invece fuori dal cubo. Lo spazio nel quale risiede tale cubo viene chiamato 'Clip Space', mentre il cubo assume solitamente il nome di 'Clip Cube' o 'Clipping Volume', ad indicare il processo di clipping della pipeline di rendering che impiega proprio tale volume per determinare quali oggetti o pezzi di oggetti ricadono all'interno dell'inquadratura e quali no, questi ultimi potranno ovviamente non essere renderizzati, e quindi scartati dalla pipeline di rendering al fine di evitare operazioni inutili. Per trasformare, o meglio per proiettare le coordinate di un vertice dal camera space al clip space queste dovranno quindi essere pre-moltiplicate per la projection matrix. L'insieme delle operazioni viste finora per trasformare le coordinate di un vertice dal local space al clip space vengono eseguite all'interno del vertex shader mettendo in catena le varie trasformazioni matriciali, pertanto la trasformazione delle coordinate di un vertice in clip space all'interno di un vertex shader avverrà in questo modo:

$$V_{\text{clip}} = M_{\text{projection}} \cdot M_{\text{view}} \cdot M_{\text{model}} \cdot V_{\text{local}} \tag{3.8}$$

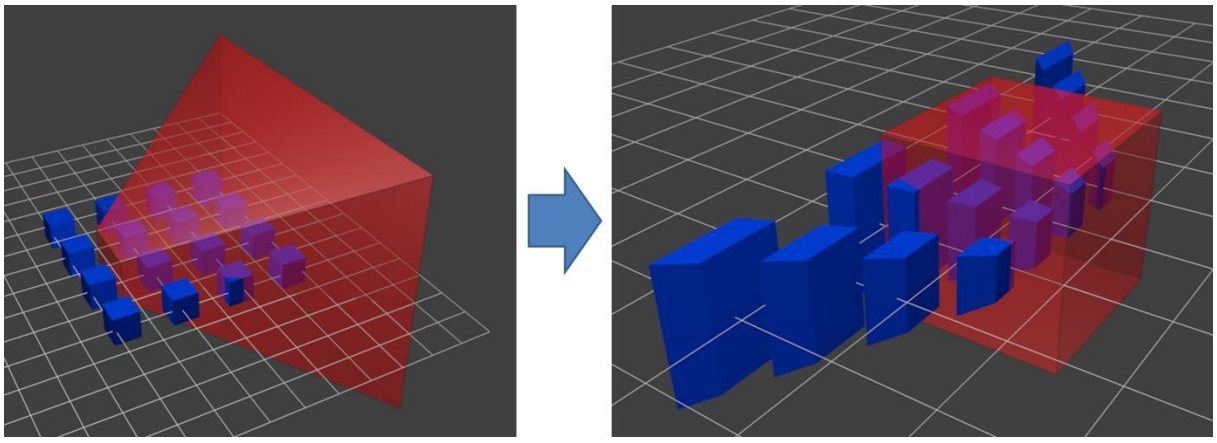

Figura 3.18: Esempio grafico di passaggio da camera-space a clip-space. Le primitive che non ricadono all'interno del clip-cube verranno scartate dalla pipeline di rendering.

NDC & Perspective Divide

Una importante caratteristica della projection matrix è la manipolazione della componente w delle coordinate omogenee di un vertice in modo tale da diventare sempre più grande all'aumentare della distanza tra quel vertice e l'osservatore, tale coordinata sarà inoltre sempre >= alle altre. Ciò è ovviamente fatto di proposito per supportare la prossima operazione nella pipeline di rendering, chiamata Perspective Divide. Tale operazione viene eseguita automaticamente a valle della trasformazione eseguita dai vertex shader e consiste nel normalizzare le coordinate dei vertici in base alla loro componente w, in tal modo oggetti più lontani dall'osservatore verranno maggiormente rimpiccioliti rispetto a quelli più vicini, simulando quindi l'effetto prospettico. Tramite l'operazione di perspective divide tutti i vertici che si trovavano all'interno del Clip Space, verranno quindi mappati su un Clip Cube che si estende nell'intervallo [-1,1] su ogni asse, e la coordinata w di ognuno di loro sarà diventata 1, così facendo si dice che le coordinate dei vertici saranno state trasformate in Normalized Device Coordinates. In una projection matrix con distorsione ortografica la componente w non viene modificata e rimarrà quindi ad 1 (si ricorda che w vale 1 per i vertici, e 0 per i vettori), pertanto in un caso del genere è possibile ignorare l'operazione di perspective divide in quanto le coordinate dei vertici in clip space saranno già delle coordinate NDC. La libreria grafica OpenGL si aspetta dei vertici in tali coordinate prima di poter dare il via al processo di rasterizzazione.

Canvas Space

Le coordinate NDC dei vertici, in quanto normalizzate, possono essere facilmente mappate sul canvas dell'applicazione e quindi trasformate in 2D dando così il via al processo di rasterizzazione. Il canvas space è anche conosciuto con il nome di 'Window Space', ad indicare la finestra nella quale sta girando eseguita l'applicazione. Esso viene spesso chiamato anche 'Screen Space', tuttavia questo ultimo nome è fuorviante, in quanto per screen si intende di solito l'intero schermo del calcolatore, mentre un'applicazione grafica come sappiamo può essere eseguita anche in una porzione dell'interno schermo, pertanto quando la dimensione del canvas/window equivarrà alla dimensione dell'intero schermo allora avrà senso parlare di screen space.

3.6.1.5 Culling & Clipping

Chiunque abbia avuto modo di dover configurare una camera all'interno di una scena avrà avuto a che fare con la distanza dei piani di clipping della camera, ovvero il near ed il far clip plane. Questi, insieme ad altri parametri, determinano l'ampiezza del view frustum, ovvero il volume che conterrà tutto ciò che sarà visibile all'osservatore, e pertanto essi saranno determinanti nell'operazione di culling e clipping. Tali operazioni vengono eseguite subito prima di effettuare la normalizzazione delle coordinate dei vertici tramite perspective divide, pertanto operano sulle coordinate dei vertici in clip space, e consistono rispettivamente nello scartare dalla pipeline di rendering tutte quelle primitive, o parte di esse, che non rientrano all'interno del clipping volume e di conseguenza non fanno parte del frustum. I vertici che cadranno all'interno del clipping volume soddisferanno le disequazioni 3.9, pertanto basterà che una di queste non sia verificata per determinare la non appartenenza di un vertice al volume di clipping.

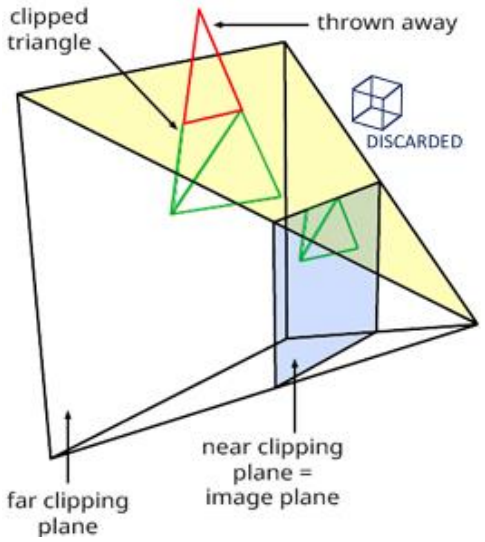

Figura 3.19: Operazione di culling e clipping.

$$
\begin{aligned}
-w \leq x \leq w \\
-w \leq y \leq w \\
-w \leq z \leq w
\end{aligned}
\tag{3.9}
$$

Le primitive completamente al di fuori del clipping volume saranno quindi soggette a culling, ovvero scartate dalla pipeline di rendering, quelle che lo intersecano parzialmente saranno soggette a clipping, ovvero 'ritagliate', in modo da buttar via la parte di primitiva che non interseca. Lo scartare una primitiva che non interseca il view-frustum viene chiamato 'view-frustum culling', tuttavia anche primitive che ricadono all'interno del frustum possono essere soggette a culling, nello specifico al 'Back-Face Culling' di cui si è già parlato in precedenza relativamente al winding-order delle primitive, tale operazione mira a scartare dalla pipeline tutte le primitive che non sono front-facing rispetto all'osservatore. Tra gli algoritmi di clipping più conosciuti citiamo l'algoritmo Cohen-Sutherland per le linee, e l'algoritmo Sutherland-Hodgman per i poligoni. Come si evince in figura 3.20, il clipping, per riuscire nel processo di ritaglio, potrebbe generare più triangoli e relativi vertici di quelli iniziali.

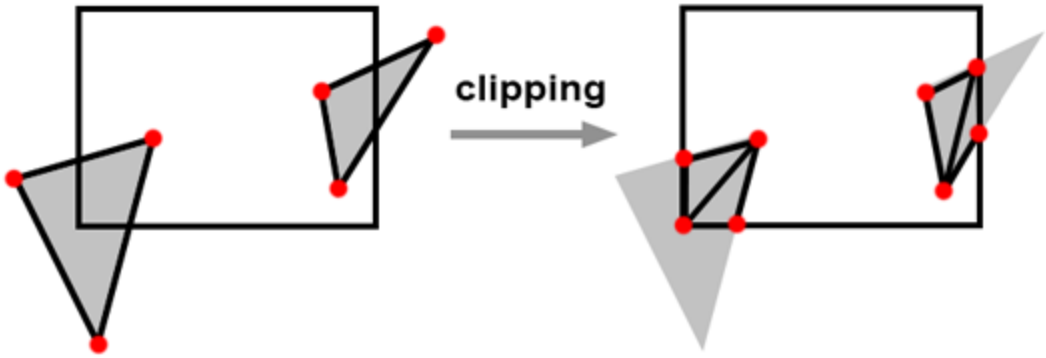

Figura 3.20: L'operazione di clipping durante il ritaglio potrebbe generare più triangoli e relativi vertici di quelli iniziali.

3.6.1.6 Rasterization

Una volta scartate tutte le primitive che non risultano visibili all'osservatore, bisognerà ricavare quali pixel del canvas ricoprono quali parti di primitive, tali pixel prenderanno il nome di 'fragment'. Questo avviene mediante il processo di rasterizzazione, già anticipato nel paragrafo relativo alle tecniche di rendering. Un banale algoritmo di rasterizzazione calcola dapprima un bounding box della primitiva, ovvero la minima area in pixel del canvas che la racchiude, dopodiché per ogni pixel all'interno di tale area viene eseguito un test che stabilisce se il centro di quel pixel ricade all'interno della primitiva, in caso affermativo verrà esso verrà selezionato come fragment. L'utilità del bounding box risiede nello scartare dal test tutti quei pixel che stanno al di fuori di esso, in quanto abbiamo la certezza matematica che essi non copriranno nessuna parte della primitiva.

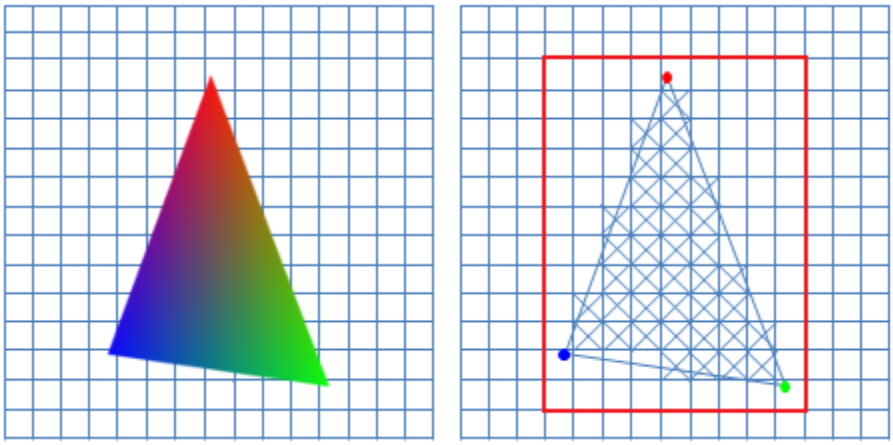

Figura 3.21: Operazione di rasterizzazione.

3.6.1.7 Interpolation

Identificati tutti i fragment di una data primitiva, i loro attributi dovranno essere interpolati a partire dagli attributi dei vertici della primitiva, ciò avviene tramite interpolazione lineare, nello specifico, tramite coordinate baricentriche. In tal modo, gli attributi di un qualsiasi fragment potranno essere ricavati mediante una combinazione lineare degli attributi dei vertici.

$$V_x = (\alpha \cdot V_0) + (\beta \cdot V_1) + (\gamma \cdot V_2) \quad (3.10)$$

dove

$$\alpha = Area(t_2)/Area(t)$$
$$\beta = Area(t_1)/Area(t)$$
$$\gamma = Area(t_0)/Area(t)$$

Una volta interpolati gli attributi, i frammenti saranno pronti per essere passati ai fragment shader per la successiva elaborazione. Questo step, insieme a quello di Rasterization, costituiscono una componente fissa della pipeline di rendering, e non saranno pertanto programmabili in alcun modo.

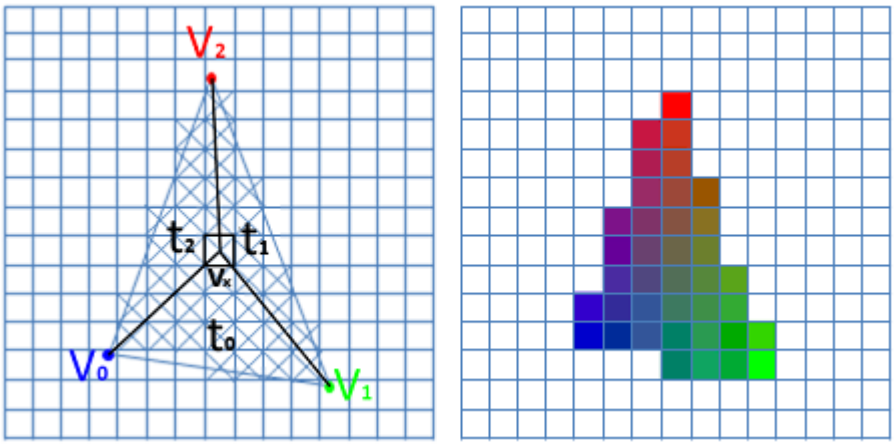

Figura 3.22: Gli attributi dei fragment di una data primitiva vengono interpolati tramite coordinate baricentriche.

3.6.1.8 Frame Buffer

Quando un fragment shader avrà terminato l'elaborazione ed il frammento avrà passato anche eventuali test come il depth test, lo stencil o l'alpha test, il colore del pixel verrà allora scritto su un buffer di pixel, ovvero la matrice di pixel che rappresenterà l'immagine finale, tale buffer viene chiamato frame buffer, o anche back buffer. Questa ultima nomenclatura suggerisce l'esistenza di un altro tipo di buffer, ovvero il front buffer, questo sarà quello che la scheda grafica espone come output a qualsiasi dispositivo voglia accederne, come ad esempio un monitor, mentre il back buffer è quello dove la scheda grafica scrive i risultati intermedi delle varie elaborazioni. Una volta che l'elaborazione di un frame sarà ultimata ed il back buffer sarà pronto, la libreria grafica comanda alla GPU di effettuare uno swap dei buffer, in tal modo viene esposto all'esterno il frame appena ultimato, e si può iniziare il render del successivo. L'approccio che fa uso di un front ed un back buffer prende tipicamente il nome il 'Double Buffering', alcune GPU supportano anche il 'Triple Buffering' che prevede l'utilizzo di due back buffer anziché uno.

3.6.1.9 Graphical Artifacts

Una volta che il frame è pronto per la visualizzazione i problemi non sono ancora finiti. Nella scheda grafica come prevedibile il tempo di rendering di un frame è variabile, non tutti i

frame richiedono lo stesso quantitativo di risorse. Tutti i dispositivi collegati in lettura ad una scheda grafica, quali monitor ed affini, utilizzano invece un refresh rate costante (tipicamente 60Hz e 120Hz) ed impiegano un certo lasso di tempo per acquisire tutto il frame buffer, frame buffer che verrà tipicamente letto verticalmente linea per linea. Ciò vuol dire che ci sarà sempre una asincronia tra il rendering-rate della scheda grafica ed il refresh-rate di un eventuale schermo. Tale asincronia può causare degli artefatti grafici [72], a volte anche molto evidenti nel caso di scene in rapido movimento.

Tearing

Lo screen tearing è esattamente l'artefatto che nasce quando l'asincronia tra rendering-rate e refresh-rate si fa marcata, come visibile in figura 3.24. In tal caso la scheda grafica potrebbe effettuare uno swap dei buffer durante l'operazione di refresh dello schermo, quest'ultimo si ritroverà pertanto con una parte di buffer vecchia ed una nuova, generando un effetto di taglio orizzontale dell'immagine. Lo screen tearing viene in genere risolto abilitando tramite la libreria grafica l'opzione 'Vertical Synchronization', spesso abbreviata in **V-Sync**, così facendo lo swap dei buffer avverrà in sincrono con il refresh-rate dello schermo, se il rendering del frame finisce prima dell'intervallo di refresh allora lo swap verrà ritardato fino alla fine dell'intervallo di refresh, se il rendering prende più tempo di un intervallo di refresh verrà comunque aspettato l'intervallo di refresh successivo. In quest'ultimo caso lo schermo mostrerà per due intervalli di refresh lo stesso identico frame come mostrato in figura 3.25, producendo un altro fenomeno indesiderato chiamato 'Stuttering', ovvero del lag introdotto dal forzato ritardo dell'operazione di swap dei buffer.

Figura 3.23: Artefatto grafico del tearing.

Stuttering

Lo stuttering è un problema causato dal V-Sync, e provoca un lag che è percepito dall'utente sotto forma di piccoli micro scatti dell'immagine, ovviamente molto fastidiosi, in quanto deteriorano la fluidità dell'applicazione. Tipicamente lo stuttering viene alleviato da un triplo buffering, in quanto una forte limitazione del V-Sync su doppio buffer è il dover attendere il tempo di refresh senza la possibilità di poter fare alcuna operazione, a meno di non fare un buffer swap. Introducendo un terzo buffer la scheda grafica può invece effettuare il buffer swap ed iniziare il rendering di un altro frame, senza dover quindi aspettare passivamente l'arrivo del prossimo ciclo di refresh.

Sia il Tearing che lo Stuttering sono oggi aggirabili tramite le nuove tecnologie per schermi NVIDIA **G-Sync** e AMD **FreeSync** [73]. Esse permettono di avere una sorta di refresh-rate adattivo, in tal modo lo schermo si aggiorna solamente quando la scheda grafica effettua lo swap dei frame buffer, eliminando quindi alla radice i problemi di asincronia.

