# CMDCarGame

Progetto del corso di Programmazione 2020/21

Angelo Galavotti, Adriano Pace, Denis Pondini

February 7, 2021

# 1 Introduzione alle classi

## 1.1 Definitions.hpp

In questa classe sono riportare tutte le **macro** utilizzate nel codice.

## 1.2 Menu.hpp

Questa classe contiene le funzioni riguardanti il display delle schermate di menù, iniziali e finali. Con essa il giocatore può decidere se abilitare la DevMode, scegliere il tipo di grafica, iniziare una nuova partita, consultare le istruzioni di gioco e i crediti, oppure uscire.Tutti gli ascii art vengono caricati dai file di testo presenti nelle cartelle di gioco e riprodotti alle coordinate designate sul terminale. Nella schermata di Game Over verranno stampati, oltre alle possibilità di restart e uscita, anche tutti i dati inerenti ai livelli superati dal giocatore, contenuti in una lista aggiornata ad ogni variazione.

## 1.3 LevelManager.hpp

Sicuramente la classe più importante di tutte, è infatti quella che permette la ripetizione degli eventi e la chiamata ad altre funzioni esterne ad ogni frame. Viene attivata quando il giocatore preme il tasto **New Game**, qui una breve introduzione ad alcune delle sue funzioni più importanti.

```
void Menu::SetWindow()
{
    _COORD coord;
    coord.X = width;
    coord.Y = height;

    _SMALL_RECT Rect;
    Rect.Top = 0;
    Rect.Left = 0;
    Rect.Bottom = height - 1;
    Rect.Right = width - 1;

    hconsole = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleScreenBufferSize(hconsole, coord);
    SetConsoleWindowInfo(hconsole, TRUE, &Rect);
```

Figure 1: la funzione SetWindow della classe **Menu** utilizza le funzioni built-in di windows.h per creare la finestra di gioco.

### 1.3.1 Update

Esattamente la funzione che contiene le chiamate a tutte le altre, la cui ripetizione costante permette al gioco di svilupparsi.

### 1.3.2 Start()

La prima funzione chiamata dal **Main**, si occupa di creare la macchina del giocatore e i suoi "boundaries" ovvero i bordi che limitano i suoi spostamenti. Inoltre viene istituita una Queue, che servirà per mantenere il numero di indici ancora disponibili in CollectableMemory (la nostra "cache"), un array che contiene il riferimento a tutti gli oggetti presenti sullo schermo. Viene poi istituito il riquadro in alto a destra con le Game Info, e colorato il background, dopo aver inizializzato i parametri di gioco.

### 1.3.3 EnviromentAnimationRenderer

Questa funzione permette di simulare la velocità grazie alla stampa su terminale, di colori(guardrails) e linee(riga di mezzeria), dettata dalla variazione di un booleano, vengono effettuati diversi controlli per evitare effetti indesiderati a livello grafico.

### 1.3.4 PlayerGameMechanics

Qui vengono definite le regole di comportamento del gioco durante l'esecuzione, si decide come aumentare o diminuire un livello, il gameover e vengono ag-

```
playerCar.setBoundaries(LEFT_SCREEN_BOUNDARY + 1, RIGHT_SCREEN_BOUNDARY - 1,
                        UPPER_SCREEN_BOUNDARY + 1, LOWER_SCREEN_BOUNDARY - 1);

indexQueue = IndexQ();

for(int i = 0; i < MAX_ON_SCREEN_OBJECTS; i++)
{
    indexQueue.enqueue(i);
    collectables[i].available = true;
    collectables[i].object = Collectable();
}

drawBackground();

UIGameInfoInit();
playerCar.renderSprite();
```

Figure 2: Estratto della funzione Start

giornati gli stats dei punteggi ogni qual volta si sorpassa o si perde un livello.

