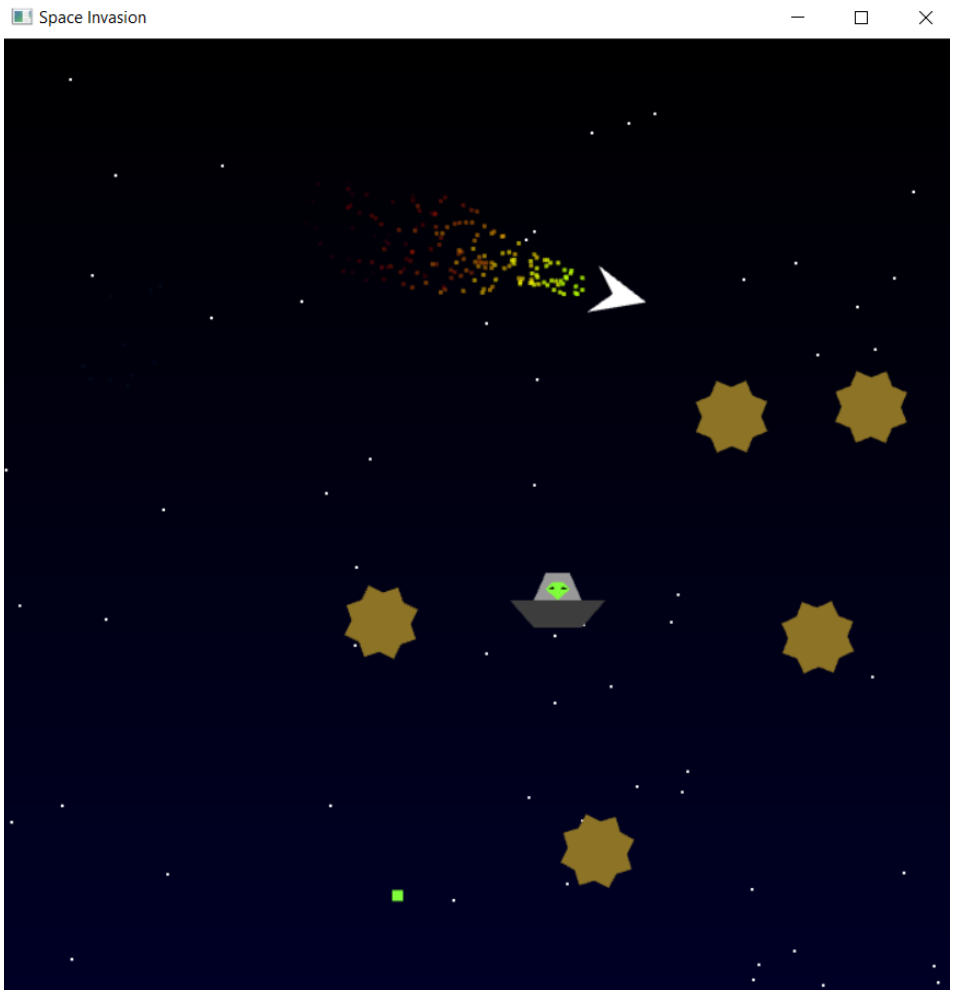
REPORT PER LAB_02

Matteo giri

Introduzione e comandi

Questo è il report per l'esercizio di laboratorio **LAB_02**. Ho chiamato la demo di animazione digitale 2D interattiva da realizzare per questo progetto **Space Invasion**, e di seguito c'è un'immagine introduttiva di esempio:

