Non Gravitar

Non Gravitar è un progetto realizzato nell'ambito di un corso di programmazione dell' Università di Bologna. Consiste nella realizzazione di un videogioco a grafica vettoriale 2D fortemente ispirato a Gravitar.

Si è deciso di dividere il progetto in due parti:

- **SGE**: una libreria con limitate funzionalità da game engine basate su SFML e Box2D
- **Non Gravitar**: la vera e propria logica di gioco, che ovviamente poggia su SGE

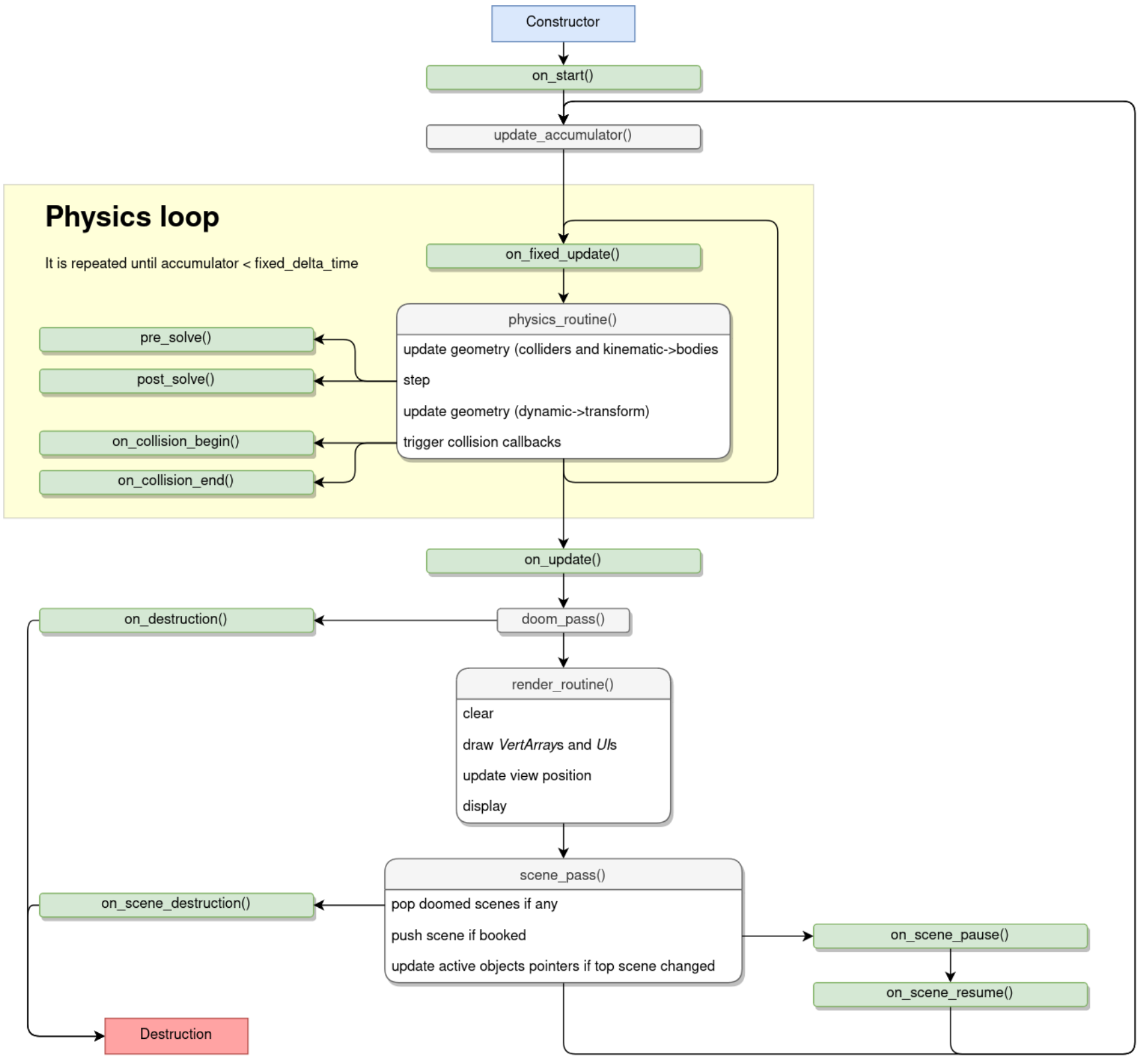
SGE

SGE (Simple Game Engine) è una libreria che offre alcune basilari funzionalità da game engine. Supporta un **ambiente grafico vettoriale** e una **simulazione fisica 2D**, una **organizzazione in scene** e una **gerarchia ad albero di oggetti** il cui comportamento è definito tramite l'aggiunta di **componenti** e **logiche** (si veda più avanti).

