Progetto SFML

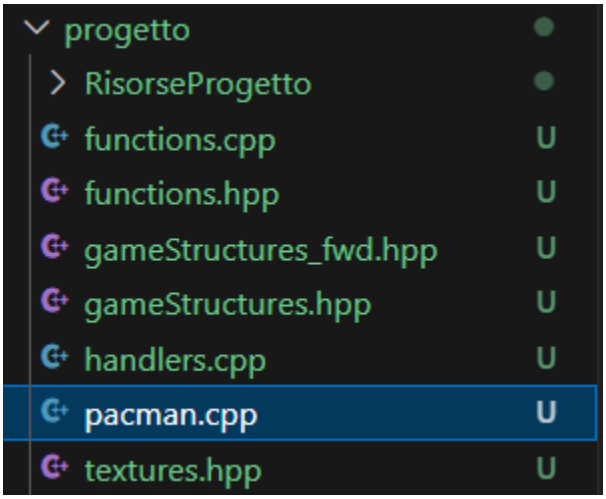
By Leonardo Necordi

Idea base

L'idea base del progetto parte dalla mia passione per il **game design**: voglio usare **SFML** per realizzare un **piccolo gioco**.
Per adattarmi alla dimensione del progetto di questo corso e alla difficoltà nell'utilizzo di SFML, ho ritenuto che un progetto di uno scope adeguato potrebbe essere un **Clone di Pacman,** dopo avergli sottoposto l'idea, il Prof. Puppo ha espresso un parere positivo a riguardo.
Ho scelto SFML innazitutto perchè rispetto alle altre proposte di progetto la programmazione "nuda e cruda" **è il motivo** per cui ho scelto questa università e poi perchè ero curioso di vedere cosa significa programmare un videogioco partendo da librerie limitate come SFML (rispetto a Engine come **Unity** su cui avevo qualche esperienza).

Organizzazione Files

Il progetto è diviso in più **files** per una migliore **leggibilità** e **navigabilità**:



pacman.cpp è il file principale: contiene la dichiarazione della maggior parte delle **variabili globali**, include le **librerie** e gli altri **file** e contiene il **main**.
textures.hpp dichiara le costanti contenenti i ritagli di **texture** utilizzati più qualche costante affine.
functions.hpp è il file **header** che definisce tutte le funzioni divise per tipo e con lo stesso ordine del file .cpp, include inoltre gameStructures_fwd.

gameStructures_fwd.hpp è il file di forwarding delle strutture: **definisce** ed **introduce** le classi e le strutture di gameStructures.hpp, in modo che le **funzioni** possano riferirsi alle **classi** o **strutture** senza avere effettivamente accesso al loro contenuto.
gameStructures.hpp contiene la matrice della **mappa** e tutti gli **enum**, **classi** e **strutture** del gioco.
functions.cpp è il corpo del progetto: contiene tutte le **funzioni** che fanno *funzionare* il gioco.
handlers.cpp contiene gli handlers di **eventi**, in particolare per gestire **chiusura**, **resize** della finestra e gli **inputs**.

RisorseProgetto cartella che contiene **immagini**, **suoni** e **font** utilizzati più due file per le licenze.

Struttura codice:

Dopo aver fatto le **inizializzazioni** e gestito gli **eventi** inizia il **Game Loop**: se ci si trova in un **menù** ne gestisce le interazioni, sennò: imposta il **deltaTime** (per sincronizzare i movimenti slegandoli dai frame al secondo), gestisce **pacman** controllandone **collisioni** ed impostandone **movimento** e **texture** per dare vita all'animazione e gestisce i **fantasmi** impostandone il **target** e muovendoli, **disegnando** infine tutto.

Storia dello sviluppo

Nota: Ogni tanto ci saranno dei riferimenti del tipo [nomeFile.cpp] o [nomeZip.zip] che indicano il nome del file o zip di backup che salva i progressi appena discussi.

Come è possibile notare il progetto non è partito da una solida architettura o design iniziale e molte scelte sono state prese tramite tentativi, questo processo sicuramente non è il migliore e in caso di creazione di un prodotto software sarebbe opportuno procedere in modo totalmente diverso.
Tuttavia questo non è un prodotto da mercato, ma bensì il progetto finale di **Fondamenti di Computer Grafica**, credo la cosa più importante qui sia imparare come utilizzare la libreria SFML ed esercitarsi a costruire un videogioco partendo da un ambiente poco conosciuto, mettendo lo studio e la sperimentazione al primo posto.
Inoltre voglio specificare che non esiste una correlazione tra sessioni/giornate di coding e suddivisione dei capitoli o paragrafi, a volte qualche salto verrà evidenziato ma la maggior parte delle volte un paragrafo coprirà lo sviluppo attraverso più sessioni.
Ho scritto questo all'inizio per tenerlo in mente quando si incontreranno scelte di design o processo non ottimali, detto questo si può iniziare: Let's play!

I) Inizio

Poichè il gioco è già noto, non mi sono dovuto occupare di una fase di design ed essendo ancora nello studio di come si struttura un codice su **SFML**, ho deciso di adottare un approccio **bottom-up**: Si inizia dal codice, implementando le cose più piccole per poi ridefinire e aggiustare lo stile e l'organizzazione via via, cercando di aderire quanto possibile con quanto visto a lezione.
La fase iniziale si è aperta con un duro scontro con il framework utilizzato, rispetto a **Unity** -con cui già avevo qualche esperienza- risulta essere molto più rigido e di basso livello.

