Nongravitar: relazione

Davide Tinti, Paolo Marzolo, Matteo Feroli

1 Introduzione

Quel che segue è una breve spiegazione delle scelte progettuali seguite durante lo sviluppo della nostra versione di NonGravitar. Come è verificabile dal codice, abbiamo da subito deciso di affrontare il progetto come una possibilità per migliorare le nostre abilità e aggiungere alle nostre conoscenze: per questo motivo, sia attraverso la storia dei commit su github, sia nel codice di adesso, non siamo stati estremamente stretti sui metodi di implementazione, e la differenza implementativa dimostra la nostra crescita come programmatori. Ciononostante, sia per pulizia del codice, sia per chiarezza implementativa, vi sono alcune linee guida che si sono formate attraverso lo sviluppo: una di queste è la relativa indipendenza delle classi, che rispettano, generalmente, uno schema che prevede un costruttore importante e una funzione di gestione, solitamente chiamata `handle()`, che riassume al suo interno tutte le operazioni necessarie per il corretto funzionamento. Questo ci ha permesso di affrontare temi molto diversi tra loro contemporaneamente e, per quanto possibile, suddividere il lavoro in modo da minimizzare la difficoltà dell'unire le varie componenti in un modo unico. Non ci siamo infatti ristretti a uno schema preciso, poiché questo ci sembrava, in primo luogo, troppo stringente per un lavoro molto creativo, e in secondo luogo perché, come primo progetto lungo, non avevamo sufficienti basi per affrontare uno sviluppo costretto in uno schema.

Le risorse del gioco hanno una classe `Resources` a parte, che utilizza `SourceLoader` per caricare texture e suoni e renderli disponibili alle varie classi che li richiedono. Inoltre, nella classe `Resources`, è presente il puntatore alla window corrente, che permette di disegnare in varie fasi del programmma piuttosto che dover costringere la fase di disegno in un momento separato dell'esecuzione. Questo, in linea con l'obiettivo di indipendenza che abbiamo esposto prima, permette, tramite un solo puntatore alla classe risorse, un accesso immediato alla funzione di disegno.

Precisiamo inoltre che, dopo una breve discussione sulla libreria grafica, saremo brevi nell'esposizione del codice: le operazioni non sono matematicamente complesse e, anche a scopo di comunicazione interna, il codice è piuttosto comprensibile dopo la breve introduzione del framework ad alto livello di questa relazione.

1.1 SFML

Prima di parlare della scelta di SFML, introduciamo la nostra motivazione dietro la scelta dell'uso di una libreria grafica. Come abbiamo menzionato nell'introduzione, l'obiettivo di questo progetto non è stato per noi lo sviluppo di un gioco rivoluzionario, o di un prodotto perfetto; piuttosto, abbiamo cercato di trarre il più possibile da questo progetto, facendo magari scelte più complesse o creative che però ci forniscono conoscenze aggiuntive. Questa filosofia ci ha guidato nella direzione di una libreria grafica: per quanto il disegno sul terminale sia una nicchia curiosa, riteniamo che la quantità e l'importanza di quel che abbiamo imparato scegliendo questa strada superi enormemente l'altra possibilità, e poiché non vi erano obblighi dal punto di vista della consegna abbiamo ritenuto di aver fatto una scelta sensata.

Una volta deciso per la libreria grafica, ci siamo trovati di fronte alla scelta di quale. OpenGL, infatti, è soltanto un insieme di specifiche di operazioni (spesso realizzate in hardware) che permettono alle applicazioni di utilizzare hardware acceleration in particolare sulla scheda grafica. Poiché vi sono molte versioni di OpenGL, spesso le primitive non sono recuperabili a tempo di compilazione, e devono essere richieste a run-time e salvare puntatori a funzione per l'uso successivo:

```
// define the function's prototype
typedef void (*GL_GENBUFFERS) (GLsizei, GLuint*);
// find the function and assign it to a function pointer
GL_GENBUFFERS glGenBuffers  = (GL_GENBUFFERS)wglGetProcAddress("glGenBuffers");
// function can now be called as normal
unsigned int buffer;
glGenBuffers(1, &buffer);
```

Se questo non bastasse, le implementazioni sono specifiche per la piattaforma: questo percorso era assolutamente insensato e impercorribile.