### 1.3.5 Spawn

La funzione Spawn regola l'avvenimento di entrata sullo schermo degli oggetti con cui il giocatore dovrà avere a che fare, questo elemento sarà scelto casualmente. L'indice della cache che l'elemento andrà ad occupare sarà scelto da **IndexQueue** che ritorna un valore di -1 se è la cache è momentaneamente piena.

```
//se dopo tutti questi passaggi, k  != -1 (quindi si  trovato un posto libero) finalizza lo spawn
if (k != -1)
{
    collectables[k].available = false;
    if (randomValue == 0) collectables[k].object = gas;
    else if (randomValue == 1) collectables[k].object = puddle;
    else if (randomValue == 2) {collectables[k].object = enemyCar; collectables[k].object.randomDir();}

    randomValue = rand() % 54 + (LEFT_SCREEN_BOUNDARY + 1);
    collectables[k].object.moveTo(randomValue, 0);

    Collider* coll = collectables[k].object.getCollider_ptr();
    if (coll->leftLine <= ROAD_CENTER && ROAD_CENTER <= coll->rightLine) collectables[k].object.moveTo(ROAD_CENTER + 1, 0);

    //fixa un bug che "cancellava" una parte del lato sinistro della strada durante la lightWeightMode.
    if (collectables[k].object.getTypeOfCollectable() == EnemyCar && randomValue <= LEFT_SCREEN_BOUNDARY + 1
        && collectables[k].object.getDir() != 0 )
        {   collectables[k].object.moveTo(LEFT_SCREEN_BOUNDARY + 2, 0);}
```

Figure 3: Estratto della funzione Spawn

### 1.3.6 CheckColliders

Questa funzione ci permette di controllare se durante lo svolgimento avvenga o meno una collisione. Il sistema collisioni è stato implementato tramite una struct Collider (in **ConsoleSprites.hpp**), collegata ad ogni oggetto di tipo ConsoleSprite, per cui definisce il "perimetro" nel quale il giocatore deve arrivare affinchè una collissione possa avvenire. Per fare questo controllo, si fa uso di un puntatore al collider stesso di ciascun oggetto. Quando una collisione avviene, il CollisionHandler si occupa di eliminare l'oggetto e aggiungere un indice alla coda.

```
void LevelManager::checkColliders()
{
    for(int i = 0; i < MAX_ON_SCREEN_OBJECTS; i++)
    {
        if (!collectables[i].available) //per ogni elemento sullo schermo...
        {
            Collider* col_ptr = collectables[i].object.getCollider_ptr();

            if (playerCar.checkCollision(col_ptr)) //...guarda se c'è una collisione.
            {
                if (collectables[i].object.getTypeOfCollectable() == Gas) gas_tanks_counter++;
                else if (collectables[i].object.getTypeOfCollectable() == Puddle) puddle_counter++;

                CollisionHandler(i); //se c'è una collisione, passo il controllo al collisionHandler.
```

Figure 4: Principio della collisione

## 1.4 ConsoleSprite.hpp

Questa classe implementa le funzioni per il rendering degli sprite, assieme alla decodifica di uno sprite da file .txt. Inoltre, contiene le informazioni sullo sprite stesso.

### 1.4.1 LoadFromFile

In questa funzione vengono letti e decodificati gli sprite salvati nella relativa cartella, essi vengono letti carattere per carattere, al fine di settare correttamente i colori nel dettaglio.

### 1.4.2 Translate

Translate, traduzione di traslare, effettua l'update delle coordinate della posizione di un oggetto. Permette il movimento di un oggetto di una posizione in ogni direzione.

### 1.4.3 moveTo

Come translate, anche questa effettua l'update delle coordinate della posizione di un oggetto, tutta via questa funzione sposta proprio l'oggetto in se in un una data coordinata.

### 1.4.4 GenerateColliders

Questa funzione permette di generare i collider in base alle coordinate dei pixel degli sprite. Un collider è essenzialmente il perimentro rettangolare di un certo sprite, e appunto rappresenta l'area in cui le collisioni vengono effettivamente registrate. In devMode, si ha una rappresentazione grafica dei vertici di questi collider.

## 1.5 Collectable

Questa classe eredita i metodi ed il costruttore della classe ConsoleSprite, è molto importante perchè generalizza le funzioni di movimento consentendo ad ogni oggetto di usarle.