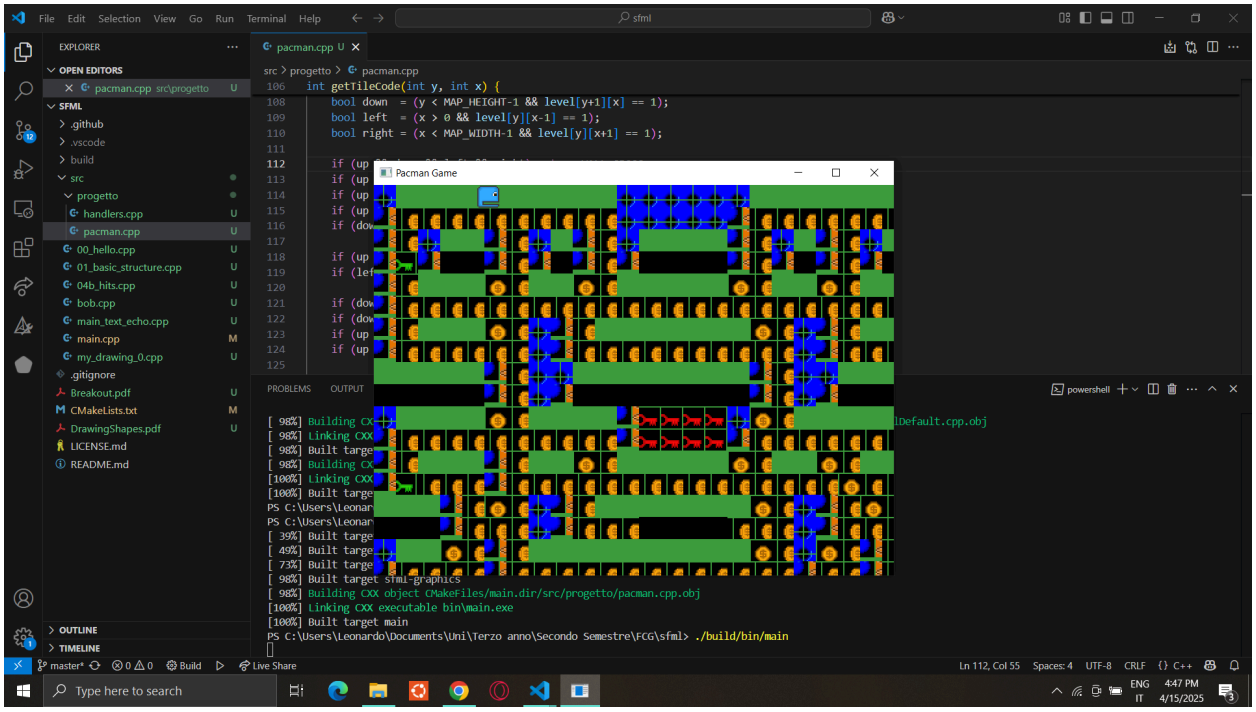
Sono allora partito da un layout di codice minimo di SFML, ho cercato su internet un file **texture** contenente tutti gli elementi di cui avevo bisogno per il gioco e sono partito dal centro del gioco: **Pacman**.
Dopo aver avuto qualche scontro con il sezionamento del file **texture** (nonostante nelle indicazioni ci fosse scritto che ogni tile era 20x20 px, questa unità di misura è risultata lievemente distorta), ho deciso di tentare di fare muovere il personaggio.
Per fare ciò, mi sono subito accorto di dover aggiustare l'handling degli stati, fra quanto trovo online e quanto spiegato a lezione trovo qualche differenza di stile, dovuto sicuramente a utilizzi di versioni diverse di SFML, decido comunque di optare per uno stile noto: l'uso di funzioni lambda per il metodo handle() e la divisione in struct degli oggetti presenti.
Dopo aver fatto **muovere** il personaggio in modo *smooth* (come visto a lezione) alla premuta dei tasti WASD e aver iniziato a spostare il codice fuori dal **main**, ho deciso di adattare il movimento del mio personaggio ad una griglia.
Prima di fare ciò ho però pensato di dividere il codice in più **file**, in modo da facilitare la navigazione e avere un maggiore ordine: il primo step è stato dividere gli handlers dal resto del codice, pianificando più avanti di dedicare anche alle struct un file a parte.
Con qualche intuizione trovata su internet, ho aggiustato il movimento in modo da cambiare direzione solo quando allineato con la griglia (composta da caselle dalla dimensione di 20 px, sempre per rimanere allineati con il file di texture).

II) La mappa

Come secondo obiettivo mi sono prefissato di impostare la **mappa di gioco**, questo comprende creare una struttura che rappresenti come la mappa è divisa in **caselle**, istanziare ogni casella del corretto tipo di muro e poi gestirne le collisioni.
Anche qui, la decisione è stata quella di rimanere fedele allo stile di sviluppo del resto del progetto: prima aggiungo la nuova feature, dopo aggiusto il codice per renderlo pulito e funzionale.
Nelle prime ore gli sforzi si sono concentrati sull'impostare l'algoritmo per la stesura della mappa e -ahimè- nuovamente a picchiarmi con le texture.
L'idea nella mappa è di avere una struttura a **matrice** che assegni un numero intero per ogni differente tipo di sprite da istanziare.

[Parte scritta dopo diverso tempo dallo svolgimento della stessa:]
Ho tentato di implementare la mappa con questa idea di base:
Ogni casella della mappa è uno **sprite** a cui associare una texture (dal foglio delle texture) in base alla sua identità: Tipo di muro, dot, superdot o vuoto.
Per fare questo ho implementato prima una funzione di **autotiling**, che assegni automaticamente il tipo corretto di muro per ogni casella e poi cercato di ritagliare il pezzo corretto di texture per ognuna.

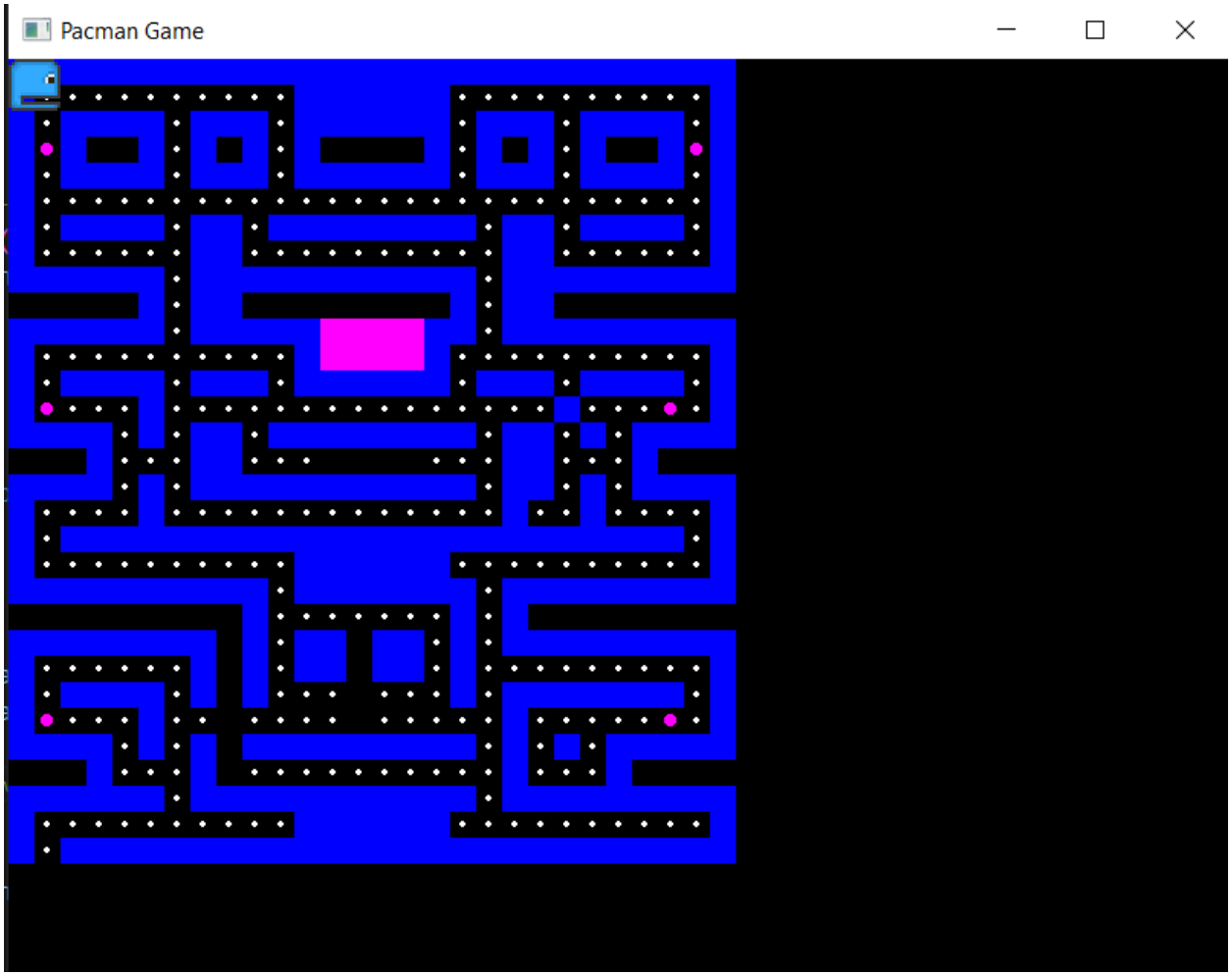
Quest'ultima parte si è rivelata fallimentare, nonostante diverse ricerche e tentativi anche aiutati da IA non sono riuscito ad arrivare ad un momento nel quale la mappa venisse disegnata nel modo giusto.
Ho riprovato per un paio di giorni, poi demotivato ho interrotto temporaneamente il progetto