Abbiamo quindi deciso di usare una libreria di livello più alto: alcune scelte possibili sono `GLFW, freeglut` per quanto riguarda quasi esclusivamente l'apertura di un context OpenGL e l'interazione con le periferiche, e `Allegro, SDL, SFML` per capacità multimediali più avanzate. Poiché Allegro e SDL hanno interfacce in C, mentre SFML usa C++, la scelta è stata ovvia: inoltre, SFML contiene anche un modulo di networking, ed è accompagnata da un breve libro di sviluppo di giochi. L'astrazione di SFML ha reso possibile lo sviluppo di NonGravitar e ci ha fornito gli strumenti per affrontare problemi molto diversi tra loro senza bisogno di lunghe sessioni di studio prima della prima riga di codice.

2 HUD

La prima componente che affronteremo è l'HUD: poiché tutti i componenti del gruppo avevano giocato a videogiochi prima di sviluppare, ci siamo trovati di fronte a problematiche che avevamo già affrontato da un lato diverso; per esempio, il tradeoff tra conoscenza, modificabilità e immersione. L'HUD è il primo e più grande esempio: le barre nella parte alta dello schermo, chiaramente riconoscibili e accompagnate da un'etichetta, permettono più flessibilità che un numero discreto di, per esempio, icone di navetta a indicare le vite. Allo stesso tempo, quanto un bunker prende danno, abbiamo deciso di non includere una piccola barra della vita: poiché i bunker muoiono con pochi colpi, questa avrebbe soltanto preso spazio. Allo stesso modo, non abbiamo incluso valori di velocità o usato valori numerici per carburante e vita. Dal punto di vista intuitivo, invece, torneremo sulle particolarità della navetta e del Boss nelle rispettive sezioni.

2.1 Bars

Le barre della vita e del carburante sono realizzate con 4 Sprite. Ogni barra è composta da 2 Sprite sovrapposte, la prima chiamata "back" rappresenta la barra vuota ed è posizionata in secondo piano, mentre la seconda chiamata "active" rappresenta la barra piena ed è posizionata sopra alla "back". Questo sistema di visualizzazione rende la modifica delle barre estremamente semplice. Le Sprite "back" restano inalterate e l'unico compito è ridotto alla modifica della porzione visualizzata di Sprite "active" ridimensionando le dimensioni del suo TextureRect in `HUD::handle()`. Abbiamo scelto di usare status bar anzichè valori interi per rappresentare i valori di carburante e vita per lasciare più flessibilità all'implementazione di features secondarie, lasciando una maggior precisione nella modifica di tali valori. Questa possibilità viene, per esempio, sfruttata con l'implementazione della difficoltà di gioco, che altera la velocità di perdita del carburante e la quantità di danni subiti rendendo il gioco più facile o difficile.

2.2 Highscore

La nostra versione di nongravitar include anche un sistema di highscores, per tenere traccia dei migliori 3 punteggi realizzati. Tale sistema viene implementato nella classe Highscore la quale basa il suo funzionamento su tre processi principali:

- la creazione, aggiornamento e visualizzazione dei punteggi all'interno dello Stage Highscore
- l'inserimento di un nuovo miglior punteggio
- l'aggiornamento del file `./resources/highscore` per salvare i punteggi tra le diverse sessioni

La visualizzazione dei punteggi utilizza una `std::list<ScoreRow>`, lista formata da ScoreRow, una struttura contenente 3 campi di testo (rank, name e score) che rappresentano le informazioni del punteggio mostrate. La lista viene riempita con 3 elementi ScoreRow, posizionati a distanze predefinite e inizializzati leggendo le righe del file d'appoggio highscore, il quale segue una formattazione particolare.