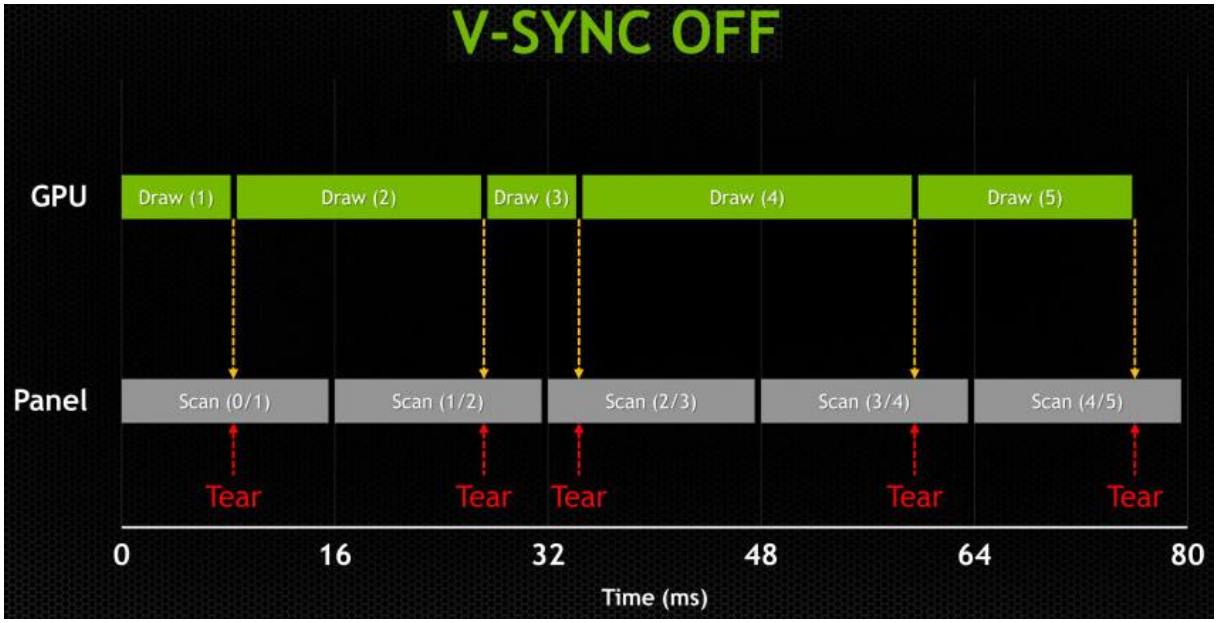

Figura 3.24: L'asincronia tra la frequenza di refresh dei monitor e la frequenza di buffer swap della GPU genera l'artefatto del tearing.

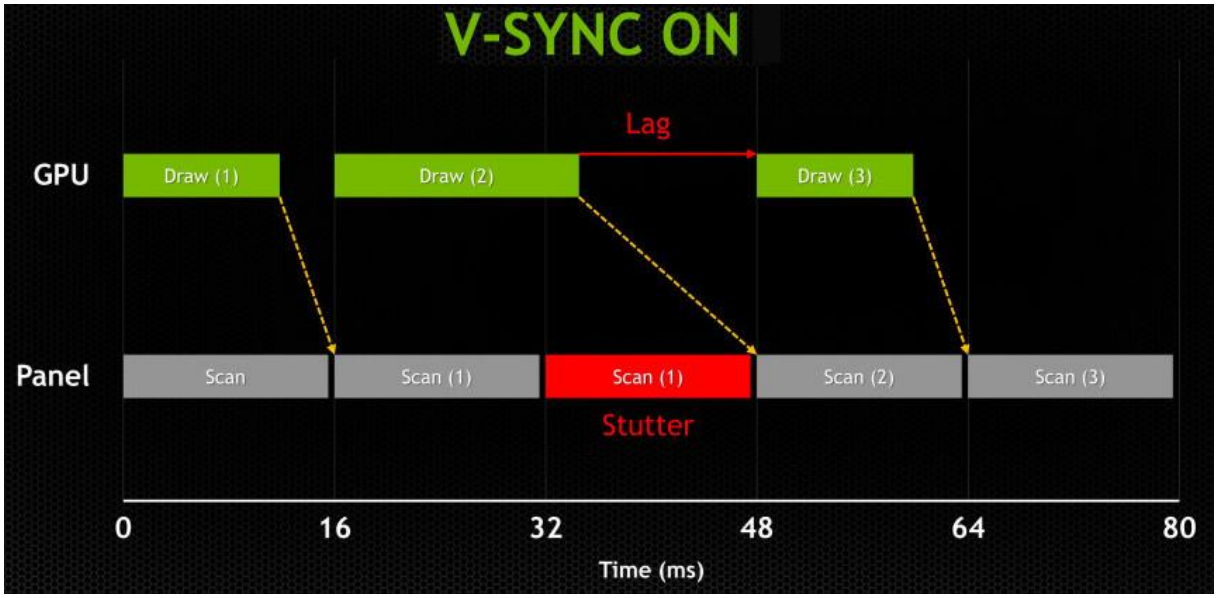

Figura 3.25: L'uso del V-Sync per risolvere problemi di tearing genera potenzialmente stuttering.

3.6.2 Graphics APIs

In questa sezione verrà fatta una breve panoramica sulle librerie che permettono di interfacciarsi con le attuali schede grafiche, dando poi maggiori dettagli sulla libreria utilizzata in questo engine, ovvero OpenGL.

3.6.2.1 Fixed VS Programmable Rendering Pipeline

Quando nacquero le prime GPU, esse si basavano su una pipeline di rendering fissa, ovvero dove i vari step della pipeline implementavano un insieme di funzionalità ben definite e non modificabili, al più parametrizzabili fino ad un certo punto. La pipeline agiva quindi come fosse una scatola chiusa, il programmatore non aveva modo di intervenire al suo interno ed era appunto limitato ad usare le funzioni che questa esponeva. Tali funzionalità erano sicuramente valide, c'era la possibilità di impostare diverse modalità di blending, di eseguire uno shading di tipo Gouraud per vertice, texturing ed uv-mapping, effetti di fog, ed anche stencil buffer per gli shadow volumes. Ciò portò alla nascita di svariati effetti grafici che per l'epoca risultavano ovviamente senza precedenti. Tuttavia il set di funzioni era comunque limitato, e cercare di aggiungere alla pipeline tutte quelle funzionalità che i programmatori necessitavano in ogni particolare caso d'uso era improponibile. Questo spinse verso la nascita di pipeline di rendering programmabili, ovvero dove il programmatore poteva scrivere dei piccoli programmi in un limitato linguaggio assembly, chiamati shader, che sarebbero stati eseguiti all'interno della GPU per ogni vertice o frammento. Presto tali linguaggi crebbero in complessità portando alla definizione di linguaggi di shading di alto livello, nacquero così GLSL [47], HLSL [70] e successivamente anche Cg [74]. Questa flessibilità fece letteralmente esplodere il numero di tecniche ed effetti grafici realizzati dalla community di sviluppatori, portando alla nascita di shader per qualunque esigenza: dal parallax mapping a modelli di luce custom e rifrazioni. Vennero sviluppati anche sistemi di illuminazione custom come il deferred shading ed il light pre-pass, che portarono alla realizzazione di complessi effetti di post-processing come lo screen space ambient occlusion e l'horizon based ambient occlusion. Anche le tipologie di shader realizzabili crebbero presto, furono introdotti shader di tipo geometry che potevano modificare la complessità geometrica di una primitiva, ed altri shader che agivano sul processo di tessellation. E' chiaro quindi come l'introduzione della pipeline programmabile e la nascita degli shader con la possibilità di controllare esattamente il modo di processare vertici, frammenti, texture, ecc, e la possibilità di eseguire questi in parallelo all'interno della GPU, fu letteralmente un punto di svolta nel mondo della computer grafica, dando spazio ad un numero virtualmente infinito di possibilità.

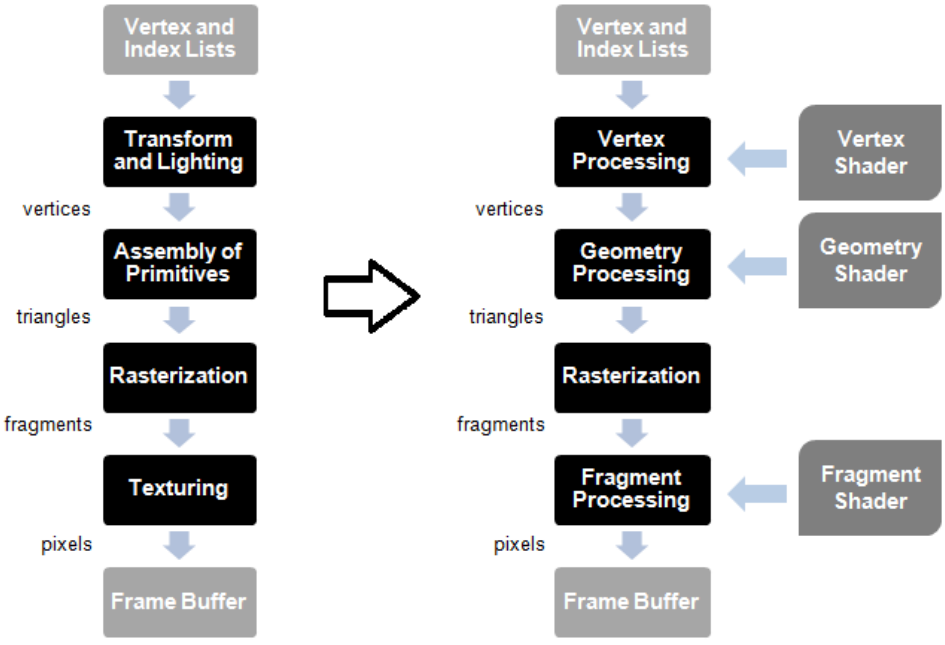

Figura 3.26: Differenza tra una pipeline di rendering fissa (a sinistra) e programmabile (a destra).

3.6.2.2 Graphics APIs

Direct3D

Rilasciata da Microsoft nel lontano 1995, Direct3D [75] fa parte di una più grande famiglia di API chiamata DirectX, queste includono funzionalità per il suono, networking, gestione periferiche di input, grafica sia 2D che 3D ed altro ancora, tuttavia il termine DirectX viene spesso associato alle funzionalità grafiche di tale libreria, soprattutto dai videogiocatori, in quanto molto spesso i giochi su piattaforma Windows includono l'installazione di tali librerie direttamente nel programma di installazione. DirectX nasce per l'industria degli sviluppatori di videogiochi, più nello specifico dall'esigenza di Microsoft di rendere appetibile ai game developer dell'epoca il nuovo sistema operativo Windows 95 in sfavore del DOS, che a quei tempi veniva considerato come una delle migliori piattaforme per realizzare videogames. Direct3D fin dalle sue prime versioni si basava su una pipeline di rendering fissa, la versione 8 vide invece il passaggio alla pipeline programmabile grazie all'introduzione dei vertex e pixel (fragment) shader, la versione 9 vide la nascita del linguaggio di shading HLSL [70] che sostituì il linguaggio assembly utilizzato fino ad allora. Con la versione 10 si ha un completo passaggio alla pipeline programmabile, la retrocompatibilità con le versioni precedenti venne troncata e vennero introdotti i geometry shader. La versione 11 invece vede il supporto alla tessellation via hardware con l'introduzione dei tessellation shaders, olte che il supporto al multithreaded rendering ad ai compute shaders. Infine con la versione 12 si ha un passaggio ad una API di più basso livello, come quanto fatto da Khronos Group [76] con Vulkan [77], in tal modo si ha una libreria più snella e veloce permettendo quindi una velocità di rendering superiore, a patto che lo sviluppatore si faccia carica di tutta la gestione della memoria. Le ultime release della versione 12 di DirectX hanno visto un primo supporto a tecniche di raytracing in real-time, reso possibile grazie al supporto hardware dei più recenti modelli di schede grafiche.

OpenGL

Tale libreria risale agli inizi degli anni 90, quando Silicon Graphics Inc. rilasciò una parte del loro prodotto IRIS GL in formato open, da cui il nome OpenGL, successivamente nell'anno 2007 la gestione passò in mano a Khronos Group [76] che ne detiene tuttora lo sviluppo ed offre sul proprio sito web la documentazione sulla API. Data la sua natura open, OpenGL [44] è stata adottata in svariate piattaforme e software, inoltre, a differenza di Direct3D, supporta la possibilità di includere estensioni vendor-specific all'interno della libreria, includendo in questo modo la possibilità di esporre nuove feature dell'hardware senza la necessità di dover attendere un rilascio ufficiale di una nuova versione di OpenGL che incorpori quelle funzionalità. OpenGL 1 implementava la già discussa pipeline di rendering fissa, mentre dalla versione 2 c'è stato il passaggio verso la pipeline programmabile tramite shaders scritti in GLSL [47] ed il linguaggio assembly ARB. OpenGL 3 segnò il passaggio totale alla pipeline programmabile introducendo i geometry shader e rimuovendo vecchie feature che erano ancora presenti per retrocompatibilità, a tal proposito vennero definiti due profili, il 'Core profile' ed il 'Compatibility profile', quest'ultimo nato allo scopo di mantenere comunque porzioni della vecchia API per compatibilità verso vecchie applicazioni. OpenGL 4 vide invece l'introduzione di shader per il controllo della tessellation, portando quindi il set di feature della libreria ad equiparare le feature offerte da Direct3D 11. Così come DirectX 12, anche le ultime release di OpenGL supportano tecniche di raytracing in real-time.

Vulkan

Annunciata per la prima volta alla Game Developer Conference del 2015 da Khronos Group [76], venne poi pubblicata nel 2016 sempre dallo stesso sviluppatore. Vulkan [77] è nata a seguito dell'esperienza accumulata da Khronos Group nel corso degli anni e si pone come successore di OpenGL, in grado quindi di sopperire a tutti i problemi e le necessità che sono sorte durante lo sviluppo ed il mantenimento di quest'ultima. Tra questi troviamo: il supporto a linguaggi diversi da GLSL [47]; il supporto a diverse piattaforme tramite un'unica API (Vulkan per l'appunto, senza la necessità di avere dei branch di libreria ad-hoc come succede per OpenGL ES e WebGL); l'essere OS-agnostic al fine di migliorare la portabilità delle applicazioni che utilizzano l'API; un supporto migliore ai sistemi moderni basati su multi-threading; l'intenzione di fondere OpenCL a Vulkan in modo da poter gestire insieme sia la parte grafica che computazionale, con un conseguente guadagno in termini di complessità.

Negli anni, le differenze tra il set di feature delle due principali librerie, ovvero OpenGL e Direct3D, sono andate sempre più assottigliandosi. Tuttavia OpenGL è l'unica a poter essere considerata multi piattaforma, infatti sia Windows che Linux che le console Play Station supportano in qualche modo OpenGL, mentre invece Direct3D è esclusiva di Windows ed Xbox. Inoltre, nel corso degli anni sono nate diverse versioni di OpenGL per poter permettere l'esecuzione in ambienti ben specifici, si pensi ad esempio ad OpenGL ES per i dispositivi mobile o WebGL per ambienti Web. Tale notorietà insieme al fatto di essere open, ha reso ancor di più OpenGL quasi il punto di riferimento per tutti i programmatori che vogliono approcciarsi al mondo della computer grafica o appartengono già al settore.

3.6.2.3 OpenGL

OpenGL funziona come una grande macchina a stati, una volta inizializzata essa si troverà in uno stato ben preciso, ovviamente modificabile dall'utente, tuttavia va tenuto a mente che tutte le operazioni eseguite tramite la libreria terranno conto dello stato durante il quale una funzione viene chiamata. Qualsiasi chiamata a funzione di libreria è asincrona, i comandi inviati verranno semplicemente salvati all'interno di una coda FIFO, pertanto l'ordine con cui l'utente invierà i comandi di draw call e state change sarà preservato, il driver della scheda grafica si occuperà poi di scodare tali comandi uno alla volta e mandarli in esecuzione sulla GPU, nel caso dei comandi di drawing, questi daranno vita all'immissione di dati nella pipeline di rendering ed avranno come risultato la scrittura di uno o più pixel all'interno del frame buffer.

La pipeline di rendering di OpenGL è programmabile e si basa su rasterizzazione [78], pertanto racchiude tutte le fasi illustrate nei precedenti paragrafi. In figura 3.27 è possibile vedere approssimativamente le varie fasi di cui si compone.

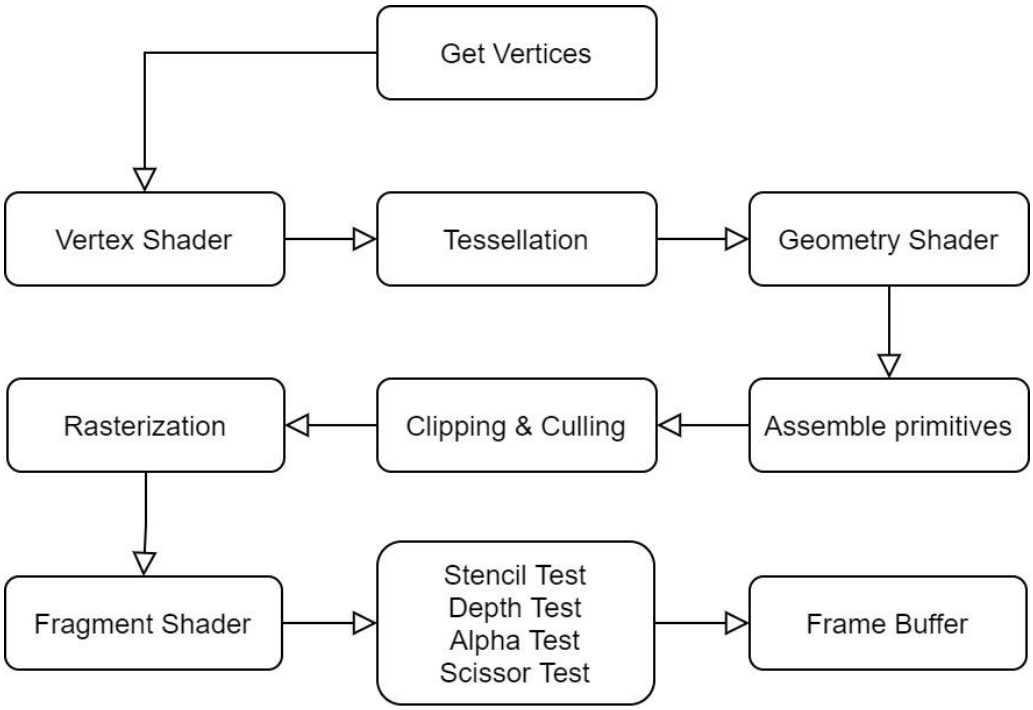

Figura 3.27: Flow approssimativo delle varie fasi della pipeline di rendering di OpenGL.