[Screenshot dello stato del gioco pre-ripresa]
[mappaScartata.cpp]

III) La ripresa

Passato un po' di tempo, ho deciso di rimettermi a lavorare al progetto.
Il sentimento di frustrazione legato al tornare a lavorare nel codice in quello stato mi demotivava particolarmente, di conseguenza ho deciso di tentare di ripartire da zero con la mappa utilizzando un altro approccio. [backUpPostRipresa.cpp]
Prima di riprendere con la mappa però, ho migliorato nuovamente l'ambiente di lavoro, aggiungendo una *struct* a parte contenente le informazioni riguardanti i personaggi (**Character**). Per adesso è un campo di **State** per rappresentare **pacman**, ma in futuro verrà utilizzato anche per gestire i **nemici**.
A questo punto, ho cambiato l'idea alla base della mappa:
Invece che rendere ogni casella uno sprite, disegnare direttamente la mappa composta da **RectangleShape**, utilizzando alla base sempre la matrice di interi provata per l'altro metodo (tuttavia aggiustata).
Ho inserito nel main un doppio *for* per costruire la mappa e a questo punto i primi risultati sono stati di gran lunga migliori.

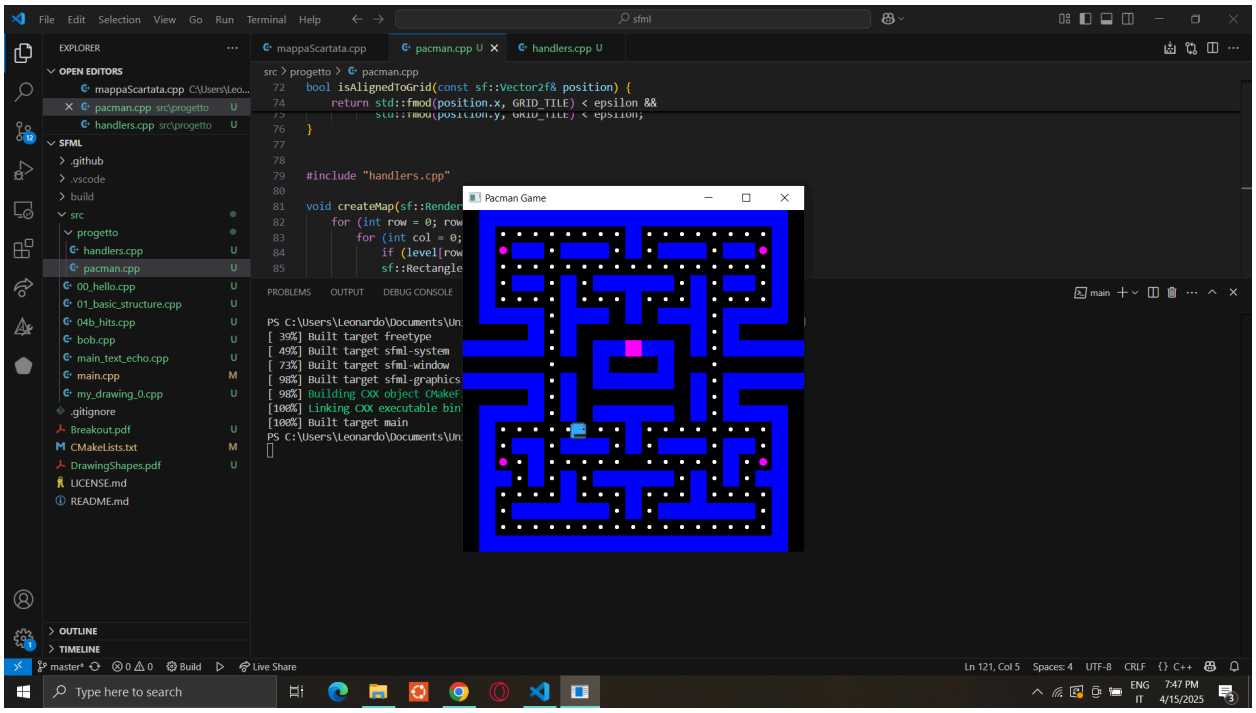


[Screenshot dello stato del primo tentativo positivo di mappa]

A questo punto però mi sono trovato con alcuni problemi da risolvere:
1) La dimensione delle caselle era diversa da quella del personaggio
2) La mappa non era giusta
3) Era presente un effetto sfarfallio della mappa
4) Pacman lasciava una scia muovendosi (oppure la mappa non appariva mai, con il window.clear())

Il primo problema è stato rimpiazzato da un altro rapidamente: Utilizzavo la costante sbagliata per dare la dimensione alle caselle, che non teneva conto del moltiplicatore di zoom, a questo punto ho però dovuto selezionare un moltiplicatore adeguato e adattare le costanti delle dimensioni della finestra rispetto a quelle della mappa.

Ho poi provato a sistemare il secondo conversando con ChatGPT o cercando una matrice corretta online ma niente da fare, l'AI non è abile in questo genere di problemi (o non so usarla sufficientemente bene).
Ho preso allora uno screenshot online della mappa classica e mi sono messo a rifare la matrice level a mano, notando che anche le dimensioni della mappa erano errate essendo quelle giuste 21x21.
Per risolvere gli altri due problemi invece è stato sufficiente sistemare la posizione del blocco di codice relativo alla creazione della mappa (che era nel main prima del game loop) e decommentare il window.clear().
Il primo passo è stato estrarre il codice in una funzione a se **createMap()**, dopo di che ho capito che la cosa opportuna era anticipare la cancellazione della finestra prima di fare alcun calcolo nel game loop e successivamente chiamare la funzione generatrice della mappa.
Nella fase successiva -in cui verrano gestite le collisioni- vedremo se avere la creazione dei vari rettangoli interni ad una funzione sia funzionale al nostro programma o se invece sia necessario mantenere le variabili create da qualche parte.