Per quanto questa libreria sia stata costruita ad-hoc per la realizzazione di Non Gravitar, si è cercato di mantenerla relativamente generale e il più riutilizzabile possibile, non tanto perchè debba essere effettivamente riutilizzata, quanto piuttosto come esercizio di buona pratica nella separazione tra engine e contenuto.

Game loop

Il cuore della libreria è il game loop, sequenza di procedure ripetuta fino alla terminazione dell'esecuzione dell'engine.



Qui sono illustrati i momenti salienti del game loop. In verde i callback che una determinata logica di gioco riceverà nell'arco della sua vita, in grigio le routine interne dell'engine.

Scene Stack

Il contenuto in SGE è innanzitutto organizzato in scene, che possono contenere un numero arbitrario di `GameObject`, organizzati in una gerarchia ad albero dove la scena funge da radice. Queste scene sono contenute in uno stack; solo la scena in cima sarà "attiva", ovvero verrà disegnata, simulata e riceverà i callback dal game-loop. Ciò permette per esempio il mantenimento dello stato di una scena di gioco mentre un menu di pausa prende le redini del game-loop, con un semplice pop della scena menù l'esecuzione della scena di gioco riprende.

Scenes

Una scena viene definita alla costruzione da un oggetto `Scene_ConstructionData` che ne specifica una serie di parametri. Di questi il più importante è l'**entry logic**, ovvero la logica che sarà attaccata al *GameObject* iniziale della scena. Questo è di fatto l'entry point di una scena, tipicamente l'utente utilizzerà `on_start()` per la definizione dei suoi contenuti (si veda in seguito).

Components

SGE si basa su una forma non particolarmente stretta di **Entity Component System**, in cui un *GameObject* non è altro che un recipiente in grado di contenere alcuni `Component` che ne specificano il comportamento. Un *GameObject* può contenere solo un *Component* di ogni tipo. Due di questi sono sempre presenti in un gameobject, dallo spawn alla distruzione:

- `Transform`: definisce le proprietà spaziali di un gameobject (*posizione*, *rotazione*, *scala*) e lo colloca in una gerarchia ad albero di gameobject che permette di applicare tali proprietà spaziali gerarchicamente
- `LogicHub`: contiene gli oggetti `Logic` e ne media l'accesso

Sono inoltre presenti alcuni **component "facoltativi"** che possono specificare ulteriormente alcuni aspetti del comportamento di un oggetto nell'ambiente di gioco:

- `VertArray`: permette di definire come l'oggetto viene disegnato in relazione allo spazio definito dalla gerarchia di *Transform*
- `UI`: permette di definire un UI tramite oggetti `UIContent` che a differenza dei *VertArray* sono disposti in *screen-space*
- `Rigidbody`: rende un gameobject parte della simulazione fisica e media l'applicazione di forze a tale oggetto. È necessario che vi sia almeno un *Collider* nello stesso *GameObject* o nei *GameObject* del suo sottoalbero
- `Collider`: definisce la geometria di un oggetto fisico (o di una sua parte) e alcune proprietà fisiche collegate ad essa quali *friction*, *restitution* e *density*

Logic

Questi sono gli oggetti gestiti dal componente *LogicHub* attraverso cui l'utente può specificare la propria logica. Ciò viene fatto tramite la creazione di classi che ereditano da `sge::Logic`, classe astratta che fornisce loro una serie di callback, caratterizzati per convenzione dal prefisso `on_`. Qui sta il cuore di questa architettura, poichè l'engine garantisce che questi callback verranno invocati in specifici momenti del game-loop per ogni Logic di ogni GameObject della scena attiva.