I vertici per renderizzare una primitiva vengono quindi letti dai buffer interni alla GPU ed inviati al vertex shader dello shader program correntemente attivo per l'elaborazione (si ricorda che uno shader program è composto da più tipi di shader). I vertici così elaborati passano attraverso una fase preliminare di primitive assembly, chiamata 'Early Primitive Assembly' allo scopo di essere trasformati in primitive base per le successive fasi di tessellation e geometry shader dalle quali potrebbero uscire più o meno primitive di quelle entrate. A questo punto le primitive complete possono essere assemblate a partire da quelle base nella fase di 'Full Primitive Assembly', per poi essere soggette a culling se ricadono fuori dal view frustum, ed a clipping se intersecano il view frustum. Dopo l'applicazione dell'operazione di perspective divide, la fase di rasterizzazione ha inizio seguita da quella di interpolazione degli attributi dei fragment, questi saranno quindi pronti per essere così inviati al fragment shader per il calcolo del colore finale del pixel che ricoprono. Una volta che l'elaborazione del fragment shader sarà conclusa avviene l'elaborazione post-shader chiamata 'Per-Sample Processing', che sottopone il fragment a diversi test prima di essere effettivamente scritto sul frame buffer. Tali test possono consistere nel verificare che il fragment rientri all'interno di una area scrivibile (scissor test), o che questo si trovi più vicino rispetto ad un altro fragment nella stessa posizione e già esaminato (depth test), o che questo possieda un valore di stencil che non corrisponde con il valore di stencil presente alla stessa posizione nello stencil buffer (stencil test). Nel caso in cui il relativo oggetto non sia perfettamente opaco, prima di essere scritto sul frame buffer un fragment potrebbe essere sottoposto ad un'altra operazione, ovvero quella di alpha blending, che consiste nel mixare il colore con quello di altri fragment non opachi che ricadono su quel pixel.

OpenGL Coordinate System

Per sistema di coordinate di una libreria grafica si intende il sistema di coordinate del relativo spazio NDC. In OpenGL, lo spazio NDC è basato su un sistema di coordinate sinistrorso (left-handed) [68]. Inoltre, la funzione di depth-testing di default di OpenGL è 'GL_LESS', ciò

vuol dire che un fragment verrà scritto nel frame buffer solamente se la sua coordinata Z è inferiore rispetto a quella già presente nel depth-buffer. In tal modo oggetti che in NDC hanno una coordinata Z inferiore verranno renderizzati sopra ad altri, e di conseguenza saranno quelli concettualmente più vicini all'osservatore, in sostanza più un vertice sarà lontano dall'osservatore maggiore sarà la sua coordinata Z in NDC e viceversa. OpenGL tuttavia utilizza come convenzione sulle superfici front-facing un winding order antiorario, che corrisponderebbe ad un sistema destrorso, questo perché OpenGL assume in realtà che tutto ciò che avviene fuori dalla pipeline di rendering si basi su un sistema di riferimento destrorso dove la direzione forward dell'osservatore coincide con l'asse Z negativo. La motivazione di tale scelta risiede nel fatto che il sistema di coordinate destrorso è praticamente uno standard in svariate applicazioni così come in geometria ed altre discipline scientifiche.

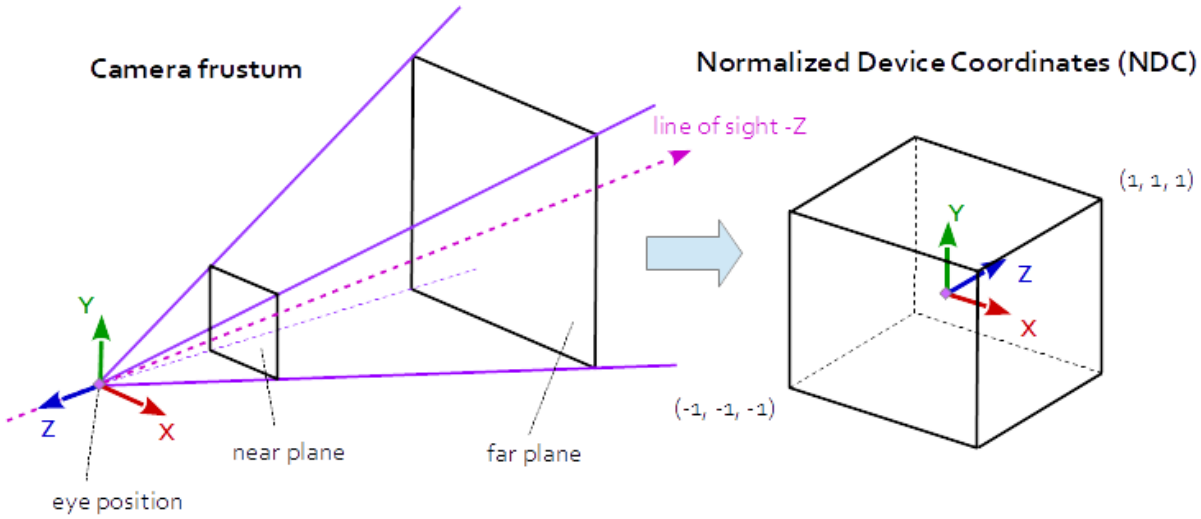

Figura 3.28: OpenGL assume un sistema di riferimento destrorso (a sinistra) al di fuori della pipeline di rendering. Nello spazio NDC OpenGL si comporta invece in modo sinistrorso (a destra).

In un sistema destrorso, più gli oggetti si allontanano dall'osservatore più la loro coordinata Z diminuirà, tuttavia essendo lo spazio NDC di OpenGL sinistrorso, esso si comporterà in modo inverso e l'utente vedrà oggetti più distanti coprire oggetti che in realtà dovrebbero essere più vicini. Pertanto in una impostazione del genere, prima di arrivare nello spazio NDC, bisognerà invertire la coordinata Z di tutti i vertici in modo da avere il comportamento corretto. Tale trasformazione solitamente avviene in modo trasparente all'utente, motivo per il quale si pensa spesso che OpenGL si basi su un sistema di coordinate destrorso, tale trasformazione avviene per l'esattezza all'interno della projection matrix e consiste in uno scaling negativo (-1) lungo l'asse Z, in modo da invertire la depth dei vertici, pertanto lo step esatto nel quale avviene tale trasformazione è il vertex shader, in quanto al suo interno i vertici vengono trasformati tramite la matrice model-view-projection.

3.6.3 Engine Rendering

In questa sezione si descriverà l'implementazione del processo di rendering all'interno dell'engine realizzato, così come i vari componenti e le classi a supporto di questo.

3.6.3.1 Coordinate System & Handedness

Il sistema di coordinate utilizzato dall'engine è sinistrorso, con gli assi globali di riferimento right, up e forward corrispondenti rispettivamente a +X, +Y e +Z. La scelta di un sistema di riferimento solitamente è solo una convenzione, pertanto si è deciso di applicare la convenzione che sembrava umanamente più intuitiva per una persona ambidestra, ovvero spostandosi a destra si ha un incremento della la coordinata x, spostandosi in alto si ha un incremento della coordinata y, ed infine spostandosi in avanti si ha un incremento della coordinata z, in tal modo oggetti più distanti dall'osservatore avranno una coordinata z maggiore. La projection matrix viene quindi costruita in modo tale da non eseguire l'inversione della coordinata Z dei vertici.

La funzione di depth-test è stata lasciata sul default, ovvero 'GL_LESS', in quanto coerente con il fatto che oggetti più vicini all'osservatore avranno una coordinata Z inferiore. Anche il winding-order è stato lasciato sul valore di default, ovvero in senso antiorario.

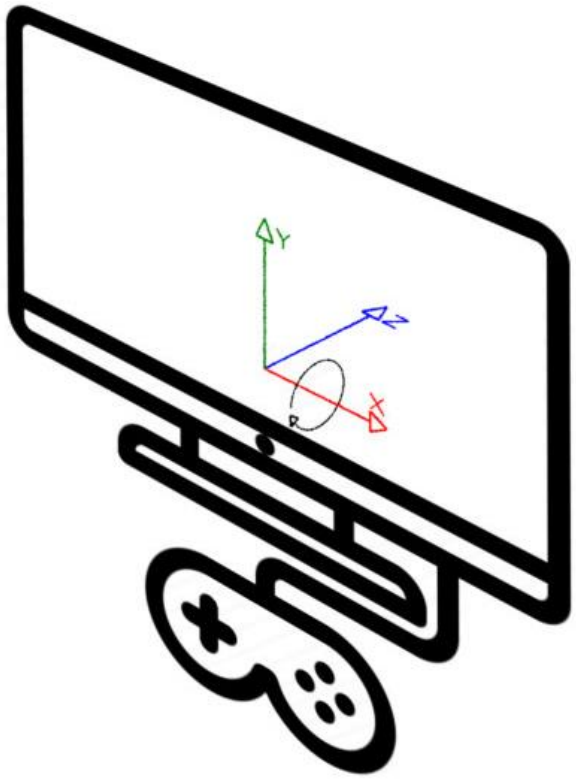

Figura 3.29: Sistema di coordinate utilizzato nell'engine.

In figura 3.29 è possibile vedere il sistema di riferimento utilizzato dell'engine allineato al punto di vista dell'osservatore.

3.6.3.2 Camera Component

Si tratta di uno dei componenti più importanti in quanto rappresenta il punto di vista dell'osservatore, ovvero la sua posizione ed orientamento all'interno della scena insieme al view frustum ed i suoi piani di clipping. Pertanto il camera component conterrà sia la view matrix che la projection matrix, che nel nostro caso consisterà in una proiezione ortografica, mentre la view matrix corrisponderà alla matrice di trasformazione del Transform component della entity al quale il camera component verrà attaccato.

Come detto nel precedente paragrafo, la matrice di proiezione viene creata in modo da non invertire la coordinata Z dei vertici, tuttavia la maggior parte delle librerie matematiche a supporto di OpenGL restituiscono automaticamente una matrice di proiezione che inverte la coordinata Z, pertanto all'interno del costruttore della classe Camera è stata specificata l'intenzione di costruire una matrice di proiezione da utilizzare in un contesto sinistrorso:

```
public Camera(float left, float right, float bottom, float top, float
zNear, float zFar) {
    this.projectionMatrix = new Matrix4f()
                .orthoLH(left,right,bottom,top,zNear,zFar);

    ...
}
```

Due utilissime funzioni del camera component sono la *screenToWorldCoords()* e la *worldToScreenCoords()* che permettono rispettivamente di trovare le coordinate del mondo corrispondenti ad un punto sullo schermo, e le coordinate dello schermo a partire da un punto all'interno del mondo. Tali funzioni si basano sulle diverse trasformazioni applicate ad un vertice durante la pipeline di rendering viste in un precedente paragrafo e sono molto utilizzate per eseguire raycast all'interno della scena a partire dai click dell'utente sullo schermo.

3.6.3.3 Window Class

Questa classe non è direttamente esposta all'utente ma viene utilizzata internamente dall'engine per creare e rappresentare la finestra di esecuzione dell'applicazione. Essa si occupa inoltre di inizializzare correttamente la libreria OpenGL e di fornire dei metodi per effettuare una chiusura pulita delle risorse allocate.

3.6.3.4 Mesh Class

La classe Mesh implementa le funzionalità per la creazione di un asset di tipo mesh all'interno dell'engine, essa conterrà i dati di tutti i vertici e comunicherà con la libreria grafica allo scopo di caricare nella GPU tutte le informazioni relative alla mesh nel formato corretto. Essa implementa inoltre la corretta metodologia di rendering della mesh in base alla tipologia di primitive di cui si compone. Le informazioni di una mesh all'interno della scheda grafica sono mantenute tipicamente all'interno di un oggetto chiamato 'Vertex Array Object' [68], esso contiene una lista di buffer contenenti gli attributi dei vertici, chiamati 'Vertex Buffer Object', questi ultimi contengono gli attributi veri e propri dei vertici di una mesh, quali posizione, colore, coordinate uv, ecc. La scheda grafica tende ad impacchettare gli attributi dei vertici di una mesh in modo contiguo in memoria, in modo tale da rendere l'esecuzione dell'applicazione molto cache-friendly.

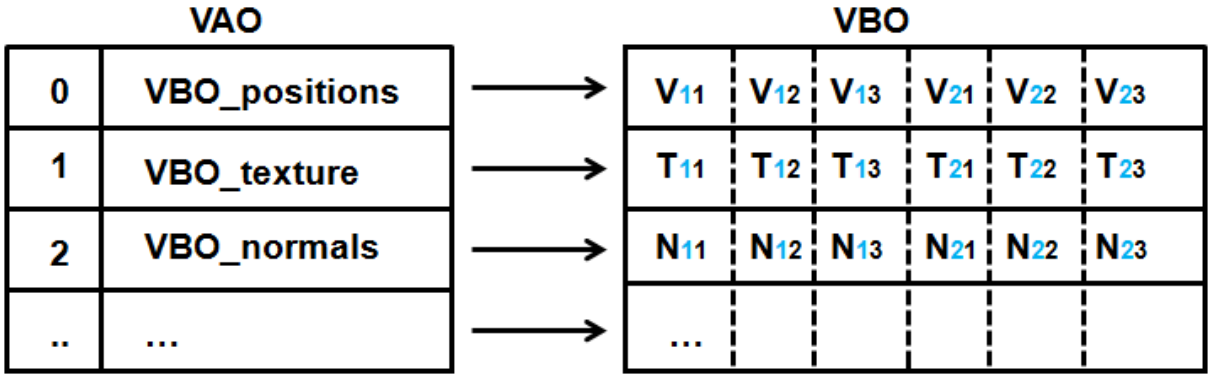

Figura 3.30: Vertex Array Objects e Vertex Buffer Objects.

Per mesh dove i singoli vertici vengono condivisi da diversi triangoli è possibile utilizzare il cosiddetto 'Indexing' al fine di evitare vertici duplicati all'interno dei buffer con conseguente spreco di memoria. L'indexing consiste nell'aggiungere alla mesh un altro buffer di dati contenente numeri interi, dove ogni numero identifica il relativo vertice all'interno di un altro buffer che conterrà invece le informazioni di ogni singolo vertice. La classe mesh come anticipato implementa anche il rendering della mesh stessa, ovvero esegue la corretta draw

call in base alle informazioni di rendering della mesh, come ad esempio la tipologia di primitive su cui essa si basa (es. triangles o triangle_strips) e l'abilitazione o meno dell'indexing. L'engine è provvisto di una mesh built-in molto utilizzata in ambito 2D, ovvero il Quad, di cui il seguente codice ne rappresenta la creazione all'interno dell'engine:

```
Quad = new Mesh(
        "Quad", //meshName
        Mesh.DrawMode.TRIANGLES, //drawMode
        new Vector3f(1,1,1), //importScale
        new HashMap<Mesh.VertexAttribute,Number[]>() {{ //vertexData
            put(Mesh.VertexAttribute.COORDS, new Float[]{
                    -0.5f,-0.5f,0f,
                    -0.5f,0.5f,0f,
                    0.5f,0.5f,0f,
                    0.5f,-0.5f,0f
            });

            put(Mesh.VertexAttribute.UV_COORDS, new Float[]{
                    0f,1f,
                    0f,0f,
                    1f,0f,
                    1f,1f
            });
        }},
        new HashMap<Mesh.MeshAttribute,Number[]>() {{ //meshData
            put(Mesh.MeshAttribute.VERTEX_INDICES, new Integer[]{0,2,1,0,3,2});
        }}
);
```

3.6.3.5 Shader Class

La classe shader rappresenta un altro degli asset creabili all'interno dell'engine ed implementa le funzionalità per la creazione ed il controllo di uno shader program all'interno della scheda grafica. Tale classe si occuperà pertanto di eseguire la lettura del codice GLSL su disco, e comunicare poi con la libreria grafica per la compilazione dei vari shader (vertex, fragment) producendo così i cosiddetti 'Shader Object', questi verranno poi attaccati ad uno 'Shader Program' che rappresenta i vari step di shading di un vertice all'interno della pipeline di rendering, quest'ultimo sarà soggetto infine all'operazione di 'Linking' che avrà come risultato uno 'shader program executable', che rappresenterà l'effettivo programma di shading. Tale programma sarà identificato all'interno di OpenGL da un numero intero, e tramite libreria potrà essere impostato come programma di shading attivo per le future chiamate di drawing eseguite sulla libreria. Infatti, nella macchina a stati di OpenGL ci sarà sempre al più un solo shader attivo al quale le operazioni di drawing faranno riferimento, quando si vorrà usare un nuovo shader program allora il precedente verrà disattivato. Questo comportamento rassomiglia il comportamento dei software di disegno: quando disegniamo staremmo utilizzando al più un solo pennello (shader program), quando cambiamo pennello per ottenere uno stile diverso rimpiazzeremo il precedente.

I vari step per la creazione di uno shader program executable, così come le relative chiamate OpenGL, sono visibili in figura 3.31.

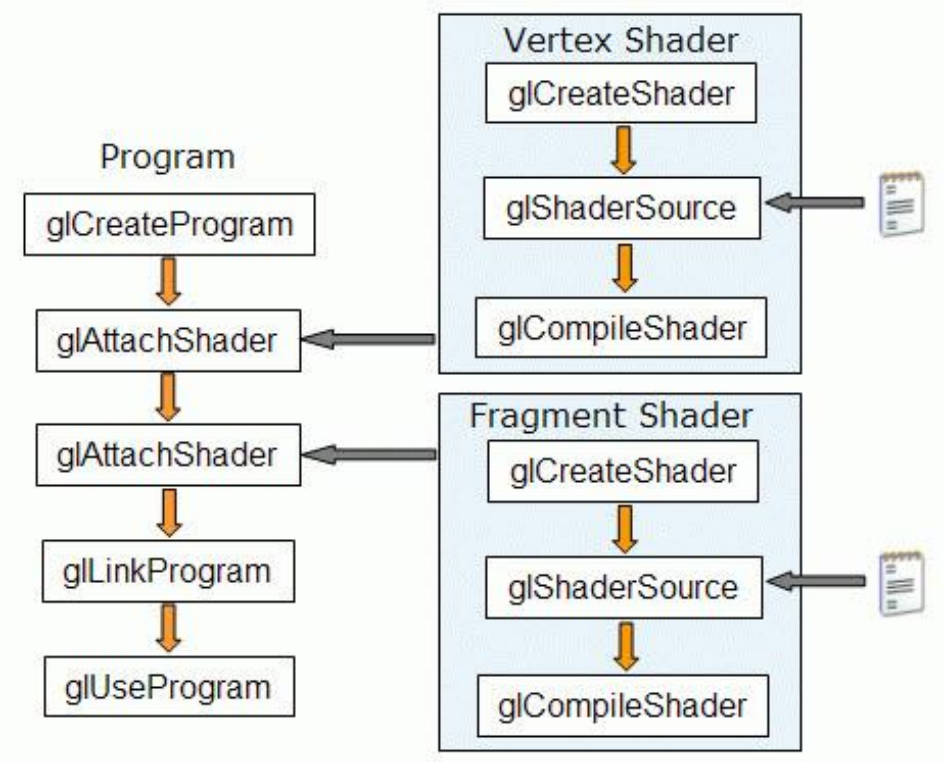

Figura 3.31: Step per la creazione di uno shader in OpenGL.