[Mappa realizzata]

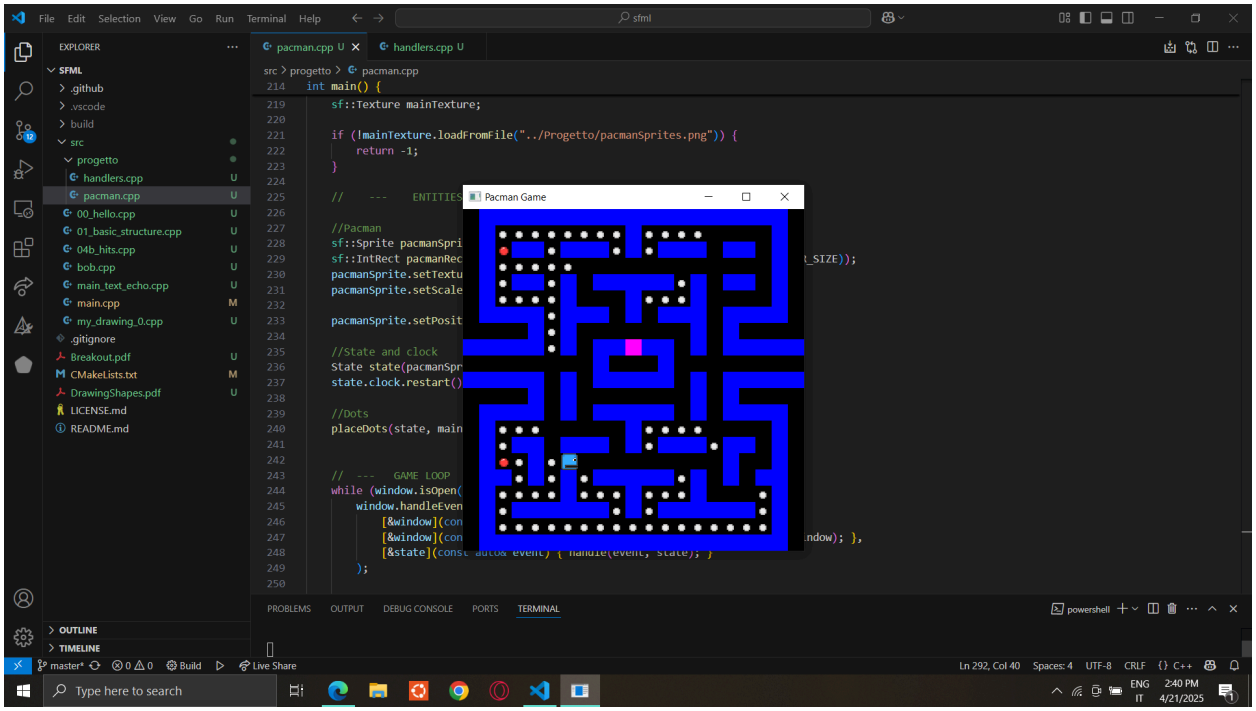
IV) Interazioni con la mappa

Arrivati a questo punto disponiamo di un personaggio che si muove e di una mappa, il prossimo step è chiaramente quello di mettere assieme questi due elementi per farlo muovere secondo essa.

L’intuizione alla base è quella di avere una funzione che determina se il personaggio si può muovere e controllarne il risultato ad ogni iterazione del game loop prima di permettere il movimento.
Il testing di questa implementazione ha anche guidato naturalmente lo sviluppo del codice nel definire alcuni costanti di posizione, in modo da avere un riferimento per l’assenza di direzione e soprattutto per il punto di partenza ufficiale per il nostro personaggio di pacman!
Durante il testing qualche linea si è rivelata essere nel posto sbagliato (in particolare, avevo scollegato il controllo delle collisioni con quello della griglia), ma ho sistemato rapidamente.
Una volta ritenuta la collisione con i muri funzionante, ho aggiunto una funzione **manageTunnel()** per gestire il caso in cui pacman si trovi ad attraversare il tunnel da un lato dello schermo all’altro.
Qui semplicemente viene controllato se pacman intenda andare nella direzione del tunnel e nel caso viene teletrasportato dall’altro lato; ho attraversato qualche iterazione di questa funzione, provando a chiamarla in punti diversi e talvolta accompagnandola da una ausiliaria, ma allo stato corrente per permettere un movimento *sufficientemente* smooth (per farlo totalmente non credo la gestione a sprite sia compatibile, visto che dovrei teletrasportare un pixel alla volta) gestisco una variabile booleana della struct di pacman, in modo da impostare il teletrasporto non appena verificata l’intenzione di attraversare il tunnel, ma effettuarlo solo quando viene effettivamente raggiunto il confine.

Gestita l’interazione con i **muri**, è arrivato il momento di gestire quella con i **dots**.
L’idea è di gestirli come sprites contenuti in un **vettore**, che vengano disegnati ad ogni frame ma generati una sola volta ad inizio main (dopo pacman e lo state).
Ho quindi scritto una funzione apposita per crearli ritagliando la loro texture dal solito foglio, aggiungendo tramite tentativi qualche costante (rassegnandomi al fatto che il file di texture sia sfasato)
Successivamente ho gestito la collisione con **pacman**: l’operazione ha impiegato un po’ di tempo perchè non riuscivo a trovare un metodo di stabilire l’intersezione tra il Pacman e i dots che mi soddisfacesse, alla fine però mi sono rassegnato ad utilizzare una funzione ausiliaria che data una posizione mi restituisca il suo allineamento con la griglia (passando da un Vector2f a un Vector2i), questo ha reso la cattura dei pallini un po’ meno fluida, ma disegnandoli sotto Pacman non si notava troppo.
Per ripulire un po’ l’ambiente di lavoro ho raggruppato la gestione del tunnel e quella dei dots in un’unica funzione **getCollisions()**, qui è sorta la tentazione di inserire in questa funzione anche le collisioni coi muri, tuttavia vedo quella parte di codice più legata al movimento che alle collisioni in sè (controllo della variabile booleana *can walk*, impostazione direzione) quindi ho preferito lasciarla fuori.
In fine ho aggiunto i super pallini, l’unica scelta importante a cui hanno portato è stata il creare una struct Pellet per i pallini, in modo da poter avere un campo isSuper che distinguesse la tipologia, per il resto si è solo trattato di aggiungere la logica per disegnarli e un caso alla

gestione dei dots, prevedo al momento dell'aggiunta dei nemici di aggiungere un campo allo stato per sapere se Pacman si trova in modalità berserk/super (ha appena mangiato un pallino super).