Al termine di ogni partita è data la possibilità al giocatore di salvare il proprio punteggio inserendo un nome personalizzato da tastiera grazie alla funzione `Highscore::insert`. Durante la digitazione viene catturato l'input del tasto premuto e trasformato nel rispettivo unicode, il carattere corrispondente viene poi accodato alla stringa `player_name`. Successivamente il testo mostrato a schermo viene aggiornato con la nuova stringa per avere un riscontro visivo di ciò che è stato digitato. Alla pressione del tasto Invio l'inserimento del nome termina ed inizia la verifica del nuovo punteggio con la funzione `Highscore::checkNewScore`. Quest'ultima confronta il contenuto degli elementi di `std::list<ScoreRow>` con l'ultimo punteggio ottenuto e in caso di nuovo miglior punteggio viene rimosso l'elemento in coda alla lista per fare spazio al nuovo. La `ScoreRow` rappresentante il punteggio appena realizzato viene poi inserita nella lista rispettandone l'ordinamento.

Ogniqualvolta viene realizzato un nuovo miglior punteggio è quindi necessario aggiornare la lista di ScoreRow, con la procedura appena descritta, e riportare il medesimo cambiamento anche nel file highscore. Quest'ultimo passaggio è realizzato accedendo al file tramite `fstream::open` e sovrascrivendo il suo contenuto seguendo quello contenuto nella nuova lista aggiornata.

2.3 Animation

Le animazioni sono realizzate grazie all'utilizzo di spritesheet, ovvero texture che rappresentano non una singola immagine bensì una serie di frame, che se riprodotti in sequenza, ovvero variando la porzione di Texture visualizzata sulla Sprite, simulano movimento. Quando un oggetto Animation viene istanziato, tra i parametri richiesti compaiono rows e columns, ovvero la quantità di righe e colonne di cui è composta la tabella di frame rappresentata sullo spritesheet (che ricordiamo essere un oggetto Texture). Grazie a questi due indici, Animation scorre la tabella aggiornando la propria Sprite. A livello superiore le animazioni sono implementate tramite una `std::list<Animation*>` contenuta in Resources la quale include tutte le animazioni in corso. Durante l'aggiunta di un elemento alla lista, una nuova Animation viene creata iniziandone l'esecuzione. Ad ogni ciclo del main, tramite la funzione `Resources::handleAnimation` ogni elemento della lista passa al frame successivo fino al termine di quest'ultimo, momento in cui viene eliminato.

2.4 nongravitar vs Game vs Universe vs GPlanet

Consideriamo ora le principali sezioni di gestione presenti nel progetto: la prima è contenuta nel file `nongravitar.cpp`, ed è il main del progetto. Qui è contenuto il *game loop* principale, che semplicemente si attiva al più $\frac{1}{timePerFrame}$ volte al secondo. Da qui, `nongravitar` si occuperà esclusivamente di chiamare la gestione di `Game` fino alla chiusura della finestra. In questo modo, l'intero gioco, compresa la schermata iniziale, sono limitati a un framerate specifico. A questo punto, `Game.handle()` disegnerà la schermata attiva al momento e chiamerà la gestione dell'`HUD` e di `Universe`.

`Universe` è una classe multifunzione, che abbiamo mantenuto nel modo in cui è emersa dallo sviluppo: svolge la funzione di entità contenente i pianeti, formata da più schermate, ma copre anche il ruolo di gestione generica della singola partita, come vedremo nel prossimo paragrafo.

3 Gestione all'esterno del pianeta

La divisione delle gestioni tra interno ed esterno del pianeta è emersa come conseguenza dello sviluppo incrementale, ma non è stata sostituita da una gestione unica perché (a noi) è subito apparsa come la

soluzione migliore e più chiara. All'esterno, `Universe.handle()`, chiama la gestione della navetta, verifica la posizione della navetta e, quando necessario, attiva l'entrata o il completamento di un pianeta. Inoltre, chiama la gestione di `GPlanet`, passando per il `Pianeta` di riferimento.