- `on_start()`: invocato quando la logica viene aggiunta a un *LogicHub* (particolarmente utile per l'inizializzazione)
- `on_destruction()`: invocato quando il GameObject a cui questa logica appartiene è distrutto
- `on_scene_destruction()`: invocato quando la scena a cui il *GameObject* di questa logica appartiene viene distrutta
- `on_update()`: invocato subito prima delle operazioni di render ad ogni iterazione del game-loop
- `on_fixed_update()`: invocato ad ogni iterazione del loop fisico interno al game-loop, subito prima dell'avanzamento della simulazione. Può essere invocato 0, 1 o più volte in un iterazione del game-loop
- `on_scene_pause()`: invocato quando la scena a cui questa logica appartiene viene "coperta" da una nuova scena nello scene-stack
- `on_scene_resume()`: invocato quando la scena a cui questa logica appartiene viene "scoperta" dalla distruzione della scena che la copriva nello scene-stack, momento a partire dal quale ricomincerà a ricevere i callback dal game-loop
- `on_collision_begin(CollisionInfo& info)`: invocato quando inizia una collisione che coinvolge il *Collider* attaccato al *GameObject* di questa logica
- `on_collision_end(CollisionInfo& info)`: invocato quando termina una collisione che coinvolge il *Collider* attaccato al *GameObject* di questa logica. Ciò può avvenire dopo multiple iterazioni del loop fisico o del game-loop

rispetto all'inizio della collisione

Entry point

Il constructor dell'engine utilizza un oggetto `Engine_ConstructionData`, che definisce una serie di impostazioni di esecuzione. Una volta costruito, l'engine deve anche essere inizializzato con un oggetto `Scene_ConstructionData`, che definirà la scena iniziale, ovvero l'unica presente nello scene stack alla prima iterazione del game loop.

```
Engine_ConstructionData engine_cd();                                    // Object that holds engine settings
// Set your engine options here
Engine engine(engine_cd);

auto entry_logic = new MyEntryLogic();                                  // Initialize the engine with an entry scene
Scene_ConstructionData scene_cd("My Entry Scene", entry_logic);
// Set your scene options here
engine.initialize(scene_cd);

while (engine.game_loop()) { }                                          // game_loop() returns false when the engine is done
return 0;
```

Note sull'implementazione

Debug

Quando SGE viene compilato con `CMAKE_BUILD_TYPE=="Debug"` (o direttamente con una direttiva al preprocessore `DEBUG`) sono disponibili alcune interfacce per il debug come una **hierarchy view** (`Ctrl+V`, navigabile con `Ctrl+<arrowkeys>`), **transform** (`Ctrl+T`), **collider** (`Ctrl+C`), **names** (`Ctrl+N`), oltre che un **contatore fps** in alto a destra.

Cache awareness

L'accesso a gameobjects e components è mediato da **handles**. Questo perchè questi oggetti sono organizzati in memoria in un vettore e dunque possono essere riallocati se tale vettore si riempie. Ciò causa un leggero overhead nell'accesso a tali oggetti, ma rende anche le istanze di un dato *Component* contigue in memoria e minimizza dunque le cache-miss durante lo scorrimento del vettore. Se si considera che un game-loop è costituito di una serie di "burst" concentrati su uno specifico tipo di component ciò può essere molto rilevante. Detto questo, SGE avrà colli di bottiglia ben più significativi e banali non essendo particolarmente ottimizzato in alcune sue parti. Anche questo è più che altro un esercizio di esplorazione di una tecnica interessante.

Smesh e spath files

SGE utilizza due semplicissimi file per salvare geometrie di triangoli (`.smesh`) e sequenze di punti (`.spath`), utilizzati rispettivamente da *VertArray* per la dichiarazione di mesh di triangoli e da *Collider* per la dichiarazione di *chain_colliders* o *polygon_colliders*. Teoricamente questi file possono essere creati e modificati manualmente, ma per facilitare tali operazioni insieme a SGE è fornito uno script python che permette l'esportazione da Blender di tali geometrie, oltre che un wrapper in bash che permette l'esecuzione dell'esportazione ricorsivamente su cartelle.