[Dots e muri funzionanti]
[postInterazioneMappa.cpp]

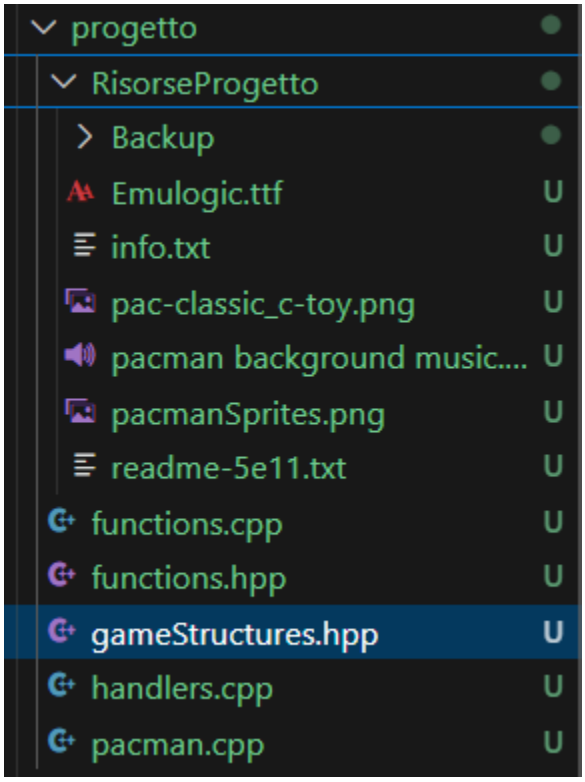
V) Game Over

Una cosa che **amo** dello sviluppo di videogiochi è che la progressione del codice si evolve in modo molto naturale, dopo aver completato un checkpoint basta far partire il gioco e notare lo stato attuale per capire come proseguire.
Adesso la mappa funziona e ci si può interagire, però raccogliere i vari pallini non serve a niente! È evidente che il prossimo passo sia gestire la fine del gioco, preparandola poi anche per la sconfitta del giocatore.
L'idea è quella di aggiungere allo **stato** l'informazione riguardante lo… *stato* -brutto gioco di parole- della partita, in particolare se il gioco è finito o meno.
Ho valutato inizialmente la possibilità di gestire la cosa come un **evento**, ma credo che complicherebbe inutilmente il codice.
La decisione è stata quindi quella di iniziare ad implementarla senza considerarla come un evento, cambiando modalità in caso di problematiche.
Quindi ho prima di tutto aggiunto lo stato di gioco, pensavo inizialmente di creare una variabile booleana isGameOver, tuttavia poi ho pensato che si potrebbero voler gestire anche altre situazioni come la pausa, pertanto ho creato invece un **enum**.

Per disegnare i menù, ho inserito un controllo nel **Game Loop** che a seconda dello stato di gioco decide di chiamare una funzione per disegnare i menù **drawMenu()** e saltare il resto del loop.
Ho quindi preso da internet un font da inserire nello State e scritto la funzione in modo da posizionare i vari elementi del menù in modo piuttosto classico con indicazioni di come far ripartire il gioco.
Ho speso un po' di tempo a cercare un modo semplice per poter chiudere la finestra direttamente dall'handle generico nel caso venisse premuto un certo pulsante (E), tuttavia non ne ho trovati e la modifica degli eventi mi è parsa alquanto complicata, di conseguenza ho optato per lasciare semplicemente la possibilità di reset a fine partita.
Per far funzionare il reset, oltre a cambiare lo stato di gioco ho creato una funzione **resetGame()** che rifacesse le inizializzazioni iniziali per portare il gioco allo stato base, l'handler in questo caso chiama quella funzione ed il gioco ricomincia da capo! [postGameover.cpp]

[Schermata Game_Won]

Prima di poter andare avanti, mi sono reso conto che il file stava diventando davvero troppo affollato e caotico per lavorare in modo efficiente, ho quindi deciso di mettermi a fare un bel refactoring separando Structs e funzioni dal resto.
Ho colto l'occasione per sistemare anche i folder, avvicinandomi alla struttura necessaria per lo zip della consegna.



[Organizzazione folder]
[postRiordine.zip]

Per concludere ho preparato menù e relativa gestione per tutti i possibili stati di gioco, includendo anche menù principale e di pausa.
Questo mi ha richiesto essenzialmente di modificare l'handler e creare funzioni ausiliarie, durante lo sviluppo della pausa ho dovuto affrontare qualche bug riguardo al personaggio che continuava ad andare avanti ignorando i muri mentre a schermo c'era il menù, successivamente non riusciva più a muoversi una volta usciti dal menù di pausa.
La soluzione a questi problemi è stata dietro allo scegliere bene cosa veniva disattivato e riattivato nel passaggio tra uno stato e l'altro, oltre che dividere effettivamente queste due situazioni.
Tutto sembra essere pronto per aggiungere il secondo scenario di terminazione del gioco: la sconfitta.

VI) I fantasmi

Dopo aver realizzato i diversi **stati di gioco** e relativi menù il gioco si può effettivamente vincere.
Che senso ha però vincere ad un gioco in cui è impossibile perdere?
Per dare un senso al menù di sconfitta (ma anche a tutto il resto, in realtà) era finalmente giunto il momento di implementare i **nemici** del gioco.