Dopo l'operazione di creazione dello shader program executable, viene eseguita su di esso una sorta di reflection al fine di ottenere i parametri configurabili esposti, in OpenGL chiamati 'Uniform', questi potrebbero essere texture, matrici, vettori e costanti. A tal proposito, la classe Shader espone una serie di funzioni per la relativa scrittura e lettura degli Uniform esposti. Rifacendoci all'analogia con i software di disegno, anche in questi i diversi pennelli espongono dei parametri per determinare la configurazione corrente di un particolare pennello, parametri che potremmo paragonare ai nostri Uniform. Di seguito vediamo il codice per la creazione di uno shader all'interno dell'engine, esso prende in input il nome che identificherà lo shader all'interno dell'engine, il path al file .glsl e gli intervalli in linee di tale file all'interno dei quali risiedono gli shader creati:

```
new Shader(
        "SimpleShader", //name
        shadersDirectory + "SimpleShader.glsl", //file path
        new HashMap<Shader.ShaderType, Vector2ic>(){{ //vertex and fragment shaders
            put(Shader.ShaderType.VERTEX, new Vector2i(1,18));
            put(Shader.ShaderType.FRAGMENT, new Vector2i(22,36));
        }}
);
```

Il codice a seguire mostra invece la definizione di un semplice shader in GLSL:

```
//VERTEX_PROGRAM

#version 330

layout (location = 0) in vec3 position;
```

```
layout (location = 1) in vec2 uv_coords;

out vec2 uvCoords;

uniform mat4 worldMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;

void main() {
    uvCoords = uv_coords;
    gl_Position = projectionMatrix * inverse(viewMatrix) * worldMatrix *
vec4(position, 1.0);
}

//VERTEX_PROGRAM_END

//FRAGMENT_PROGRAM

#version 330

#extension GL_ARB_shading_language_420pack : require

in vec2 uvCoords;

out vec4 fragColor;

layout(binding = 0) uniform sampler2D textureSampler0;

void main() {
    fragColor = texture(textureSampler0, uvCoords);
}

//FRAGMENT_PROGRAM_END
```

La parte iniziale del vertex program specifica che esso avrà accesso alle informazioni dei vertici contenute nei VBO con id 0 ed 1 della mesh, e tali dati verranno parsati all'interno di due variabili, rispettivamente position ed uv_coords. E' importante che il tipo di tali variabili corrisponda al tipo dei dati contenuti nei relativi VBO.

Il main() è la parte dove risiederà il codice dello shader, in questo caso il vertex program sta trasformando le coordinate del vertice per mezzo delle matrici mvp definite come uniform configurabili dall'utente, e sta inoltre mandando in output le coordinate UV del vertice in esame, tali coordinate saranno accessibili dai prossimi shader della pipeline di rendering, in questo caso dal fragment shader.

Definendo una variabile di ingresso con lo stesso nome della variabile di output del vertex shader, ovvero uvCoords, il fragment shader prenderà quindi in ingresso le coordinate UV del frammento (queste non saranno quelle inviate in output dal vertex shader, ma saranno quelle interpolate dallo step di Interpolation descritto nel relativo paragrafo) e manderà in uscita il colore del frammento tramite la variabile fragColor.

Nel fragment shader notiamo anche la presenza di uno Uniform di tipo 'sampler2D', questo rappresenta una texture, o meglio un oggetto che permette di campionare dei punti su una texture, e viene utilizzato dal fragment program per campionare il valore della texture attualmente attiva sul texture slot 0 (binding = 0) alle coordinate uvCoords. L'esatto id di binding di un sampler2D, così come i corretti id dei VBO da cui il vertex shader attinge le informazioni sui vertici, derivano dalla convenzione utilizzata nell'engine.

3.6.3.6 Material Class

Come visto nel precedente paragrafo, uno shader può essere parametrizzabile tramite l'esposizione di uniform. Pertanto è naturale pensare che lo stile grafico implementato da uno shader possa essere condiviso da più oggetti della scena, al netto di alcuni parametri di ingresso come potrebbero essere il colore di base, o la percentuale di luce riflessa. Per concretizzare questa idea nasce il concetto di 'Material', ovvero un altro degli asset creabili all'interno dell'engine che rappresenta il set di valori degli uniform da attribuire ad un dato shader, ragion per cui un material viene spesso visto come "istanza" di uno shader in quanto ne valorizza gli uniform. Più material possono condividere quindi lo stesso shader, ma avere dei diversi valori per gli uniform. A livello di libreria grafica, questo si traduce nel poter riutilizzare uno stesso shader program per diverse mesh cambiandone solamente i parametri, questo fa ovviamente risparmiare tempo rispetto a cambiare completamente lo shader program con il quale verranno eseguite le draw call [79]. Di seguito un esempio di creazione di un material all'interno dell'engine:

```
new Material(
    "BoardMaterial",
    gameEngine.assetManager.getShader("TileShader"),
    new
Pair<>("diffuse",gameEngine.assetManager.getTexture("BoardTexture")),
    new Pair<>(UniformElement.worldMatrix.getName(),
UniformElement.worldMatrix.getValuem4f()),
    new Pair<>(UniformElement.viewMatrix.getName(),
UniformElement.viewMatrix.getValuem4f()),
    new Pair<>(UniformElement.projectionMatrix.getName(),
UniformElement.projectionMatrix.getValuem4f()),
    new Pair<>("spritesheetRows",1),
    new Pair<>("spritesheetCols",1),
    new Pair<>("initialTileNumber",0),
    new Pair<>("finalTileNumber",0),
    new Pair<>("animationProgress",0f),
    new Pair<>("xAlphaSlide",0f),
    new Pair<>("yAlphaSlide",0f),
    new Pair<>("xMirror",0),
    new Pair<>("forcedRow",-1f),
    new Pair<>("postMultipliedAlpha",1f)
)
```

Come si evince dal codice, il material sarà identificato da un nome e conterrà il riferimento allo shader da utilizzare insieme ai valori degli uniform da settare su quest'ultimo. La classe material espone inoltre degli utili metodi per poter ricavare ed assegnare il valore di un particolare uniform a runtime.

3.6.3.7 Texture Class

La classe Texture serve per permettere il caricamento per l'appunto di texture all'interno della scheda grafica e definirne delle proprietà, come ad esempio il tipo di algoritmo da utilizzare quando un texel viene mappato su più pixel, anche chiamato "Magnification Filtering", il tipo di algoritmo da utilizzare quando più texel vengono mappati su di un solo pixel, chiamato "Minification Filtering", e l'abilitazione all'uso di mipmaps nell'algoritmo di minification. Il nome 'mipmaps' fa riferimento all'omonima tecnica che prevede l'utilizzo di sequenze di texture pre calcolate a risoluzione dimezzata al fine di evitare per quanto possibile fenomeni

di aliasing che si hanno quando una texture viene proiettata a schermo in modo rimpicciolito rispetto alla sua risoluzione originale usando degli algoritmi di minification real-time.

Figura 3.32: Esempio di mipmaps di una texture.

Le mipmaps vengono suddivise in livelli come mostrato in figura 3.32, uno per ogni volta che si dimezza l'immagine di partenza fino ad arrivare alla mipmap di dimensione 1x1, dove il livello 0 rappresenta la texture originale, e vengono calcolate usando degli algoritmi più avanzati rispetto a quelli utilizzati a runtime in quanto offline non si hanno vincoli temporali di alcun tipo ed è quindi possibile ottenere una qualità migliore. La libreria grafica sceglie quale livello di mipmap utilizzare in base al rateo di variazione tra le coordinate uv dei fragment e la loro posizione a schermo. Le mipmaps sono uno dei pochi esempi in ambito rendering dove una tecnica porta miglioramenti sia a livello visuale che di performance dell'applicazione [80]. Di seguito è mostrato come creare una texture all'interno dell'engine:

```
new Texture(
        "EnemiesTilemap", //texture name
        texturesDirectory + "SuperMarioBrosEnemiesTiles.png" //texture path
);
```

3.6.3.8 Mesh Renderer Component

Il Mesh Renderer è il component tramite il quale ad una entity viene attaccata una forma poligonale, rappresentata da una mesh, le cui proprietà grafiche saranno specificate dal material assegnato component stesso. Il MeshRenderer avrà quindi il compito di renderizzare la mesh tramite le proprietà grafiche del material. L'origine del sistema di riferimento della mesh corrisponderà alla posizione in word space della entity. Il mesh renderer rappresenta l'unità logica su cui lavora il Rendering Engine e può definire altre proprietà utili a quest'ultimo per comporre correttamente il rendering finale, inoltre il mesh renderer si presta facilmente ad estensioni ad-hoc al fine di implementare il rendering di mesh particolari come una tilemap o un sistema particellare.

3.6.3.9 The Rendering Engine

L'engine di rendering entra in gioco nella fase finale del game loop, una volta che fisica, animazioni e logica dell'utente sono state applicate, e costituisce quel modulo che gestisce e tiene traccia dei vari mesh renderer attivi in scena orchestrandoli in modo da renderizzare un frame coerente, cercando al tempo stesso di minimizzare il costo computazionale del processo di rendering. Il processo di rendering implementato dall'engine è abbastanza semplice, in figura 3.33 è possibile vedere il flow delle operazioni eseguite:

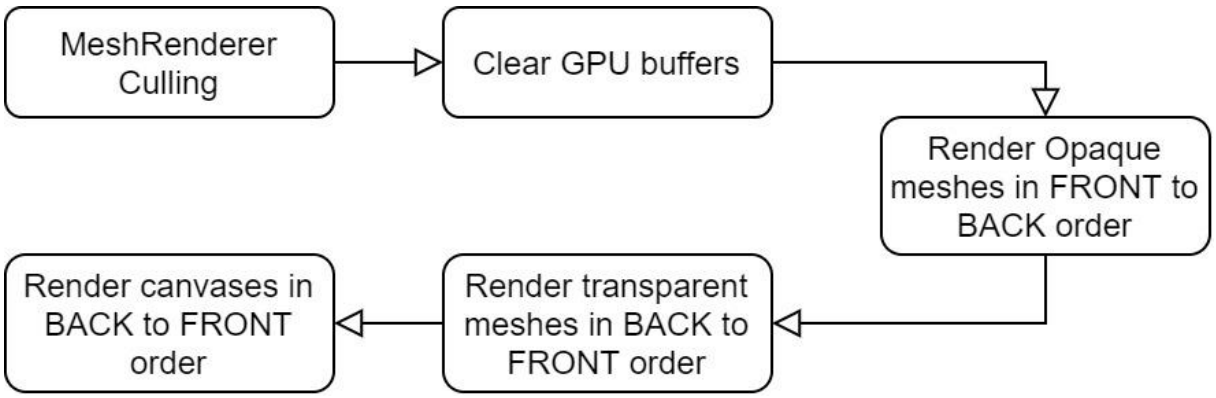

Figura 3.33: Flow delle operazioni eseguite durante il processo di rendering dell'engine realizzato.

Come prima operazione vengono scartati tutti i mesh renderer che non ricadono all'interno del view frustum ed i buffer della GPU vengono ripuliti. A questo punto si esegue il render dei mesh renderer opachi a partire da quello più vicino alla camera, in tal modo le mesh più lontane verranno scartate dal depth test facendo risparmiare cicli di fragment shader. Fatto ciò, è il turno dei mesh renderer semitrasparenti che verranno renderizzati a partire dal più lontano, in modo da permettere all'alpha blending di funzionare correttamente. Infine, vengono renderizzati i vari canvas a partire dal più lontano e con depth test disabilitato, così da assecondare la logica di rendering dei canvas.

3.6.3.10 Sprite Renderer Component

E' una specializzazione del mesh renderer component dove come mesh viene implicitamente usata un Quad, al fine di permettere il render delle cosiddette "sprite". Tale component gode anche di una proprietà aggiuntiva chiamata 'z-sorting' che permette di dare un ordinamento di rendering a diversi sprite renderer che si trovano eventualmente alla stessa distanza dalla camera.

3.6.3.11 TileMap Component

Il tilemap component viene utilizzato per creare e renderizzare una griglia di tile, questa può essere vista come un'unica grande sprite, per questo motivo tale component estende lo sprite renderer visto in precedenza in modo da ereditarne anche la proprietà di z-sorting, molto utile quando si ha la necessità di creare più tilemap sovrapposte tra le quali si vuole definire un ordine di rendering ben preciso. In figura 3.34 si può vedere un esempio di tilemap component in azione.

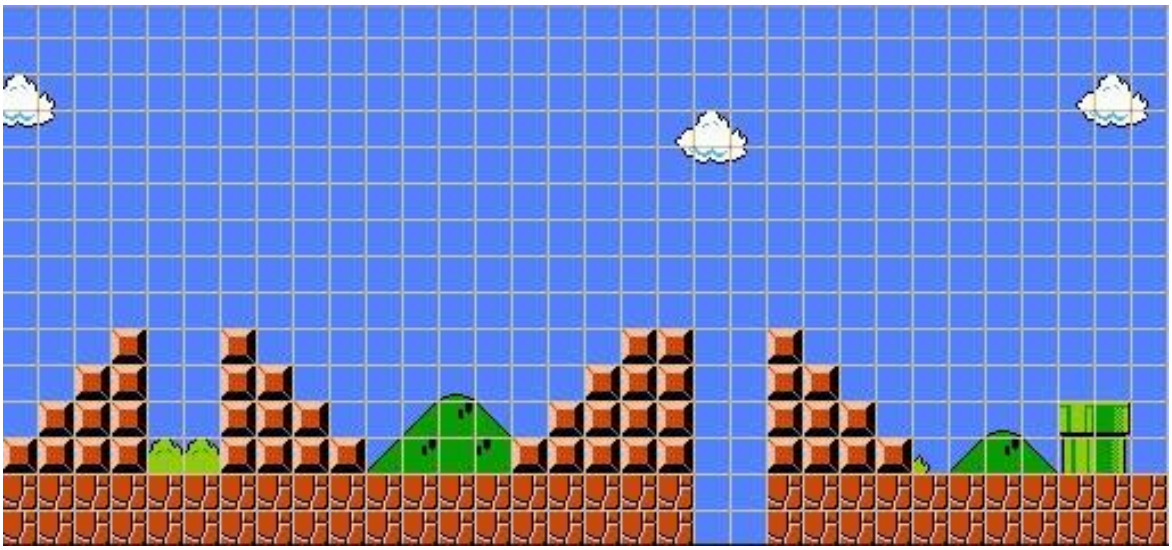

Figura 3.34: Esempio di tile map generata tramite il TileMap component.

Per la sua creazione, il TileMap component necessita di:

- un material che verrà utilizzato come material delle singole tile.
- una griglia di id, dove ogni id rappresenta l'id della tile che verrà creata dal component. Tale griglia sarà fornita dall'utente tramite un file di testo .csv.
- un oggetto di tipo TileBuilder dove l'utente avrà definito programmaticamente come dovranno essere create le varie tile in base al loro id.

Per ogni singola tile della griglia verrà creato un GameObject alla posizione corrispondente, dove la tile in basso a sinistra nella griglia verrà posizionata nell'esatta posizione della entity, inoltre a tale gameobject verrà automaticamente attaccato un mesh render component dummy, in modo che l'utente possa comunque modificare i parametri del material della tile. Una volta che un gameobject corrispondente ad una tile è stato creato, esso viene passato all'oggetto TileBuilder fornito dall'utente, insieme ad altri dati come l'id della tile cui corrisponde il gameobject e la sua riga e colonna all'interno della griglia, in tal modo l'utente ha la possibilità di arricchire il gameobject con altri component ed eventualmente della logica di scripting. Le tile una volta create vengono inserite all'interno di una struttura di partizionamento spaziale, in questo caso una griglia, in modo da poter agevolmente recuperare le sole tile che ricadono all'interno del view frustum durante il processo di rendering. Il tilemap component si occuperà poi di renderizzare le varie tile quando richiesto dal rendering engine, durante tale processo si utilizzerà un solo shader, in quanto i material delle tile sono clonati da uno stesso material che punterà ad un solo shader, ciò comporta un grande vantaggio in termini di prestazioni.

3.6.3.12 ParticleSystem Component

Il particle system component abilita una entity a fungere come sorgente di particelle, tali particelle saranno delle sprite dove la texture viene ritagliata da uno spritesheet in base al tempo di vita della particella, inoltre le particelle potranno godere di fisica a tempo continuo e pertanto interagire con l'ambiente. Tale sistema particellare è quindi spritesheet based e come particelle utilizza delle mesh di tipo quad, ciò lo rende molto leggero in termini di rendering.

Il particle system component implementa l'interfaccia dei mesh renderer pur non estendendo direttamente il mesh renderer base, ciò lo rende quindi riconoscibile dal rendering engine, inoltre implementa delle funzionalità di scripting, in quanto avrà bisogno di generare nuove particelle col passare del tempo.

Come material per le particelle ne viene creato uno a runtime basato su uno shader built-in creato ad-hoc per il rendering di particelle, esso esporrà infatti degli uniform per controllare il corretto uv mapping della particella sullo sprite-sheet fornito in base al tempo di vita delle particelle stesse.

Tra i principali parametri necessari a creare un particle system component troviamo la texture che fa da spritesheet insieme al numero di righe e colonne di cui si compone, un flag che indica se le particelle godranno di fisica o meno insieme al layer di collisione da utilizzare, il numero di particelle da emettere per secondo insieme al loro tempo di vita, diversi parametri geometrici come direzione, velocità, rotazione e scala iniziale delle particelle, il colore da mixare alla texture durante la vita della particella, la scala della billboard della particella durante il suo ciclo di vita, e delle callback che l'utente può impostare per controllare la nascita di una particella ed il suo eventuale comportamento fisico, quest'ultimo è impostato di default sulla distruzione di una particella al contatto con un altro oggetto.