Non Gravitar

Flusso di gioco

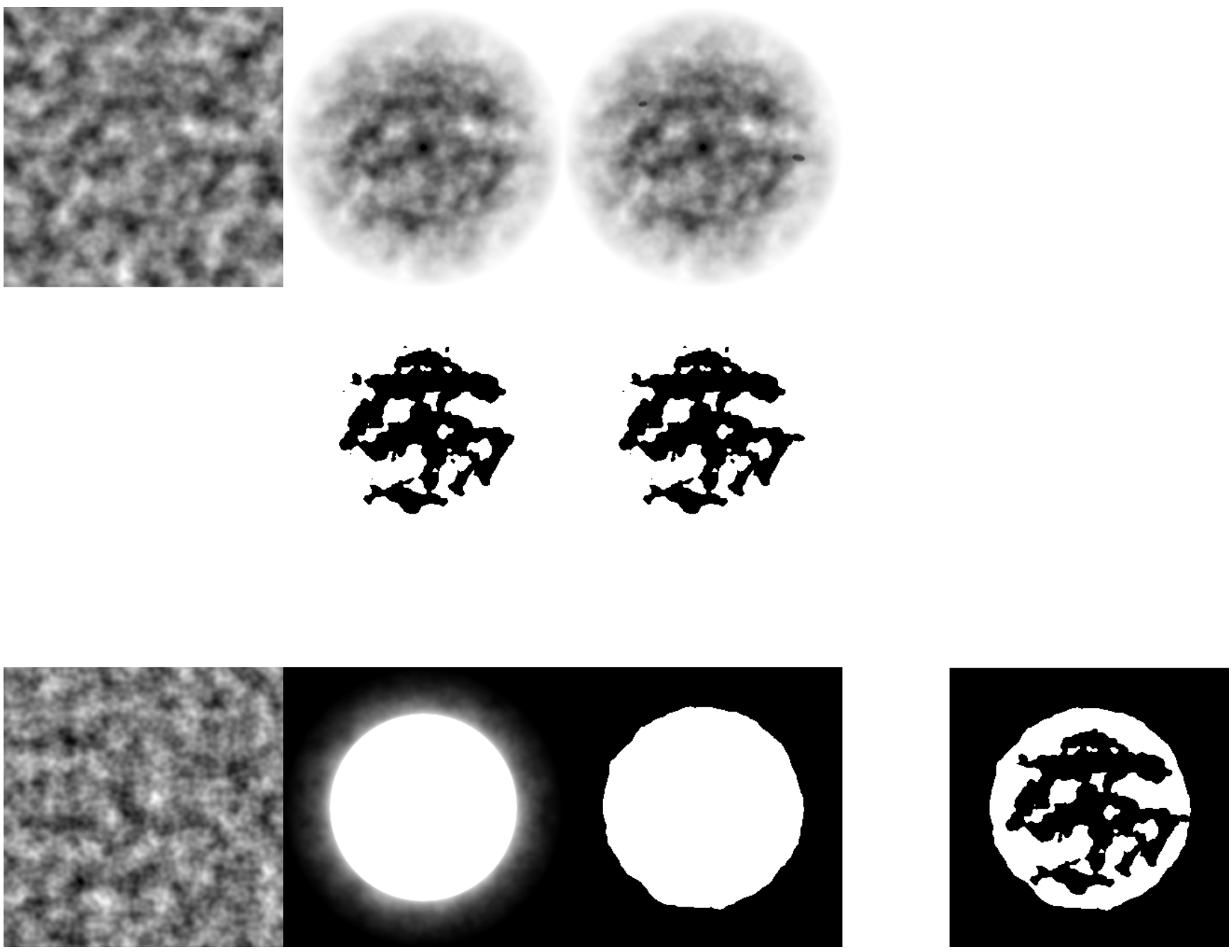
All'apertura il giocatore si trova in una scena `SolarSystem` con delle versioni scalate dei pianeti. Durante l'inizializzazione di tale scena vengono generati i dati salienti dei pianeti, contenuti in degli oggetti `PlanetoidPersistentData` gestiti dal `PlanetoidManager`. Al contatto con un pianeta il giocatore entra nella vera e propria scena di gioco, dove può esplorare il pianeta, distruggendo i nemici e raccogliendo le casse. Alla distruzione di tutti i nemici di un pianeta quest'ultimo è considerato completato. Quando tutti i pianeti di un sistema solare sono completati può essere generato un nuovo sistema solare. Ciò dietro le quinte consiste nel pop della scena SolarSystem completata e nel push di una nuova scena sullo stack, di fatto una sostituzione.

Planetoids

Si è scelto di rendere i pianeti cavi e concentrare gran parte del gameplay al loro interno, poichè in un ambiente senza gravità la navigazione di uno spazio confinato tende a essere più interessante del mantenimento di orbite intorno a oggetti in uno spazio vuoto.

I pianeti sono generati tramite la manipolazione di perlin noise bidimensionale e l'applicazione di un algoritmo marching squares alla matrice risultante. La manipolazione consiste nell'applicazione di gradienti per garantire per

esempio la circolarità del pianeta o il collegamento tra le regioni interne. L'algoritmo marching squares genera poi una lista di triangoli che viene passata a un *VertArray* e una lista di `sge::Path` che generano una serie di *Collider* (di tipo chain), tutti figli di un *RigidBody* di tipo *static*.