3.1 Gestione schermate

Le schermate sono il secondo oggetto istanziato dal costruttore di `Universe`, dopo la navetta. Per poter tenere traccia di tutte le schermate, esse fanno parte di una lista monodirezionale, di cui sono mantenuti puntatori al primo e ultimo elemento come variabili di classe. Le schermate sono identificate da una coppia (x, y), che rappresentano la posizione della schermata su un piano astratto. Teniamo a specificare che l'implementazione e l'uso della lista è stato mantenuto come formulato inizialmente: benché questa implementazione sia sicuramente non efficiente, è stata preservata per mostrare lo sviluppo del gruppo; infatti, questa sezione è una delle prime che sono state sviluppate. Nel resto del codice, useremo invece soprattutto `std::list<>`

3.2 Generazione pianeti

I pianeti di ogni schermata fanno parte di una classe apposita (appunto, `Pianeti`), e sono contenuti in una `std::list<SinglePlanet>`: ciascun elemento di essa rappresenta sia l'oggetto disegnato, sia l'entità logica; questo significa che ogni istanza di `SinglePlanet` contiene una `Sprite`, alcuni valori di posizione, un `GPlanet` (come menzionato sopra, esso svolge le funzioni di gestione all'interno del pianeta) e il numero di schermate del pianeta. Il numero di schermate ha in realtà molteplici funzioni: identifica anche il colore e il punteggio associato al pianeta. La sprite di ogni pianeta è posizionata in una posizione casuale all'interno di una griglia predeterminata per evitare sovrapposizioni.

3.3 Movimento navetta

Poiché questo è un argomento centrale del progetto e le scelte possibili sono molteplici, abbiamo deciso di dedicare una sezione a parte. La nave che abbiamo implementato si muove utilizzando vettori direzionali: `(dx, dy)` sono una coppia di variabili nel codice che rappresentano la proiezione del vettore direzione sulle due assi. La nave è libera di ruotare su se stessa, e farlo non consuma carburante. Il consumo di carburante è invece legato all'accelerazione. Come si può vedere dal codice, abbiamo deciso di non avere una sezione del codice interamente dedicata al processing degli input, ma piuttosto di affrontarli quando è stato necessario: questa decisione è derivata dalla scarsità di momenti in cui gli input sono usati (movimento navetta e sparo). Quando lo spazio è premuto e la navetta accelera, l'opacità viene resettata al massimo, e cala progressivamente finché lo spazio non è premuto di nuovo. A ogni ciclo di clock vengono quindi ricalcolati dx e dy e rivalutata la posizione della navetta.

4 Gestione all'interno del pianeta

Poiché le collisioni (tranne quelle per l'entrata in un pianeta) sono possibili solo all'interno di un pianeta, `checkCollision()` è una funzione di `GPlanet`, chiamata all'inizio della gestione. Oltre a questo, la gestione interna del pianeta si occupa della collisione con il terreno, dei bunker, del boss e del raggio traente. Poiché il buco per accedere al boss è semplicemente disegnato eliminando una delle sezioni di superficie e disegnando al suo posto l'entrata di una caverna, non è necessaria nessuna verifica particolare per l'entrata al boss: l'unico momento in cui la navetta possa uscire al di sotto della schermata è dopo che la superficie è stata modificata; per questo motivo, l'entrata nel boss è gestita a livello superiore, in `Universe`.

4.1 Schermate

La gestione delle schermate in `GPlanet` differisce sostanzialmente da quella in `Universe`: la lista è circolare, e le direzioni di cui tenere conto sono solo due (destra e sinistra), e tutte le schermate sono istanziate nello stesso momento. Poiché l'istanziazione è (relativamente) costosa, è ritardata all'entrata nel pianeta, anziché mantenerla all'interno del costruttore (chiamato all'istanziazione del pianeta corrispondente). Vi è poi una schermata separata dalle altre, la schermata del Boss. Ha uno sfondo diverso, e un comportamento diverso: uscire ai lati è impossibile, quindi non è nemmeno contemplato.