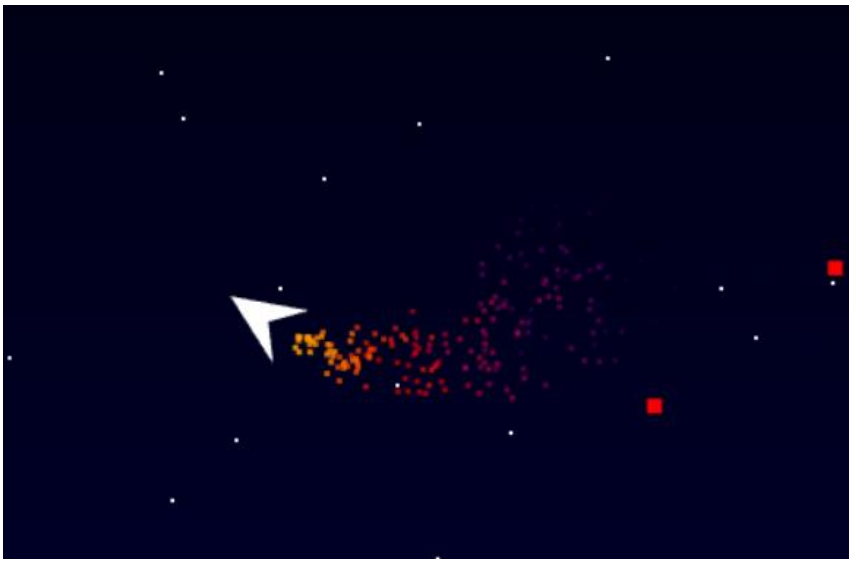


La demo riguarda un gioco ambientato nello spazio nella quale bisogna guidare una navicella a propulsore nell'ambiente e sparare ai nemici che compariranno nello spazio. I nemici da distruggere sono di due tipi: **asteroidi**, che richiedono un solo colpo per essere distrutti; **ufo**, che richiedono 5 colpi per essere distrutti e sparano proiettili verdi che possono colpire la navicella. Se la navicella viene colpita da un proiettile di un ufo o se si scontra con un asteroide o ufo, esplode e il gioco termina. Il punteggio e la condizione di game over vengono indicati nella console. Possono essere presenti a schermo fino a 5 asteroidi alla volta e un ufo (parametri modificabili da codice, insieme a velocità di sparo dell'ufo e vita dell'ufo, ed altri…).

Per girare la navicella a destra e sinistra si usano i tasti *a* e *d*, mentre per farla muovere si accende il propulsore con *w*. Si possono sparare proiettili premendo la *barra spaziatrice*. Se la navicella colpisce il bordo dello schermo viene "rimbalzata" nella posizione opposta.

Dettagli realizzativi

Navicella e propulsore



Tutti gli elementi della demo hanno dei parametri a loro associati, a partire dalla **navicella**:

```
//Parametri per la navicella
int vertices_Navicella = 6;
Pointxy* Navicella = new Pointxy[vertices_Navicella];
double velocità_max = 10;
double velocità = 0;
int delta_rot = 4;  //velocità di rotazione
int navicella_width = 35;
int navicella_height = 40;
float navicella_posx = float(width) / 2;
float navicella_posy = float(height) / 2;
float navicella_angle = 0; //angolo in cui è disegnata la navicella
float navicella_moving_angle = 0; //angolo di movimento della navicella (dive
bool navicella_alive = true; //true se la navicella non è esplosa (game over)
```

Vertices_Navicella e Navicella sono parametri per il disegno della navicella, mentre gli altri sono parametri per la "logica". Navicella_angle e navicella_moving_angle sono due parametri diversi che descrivono il primo l'angolo della navicella, e il secondo l'angolo di movimento. Sono separati perché la navicella può girarsi e sparare in diverse direzioni mentre si muove nello stesso angolo di direzione (quando non sto premendo il propulsore). Il parametro navicella_alive indica se la navicella è ancora in vita o no (se non è in vita non sarà più disegnata a schermo, non potrà essere mossa e non si potranno più sparare proiettili).

La geometria della navicella è realizzata nella funzione *disegna_navicella()*, ed è costituita da due semplici triangoli colorati di bianco. Il disegno su schermo è gestito nella *drawscene*, dove la navicella viene disegnata nella giusta posizione e rotazione modificando la matrice di modello con i giusti parametri.