A) Creazione e allocazione

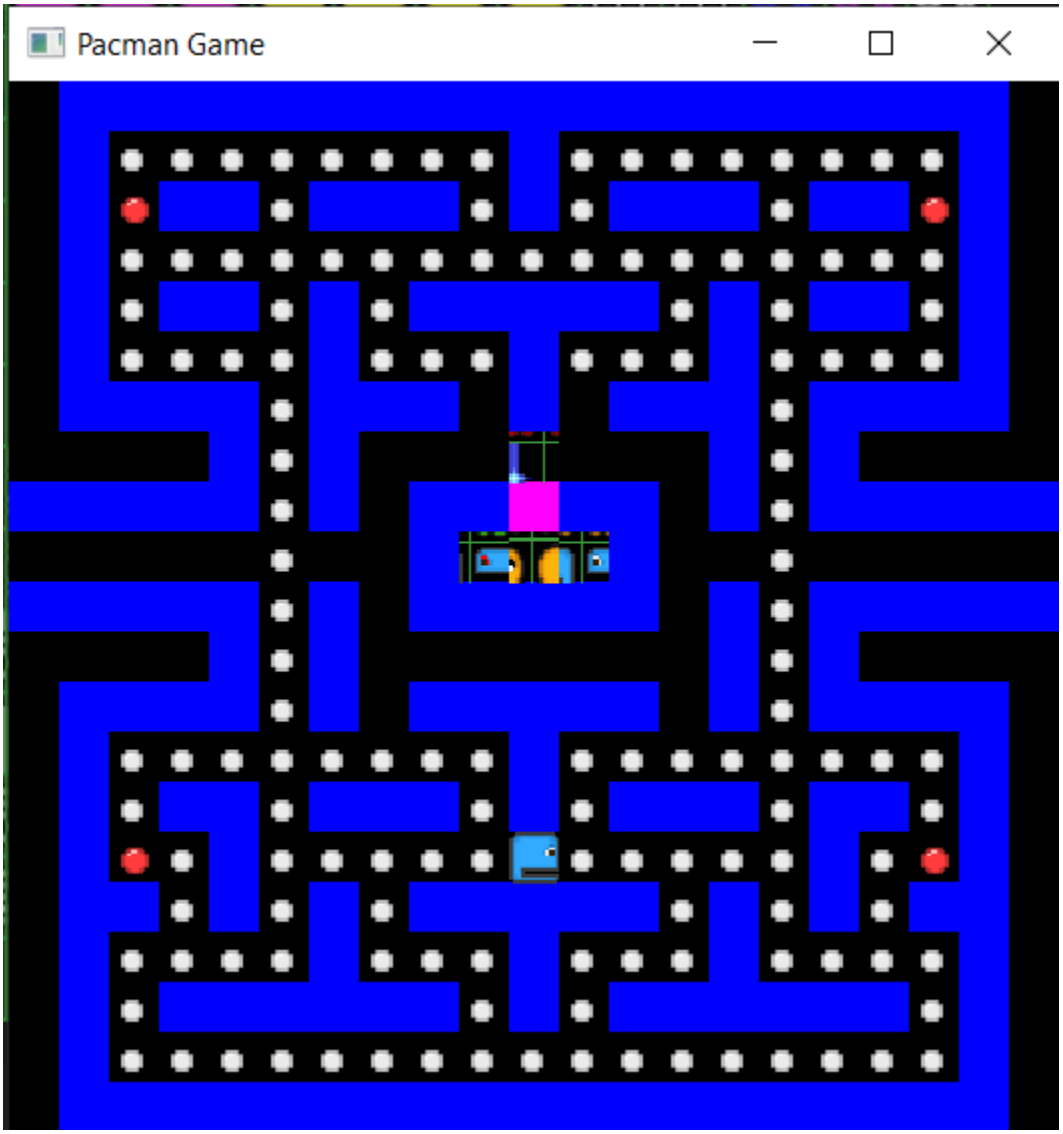
Il primo task -di dimensioni non banali, nonostante le apparenze- è stato quello di aggiungere strutture per i nemici e gestire la loro inizializzazione, piazzamento e disegno.
Il punto di partenza non era affatto male: la struttura character era già generale a sufficienza da avere tutte le componenti necessarie per la generazione e movimento dei fantasmi, ho giusto pensato di aggiungere un booleano isAlive per poi poter gestire le interazioni con i superPellet.
Ho poi pensato che così come nello State c'è un riferimento a pacman sarebbe opportuno anche averne uno per i fantasmi, ancora meglio se tutti assieme.
Una qualche struttura tipo array o vector poteva essere indicata, ma ho preferito usare le solite *structs* sia per una questione di consistenza, sia perchè mi permetteranno di riferirmi ai fantasmi non tramite la loro posizione e ancora per poter implementare dei costruttori.
Così è nata la struct **Enemies**, contenente quattro character chiamati in base al loro colore, il quale fra l'altro ho definito anche in un enum visto che sarà un dato importante sia per gestirne la generazione sia per -in futuro- gestirne personalità ed algoritmo.
Giunto il momento delle inizializzazioni, ho notato che il main stava crescendo notevolmente, ho deciso allora di racchiudere l'inizializzazione di pacman in una funzione a sé e mi sono impegnato a fare lo stesso per i fantasmi e le loro sottofunzioni.

```
Character initializePacman(const sf::Texture& texture);

sf::IntRect initializeGhostRect(GhostType ghostType);

sf::Vector2f initializeGhostSpawn(GhostType ghostType);

Character initializeGhost(const sf::Texture& texture, GhostType type);

Enemies initializeEnemies(const sf::Texture& texture);
```

A questo punto lo sviluppo è stato abbastanza intuitivo: una funzione per generare la struct dei nemici chiama una funzione per **generare** il singolo fantasma, la quale,invece che contenere enormi switch case per impostare punto di generazione e ritaglio di texture a seconda del fantasma, **delega** questi compiti ad altre due sottofunzioni.

Pronto per compilare e testare tutto, ho aggiustato nelle struct i relativi costruttori ed iniziato a scrivere nel main l'utilizzo di queste funzioni ma nel farlo mi sono reso conto gradualmente che disponevo già degli ingredienti per compattare sempre di più questa parte.
Il compilatore mi ha subito dimostrato la fallacia dietro a tutto il mio bel ragionamento, quasi volesse infrangere un sogno troppo bello per essere vero: la catena di inclusioni.
Ci ho messo un po' a capire dove fosse il problema (senza contare che alla fine ne ho speso decisamente troppo per capire che avevo anche invertito due lettere nel nome di una funzione):
Le strutture in **gameStructures.hpp** a questo punto avevano dei meravigliosi costruttori che utilizzano le funzioni di **functions.cpp** definite in **functions.hpp**, tuttavia quelle stesse funzioni necessitano delle strutture di gameStructures!
Tutto sembrava cadere, come un serpente o un cane che si morde la coda ogni pezzo di codice aveva bisogno di qualcosa presente nell' altro, tuttavia non potevo cedere al mio sogno di codice pulito ed efficace così facilmente: doveva esserci un altro modo.
Ho così cercato una soluzione per l'internet e ho scoperto un sistema che non conoscevo: le **forward declarations**!
Dire che non le conoscevo non è del tutto esatto, in realtà è un meccanismo praticamente uguale a quello tra functions.cpp e functions.hpp: essenzialmente si **spezza** il modulo in due, avendone uno che anticipa ciò che verrà dopo, in modo da presentare i nomi di ciò che verrà creato in modo da poterli utilizzare senza ancora saperne il funzionamento.
Così ho creato **gameStructures_fwd.hpp**, che mi ha consentito di risolvere il circolo vizioso, utilizzando liberamente nelle funzioni i nomi delle mie strutture nonostante esse venissero incluse solamente dopo.