4.2 Terreno

Passiamo ora alla generazione del terreno. Il terreno è formato da una serie di `ConvexShape`, un oggetto grafico fornito da `SFML`, *wrappato* in un `soil`; queste sezioni di terreno sono contenute in un'unica lista monodirezionale, e posizionate adiacenti. La funzione `spriteSetup()`, una caratteristica comune a varie classi del progetto, inizializza la sprite del terreno, utilizzando sempre lo stesso puntatore a texture, memorizzato in una variabile di classe. Per motivi di giocabilità, quando la nave colpisce il terreno anziché morire all'istante (un po' noioso e inutilmente punitivo, a nostra opinione) rimbalza verso l'alto prendendo danno. Infine, le funzioni `prepareForHole()` e `isBoss()`, dal nome piuttosto autoesplicativo, servono la prima a creare il buco e posizionare la sprite della "cava" nella posizione corretta (infatti, benché la grandezza dei `soil` sia costante in ciascun pianeta, varia tra pianeta e pianeta), e la seconda a verificare se l'istanza di `Terreno` corrente è in realtà una classe posticcia, e la schermata che la contiene è quindi quella del boss. La funzione più usata è invece `getTerrainY(double x)`: data una `x` restituisce l'altezza del terreno corrispondente. Questo causa vari scorrimenti della lista delle `soil`: a questo proposito teniamo a far notare che la lunghezza della lista ha un limite superiore (abbiamo scelto 23) e che, sebbene sia possibile scrivere una versione molto efficiente basata

su vettori istanziati insieme alla classe, questo avrebbe reso la funzione inutilmente complessa (non ha un impatto osservabile sulla performance).

4.3 Bunker

Consideriamo brevemente i nemici da colpire. La classe `Bunker` è stata rivisitata varie volte, in parte anche perché serve da classe base per i `BossBunker`: grazie a questo, usa finalmente una `std::list<bunkerlist>` per la lista dei bunker attivi. All'interno vi sono tutti i bunker attivi in una schermata, che possono essere di quattro tipi sulla base di due varianti: possono seguire o non seguire il giocatore, e usare uno sparo singolo o triplo. Alla distruzione, i bunker esplodono, grazie a una animazione gestita dalla classe `Animation`. Le `checkCollision()` *overrrided* permettono di verificare la collisione di un corpo con un bunker o direttamente tutti i proiettili all'interno di una classe `Bunker`. Vi è poi una terza funzione di verifica collisione, cioè la verifica che un corpo collida con uno dei proiettili dei bunker: questa scorre tutte le liste dei proiettili, elimina tutti i proiettili in collisione e restituisce il valore del danno più alto (poiché la nave ha un breve periodo di invulnerabilità dopo ogni colpo, se due proiettili sono rilevati nello stesso istante ci sembrava controintuitivo danneggiare due volte la nave).

4.4 Proiettili

L'implementazione di `Bullets`, ancora successiva, rappresenta il nostro utilizzo più completo del paradigma ad oggetti. Le due classi derivate `SingleStraightBullets` e `TripleBullets` ereditano dalla classe base *astratta* `Bullets`. Questo ci permette di utilizzare puntatori di tipo statico `Bullets *` che a tempo dinamico indirizzano oggetti di uno dei due tipi derivati. In questo modo, diminuisce molto la complessità di gestione dei possibili proiettili: le classi derivate sovrascrivono i metodi della classe base che sono poi chiamati normalmente. Addirittura, la classe `TripleBullets` non implementa in realtà nulla, ma semplicemente "gestisce" (per quella poca gestione necessaria) tre classi `SingleStraightBullets` al suo interno. I proiettili hanno un tempo di *esistenza* massimo, ma vengono eliminati anche quando escono dalla schermata, dalla funzione `cleanup()`. In ultimo, menzioniamo che la gestione del movimento dei proiettili è molto più semplice, poiché non dipende da user input e si muove in linea retta; infatti, la funzione di movimento dei proiettili è molto più breve di quella della nave.