Tutta la logica è nella funzione *update_navicella()*, che viene chiamata ciclicamente ogni 24 millisecondi. La posizione della navicella viene aggiornata con una funzione che considera la velocità e l'angolo di movimento della navicella:

```
//la posizione della navicella è aggiornata in base alla velocità e all'angolo di movimento
navicella_posx += velocità * sin(radians(navicella_moving_angle));
navicella_posy += velocità * cos(radians(navicella_moving_angle));
```

Se stiamo premendo *w* (che possiamo sapere grazie a una variabile booleana che è true se il tasto è premuto e non rilasciato) allora aumentiamo la velocita della navicella, mentre se non lo stiamo premendo la diminuiamo (fino a un minimo e un massimo). Se stiamo premendo *w,* inoltre, l'angolo di movimento diventa l'angolo della navicella. Se stiamo premendo *a* o *d* bisogna modificare l'angolo della navicella (non di movimento) di un valore *delta_rot.* Se la navicella sbatte su uno dei bordi viene rimbalzata nella direzione opposta in maniera molto fluida andando a cambiare l'angolo in base all'angolo con la quale ha sbattuto. La velocità viene diminuita di una percentuale quando questo accade. Grazie a tutte queste istruzioni si riesce ad ottenere un movimento che rappresenta in maniera "organica" il movimento di una navicella nello spazio.

All'interno di questa funzione vengono anche gestite le collisioni con gli asteroidi e gli ufo: per ogni ufo e asteroide si va a controllare se il box della navicella si trova all'interno del box di uno degli ufo o asteroidi; Se ciò accade, la navicella viene distrutta e il gioco termina, chiamando la funzione *killNavicella():*

```
//funzione chiamata quando la navicella esplode
void killNavicella() {
    navicella_alive = false;
    spawn_frammenti(navicella_posx, navicella_posy, col_white, 100); //spawno i frammenti della navicella distrutta
    cout << endl << "\nGAME OVER" << endl;
    cout << "Asteroidi distrutti: " << score_asteroidi << endl;
    cout << "Ufo distrutti: " << score_ufo << endl;
}
```

Quando la navicella viene distrutta vengono spawnati al suo posto dei frammenti tramite la funzione *spawn_frammenti()*, che vedremo dopo.

Infine, se stiamo premendo il tasto *w* (ci stiamo movendo), bisogna spawnare il **propulsore** della navicella, e questo viene fatto con la funzione *spawn_propulsore()*, che crea un effetto particellare che cambia colore nel tempo ed è stato preso dal progetto di esempio 2D_PS. Il propulsore viene spawnato sotto la navicella e per farlo c'è da tenere in considerazione non solo la posizione della navicella, ma anche l'angolo con cui è disegnata, per fare in modo che indipendentemente dall'angolo il propulsore sembri sempre uscire dal retro della navicella.

Asteroidi

Come per la navicella, anche gli **asteroidi** hanno dei parametri a loro associati:

```
//struttura dell'asteroide (posizione, angolo di rotazione, variazione in x e y del movimento, lato della finestra in cui spawna l'asteroide, velocità)
typedef struct {
    float xpos, ypos, angle;
    int deltax, deltay, spawn_side, speed;
} Asteroid;

//parametri per gli asteroidi
int vertices_asteroide = 12;
int max_asteroidi = 10; //numero massimo di asteroidi a schermo
int number_asteroidi = 0;
int asteroide_width = 40;
int asteroide_height = 40;
Pointxy* Asteroide = new Pointxy[vertices_asteroide]; //struttura che contiene i vertici per disegnare un singolo asteroide
vector <Asteroid> asteroidi; //struttura che contiene gli asteroidi in scena
```

Per mantenere traccia della posizione, velocità, direzione di movimento e angolo dei singoli asteroidi ho creato una struttura apposita *Asteroid*. *Max_asteroidi* è il numero massimo di asteroidi che possono essere presenti in scena, *number_asteroidi* è il numero di asteroidi in scena e *asteroidi* è l'array che contiene le strutture degli asteroidi che al momento sono in scena.

La geometria dell'asteroide è realizzata nella funzione *disegna_asteroide()*, e consiste in due quadrati di cui uno è ruotato di 45 gradi. Il disegno su schermo è gestito nella drawscene, dove per ogni *Asteroid* all'interno del vettore degli asteroidi si va a disegnare un asteroide nella posizione e rotazione scritta nella struttura.