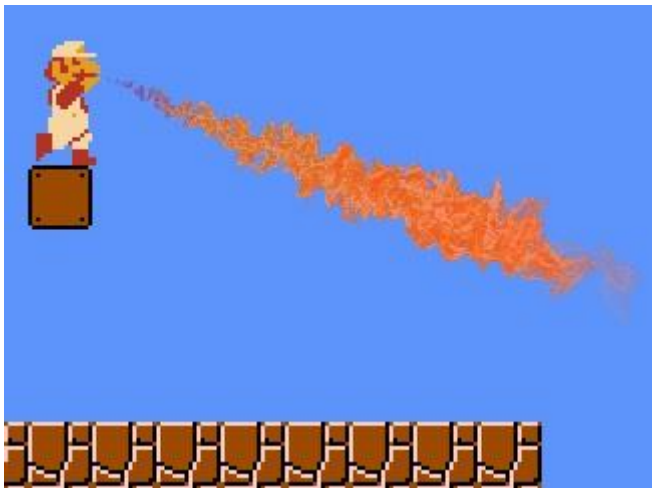

Figura 3.35: Esempio di un effetto lanciafiamme realizzato con il ParticleSystem component.

3.6.3.13 Canvas Component

Anche il canvas component implementa l'interfaccia dei mesh renderer e sarà pertanto riconosciuto dal rendering engine. Esso serve principalmente per renderizzare elementi di UI, quali forme e caselle testuali. Tali elementi di UI consisteranno in dei GameObject ai quali è stato attaccato un component di tipo CanvasElement, pertanto avranno una posizione all'interno dello spazio data dal loro transform component, e conterranno un elemento di UI definito dal canvas element component, inoltre essi dovranno essere imparentati al gameobject contenente il canvas component, al fine di essere riconosciuti e renderizzati da quest'ultimo.

L'ordine di parenting dei child di un canvas è determinante al momento del rendering, in quanto questo verrà eseguito visitando il grafo di parentela in modo depth-first pre-order.

Le coordinate spaziali di un canvas element component verranno direttamente interpretate in Canvas space (window space), pertanto è normale che queste siano spesso nell'ordine delle centinaia.

Peculiarità degli elementi di un canvas deve essere quella di adattarsi ad ogni tipo di risoluzione, sia per quanto riguarda le forme, sia per il testo, per tali motivi il canvas component realizzato, così come i vari canvas element offerti, si basano sulla libreria NanoVG [81], ovvero una piccola e leggera libreria per OpenGL che permette il rendering di grafica vettoriale con antialiasing.

Canvas Element Components

Tra gli elementi ui realizzati nell'engine troviamo box di varia forma con sfondo unica tinta e caselle di testo:

- RectBox: si tratta di una forma rettangolare che è possibile riempire con un colore pieno. Viene tipicamente utilizzata per fare da sfondo per eventuali caselle di testo.
- RoundedRectBox: come la RectBox, ma è possibile specificare una curvatura per gli angoli.
- ImageBox: sostanzialmente una RectBox con un'immagine di sfondo anziché un colore pieno. E' possibile mantenere l'aspect ratio dell'immagine originale o lasciare che questa di stiri lungo le dimensioni specificate.
- CircleBox: una forma circolare che è possibile riempire con un colore pieno. Viene solitamente utilizzata in combinazione con altre CircleBox per realizzare dei pulsanti circolari.
- EllipseBox: una forma ellittica che è possibile riempire con un colore pieno.
- TextBox: una casella di testo rettangolare per la quale è possibile specificare un font, così come il colore del testo ed altre sue proprietà come l'allineamento orizzontale e verticale all'interno della box.

Canvas Ray Casting

Il canvas raycasting serve per determinare quale canvas element si trova sotto il puntatore del mouse, operazione molto utile quando l'utente vuole interagire con degli elementi di UI. Un modo per effettuare tale operazione potrebbe essere quello di usare il ray casting offerto dal physics engine, generando quindi un raggio a partire dalla posizione del click sullo schermo, effettuando poi le dovute trasformazioni e vedere chi tra i canvas element interseca per primo il raggio generato. Questo approccio presenta una serie di problemi: innanzi tutto i canvas element dovrebbero essere dotati di un rigid body, in modo da essere presenti all'interno del physics engine ed essere presi in esame nel test di collisione, solo questo punto costituisce già un controsenso in quanto non avrebbe senso equipaggiare i canvas element di un rigid body; inoltre il fatto che i canvas element si trovano spesso uno sopra l'altro, ovvero lungo la stessa coordinata Z, renderebbe impossibile determinare quale elemento è stato intersecato per primo dal raggio. Per queste ragioni, si è deciso di adottare un approccio che non prevede nessun utilizzo della fisica, anzi, si basa esclusivamente sul processo di rendering, tale approccio prende il nome di 'Frame Buffer Object Picking', e come suggerisce il nome prevede l'utilizzo del frame buffer per determinare quale oggetto risiede sotto il puntatore.

Nel frame buffer object picking si esegue un render di tutti gli oggetti che si desidera testare, ogni oggetto verrà renderizzato con un colore ben preciso, una volta finito il rendering verrà estratto il colore del pixel corrispondente alla posizione del puntatore e di conseguenza verrà identificato l'oggetto appartenente a quel colore. Tale tecnica può essere usata per effettuare il picking di qualunque oggetto di scena, non solo di oggetti presenti nel canvas, e permette il picking in tempo reale di oggetti di forma arbitraria come si evince in figura 3.37. L'unica limitazione del frame buffer object picking consiste nel massimo numero di colori rappresentabili da un pixel del frame buffer che rappresenterà quindi il massimo numero di oggetti testabili.

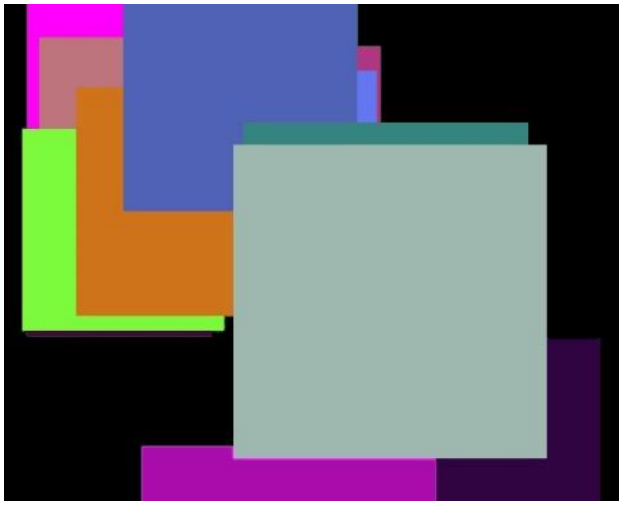

Figura 3.36: Esempio di frame buffer object picking su oggetti di tipo Quad.

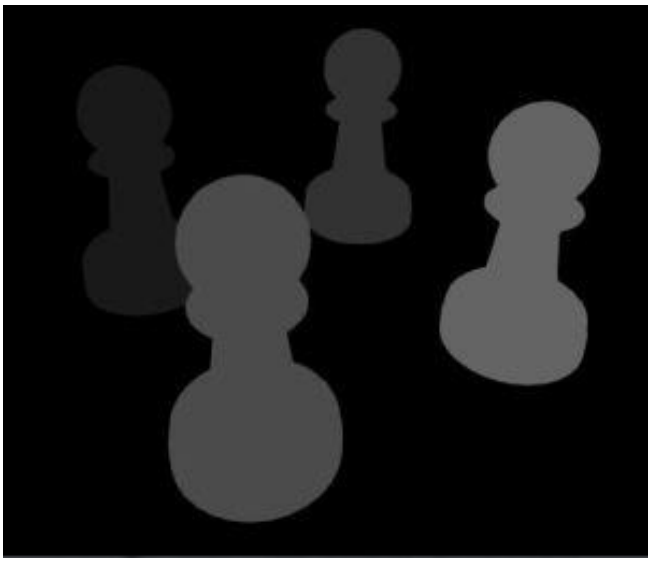

Figura 3.37: Tecnica del frame buffer object picking eseguita su dei pedoni di una scacchiera.

3.7 Animation

Il supporto alle animazioni è una caratteristica fondamentale di ogni game engine. Esse permettono di definire in modo esatto il cambiamento di una o più proprietà di un oggetto in funzione del tempo ed il loro utilizzo aggiunge quindi una certa dinamicità alla scena. Tali proprietà possono avere diversa natura, come ad esempio un colore, o un semplice valore scalare esposto da uno script, o ancora la posizione e l'orientamento di un oggetto. Il modo in cui un animatore crea un'animazione è tramite la definizione di keyframe su di una o più proprietà, dove un keyframe rappresenta il valore di quella proprietà ad un dato istante di tempo, e quest'ultimo viene contato a partire da un istante zero. L'engine si occuperà poi di determinare il valore di quella proprietà tra due keyframe, ovvero tra due istanti di tempo, questo avviene tipicamente tramite metodi di interpolazione che fanno uso di curve polinomiali parametriche come ad esempio le curve di Bezier, dove l'utente ha quindi la possibilità di intervenire sull'andamento della curva generata.

3.7.1 Interpolation Algorithms

L'engine realizzato fornisce due tipi di interpolazione tra i keyframe di un'animazione: Costante e Lineare. Nella prima, il valore della proprietà rimane costante fino al successivo keyframe. Nella seconda, il valore viene interpolato tramite interpolazione lineare secondo la formula: v = a + (b - a) * i; dove a e b rappresentano il valore della proprietà al keyframe attuale e successivo, ed i è un valore appartenente all'intervallo [0, 1] che rappresenta la distanza temporale dal keyframe attuale normalizzata nell'intervallo temporale tra i due keyframe. Le figure 3.38 e 3.39 mostrano un esempio grafico di entrambi i tipi di curve. Nell'interpolazione costante la proprietà cambierà di netto al sopraggiungere di un nuovo keyframe, nell'interpolazione lineare la proprietà varia con una velocità costante tra i due keyframe.

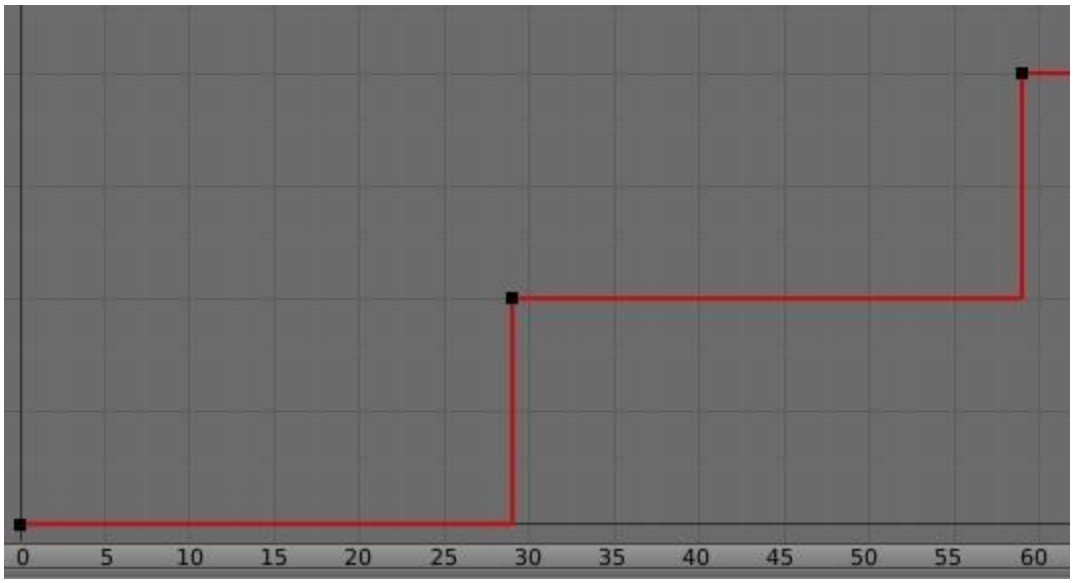

Figura 3.38: Esempio di keyframe con interpolazione costante.

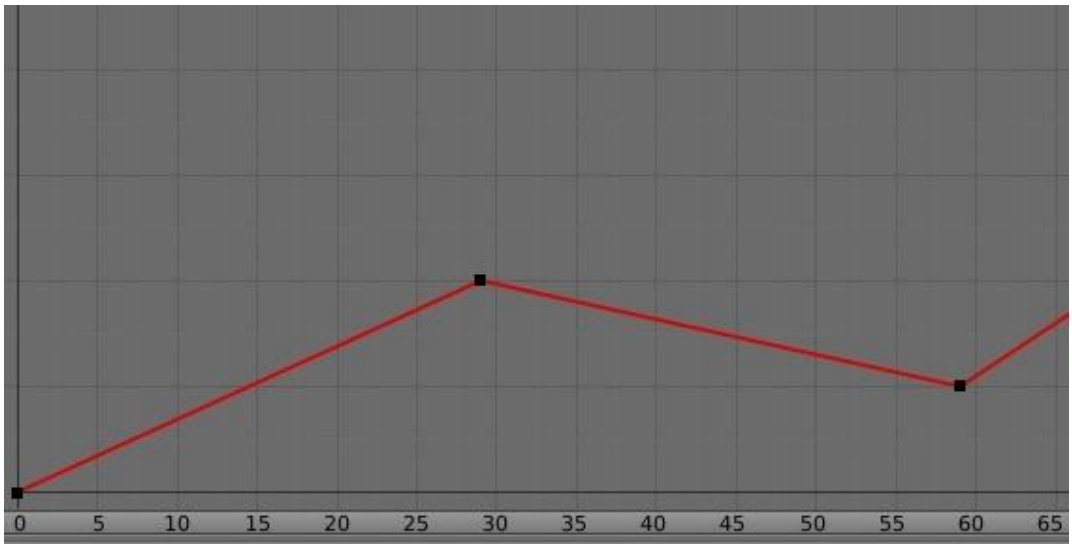

Figura 3.39: Esempio di keyframe con interpolazione lineare.

3.7.2 Animation Class

La classe Animation permette di creare all'interno dell'engine un asset che rappresenta un'animazione. Pertanto permette di specificare i vari keyframe di cui un'animazione verrà composta insieme alle proprietà su cui questi agiscono. Di seguito un esempio mostrante il codice necessario alla creazione di un'animazione:

```
Animation marioPowerupsToSuper = new Animation(
        "marioPowerupsToSuper",
        new HashSet<>(Arrays.asList(
            new KeyframedProperty(
                 "initialTileNumber",
                 new KeyFrame(0,181,InterpolationFunction.CONSTANT),
                 new KeyFrame(0.67f,181,InterpolationFunction.CONSTANT)
            ),
            new KeyframedProperty(
                 "finalTileNumber",
                 new KeyFrame(0,190,InterpolationFunction.CONSTANT),
                 new KeyFrame(0.67f,191,InterpolationFunction.CONSTANT)
            ),
            new KeyframedProperty(
                 "animationProgress",
                 new KeyFrame(0,0,InterpolationFunction.LINEAR),
                 new KeyFrame(0.67f,0.9545f,InterpolationFunction.LINEAR),
                 new KeyFrame(0.7f,1,InterpolationFunction.LINEAR)
            )
)));
```

Come si può vedere un'animazione sarà identificata da un nome e sarà composta da diversi keyframe per diverse proprietà. Chiaramente, il sistema si aspetta che l'oggetto sul quale verrà applicata tale animazione possieda tali proprietà.

3.7.3 AnimableObject & AnimableField

La classe AnimableField è un'interfaccia che viene sotto forma di annotazione Java e rappresenta il modo programmatico tramite il quale un campo di un oggetto può essere marchiato come animabile.

La classe AnimableObject è un'interfaccia e rappresenta il modo programmatico tramite il quale un oggetto può essere marchiato come animabile.

L'engine programmer utilizzerà tipicamente queste due interfacce per marchiare quali oggetti potranno essere riconosciuti dal sistema di animazione come target di un'animazione (gli AnimableObject), e quali saranno le proprietà animabili di un oggetto animabile (gli AnimableField).

Gli AnimableObject potranno anche essere annidati l'uno dentro l'altro, pertanto per indicare come proprietà di un keyframe un AnimableField annidato all'interno di più AnimableObject si utilizzerà la notazione puntata "AnimObj1AnimObj2.AnimField".

3.7.4 AnimationPlayer

L'AnimationPlayer è quella classe tramite la quale un'animazione viene applicata ad un oggetto. Essa contiene informazioni e metodi utili alla corretta riproduzione dell'animazione ed al suo aggiornamento, così come il suo attuale stato (playing, paused).

Tra le più utili informazioni contenute nell'AnimationPlayer è di particolare interesse il PlaybackType che determina il comportamento dell'animazione una volta che questa raggiunge la sua fine, ovvero l'ultimo keyframe. Il PlaybackType può essere di tipo FINISH_STOP, FINISH_RESET e CYCLIC. Nel caso FINISH_STOP l'animazione si fermerà sull'ultimo keyframe, nel caso FINISH_RESET una volta arrivata all'ultimo frame l'animazione si resetterà al primo frame, nel caso CYCLIC l'animazione ricomincerà dall'inizio.