4.5 Fuel

La rigenerazione di carburante durante il gioco è resa possibile utilizzando il raggio traente per attrarre piccole taniche disposte randomicamente sulla superficie del pianeta. La classe che le gestisce, `Fuels`, è inserita in `planet_screen` ed è implementata con l'utilizzo di una `std::list<fuel>`. La struttura `fuel` rappresenta la singola tanica di carburante e comprende i campi:

- coordinate di posizione
- `power`, un intero rappresentante la quantità di carburante rigenerato qualora venisse utilizzato
- una Sprite per visualizzare l'entità a schermo, colorata in modi differenti in base al valore di power.

Al momento della creazione della classe, la lista vuota viene riempita con fuel, generandone le caratteristiche (posizione, esistenza e potenza) casualmente. La posizione è scelta partendo da una griglia predeterminata (per evitare sovrapposizioni con i Bunker) mentre esistenza e potenza sono scelte in scala percentuale (mediante percentuali scelte per mantenere un corretto equilibrio della difficoltà del gioco), rispettivamente 50% per l'esistenza e 33% per la possibilità di avere una potenza superiore allo standard.

Gli oggetti creati vengono poi gestiti all'interno di `GPlanet::tractorRay`, funzione che verifica se il raggio traente è in uso e se interseca una tanica (iterando sulla lista di fuel). In caso positivo le coordinate della tanica (ricordiamo essere la Sprite contenuta all'interno della struttura fuel, contenuta a sua volta nella lista della classe Fuels) vengono aggiornate avvicinandone la posizione a quella della Sprite della nave. In caso negativo, se la coordinata y della tanica è superiore a quella del terreno sotto di essa (ed è quindi sospesa), viene diminuita progressivamente riportandola al livello del terreno.

4.6 Boss

Abbiamo deciso di includere un boss per dare spazio alla creatività: grazie alle classi già implementate finora, dal punto di visto gestionale non poneva problemi complessi, ma la rotazione aumentava molto l'aspetto di "novità" che il boss presentava. La nave compare al centro dello schermo. Il movimento è limitato dal boss, e colpirlo con la navetta significa rimbalzare verso il centro e prendere danno. Inizialmente, il boss non può essere colpito: i colpi vengono "assorbiti" da una barriera blu; è possibile però colpire le torrette su di esso. I proiettili non hanno nulla di speciale, ma poiché la loro fonte ruota evitarli è più difficile che sulla superficie. Il carburante non cala durante la battaglia: ci è sembrato inutilmente complicato, sia da un punto di vista implementativo (richiedeva scrivere una nuova funzione di generazione, con funzionamento radicalmente diverso dalla precedente) sia da un punto di vista di giocabilità (anche per noi che abbiamo scritto il gioco sconfiggere il boss non è semplice). Quando tutte le torrette sono state distrutte, si può sparare al boss, che dopo un certo numero di colpi esplode: la navetta viene poi trasportata fuori dal pianeta, e anche il pianeta esplode.

4.6.1 BossBunker

Grazie all'ereditarietà, `BossBunker` è in realtà una classe molto semplice, che presenta due principali funzioni: `updatePosition()` e `updateRotation`. A titolo informativo, le torrette del boss sono raggiungibili sia dal puntatore a `enemies` nella schermata del boss (cioè visibile da `GPlanet`), ma sono anche un campo apposito del boss: in questo modo la gestione può essere svolta dal boss, ma la verifica delle collisioni è svolta sempre dalla stessa funzione `checkCollision()` in `GPlanet`.