### 1.5.1 Movement

Movement permette agli oggetti di muoversi in ogni direzione nella mappa, solo le macchine si muovono in diagonale per una scelta implementativa di difficoltà del gioco. N.B. -¿ il movement degli oggetti è diverso dal movement della player car.

## 1.6 Pixel.hpp

Perchè **Pixel**? Abbiamo deciso di chiamare questa classe così per cercare di definire una nostra "unità di misura". Siccome ciascun pixel rappresenta un carattere del terminale, il nostro pixel è molto più grande e definisce una porzione di schermo decisamente guardabile ad occhio nudo. Ogni oggetto

```
    Il modo in cui si muove dipende dal valore di direction.

    E' molto diverso dal metodo Movement della classe Car.
*/
void Collectable::Movement()
{
    if (rect_collider.bottomLine <= LOWER_SCREEN_BOUNDARY && !stop)
    {
        if (rect_collider.leftLine < LEFT_SCREEN_BOUNDARY + 2
         || rect_collider.rightLine > RIGHT_SCREEN_BOUNDARY - 2) {direction *= -1;}

        deleteSprite();
        translate(direction, 1);
        renderSprite();

        //impedisce a un elemento essere bloccato
```

Figure 5: Estratto della funzione movement

di tipo consoleSprite (quindi per ereditarietà Collectable e Car) possiede un suo array di Pixel che permettono quindi la renderizzazione su schermo.

```
Pixel::Pixel()
{
    position.X = 20;
    position.Y = 20;
    pixChar = ' ';
    color = BLACK_B_BLACK_F;
}

Pixel::Pixel(char pix)
{
    position.X = 0;
    position.Y = 0;
    pixChar = pix;
    color = BLACK_B_BLACK_F;
}
```

Figure 6: Costruttore del Pixel

## 1.7 Car.hpp

Classe che implementa i metodi necessari al corretto funzionamento delle dinamiche di gioco della player car. Anche questa eredita la classe consoleSprite.

```
/** Metodo che controlla il movimento della macchina.
*   Permette il movimento da tastiera, attraverso diversi keybindings.
*/
void Car::optimized_Movement()
{
    int x = 0, y = 0;

    bool hasMoved = true;
    if (GetAsyncKeyState(VK_UP) && rect_collider.topLine > up_wall) {x = 0; y = -1;}
    else if (GetAsyncKeyState(VK_LEFT) && rect_collider.leftLine > left_wall ) {x = -1; y = 0;}
    else if (GetAsyncKeyState(VK_RIGHT) && rect_collider.rightLine < right_wall ) {x = 1; y = 0;}
    else if (GetAsyncKeyState(VK_DOWN) && rect_collider.bottomLine < down_wall) {x = 0; y = 1;}
    else hasMoved = false;

    if (hasMoved)
    {
        deleteSprite();
        translate(x, y);
        renderSprite();
    }
}
```

Figure 7: Come si muove la macchina del giocatore

## 1.8 List.hpp

Classe dedicata alla gestione delle code del gioco, usate per tenere il conto delle statistiche (stampate ricorsivamente nel game-over menu), e per contenere gli indici di elementi disponibili allo spawn.

## 1.9 ASCII Mode

Questa modalità permette di giocare il gioco con un aspetto più minimale, in cui i caratteri (che solitamente sono spazi vuoti) degli sprite vengono sostituiti con lettere e simboli. Siccome in questa modalità la riga di mezzeria (e quindi anche i controlli con le macchine) viene eliminata e l'animazione dei guardials viene fermata, la prestazioni sono più alte rispetto alla modalità di gioco tipica.

## 1.10 DevMode

Per gioco, per divertimento, ma perchè no, anche per utilità. Con la DevMode siamo in grado di acquisire diverse informazioni sulle posizioni dei pixel e dei collider ad ogni movimento, possiamo aumentare e decrementare la velocità di gioco a nostro piacimento, inoltre godiamo di immortalità. Oltre

a ciò, abbiamo anche una rappresentazione grafica della coda degli indici liberi. A causa del rendering dei collider e della stampa continua delle info, le prestazioni sono leggermente ridotte.

The End.