Tutta la logica è nella funzione update_asteroidi(), che viene chiamata ciclicamente ogni 24 millisecondi. Se nella scena sono presenti un numero di asteroidi minore del numero massimo, allora viene spawnato un asteroide fuori da uno dei quattro lati della finestra. All'asteroide viene data una velocità (che può variare da più lento a più veloce), una direzione (deltax,deltay) e un angolo (l'angolo è solo per far ruotare il disegno mentre si muove). La direzione è scelta in modo che l'asteroide vada sempre a finire all'interno della finestra quando si muove. Poi per ogni asteroide si va a modificare la sua posizione e angolo in base ai suoi parametri:

```
//cambia posizione asteroidi
for (int i = 0; i < asteroidi.size(); i++) {
    asteroidi.at(i).xpos += asteroidi.at(i).speed*asteroidi.at(i).deltax;
    asteroidi.at(i).ypos += asteroidi.at(i).speed*asteroidi.at(i).deltay;
    asteroidi.at(i).angle = (int)(asteroidi.at(i).angle + 1) % 360; //ruoto l'asteroide
```

E infine per ogni asteroide si va a verificare se l'asteroide è ancora nello schermo: infatti se l'asteroide è uscito di troppo dallo schermo viene eliminato rimuovendolo dal vettore, per evitare che l'array e la memoria si riempiano di oggetti inutili.

Ufo

I parametri associati agli **ufo** sono i seguenti:

```
typedef struct { //struttura dell'ufo
    float xpos, ypos;
    int deltax, deltay, spawn_side, speed, life;
    int attack_time;
} UFO;

//parametri per gli ufo
int vertices_ufo = 36;
int max_ufo = 1; //numero massimo di ufo a schermo
int number_ufo = 0;
int ufo_width = 70;
int ufo_height = 40;
int ufo_life = 5;
Pointxy* Ufo = new Pointxy[vertices_ufo]; //struttura che contiene i vertici per disegnare un singolo ufo
vector <UFO> ufos; //struttura che contiene gli ufo in scena
```

Come per gli asteroidi, anche gli ufo hanno una loro struttura che tiene conto di posizione, direzione, velocità, vita e tempo d'attacco.

I parametri sono molto simili a quelli per gli asteroidi, con l'aggiunta di *ufo_life* che indica i punti vita con cui spawnano gli ufo.

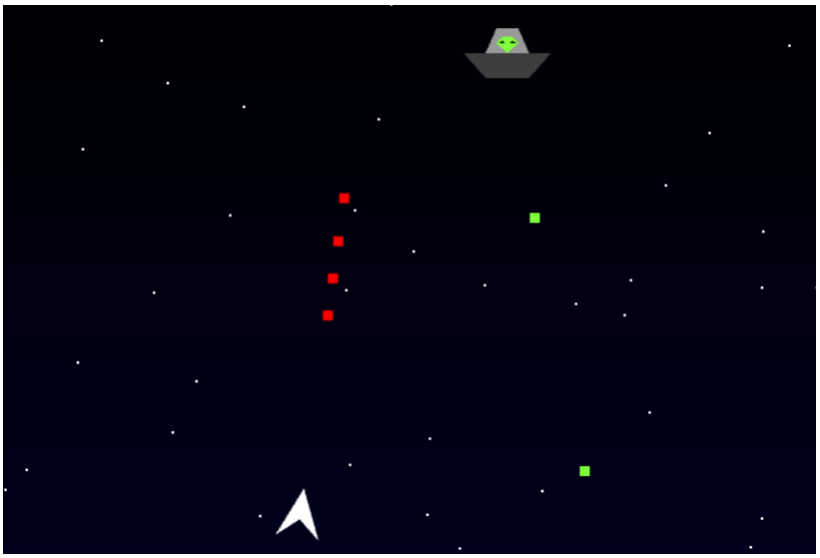
La geometria dell'ufo è realizzata nella funzione *disegna_ufo()*, e consiste in tre parti: la parte sotto dell'ufo composta da 4 triangoli colorati di grigio, la parte sopra composta da 3 triangoli colorati di grigio chiaro e l'alieno composto da 5 triangoli. Il disegno su schermo è gestito nella drawscene, dove per ogni *UFO* all'interno del vettore degli ufo si va a disegnare un ufo nella posizione scritta nella struttura.