3.7.5 Animator Component

Questo component è quello che abilita una entity a ricevere delle animazioni, queste vengono attivate tramite la creazione di un AnimationPlayer che verrà aggiunto all'Animator component. Questo, oltre a contenere quindi tutti i vari AnimationPlayer attivi sulla entity, espone anche degli utili metodi a controllare in modo semplificato il playback dei vari AnimationPlayer, così da mettere in play, in pausa, o settare il tempo di riproduzione di un'animazione senza dover scendere nei singoli AnimationPlayer contenuti nell'Animator. L'Animator svolge infine l'importante compito di interfacciarsi con l'AnimationEngine, esponendo a quest'ultimo utili informazioni di playback sugli AnimationPlayer interni, oltre a metodi per permettere l'aggiornamento degli AnimationPlayer interni in base a determinati filtri. Di seguito possiamo vedere il codice necessario a creare un Animator e lanciare un'animazione:

```
Animator goombaAnimator = new Animator();
goombaGameObj.addComponent(animationController);

goombaAnimator.addAnimationPlayer(new AnimationPlayer(
    goombaGameObj.getComponent(MeshRenderer.class).getMaterial(), //target
AnimableObject
    AssetManager.instance.getAnimation("enemyWalkingAnimation"),
//animation asset
    PlaybackType.CYCLIC, //PlaybackType
    UpdateType.CONTINUOUS //UpdateType
));

animationController.playAnimation("enemyWalkingAnimation");
```

3.7.6 The Animation Engine

L'Animation Engine è stato creato con lo scopo di avere un modulo a se stante che si occupasse di aggiornare in modo coerente le varie animazioni attive sulle entity. Esso tiene traccia di tutti gli Animator component attivi presenti nella scena ed è in grado di determinare quale sarà la prossima animazione a tempo continuo a terminare per prima entro un determinato lasso temporale. Quanto detto si riflette esattamente nella sua interfaccia lato codice, in quanto tra i metodi esposti troviamo rispettivamente quelli per aggiungere/rimuovere un Animator dalla lista degli Animator attivi, un altro che permette di ricavare il lasso di tempo del prossimo eventuale evento di animazione a tempo continuo ed infine un metodo update(dt, [filters]) che aggiorna in modo coerente le varie animazioni degli

Animator per un dato deltatime ed eventuali filtri. L'animation engine viene quindi coordinato dal game engine proprio tramite tale interfaccia ed in figura 3.40 è possibile vedere cosa accade a grandi linee durante l'aggiornamento.

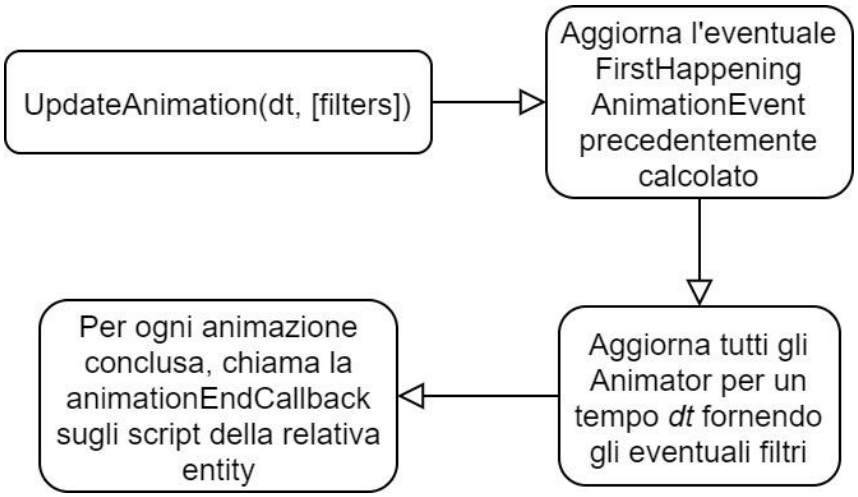

Figura 3.40: Operazioni eseguite dall'AnimationEngine a seguito di una richiesta di aggiornamento delle animazioni.

3.8 Game Logic

La parte di game logic consiste in tutti quei punti all'interno del ciclo di esecuzione del gioco all'interno dei quali l'utente può inserire la sua logica tramite codice. Tipicamente questo avviene tramite l'esposizione di un'interfaccia da parte dell'engine che l'utente può implementare e customizzare quindi a suo piacimento, dove ogni porzione dell'interfaccia avrà un contesto di esecuzione ben preciso e verrà eseguita in un determinato punto all'interno del game loop. Tramite la api dell'engine l'utente potrà poi apportare tutta una serie di modifiche alle varie entity presenti in scena così come ai loro component.

3.8.1 Script Component

L'engine realizzato permette lo scripting della logica di gioco tramite un altro component, chiamato Script, questo viene in genere esteso dall'utente che andrà poi ad implementare uno o più metodi dell'interfaccia esposta. Estendere la classe Script vuol dire concettualmente creare un determinato comportamento che un game object potrà avere all'interno della scena.

La classe Script possiede per comodità un riferimento al Transform component del game object, in quanto esso è spesso soggetto a manipolazioni da parte dell'utente, inoltre espone una serie di metodi che l'utente può sovrascrivere per implementare la logica di gioco:

- *constructor()*: all'interno del costruttore della classe il programmatore ha libero accesso alla api dell'engine. L'unica cosa alla quale prestare attenzione è non fare riferimento a game objects che non sono ancora stati creati. L'ordine di costruzione dei game object è quello specificato dal programmatore nel codice che crea la scena, per cui si ha il completo controllo anche su questo aspetto.
- *awake()*: questo è un metodo di inizializzazione. Viene chiamato una sola volta prima dei metodi start() ed update() di tutti gli script presenti in scena, viene chiamato anche

se lo Script cui appartiene non è abilitato. Tuttavia se l'intero game object è disabilitato ad inizio scena allora neanche il metodo awake() verrà invocato.

- *start()*: altro metodo di inizializzazione. Viene chiamato una sola volta subito dopo la fase di awake() ma prima della fase di update(). Viene chiamato solo se lo script risulta abilitato così come anche il suo game object.
- *update(float deltaTime)*: questo è il metodo tipicamente più utilizzato della classe Script e viene invocato ad ogni ciclo del game loop dopo che l'engine ha aggiornato la simulazione fisica e le animazioni, prima di eseguire la fase di rendering. Come argomento viene passato il deltaTime dal precedente ciclo di aggiornamento in modo tale da poter effettuare delle modifiche in funzione del tempo.
- *onContinuousCollisionEnter(CollisionInfo collisionInfo, Vector3fc collisionImpulse)*: questo metodo viene chiamato durante la fase di collision detection a tempo continuo, nel momento in cui il RigidBody associato al gameobject entra in contatto con un altro RigidBody. Durante l'invocazione di questo metodo il RigidBody sarà esattamente nella posizione di collisione con l'altro RigidBody. L'utente riceve diverse informazioni sulla collisione come la normale d'impatto e l'impulso generatosi, utili sia ad implementare un qualche tipo di logica sia per generare una risposta custom alla collisione nel caso il RigidBody del gameobject sia di tipo DYNAMIC_CUSTOM.
- *onContinuousTriggerEnter(CollisionInfo collisionInfo)*: metodo invocato durante la fase di collision detection a tempo continuo, quando uno o entrambi i RigidBody coinvolti nella collisione sono dei Trigger, nel momento in cui il RigidBody del gameobject entra in contatto con un altro RigidBody. Durante l'invocazione di questo metodo il RigidBody sarà esattamente nella posizione di contatto con l'altro RigidBody. L'utente riceve diverse informazioni sulla collisione utili ad implementare la logica di gioco.
- *onContinuousTriggerExit(CollisionInfo collisionInfo)*: metodo invocato durante la fase di collision detection a tempo continuo, quando uno o entrambi i RigidBody coinvolti nella collisione sono dei Trigger, nel momento in cui il RigidBody del gameobject esce dal contatto con un altro RigidBody. Durante l'invocazione di questo metodo il RigidBody sarà esattamente nella posizione di non contatto con l'altro RigidBody. L'utente riceve diverse informazioni sulla collisione utili ad implementare la logica di gioco.
- *animationEndCallback(String animationName)*: questo metodo viene invocato durante la fase di aggiornamento delle animazioni, quando un AnimationPlayer attivo all'interno dell'Animator del gameobject ha concluso la riproduzione di un'animazione. L'utente riceve il nome dell'animazione che ha terminato.

3.9 Audio

Il sistema audio, seppur secondario, rappresenta un'altra feature importante per un game engine in quanto l'aggiunta di tracce ambientali e sound effects permette di dare all'utente quel feedback in più che migliora il coinvolgimento con l'intera applicazione.

Oggi esistono sistemi per applicare alle tracce audio diversi filtri e trasformazioni, tra i più importanti sicuramente vi è la spazializzazione dell'audio all'interno della scena 3D.

Data la sua natura, l'engine realizzato implementa un semplice playback di audio 2D (ovvero senza spazializzazione) con la possibilità di loop, e supporta file audio con codifica

pcm ed mp3. Per la realizzazione di queste funzionalità è stata utilizzata la libreria OpenAL [45] inclusa all'interno di LWJGL [1].

3.9.1 AudioClip Component & AudioSource Component

La classe AudioClip permettere di caricare un file audio con codifica pcm o mp3 all'interno dell'engine, ed all'interno di quest'ultimo costituisce quell'asset che rappresenta quindi una traccia audio. L'AudioSource component permette invece di dotare un gameobject di funzionalità audio, e funge da player per un dato AudioClip. Pertanto l'AudioSource component oltre a richiedere come input un asset di tipo AudioClip, espone all'utente una serie di funzioni per controllare il playback ed il volume di una traccia audio. Di seguito viene mostrato un codice di esempio per aggiungere nell'engine un asset di tipo AudioClip e creare un AudioSource component:

```
AssetManager.instance.addAudioClip(
        new AudioClip(
                "MarioJumpSFX",
                soundsDirectory + "Jump.wav"
));

marioGameObj.addComponent(
        new AudioSource(AssetManager.instance.getAudioClip("MarioJumpSFX"))
);
```

3.10 Input

Anche il sistema di input rappresenta un elemento fondamentale di un game engine, soprattutto oggi in quanto i dispositivi di input si sono parecchio evoluti e diversificati, basti pensare ad esempio ai dispositivi di realtà virtuale basati su giroscopi, accelerometri e sensori infrarossi o di profondità, o ai guanti aptici.

L'input supportato dall'engine realizzato si basa su normale input da mouse e tastiera, ed a livello programmatico sfrutta la libreria GLFW [46] inclusa in LWJGL [1].

3.10.1 InputManager

Questa classe offre esclusivamente metodi statici, per tali ragioni non ha bisogno di essere istanziata. Essa fornisce metodi per ricavare la posizione del puntatore del mouse in canvas space, così come lo stato dei vari pulsanti di mouse e tastiera (PRESSED, RELEASED), oltre che funzioni di utilità per determinare se un tasto è appena passato da uno stato RELEASED a PRESSED e viceversa, queste ultime molto utili per determinare semplici click o singole pressioni di un tasto della tastiera.

3.11 How to use

L'engine sviluppato è utilizzabile sotto forma di libreria Java, questa va quindi aggiunta al proprio progetto per poterne usufruire. Come si evince in figura 3.41, un progetto avrà tipicamente una cartella 'resources' contenente gli asset, ed un'altra cartella contenente il codice di gioco. Quest'ultimo è tipicamente composto da una classe MyScenes contenente i metodi statici atti alla creazione delle scene pensate dall'utente, la classe Main che comprende

l'inizializzazione ed il caricamento degli asset all'interno dell'engine insieme all'avvio del game loop, ed una cartella 'scripts' contenente le varie classi (tipicamente degli script component) che implementano la logica di gioco dell'utente. Tale struttura di progetto, così come la nomenclatura di file e cartelle, sono completamente arbitrarie, quelle mostrate nell'immagine sono solamente una convenzione utilizzata dallo scrittore per mantenere un certo ordine logico e concettuale.

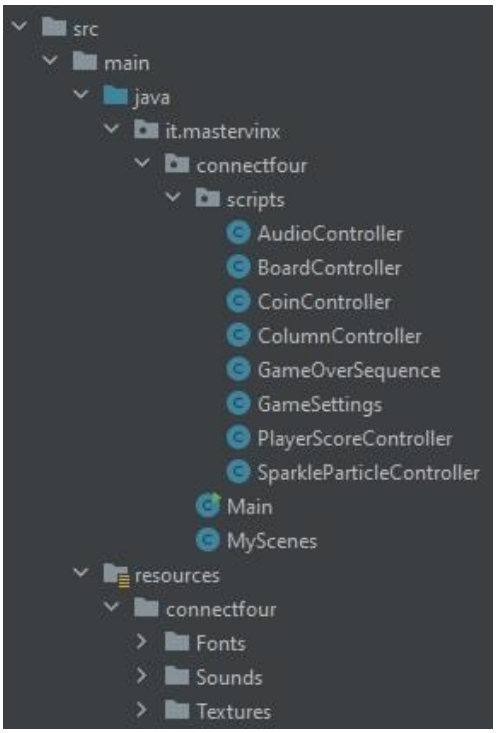

Figura 3.41: Tipica struttura di un progetto realizzato con l'engine.

3.11.1 User Scenes

La classe MyScenes come detto raggruppa il codice necessario alla creazione delle varie scene e contiene una sfilza di metodi statici, uno per scena, che saranno invocati dall'engine per caricare una scena quando richiesto dall'utente. In figura 3.42 si può vedere uno stralcio della classe MyScenes che contiene le scene del gioco clone Super Mario Bros.

```
public class MyScenes {
    public static void MainMenu() {...}
    public static void CurrentLevelAndLivesSummary() {...}
    public static void TimeUp() {...}
    public static void GameOver() {...}
    public static void TransitionToUnderworld() {...}
    public static void Scene1_1() {...}
    public static void Scene1_2() {...}
    {...}
}
```

Figura 3.42: Esempio di una tipica classe contenente le scene definite dall'utente.

3.11.2 User Main

La classe Main è solitamente quella dove viene creata l'istanza del game engine e specificati i parametri della finestra:

```
GameEngine gameEngine = new GameEngine(
        "ConnectFour", //ApplicationName
        1024, //window width
        576, //window height
        false //is vsync on?
);
```

Al suo interno vengono inoltre specificati i collision layer e creata la collision layer matrix, sempre ammesso che si voglia utilizzare il physics engine ed i RigidBody component:

```
PhysicsEngine.instance.collisionLayerMatrix.setCollisionLayersNames(
        "None",
        "All",
        "Terrain",
        "Coin",
        "BoardCellTrigger"
);

PhysicsEngine.instance.collisionLayerMatrix.setCollisionLayerMatrix(
        new int[][]{ //maxSize: 32x32. It must be squared. The bottom left
triangle (except the diagonal) must be all 0.
                                    //   L1,L2,L3,..   (L stands for Layer)
                {0,0,0,0,0},        //L1 1 , 1, 1,..
                {0,1,1,1,0},        //L2 0 , 1, 1,..
                {0,0,0,1,0},
                {0,0,0,1,0},
                {0,0,0,0,0}});      //Ln 0 , 0, 0,..
```

La classe Main procede anche al caricamento degli asset all'interno dell'engine tramite l'apposita classe AssetManager (facente parte della libreria), che si occupa di referenziare questi ultimi tramite dei nomi o degli id in base al tipo di asset. Fatto ciò vengono aggiunte alla classe Scenes (facente parte della libreria) le scene create dall'utente referenziando i metodi statici che le creano.

Infine viene chiamato il metodo *start()* sull'istanza del game engine, facendo così partire il game loop.

3.11.3 User Scripts

La cartella Scripts, come accennato precedentemente, contiene la logica dell'utente e sarà tipicamente composta dalle varie classi che ereditano da Script, oltre che da tutte le altre classi che l'utente creerà per i propri fini.

4. Games made with the In-House Game Engine

Per mostrare l'engine in azione insieme alle sue funzionalità sono stati realizzati tre giochi: un clone del Super Mario Bros del 1985 che include solamente i primi due livelli, un clone del famoso gioco mobile Flappy Bird, ed un ultimo gioco che si ispira al famoso gioco da tavolo ConnectFour conosciuto in Italia come Forza4. Essi verranno adesso presi in esame, facendo una breve disamina su quali feature dell'engine utilizzano e sulla loro realizzazione in generale.

4.1 Super Mario Bros Clone

Questo costituisce sicuramente il gioco più elaborato e completo rispetto agli altri realizzati e fa uso di praticamente tutte le feature esposte dall'engine.

4.1.1 Assets Preparation

Per la elaborazione degli asset grafici si è utilizzato il software Gimp [82], questi sono stati ricavati dal web e sono stati aggiustati graficamente laddove necessario, successivamente sono stati costruiti degli spritesheet ad-hoc contenenti tutte le varie texture animate e statiche degli elementi di gioco.

Per calcolare la corretta durata delle animazioni, per identificare esattamente da quale ciclo di sprite fosse composta un'animazione e per costruire i relativi spritesheet nel modo corretto, è stata eseguita una minuziosa analisi in slow-motion del gioco originale effettuando delle registrazioni di gameplay e riproducendo poi a velocità estremamente ridotta il video ottenuto.

Anche gli asset audio sono stati ricavati dal web e sono stati elaborati e sistemati laddove necessario tramite il software Audacity [83].

Il gioco è basato su tilemap, ovvero delle griglie di numeri interi che rappresentano le varie tile di cui si compone, tali griglie sono state costruite graficamente tramite il software Tiled [84], che permette la creazione di tilemap a partire da degli spritesheet, le tilemap dei due livelli di gioco realizzati sono state costruite tramite tale software.

4.1.2 Exploited Features

Il gioco sfrutta il TileMap component per generare sia il mondo di gioco che i vari character, grazie a tale component ogni tile è stata arricchita dalla logica e dagli ulteriori component specificati via codice. Tale logica è identica a quella presente nel titolo originale ed è stata derivata dopo un'attenta analisi del gameplay e delle meccaniche di gioco.

Diversi elementi di gioco, quali il personaggio principale ed i vari nemici, box distruttibili, powerup ed elementi di ui utilizzano l'Animator component per eseguire delle animazioni sia sul Material che sul Transform.

Il gioco sfrutta la collision matrix e la fisica a tempo continuo offerta dall'engine, pertanto tutti i bug presenti nel gioco originale che derivavano da un calcolo della fisica a tempo discreto quali wall-jumps e wall-sliding non sono presenti in questa versione.

I vari elementi di gioco sfruttano i RigidBody component in tutti i loro tipi, da quelli DYNAMIC dove la fisica è calcolata dall'engine, a quelli DYNAMIC_ANIMATED come diversi box dove la fisica è calcolata a partire da un'animazione attiva sul transform dell'oggetto, a quelli DYNAMIC_CUSTOM dove la fisica è dettata dall'utente, a quelli STATIC come tutto il mondo di gioco che non risulta animabile in nessun modo. Inoltre sono presenti anche RigidBody con la funzione di trigger, come i box invisibili interagibili solamente se spinti dal basso.

La funzione di raycasting è parecchio utilizzata dal personaggio principale per stabilire se si trova su di un oggetto o in volo. La funzione di raycasting è anche utilizzata a livello di canvas per muoversi con il mouse sui bottoni presenti nel menù principale.

Il particle system offerto dall'engine è utilizzato per realizzare l'effetto lanciafiamme del personaggio principale mostrato in azione in figura 4.1. Le particelle godono inoltre di fisica continua in modo da interagire con l'ambiente circostante.

E' fatto grande utilizzo anche degli AudioSource component per riprodurre sfx e musiche ambientali sia in loop che non, attingendo da file audio sia aventi codifica mp3 che pcm.