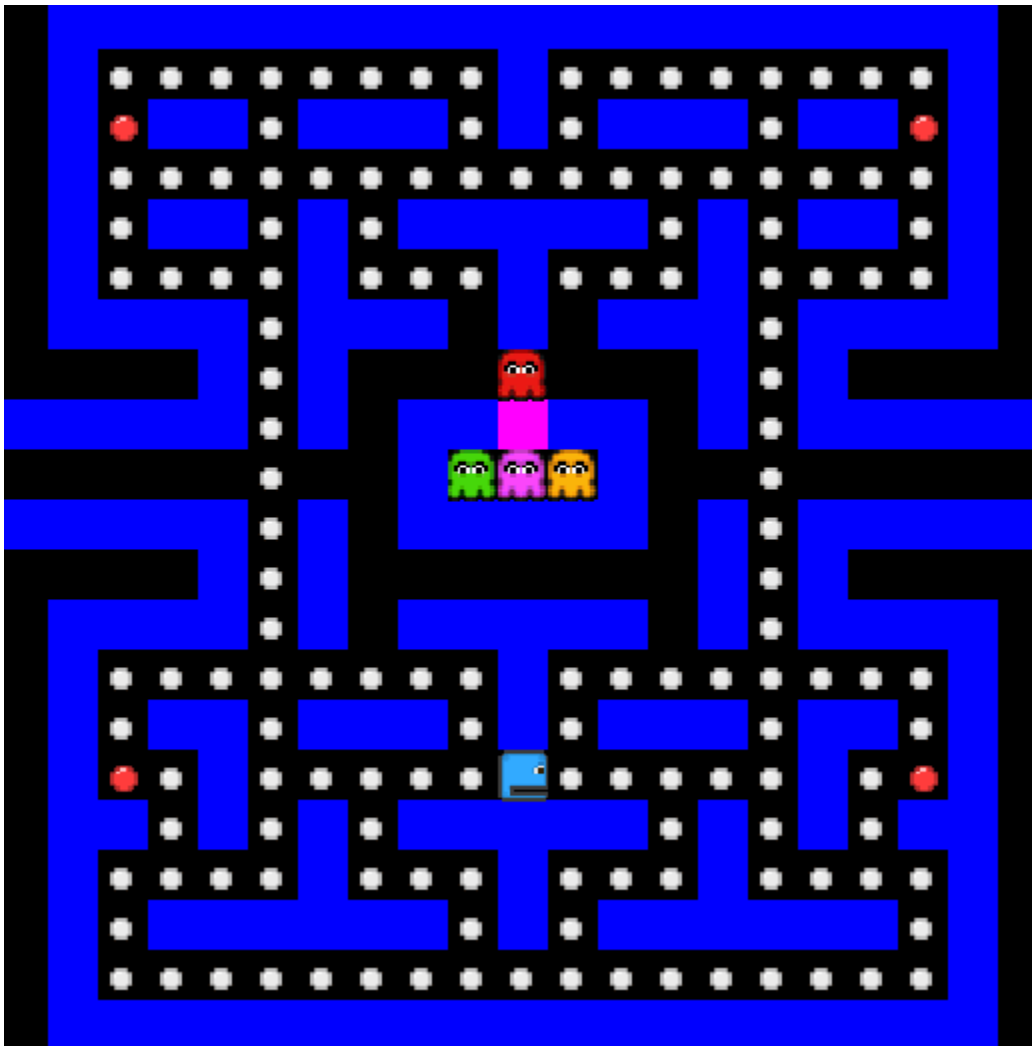
Non dispongo di screenshot riguardanti i rapidi passaggi della fase che venne dopo, però è stato un soddisfacente e fluido concatenarsi di scritture di codice sempre più ristrette, fino a realizzare che l'unico argomento che mi serviva davvero per inizializzare tutti i personaggi era il foglio di **texture**, ho allora creato un nuovo **costruttore** che prendendendo come argomento solo la texture inizializzasse a cascata nemici e pacman chiamandone le relative funzioni, in modo da poter gestire tutta questa parte nel main semplicemente dichiarando `State state(mainTexture);`



[Prima generazione dei fantasmi]

Il risultato era atteso, ma è stato ugualmente divertente: il ritaglio del foglio di texture era sfasato.
Ho quindi allineato a mano il ritaglio di tutti i fantasmi e nel farlo sono forse giunto finalmente a comprendere quale fosse l' "algoritmo di allineamento" delle texture, non comprendo tutt'ora se questa problematica sia imputabile al foglio di texture, alle costanti di ingrandimento o ad altro, però credo che sia sufficiente aver trovato il modo di risolverlo volta per volta.

```
const sf::Vector2i RED_GHOST_TEXTURE(TILE_SIZE * 11-7, TILE_SIZE * 2+5);
const sf::Vector2i PINK_GHOST_TEXTURE(TILE_SIZE * 7 - 11, TILE_SIZE +4);
const sf::Vector2i GREEN_GHOST_TEXTURE(TILE_SIZE * 2+5, TILE_SIZE * 2+5);
const sf::Vector2i ORANGE_GHOST_TEXTURE(TILE_SIZE * 7-11, TILE_SIZE * 2+5);
//Credo per allineare sia: y + (nriga + 4) , x -(ncolonna +4)
```



[Costanti delle texture dei fantasmi e la loro generazione sistemata]
[disegnoFantasmi.zip]

B) Interazione e movimento

Le sottofasi successive saranno probabilmente le più complicate di tutte: gestire non solo le collisioni con i fantasmi, ma anche i loro algoritmi e movimenti.
Prima di iniziare ho fatto il consueto refactoring di inizio nuovo task, questa volta seguendo il filone soddisfacente della pulizia del main: adesso il costruttore dello stato inizializza anche il vector di dots, diminuendo gli argomenti in ingresso della funzione **placeDots()** e facendola

ritornare l'array invece che posizionarlo direttamente all'interno dello stato passato per riferimento.
Con lo stesso spirito ho poi riunito le varie draw (che andavano aumentando) in un'unica funzione **drawAll()** mettendoci anche **createMap()** adesso rinominata in **drawMap()**, visto che effettivamente non creava nulla ma si occupava solo del disegno basandosi sulla matrice della mappa.
La gestione della collisione diretta con i fantasmi è stata quasi immediata: fortunatamente avevo già tutto.
Mi è bastato creare una funzione **ghostsCollisions()** analoga a **manageDots()** che sfruttasse **getGridCoords()** per verificare se pacman si trovi nella stessa casella di un fantasma e in tal caso impostare lo stato di gioco a **GAME_OVER**, che avevo già gestito nel capitolo precedente! Ho chiaramente aggiunto una chiamata a questa funzione in **checkCollisions()** e tutto è andato al proprio posto.
La mia ricerca per un codice efficiente ed elegante è andata avanti, per cui ho creato una funzione **doCollide()** che si occupa di stabilire se due sprites collidono (per semplificare le funzioni sopra), nello scriverla tuttavia mi son accorto che era necessaria una scelta: passando alla funzione solamente i due sprite in questione il codice sarebbe stato sicuramente decisamente elegante e chiaro dividendo bene i confini tra cosa fa una funzione e cosa fa l'altra, tuttavia così facendo, le coordinate di pacman verrebbero calcolate tante volte (una per fantasma + una per dot attivo) rendendo il codice meno efficiente del necessario.
Ho allora deciso di optare per un codice un po' meno pulito ma decisamente (spero) più efficiente, creando una seconda **doCollide()** che riceve come argomento un solo sprite e le coordinate di pacman, in modo da calcolarle un'unica volta in **checkCollisions()** e passarle a tutte le funzioni sotto.