Tutta la logica è nella funzione update_ufos(), che viene chiamata ciclicamente ogni 24 millisecondi. Se nella scena sono presenti un numero di ufo minore del numero massimo, allora viene spawnato un ufo fuori da uno dei due lati della finestra. All'ufo viene data una velocità (che può variare da più lento a più veloce), una direzione (deltax,deltay), i punti vita e il tempo d'attacco. La direzione è scelta in modo che l'ufo vada sempre a finire all'interno della finestra quando si muove. Poi per ogni ufo si va a modificare la sua posizione in base ai suoi parametri secondo una funzione sinusoidale, che lo fa muovere ad onda da una parte all'altra dello schermo:

```
//cambia posizione ufos
for (int i = 0; i < ufos.size(); i++) {
    ufos.at(i).xpos += ufos.at(i).deltax*ufos.at(i).speed;
    ufos.at(i).ypos += 0.9*sin(ufos.at(i).xpos/10);
```

Per ogni ufo si va poi ad aumentare il tempo d'attacco di 1 e se raggiunge una certa soglia si fa sparare un proiettile all'ufo verso il basso e si riporta il tempo d'attacco a 0. Il proiettile viene generato con la funzione *spawn_proiettile* che vedremo dopo. Infine, vengono eliminati dal vettore di *UFO* tutti gli ufo che sono usciti dallo schermo, come per gli asteroidi.

Proiettili



I parametri dedicati ai **proiettili** sono i seguenti:

```
typedef struct { //struttura del proiettile (posizione, angolo di movimento, colore)
    float xpos, ypos, angle;
    vec4 col;
    bool enemy; //true se il proiettile è stato sparato da un nemico
} Projectile;

//parametri per i proiettili
int max_proiettili = 20;
int proj_speed = 5;
int proj_size = 8.0;
vector <Projectile> proiettili; // vettore che contiene i proiettili
Pointxy* Proiettile = new Pointxy[max_proiettili]; //vettore che contiene i proiettili aggiornati da essere disegnati a schermo
void spawn_proiettile(float xpos, float ypos, float angle, vec4 color, bool enemy);
```

Anche i proiettili hanno una loro struttura che contiene posizione, angolo con cui sono stati sparati, colore, e un boolean per differenziare se il proiettile è stato sparato da un nemico o da noi (i proiettili sparati dal giocatore possono fare danno ai nemici ma non al giocatore, e i proiettili sparati dai nemici possono fare danno al giocatore ma non ad altri nemici). *Max_proiettili* indica il numero massimo di proiettili che possono essere a schermo.

In questo caso il proiettile è un semplice punto con una certa dimensione, e viene disegnato a schermo nella drawscene usando l'array *Proiettile.*

Un proiettile può essere generato utilizzando la funzione *spawnProiettile()*, che crea un nuovo oggetto *Projectile* con posizione, angolo di sparo, colore e chi l'ha sparato passati come parametro, e lo inserisce nel vettore.

Tutta la logica del proiettile è contenuta in *update_proiettili()* che viene chiamato ogni 24 millisecondi. Per ogni proiettile in scena si va a modificarne la posizione tramite una traiettoria rettilinea definita da una funzione che tiene conto dell'angolo di sparo:

```
//sposto il proiettile in avanti (in base all'angolo con cui è stato sparato)
proiettili.at(i).xpos += proj_speed * sin(radians(proiettili.at(i).angle));
proiettili.at(i).ypos += proj_speed * cos(radians(proiettili.at(i).angle));
```

Poi per ogni asteroide e ufo in scena, vado a controllare se il proiettile lo ha colpito vedendo se è entrato nel box del nemico. In caso positivo il proiettile e il nemico vengono cancellati e si aumenta il punteggio. Successivamente si controlla anche se il proiettile ha invece colpito la navicella (se è un proiettile nemico), in caso positivo si cancella il proiettile e si distrugge la navicella con *killNavicella()* visto precedentemente. Quando il proiettile distrugge un'entità, al suo posto spawnano frammenti di colori diversi con la funzione *spawn_frammenti()*. Infine, se il

proiettile non ha colpito nulla si controlla che non sia uscito dallo schermo. Se è uscito si elimina dal vettore, mentre se non è uscito si aggiorna la struttura usata per disegnarlo.