L'input è gestito in modo molto semplice sia da mouse che da tastiera grazie al supporto dell'engine.

Nel gioco sono state realizzate diverse scene, il menù principale, il primo livello (si veda figura 4.1), il secondo livello (si veda figura 4.2), le scene di intermezzo tra un livello ed un altro, le scene di game over e timeout, tutte sono state definite via codice ed aggiunte alla classe Scenes dell'engine, ed infine richiamate via codice quando necessario.

Figura 4.1: Screenshot tratto dal primo livello del gioco Super Mario Bros Clone.

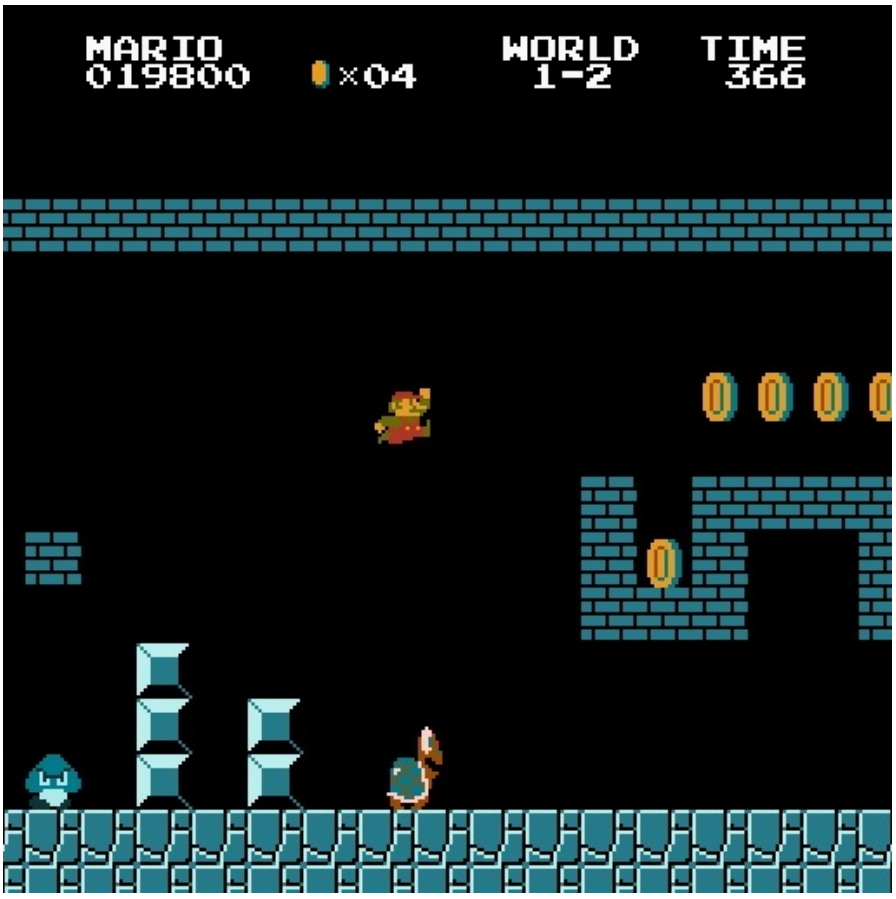

Figura 4.2: Screenshot tratto dal secondo livello del gioco Super Mario Bros Clone.

4.2 Flappy Bird Clone

Questo gioco è stato realizzato per dimostrare come l'engine si adatti bene anche ai tap-games 2D, giochi solitamente mobile rivolti ad un utenza casual.

4.2.1 Assets Preparation

Gli asset grafici sono stati ricavati dal web ed elaborati con Gimp [82]. L'unica spritesheet costruita in questo caso è quella del personaggio principale.

Anche in questo caso per il calcolo delle tempistiche delle animazioni e per la realizzazione dei vari effetti è stata eseguita un'attenta analisi del gioco originale, tuttavia a differenza di Super Mario Bros Clone non è stata necessaria un'analisi in slow-motion in quanto gli effetti grafici sono risultati molto più semplici.

Gli asset audio sono stati anch'essi ricavati dal web ed elaborati tramite Audacity [83].

4.2.2 Exploited Features

Tra le feature dell'engine sfruttate dal gioco troviamo lo SpriteRenderer per renderizzare i vari elementi grafici e l'Animator per applicare degli effetti di animazione sul personaggio principale e su elementi di scenario e di ui.

I vari elementi interagibili di giochi sono dotati di RigidBody di diverso tipo in quanto nel gioco sono presenti elementi sia statici che dinamici, elementi le cui collisioni sono regolate tramite la collision matrix.

Il gioco utilizza le utilità di input offerte dall'engine per ricevere un tap sia da mouse che da tastiera.

Viene utilizzato il Canvas per mostrare a schermo una semplice ui di game over con possibilità di ricominciare o terminare la partita, a tal proposito viene utilizzato il canvas raycasting per intercettare gli eventi di mouse hover sopra i bottoni della ui.

Il gioco sfrutta ovviamente gli AudioSource components per la riproduzione di effetti sonori.

Le scene realizzate sono solamente una di cui si può vedere uno stralcio in figura 4.3, pertanto questa viene ricaricata quando il giocatore perde e decide di ricominciare.

Figura 4.3: Screenshot tratto dal gioco Flappy Bird Clone.

4.3 Connect Four

Questa riproduzione con regole customizzate del gioco Forza 4 differisce rispetto ai giochi precedenti soprattutto per quanto riguarda la realizzazione degli asset, in quanto in tal caso si è voluto dare all'applicazione un aspetto più realistico anziché cartoon.

4.3.1 Assets Preparation

Il tabellone di gioco è generato a partire da un modello 3D realizzato su Blender [85], insieme al modello delle monete, dopodichè sono stati eseguiti dei render isolati sia del tabellone che delle monete in modo da avere delle sprite in trasparenza.

L'ombra del tabellone è stata poi generata su Blender tramite shadow catching e compositata su di una foto che è stata usata come sfondo del gioco.

Lo spritesheet utilizzato dall'effetto particellare è stato invece generato ed assemblato su Gimp [82].

Gli asset audio degli effetti sonori sono stati reperiti sul web ed elaborati con Audacity [83].

4.3.2 Exploited Features

Il gioco sfrutta lo SpriteRenderer per renderizzare secondo un certo ordine i tre layer che compongono l'applicazione, ovvero il background, il layer delle monete ed il layer del tabellone.

Le monete sono dotate di RigidBody di tipo DYNAMIC o DYNAMIC_ANIMATED in base al flusso di gioco e vengono lasciate cadere al click del mouse sopra una delle varie colonne che compongono il tabellone di gioco. Tali colonne sono equipaggiate con un RigidBody che viene utilizzato per effettuare il test di intersezione con il raggio generato dal click del mouse. Alla base del tabellone di gioco giace un box invisibile equipaggiato con un RigidBody in modo che le monete una volta lasciate cadere vadano a posarsi su tale base.

L'effetto particellare di sparkling che si attiva quando il giocatore combina quattro o più monete adiacenti è composto da tre ParticleSystem component in modo da avere un effetto più variegato.

Gli AudioSource component sono utilizzati per riprodurre gli effetti sonori del gioco.

Il Canvas component è utilizzato per mostrare una semplice ui contenente gli score dei giocatori.

L'Animator component è infine utilizzato sia per riprodurre l'animazione di floating sulla moneta che il giocatore sta posizionando sia per animare il parametro alpha di un elemento di ui che viene attivato alla vittoria di uno dei due giocatori.

Il gioco si svolge in un'unica scena, di cui è possibile vedere uno screenshot in figura 4.4.

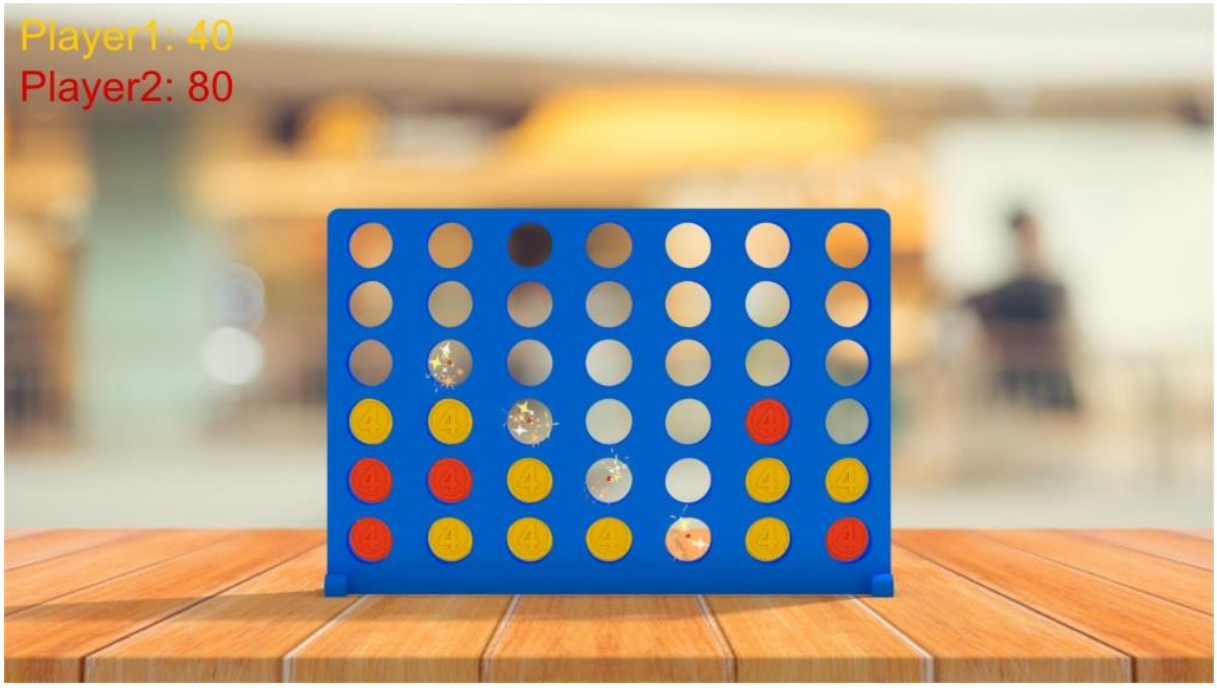

Figura 4.4: Screenshot tratto dal gioco Connect Four.

5. Teaching

La relativa semplicità dell'engine realizzato (che si propone come motore per giochi 2D), così come il numero di feature ed una codebase ridotti (circa 10K righe di codice escludendo commenti, linee vuote ed i giochi realizzati) rispetto a engine enormemente più grandi come quelli utilizzati a livello industriale discussi nei primi paragrafi, e l'utilizzo di uno tra i più popolari e newbie-friendly linguaggi di programmazione ad oggetti fanno di questo progetto un ottimo strumento per fini didattici.

5.1 Sweet Implementation Override Spots

All'interno dell'engine ci sono diversi punti dove è possibile modificare l'implementazione di una particolare funzionalità senza la necessità di metter mani o avere conoscenza della logica di numerose altre classi, tra questi:

- Integration Algorithms: si tratta di un'interfaccia che definisce le funzioni esposte da un algoritmo di integrazione e viene utilizzata durante la stima delle proprietà fisiche cinematiche del corpo rigido per un dato lasso di tempo. L'utente può così implementare tale interfaccia creando altri algoritmi di integrazione, accompagnando magari il tutto con una semplice scena di test per verificare le prestazioni e la resa del suo algoritmo.
- Space Partitioning Structures: come discusso nel paragrafo riguardante la fisica, il physics engine realizzato al momento sfrutta una griglia 2D uniforme come struttura di partizionamento spaziale. Tale struttura è utilizzata per ridurre il numero di confronti eseguiti durante la fase di collision detection ed implementa un'interfaccia che definisce i metodi che ogni struttura di partizionamento dovrebbe implementare. L'utente può quindi dilettarsi nella creazione di altre strutture di partizionamento, ad esempio i Quadtree, semplicemente implementando tale interfaccia, ed esaminando con una scena di test le performance ottenute.
- Broad & Narrow Phases: anche in questo caso è stata realizzata un'interfaccia che definisce i metodi usati dal physics engine per eseguire la broad e la narrow phase. L'implementazione di una broad phase è tipicamente molto correlata alla struttura di partizionamento spaziale utilizzata: è la struttura che conosce come i dati sono organizzati al suo interno e sa come poter determinare i corpi rigidi in collisione effettuando il minor numero di test. Pertanto l'interfaccia di broad phase è implementata tipicamente dalla struttura di partizionamento, nel nostro caso la griglia uniforme 2D. L'utente può quindi modificare tale implementazione, o creare un'altra struttura di partizionamento con la sua implementazione di broad phase, e sperimentare come le varie implementazioni di broad phase possano inficiare sulla complessità computazionale dell'intera fase di collision detection.
- Render Algorithm: il metodo render() del Rendering Engine è un altro interessante punto di studio, dove l'utente può implementare liberamente la sua logica senza la necessità di conoscere a fondo il contesto. Esso può quindi modificare il modo in cui il processo di render viene effettivamente eseguito, ad esempio modificando l'ordine di rendering dei vari MeshRenderer gestiti dal rendering engine, vedendo poi a schermo il risultato della sua logica.

5.2 Key Topics

Oltre ai punti discussi nel precedente paragrafo, molte altre parti dell'engine sono di grande interesse didattico.

E' possibile apprendere nozioni fondamentali su come la fisica viene simulata ed aggiornata all'interno di un engine, sui vari step e le criticità che compongono la fase di collision detection, su come il raycasting viene effettivamente implementato, sull'importanza di avere delle strutture di partizionamento spaziale, sui vari tipi di collider esistenti e sui vantaggi computazionali che ognuno di loro può portare nelle varie fasi di aggiornamento della simulazione.

Inoltre si può vedere nel dettaglio da cosa è composta un'animazione a livello grezzo, come questa viene aggiornata dall'engine, e l'importanza delle tecniche di interpolazione in materia di animazione.

Lato rendering, viene mostrato come avviene l'interfacciamento con una delle librerie grafiche oggi più utilizzate, ovvero OpenGL, inoltre vengono sviscerati concetti come quello di mesh, texture, shader e material, fornendo delle implementazioni così da capire a livello software di quali dati le schede grafiche hanno bisogno e come questi vengono utilizzati. A proposito di shader, gli studenti potrebbero dilettarsi nella realizzazione di semplici shader GLSL per prendere padronanza con il linguaggio ed il concetto di Uniform, questo li renderebbe più consapevoli sia sulla funzione delle matrici model-view-projection, sia sulla pipeline di trasformazione di un vertice da word space a canvas space, sia sui vari tipi di shader esistenti ed il loro utilizzo all'interno della pipeline di rendering.

Dal puro punto di vista informatico, l'engine risulta sicuramente utile come esempio di applicazione di diversi pattern e concetti di programmazione ad oggetti e programmazione funzionale: è possibile mostrare l'applicazione del pattern architetturale Entity-Component System che trova riscontro nelle classi GameObject e GameObjectComponent, così come l'utilizzo di numerosi altri pattern quali Lazy Initialization, Singleton, Container, Facade, Private Class Data, Iterator, Observer, Template Method, ed altri, che trovano impiego nelle implementazioni dei diversi component, sub-engine e strutture dati di supporto; all'interno del codice vengono sfruttati diversi aspetti sia object-oriented che funzionali del linguaggio Java, tra cui interfacce, classi astratte, annotazioni, polimorfismo tramite override di metodi della classe base, metodi statici, tipi template, delegati tramite costrutti ad-hoc, reflection, funzioni lambda, monadi e funtori.

6. Possible Improvements

Come accennato nella sezione relativa al teaching, l'engine realizzato è comunque piccolo se paragonato ad engine usati in ambito industriale, ciò lascia spazio ad un ampio ventaglio di miglioramenti sia delle feature già esistenti sia in termini di nuove funzionalità che potrebbero essere aggiunte, di cui adesso si farà una breve disamina.

6.1 Discrete Collision Detection

Il tipo di fisica attualmente supportato dall'engine è a tempo continuo, ciò permette all'utente di sapere con precisione molto elevata l'esatto istante di collisione di due RigidBody all'interno della simulazione. Tuttavia tale fisica è comunque computazionalmente onerosa da sostenere, pertanto sarebbe opportuno integrare nell'engine anche la possibilità di poter simulare una fisica a tempo discreto, in tal modo le collisioni verrebbero rilevate solamente ad intervalli ben precisi di tempo, tuttavia ciò è spesso sufficiente per un gran numero di applicazioni.

L'engine implementa già gran parte delle funzionalità che servirebbero per supportare una fisica a tempo discreto, ad esempio la collision detection a tempo discreto si basa su semplici intersection test per determinare se due corpi stanno compenetrando o meno, in questo caso l'engine implementa già l'intersection test AABBvsAABB basato su SAT di cui si è già discusso nel paragrafo relativo alla collision detection a tempo discreto.

Relativamente alla collision resolution a tempo discreto sarebbe possibile implementare un approccio basato sul ***metodo della proiezione e dell'impulso***. Il metodo della proiezione prevede che la compenetrazione tra due corpi rigidi venga risolta spostando entrambi i corpi lungo la direzione normale alla collisione in modo proporzionale alla massa di ognuno, il metodo dell'impulso prevede che i corpi subiscano una variazione istantanea di velocità a seguito dell'impatto, ovvero un impulso, calcolato in base alla quantità di moto dei due oggetti, in modo tale che i corpi non entrino di nuovo in collisione alla ripresa della simulazione. Nel caso di corpi aventi collider di tipo OBB, bisognerà tener conto anche della posizione d'impatto rispetto al baricentro dei corpi, in quanto con molta probabilità verrà generato un impulso angolare oltre che lineare.

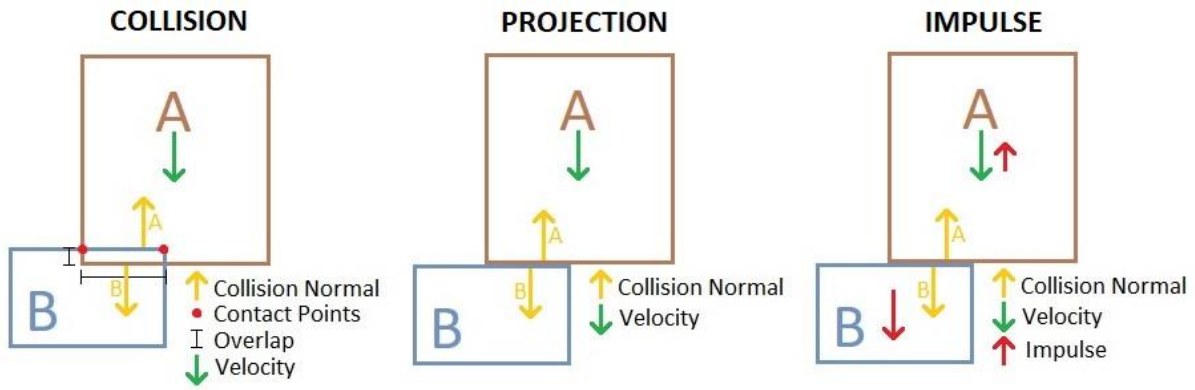

Figura 6.1: Risoluzione di una collisione applicando in cascata il metodo della proiezione ed il metodo dell'impulso.

