Non - Gravitar

Ide: Visual Studio 2017
Gruppo: Arnone William, Cucci Michele, Pellegrino Alessio

IL PROGETTO:
Il progetto Non-Gravitar utilizza una libreria grafica molto basilare chiamata “oldConsoleGameEngine” che permette di disegnare nella console di Windows.
Per fare ciò utilizza un carattere Unicode che occupa un singolo pixel e lo colora di un colore scelto, poi sfrutta la console come se fosse un piano cartesiano per identificare ogni pixel attraverso un sistema di assi denominati X ed Y.

Il progetto è così strutturato:

- Origine.cpp è il file che contiene la funzione **Main**: il suo unico compito è di far partire la console con le specifiche richieste.

- Gravitar.cpp contiene la classe principale del progetto che deriva direttamente da quella messa a disposizione dalla libreria grafica e per questo è l’unica con la possibilità di disegnare a schermo, quindi si occupa di aggiornare e gli oggetti e visualizzarli a schermo. Inoltre, essendo oldConsoleGameEngine una classe virtuale è stato necessario sovrascrivere le funzioni “OnUserCreate” e “OnUserUpdate”,che partono rispettivamente all’avvio e ad ogni ciclo del programma. Di base quindi questa classe gestisce il flusso di gioco e anche la creazione di nuovi oggetti e la distruzione degli oggetti non utilizzati.

- Pianeta.cpp contiene la definizione della classe **Pianeta**. Il costruttore genera un nuovo pianeta il quale è composto da un insieme di “Aree", ossia le zone visitabili una volta che si entra nel pianeta. La funzione “isEnded” invece controlla quando tutte le aree sono prive di nemici e restituisce “*true*” nel caso la condizione si avveri.

- Area.cpp contiene la definizione della classe **Area,** ossia il singolo ambiente del pianeta visitabile. Oltre al costruttore per la creazione di un’area, contiene varie funzioni tra cui: “CreaPuntoTerreno” che, dati certi parametri, restituisce un punto del terreno; “FindY” che restituisce l’altezza del terreno corrispondente ad una “*X*” data: essa viene usata per posizionare le torrette, i carburanti e per controllare

le collisioni con il terreno; ”CreaTorretta”, che restituisce una torretta da aggiungere al vettore di torrette, così come “CreaCarburante” che restituisce un carburante da aggiungere al vettore di carburanti; infine “CreaOggetti” si occupa di calcolare quante torrette e carburanti mettere nell’area.

- Carburante.cpp contiene la definizione della classe **Carburante** che ha solamente il costruttore con le coordinate e la quantità di carburante che il giocatore ottiene quando assorbe un carburante.

- CarburantePro.cpp contiene la definizione della classe **CarburantePro** che deriva dalla classe **Carburante** e modifica solo la quantità di carburante che il giocatore può ottenere.

- Torretta.cpp contiene la definizione della classe **Torretta** che ha il costruttore e la funzione Update, che controlla quando far sparare la torretta utilizzando un timer che decrementa e reimposta al valore iniziale dopo ogni sparo.

- TorrettaPro.cpp contiene la definizione di **TorrettaPro** che deriva da **Torretta**, con la sola differenza che nella funzione Update vengono sparati 3 proiettili e non 2.

- ObjGame.cpp contiene la definizione della classe **ObjGame**, che ha solamente il suo costruttore.

ObjGame è una delle più importanti classi del progetto: infatti ogni punto che si va a definire nel gioco è un objGame (es.: il centro di una torretta, il centro della navicella, i punti del terreno, i proiettili, etc.) e il parametro *Size* di objGame permette di conoscere ed impostare la grandezza di dimensioni dell’oggetto desiderato. Usando il parametro *Size* come raggio e le variabili (X, Y) come coordinate del centro si possono calcolare cerchi per poter creare delle hit-box necessarie nel gioco, anche se un po’ approssimate.
In questo modo ogni collisione può essere trattata come un’ intersezione tra due cerchi. (es.: un proiettile è un objGame di *Size* 0 mentre l’astronave è un objGame di *Size* 3, quindi quando il proiettile definito solo da un punto come cerchio si interseca con il cerchio di raggio 3 pixel della navicella allora c’è collisione).
In questo modo sono state trattate anche le collisioni con il terreno: grazie alla funzione “FindY” si può ottenere la proiezione del centro della navicella sul terreno e quando il punto proiettato sul terreno si interseca con la navicella allora c’è collisione e contatto.

Grazie a questo metodo ogni collisione può essere riportata ad un solo caso e ciò semplifica molto la stesura di codice.

Tutto il gioco è sviluppato in modo da risultare scalabile. Grazie alle funzioni di libreria “screenwidth” e “screenheight” si ottiene infatti la grandezza della schermata che si può ridimensionare.
In questo modo è possibile trasportare l’intero gioco sia su risoluzioni più grandi che più piccole. Per motivi tecnici le schermate di avvio e di gameover del gioco non sono scalabili.

Nel codice si fa spesso uso della variabile *fElapsedTime*: questa variabile indica il tempo trascorso dall’ultimo frame, quindi prendendola come punto di riferimento è possibile mantenere il gioco a velocità costante nonostante il numero di processi attivi o la potenza del processore della macchina utilizzata.

Per lo sviluppo del progetto sono state necessarie alcune funzioni matematiche qui elencate:

- ➢ funzione della retta passante per due punti:
  - usata per la funzione FindY.
- ➢ funzione della intersezione di due circonferenze:
  - usata per la funzione Collisioni.
- ➢ traslazione di un punto 2d tramite matrice:
  - usata per disegnare torrette e navicella.
- ➢ calcolo dell’angolo di un segmento rispetto all’asse delle ascisse:
  - usata per calcolare l’angolo di inclinazione delle torrette.
- ➢ trigonometria di base:
  - usata per calcolare lo spostamento sugli assi X ed Y per gli oggetti in movimento.