Qui è mostrato come in realtà la generazione del pianeta avvenga utilizzando due diverse matrici di perlin noise. La prima per le caverne interne e la seconda per la superficie.

Popolamento dei pianeti

In `src/GAME.hpp` è possibile trovare i principali parametri che definiscono il popolamento di un pianeta. Tali parametri sono complessati con un coefficiente di difficoltà che crescerà di pianeta in pianeta e di sistema in sistema. Un coefficiente di difficoltà maggiore per esempio porterà a una maggiore densità di nemici e una minore densità di casse.

Player

Il giocatore controlla una navicella la cui inerzia è fisicamente simulata, ma la cui rotazione è gestita tramite rotazioni geometriche, generando una sorta di ibrido tra un *kinematic-rigidbody* e un *dynamic-rigidbody*. Questo garantisce un maggiore controllo sulla direzione della spinta del motore senza rinunciare alla simulazione fisica degli altri aspetti del movimento del giocatore. Ciò causa comportamenti poco realistici quando il giocatore ruota a contatto con altri oggetti, ma in virtù delle meccaniche di gioco questo accade raramente.

La navicella è dotata di un cannone che spara proiettili generando un piccolo contraccolpo sulla navicella. Sparare richiede una certa quantità di **stamina**, che si rigenera ad una velocità costante ed è rappresentata da una barra in basso, insieme alla quantità di **fuel**.

Enemies

I nemici hanno una struttura a torretta con una base poggiata su un bordo del pianeta e una testa, che contiene la logica specifica per ogni nemico. Ve ne sono tre tipi:

- `BasicEnemy`: ha un solo cannone, spara a intervalli regolari in direzioni casuali.
- `MultiShotEnemy`: ha 3 cannoni, una dimensione maggiore e un comportamento identico a quello di BasicEnemy. È un buon esempio di riutilizzo di codice, poichè la logica della testa eredita da quella del *BasicEnemy* e compie

solo un paio di override per aggiungere i cannoni laterali all'inizializzazione e per garantire che sparino insieme a quello centrale.

- `SniperEnemy`: prima di sparare calcola approssimativamente la posizione in cui il giocatore sarà quando il proiettile raggiunge la sua area, proietta un laser (utilizzando un semplice raycasting) e spara un proiettile veloce.

I costruttori di queste logiche richiedono un puntatore a un oggetto che eredita da `EnemyPersistentData`, una classe dati che definisce alcune caratteristiche del nemico e mantiene alcune variabili di stato (per esempio `destroyed`, boolean che mantiene lo stato di distruzione del nemico per tutta l'esistenza della scena SolarSystem, infatti questi oggetti sono contenuti in logiche appartenenti a questa scena).

Crates

Nei pianeti sono disseminati contenitori che al contatto con la navicella del giocatore rigenerano una certa quantità di carburante e in certi casi applicano ulteriori effetti positivi:

- `FuelCrate`: rigenera una certa quantità di carburante
- `ExtraLifeCrate`: rigenera una piccola quantità di carburante e aggiunge una vita al giocatore
- `MaxFuelCrate`: aumenta il carburante massimo e lo rigenera completamente
- `MaxStaminaCrate`: rigenera una piccola quantità di carburante e aumenta la stamina massima

In maniera simile ai nemici, queste logiche richiedono puntatori a oggetti che ereditano da `CratePersistentData` e che risiedono nella scena SolarSystem. Definiscono caratteristiche come l'effetto applicato al giocatore e mantengono variabili di stato simili a quelle dei nemici.

Breakable objects

In Non Gravitar oggetti come i nemici, i proiettili e la navicella del giocatore sono resi distruttibili con l'aggiunta di alcune logiche (`BreakHandler`, `BreakTrigger` e `BreakGenerator`) che ne definiscono una causa di rottura e il comportamento quando questa rottura avviene. In generale la rottura di un oggetto comporta la generazione di aggregati di triangoli la cui geometria è recuperata da *VertArray* con primitive triangolari, questi aggregati possono essere ulteriormente rotti, ovvero separati ulteriormente a un nuovo contatto. Quando un aggregato contiene solo un triangolo è applicata una logica `Fading` che anima appunto il fading dei *VertArray* dell'oggetto (ricorsivo nella sua gerarchia) fino alla sua distruzione, quando l'*alpha* raggiunge 0.