6.2 OBB Colliders

Gli oriented bounding boxes rappresentano la naturale evoluzione degli AABB e porterebbero numerosi vantaggi, sia lato utente in quanto si avrebbero dei box che potrebbero ruotare, sia lato engine. In questo ultimo caso, il vantaggio più considerevole lo si avrebbe probabilmente durante la fase di collision detection di corpi dinamici.

Dai precedenti paragrafi si ricorda che la broad phase nellla collision detection a tempo continuo tra corpi statici e dinamici sfrutta dei semplici intersection test basati sul SAT, in tal caso i corpi dinamici vengono equipaggiati con un motionSweptCollider che incapsula l'interno movimento del corpo durante il deltatime. Usando degli AABB come motionSweptCollider si andrà a creare molto spazio inutilizzato che produrrà numerosi intersection test falsi positivi, gli OBB risultano invece ottimi per incapsulare il moto di un corpo durante il deltatime, di conseguenza la broad phase produrrà un numero molto inferiore di falsi positivi risultando molto più efficace. L'uso di un OBB per incapsulare il moto di un oggetto risulta molto buono anche durante la broad phase del raycasting, in quanto si potrebbero dapprima determinare gli oggetti che intersecano il raggio tramite un semplice intersection test tra gli oggetti ed un OBB con extent nullo che rappresenta quindi un raggio, e successivamente si potrà effettuare la near phase (che ritorna l'istante di collisione tra il raggio ed un corpo) tra il raggio ed i corpi che hanno superato la broad phase in modo da trovare il corpo rigido che interseca per primo il raggio. Attualmente, utilizzare un AABB come broad phase per il raycasting genera molto spazio inutilizzato all'interno del bounding volume che causa un gran numero di intersection test falsi positivi.

Sebbene l'uso di OBB porti numerosi vantaggi in generale la loro introduzione all'interno dell'engine non sarebbe banale, molte implementazioni e considerazioni all'interno dell'engine si basano sull'assunzione di star usando degli AABB, l'utilizzo di OBB non solo invaliderebbe tali assunzioni ma complicherebbe anche la fase di collision resolution in quanto i corpi potranno ruotare.

6.3 Bezier Animation Curves

Attualmente l'engine supporta curve di animazione basate su interpolazione lineare e costante. Ovviamente ciò costituisce un grande limite, soprattutto per gli animatori, in quanto per realizzare delle curve di ease-in/out si è costretti a creare un'animazione con numerosi keyframe con interpolazione lineare che approssimino l'andamento di una tale curva. L'introduzione delle curve di Bezier aiuterebbe a superare questo limite e renderebbe il processo di definizione di un'animazione basata su ease-in/out molto più semplice.

6.4 Audio Effects

Gli AudioSource component presenti nell'engine permettono la riproduzione delle tracce audio così per come sono state realizzate, agendo al più sul volume della traccia. Tuttavia la libreria utilizzata internamente all'engine, ovvero OpenAL, supporta la possibilità di applicare diversi effetti alle tracce audio, così come la possibilità di spazializzare una sorgente audio all'interno della scena. Esporre ed integrare tali funzionalità all'interno dell'engine porterebbe delle migliorie non da poco. Inoltre, anche nel caso dei migliori game engine in circolazione, molto spesso vengono utilizzati dei middleware di terze parti quali Wwise [86] o FMOD [87] come motori audio esterni anziché le funzionalità audio integrate all'engine. Sarebbe interessante fare uno studio su quali sono le feature che spingono le software house ad

utilizzare engine audio di terze parti e cercare di implementare tali feature all'interno del game engine stesso.

6.5 More Input Devices

Al momento l'engine supporta la ricezione di input tramite mouse e tastiera. Tuttavia la libreria GLFW [46] supporta anche la possibilità di ricevere input tramite joypad, pertanto integrare tali funzionalità all'interno dell'engine non dovrebbe risultare così complesso.

6.6 Graphical Assets Importing

E' noto che gli artisti creano i loro modelli, materiali ed animazioni su software esterni, pertanto negli anni sono stati creati diversi formati standard per l'importazione/esportazione di tali asset tra un software ed un altro, tra i quali i più noti sono STL, OBJ, FBX, DAE, 3DS e glTF. L'engine realizzato permette di creare mesh ed animazioni via codice, ciò è fattibile in un contesto 2D nel quale la mesh principalmente utilizzata è un banale Quad, tuttavia per mesh ed animazioni complesse tale processo risulta impossibile, e gli asset vanno quindi importati da file esterni e parsati all'interno delle relative classi definite nell'engine. In questo caso si hanno due possibilità: la prima prevede di implementare dei parser proprietari ad-hoc, la seconda possibilità è quella tipicamente più utilizzata e prevede l'utilizzo di librerie di terze parti per la lettura di asset prodotti ed esportati nei formati sopra citati. Una di queste librerie è Assimp [88] (Open Asset Import Library), nativamente scritta in C++ ma per la quale esistono porting anche per Kotlin e Java [89], pertanto questa potrebbe essere utilizzata all'interno dell'engine per favorire l'import di asset grafici da file.

6.7 Scenes Import/Export

Al momento le varie scene vengono definite dall'utente via codice e ciò permette la massima flessibilità di creazione dei vari elementi, tuttavia non si ha un vero e proprio formato tramite il quale una scena creata all'interno dell'engine può essere esportata/importata, più nello specifico non si ha un asset di tipo scena che permette il caricamento di una scena da file. Si potrebbe pensare di utilizzare gli stessi formati per l'import/export degli asset grafici, tuttavia tali formati non hanno conoscenza dei vari component offerti dall'engine, ne degli eventuali component custom definiti dall'utente e neanche dei loro eventuali campi di input. Alcuni di tali formati come ad esempio glTF [90] permettono la definizione di attributi extra, ciò consentirebbe di supportare attributi in formati custom definiti dalla particolare applicazione, tuttavia tali formati rimangono comunque dei contenitori di asset grafici, ovvero contengono informazioni raw sulla composizione dei vertici di una mesh, o dei keyframe di un'animazione. Un formato che definisce una scena all'interno dell'engine dovrebbe invece solamente fare riferimento all'utilizzo di una particolare mesh, così come all'utilizzo di una particolare animazione o un component, senza contenere i dati che definiscono quell'asset stesso, in quanto tali dati saranno appunto contenuti in un asset a parte. Per realizzare ciò si potrebbe definire uno schema custom basato ad esempio su json o xml, che permetterebbe di definire i vari GameObject presenti in scena insieme ai loro component ed i campi di input di questi ultimi, che potrebbero far riferimento ad altri asset che ci si aspetta di trovare all'interno dell'engine.

6.8 GUI

L'interfaccia grafica è sicuramente un elemento fondamentale di ogni strumento software, e questo game engine non fa eccezione. La creazione di un'interfaccia grafica sarebbe di aiuto nel setup di scena così come nella fruizione generale dell'engine da parte di utenti che non hanno basi di programmazione. Chiaramente la realizzazione di una GUI comporta una serie di requisiti alcuni dei quali sono già stati discussi come possibili miglioramenti, come la necessità di definire un formato per l'import/export di una scena, in quanto l'utente avrà la necessità di riaprire e modificare una scena tra diverse istanze dell'applicazione. Una interfaccia grafica per l'engine potrebbe essere realizzata tramite l'engine stesso, anzi costituirebbe un ottimo progetto tramite il quale altri utenti potrebbero imparare ad utilizzare l'engine o prendere spunto per la realizzazione di alcuni effetti o imparare ad utilizzare determinate feature.

6.9 3D

Trasformare l'engine da 2D a 3D costituisce un lavoro molto impegnativo. Alcune funzioni dell'engine sono già 3D ma risultano adattate al 2D, altre funzionalità ed implementazioni si basano algoritmicamente sull'assunzione di essere in 2D e pertanto non funzionerebbero più una volta passati al 3D. Tuttavia il passaggio al 3D incrementerebbe a dismisura il ventaglio di feature e migliorie implementabili all'interno dell'engine, ed in tal caso un confronto con le feature offerte dai game engine utilizzati a livello industriale potrebbe iniziare ad aver senso. Con il 3D alcune funzionalità di cui non si è tenuto minimamente in considerazione nella realizzazione di questo engine potrebbero prender vita, come ad esempio audio spazializzato, illuminazione, ombre ed effetti grafici avanzati che necessitano del concetto di luce per funzionare come ad esempio il normal mapping ed il physically based rendering. Anche l'utilizzo di alcune funzionalità della pipeline di rendering come ad esempio il depth testing comincerebbero ad avere più senso. L'engine potrebbe essere utilizzato per fare dei veri e propri esperimenti su tecniche di rendering innovative che risultano impossibili o comunque molto limitate in un contesto 2D per ovvie ragioni. Anche il ventaglio di collider disponibili dovrà essere incrementato per il passaggio al 3D, rendendo praticamente obbligatoria l'implementazione degli OBB e delle relative funzioni per il collision testing. L'engine realizzato costituisce sicuramente un'ottima base per capire a fondo il funzionamento interno di un motore di gioco così come dei vari sub-engine (physics, rendering) e le varie fasi di cui si compone il game loop, ed aiuta a rendersi conto di quali sono le sfide e le criticità che vanno affrontate così come la loro entità, tuttavia alcuni algoritmi risultano semplificati per via della natura bidimensionale per i quali sono pensati e la loro implementazione in 3D potrebbe richiedere ulteriori ottimizzazioni necessarie per restringere quanto più possibile la complessità computazionale.

7. Riferimenti

[1] Lightweight Java Game Library official website. https://www.lwjgl.org/

[2] idSoftware official website. https://www.idsoftware.com/

[3] Epic Games official website. https://www.epicgames.com/site/

[4] Unreal Engine official website. https://www.unrealengine.com/

[5] Forging new paths for filmmakers on "The Mandalorian". https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian

[6] T. De Goussencourt, J. Dellac and P. Bertolino, "A game engine as a generic platform for real-time previz-on-set in cinema visual effects", International Conference on Advanced Concepts for Intelligent Vision Systems (ACIVS 2015), Oct. 2015.

[7] Mr. Stuart Armstrong, "Game Engine Review", Research Parkway, Suite 350 Orlando FL, USA

[8] Samčović. A, "Serious games in military applications", Vojnotehnicki Glasnik, 66(3), 597–613, 2018.

[9] Leitão, A, Castelo-Branco, R, Santos, G. Game of renders: the use of game engines for architectural visualization. In: Intelligent & Informed: Proceedings of the 24th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA)- volume 1 (eds Haeusler, MH, Schnabel, MA, Fukuda, T), 15–18 April 2019, pp. 655–664. Wellington, New Zealand: Victoria University of Wellington.

[10] Edwards G, Li H, Wang B (2015) BIM based collaborative and interactive design process using computer game engine for general end-users. Vis Eng 3(1):1.

[11] Marks S, Windsor J, Wünsche B. Evaluation of game engines for simulated clinical training. In: New Zealand Computer Science Research Student Conference 2008; 2008. p. 92–99.

[12] Crytek official website. https://www.crytek.com/

[13] CryEngine official website. https://www.cryengine.com/

[14] Unity official website. https://unity.com/

[15] Godot official website. https://godotengine.org/

[16] Amazon Lumberyard official website. https://aws.amazon.com/it/lumberyard/

[17] Dice official website. https://www.dice.se/

[18] Frostbite official website. https://www.ea.com/frostbite/engine

[19] Valve Corporation. https://www.valvesoftware.com/

[20] Source Engine wiki. https://developer.valvesoftware.com/wiki/Source

[21] id Tech engine history. https://fat-studios.medium.com/the-history-of-the-id-tech-engine-4dbf7c70ef5b

[22] IW Engine wiki. https://www.wikiwand.com/it/IW_Engine

[23] Infinity Ward official website. https://www.infinityward.com/

[24] Ubisoft official website. https://www.ubisoft.com/

[25] Anvil engine wiki. https://assassinscreed.fandom.com/wiki/Anvil_(game_engine)

[26] Cocos official website. https://www.cocos.com/

[27] GameMaker Studio official website. https://www.yoyogames.com/en/gamemaker

[28] jMonkeyEngine official website. https://jmonkeyengine.org/

[29] FXGL official github. https://github.com/AlmasB/FXGL

[30] LITIEngine official website. https://litiengine.com/

[31] libGDX official website. https://libgdx.com/

[32] LionEngine official website. https://github.com/b3dgs/lionengine

[33] SilenceEngine official github. https://github.com/sriharshachilakapati/SilenceEngine

[34] Golden T Game Engine official website. http://goldenstudios.or.id/products/GTGE/

[35] AndEngine official github. https://github.com/nicolasgramlich/AndEngine

[36] FIFE Engine official github. https://github.com/fifengine/fifengine

[37] Castle Game Engine official website. https://castle-engine.io/

[38] Tilengine official website. http://www.tilengine.org/

[39] PhysX SDK official webpage. https://developer.nvidia.com/physx-sdk

[40] Bullet Physics SDK official github. https://github.com/bulletphysics/bullet3

[41] Havok Physics official website. https://www.havok.com/havok-physics/

[42] Box2D official github. https://github.com/erincatto/box2d

[43] ECS pattern. https://www.gamasutra.com/blogs/TobiasStein/20171122/310172/The_EntityComponentSystem__An_awesome_gamedesign_pattern_in_C_Part_1.php

[44] OpenGL official website. https://opengl.org/

[45] OpenAL official website. https://openal.org/

[46] GLFW official website. https://www.glfw.org/

[47] GLSL wiki. https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language

[48] Numerical integration methods for game physics. https://gafferongames.com/post/integration_basics/

[49] S. Gottschalk, Separating axis theorem. Technical Report TR96-024, Dept. of Computer Science, UNC Chapel Hill, 1996

[50] Sweep tests for moving objects. https://www.gamasutra.com/view/feature/131790/simple_intersection_tests_for_games.php

[51] Spatial partitioning overview. https://gameprogrammingpatterns.com/spatial-partition.html

[52] Some grid and quadtree implementations. https://stackoverflow.com/questions/41946007/efficient-and-well-explained-implementation-of-a-quadtree-for-2d-collision-det

[53] C. Ericson, Real-time collision detection (CRC Press, 2004)

[54] Broad-Phase Collision Detection with CUDA. https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-32-broad-phase-collision-detection-cuda

[55] Swept AABB Collision Detection and Response. https://www.gamedev.net/tutorials/_/technical/game-programming/swept-aabb-collision-detection-and-response-r3084/

[56] SAT theory and implementation. https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169

[57] D. Kodicek, Mathematics and Physics for Programmers, 2nd Edition

[58] Ray Tracing vs Rasterized Rendering – Explained. https://appuals.com/ray-tracing-vs-rasterized-rendering-explained/

[59] Rendering an Image of a 3D Scene: an Overview. https://www.scratchapixel.com/lessons/3d-basic-rendering/rendering-3d-scene-overview/visibility-problem

[60] An Overview of the Ray-Tracing Rendering Technique. https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview

[61] Rasterization: a Practical Implementation. https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm

[62] Nvidia RT Cores vs. AMD Ray Accelerators – Explained. https://appuals.com/nvidia-rt-cores-vs-amd-ray-accelerators-explained/

[63] D. Luebke, S. Parker, Interactive Ray Tracing with CUDA (NVIDIA Research 2008)

[64] NVIDIA official website. https://www.nvidia.com/

[65] AMD official website. https://www.amd.com/en

[66] What's the Difference Between Ray Tracing and Rasterization?. https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/

[67] M. Abrash, Rasterization on Larrabee. Dr. Dobbs Portal, 2009

[68] OpenGL basics learning website. https://learnopengl.com/Getting-started/

[69] Advanced OpenGL learning website. https://learnopengl.com/Advanced-OpenGL/

[70] HLSL official programming guide. https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl

[71] Texture Compression Techniques and Tips. https://www.gamasutra.com/view/feature/130906/texture_compression_techniques_and_.php

[72] Graphical artifacts live preview. https://www.testufo.com/stutter

[73] G-Sync vs FreeSync Explained. https://www.viewsonic.com/library/entertainment/g-sync-vs-free-sync-explained/

[74] Cg shading language wiki. https://www.khronos.org/opengl/wiki/Cg

[75] Direct3D wikipedia webpage. https://en.wikipedia.org/wiki/Direct3D

[76] Khronos Group official website. https://www.khronos.org/

[77] Vulkan official website. https://www.vulkan.org/

[78] OpenGL rendering pipeline overview. https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[79] Relative costs of OpenGL state changes. https://developer.nvidia.com/content/how-modern-opengl-can-radically-reduce-driver-overhead-0

[80] An introduction to shader derivative functions. http://www.aclockworkberry.com/shader-derivative-functions/

[81] NanoVG official github. https://github.com/memononen/nanovg

[82] Gimp official website. https://www.gimp.org/

[83] Audacity official website. https://www.audacityteam.org/

[84] Tiled Map Editor official website. https://www.mapeditor.org/

[85] Blender official website. https://www.blender.org/

[86] Wwise official website. https://www.audiokinetic.com/products/wwise/

[87] FMOD official website. https://www.fmod.com/

[88] Assimp official github. https://github.com/assimp/assimp

[89] Assimp porting for Java, official github. https://github.com/kotlin-graphics/assimp

[90] glTF official website. https://www.khronos.org/gltf/

[91] "Can it run Crysis?" meme. https://www.dictionary.com/e/memes/but-can-it-run-crysis/

8. Ringraziamenti

Ho il piacere di ringraziare coloro che hanno contribuito in maniera indiretta alla realizzazione di questo lavoro. La mia famiglia, che mi ha sempre sostenuto sia economicamente che moralmente durante tutto il mio percorso accademico ed alla quale devo molto, e la mia fidanzata, che non smette mai di credere in me e mi è stata sempre vicino durante le giornate trascorse lavorando al computer.

Grazie