Frammenti

I **frammenti** sono entità praticamente identiche all'entità del propulsore, in quanto usano lo stesso algoritmo per essere creati e disegnati. La differenza con il propulsore è che può essere scelto il colore da utilizzare per i frammenti da spawnare, e i parametri di creazione usati sono un po' diversi dal propulsore.

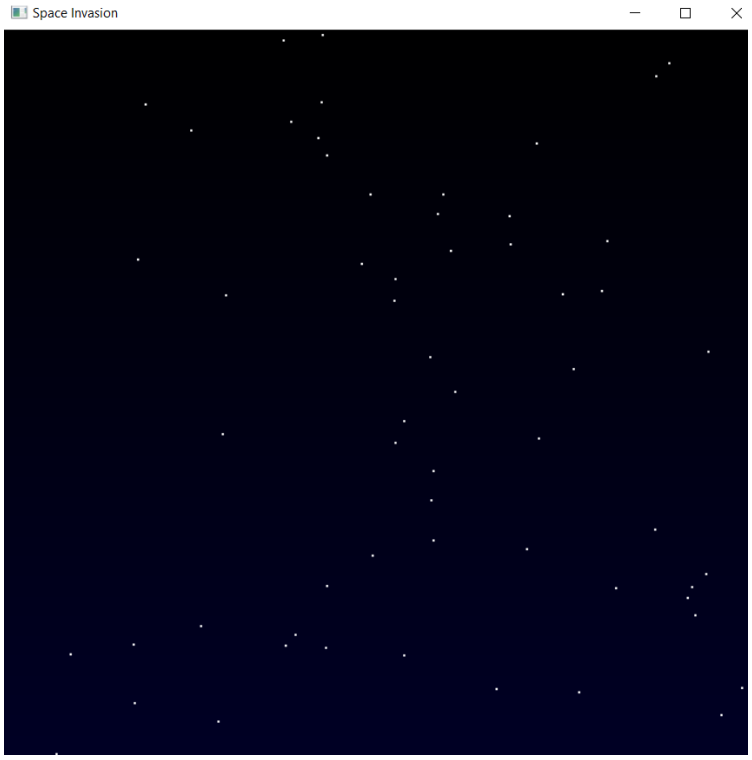
I frammenti possono essere creati con la funzione *spawn_frammenti():*

```
//funzione che permette la creazione dei frammenti
void spawn_frammenti(float xpos, float ypos, vec4 color, int num_particles) {
    for (int i = 0; i < num_particles; i++) {
        PARTICLE p;
        p.x = xpos;
        p.y = ypos;
        p.alpha = 1.0;
        p.drag = 1.0;
        p.xFactor = (rand() % 1000 + 1) / 300.0 * (rand() % 2 == 0 ? -1 : 1);
        p.yFactor = (rand() % 1000 + 1) / 300.0 * (rand() % 2 == 0 ? -1 : 1);
        p.color.r = color.r;
        p.color.g = color.g;
        p.color.b = color.b;
        // Adds the new element p at the end of the vector, after its current last element
        frammenti.push_back(p);
    }
}
```

La funzione crea una serie di frammenti nella posizione, colore e numero passati come parametro, e li inserisce in un'array *frammenti*.

La logica dei frammenti è in *disegna_frammenti(),* dove viene aggiornata posizione e tempo di vita dei singoli frammenti.

Elementi di background



Il **cielo** è realizzato semplicemente con due triangoli i cui vertici superiori ho assegnato come colore il nero e ai vertici inferiori un blu scuro. Lo shader si occupa automaticamente di interpolare i due colori.

Le **stelle** sono realizzate semplicemente con dei punti bianchi disegnati in posizione randomiche della finestra dalla draw scene.