Nell'iniziare a pensare a come implementare i fantasmi, mi sono presto accorto della necessità di differenziare di molto la struttura dei fantasmi da Character, ho quindi pensato di utilizzare il polimorfismo e dar loro una struttura che derivi da character.
Nel fare ciò mi sono però reso conto di ulteriori ottimizzazioni: **initializeGhost() e initializeEnemies()** adesso non erano più necessarie: bastava mettere i costruttori dedicati!
Una delle cose che mi gratifica di più in questo progetto è vedere come ogni step aggiuntivo facilita la scrittura del codice futuro e passato, ogni pezzo del puzzle se fatto bene aiuta anche a rendere pulito ed efficiente tutto il resto.

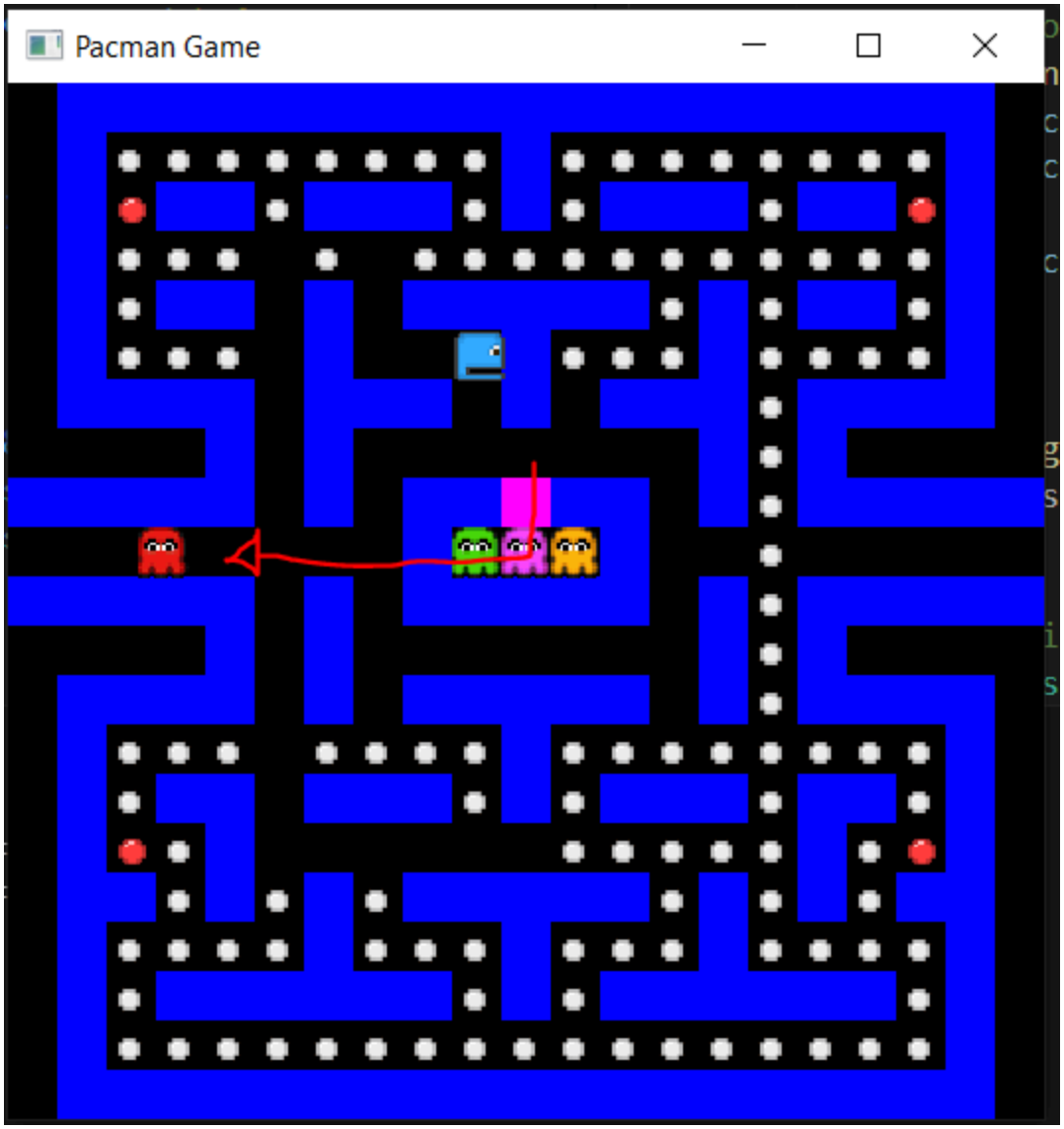
La fase successiva è stata difficile nello sviluppo e sarà difficile anche nella scrittura del resoconto, dal momento che i vari tentativi hanno preso diverso tempo ed è difficile trovare il momento giusto per documentarli: Se attendo troppo li dimentico -soprattutto essendo stati travagliati- ma attendendo poco non ho ancora la visione complessiva del modulo terminato.
Cercando di farla breve, per scrivere il movimento dei fantasmi ho deciso di iniziare da **Red** o "blinky" il cui algoritmo consiste "semplicemente" nel seguire il giocatore.

Mi sono ispirato ad una descrizione ad alto livello dell'algoritmo trovata su internet che essenzialmente controlla le direzioni possibili - quella di provenienza e ne calcolava la distanza tramite la formula **manhattanDistance** da pacman, scegliendo poi quella più vicina.
Un po' sospettoso di come ciò funzionasse ignorando i muri, ho deciso comunque di fare un tentativo, mi sono messo quindi a scrivere una -per ora generica- **moveGhost()** accompagnata da **getPossibileDirections()** e **manhattanDistance()**, ho poi iniziato a riusare il codice scritto precedentemente per quanto riguarda il controllo dei muri tramite **canWalk()** e… I nodi sono venuti al pettine.
Nel momento della scrittura di **canWalk()** ho infatti violato un principio di buona programmazione delle funzioni **"Have No Side Effects"**, infatti la funzione oltre a determinare se il personaggio in questione potesse camminare (non avendo quindi muri davanti) si occupava anche della traversata nel tunnel, al momento della scrittura mi sembrava il suo posto naturale (non esisteva ancora **checkCollisions()** e non avevo una funzione per ottenere le coordinate allineate alla griglia), però adesso faceva più cose di quelle che mi servivano.

```cpp
//Checks collision with walls and tunnel
bool canWalk(Character& character, const sf::Vector2f& direction) {

    sf::Vector2f nextPosition = character.sprite.getPosition() + (direction  * static_cast<float>(GRID_TILE));

    int row = static_cast<int>(nextPosition.y / GRID_TILE);
    int col = static_cast<int>(nextPosition.x / GRID_TILE);

    if ( row == 9 && (col == 0 || col == 20)) {     //Checks if pacman is in the tunnel
        character.traversingTunnel = true;
        return true;
    }

    return level[row][col] != 1; // 1 = wall
}
```

[Vecchia versione di canWalk(), presa dal backup "postGameover.cpp"]

Da qui è quindi iniziato un nuovo refactoring delle funzioni, dividendo meglio le responsabilità e rendendole più chiare e compatte.
Poi ho sistemato qualche errore di compilazione, runnato il programma e… ehy! Ma dove stai andando!?



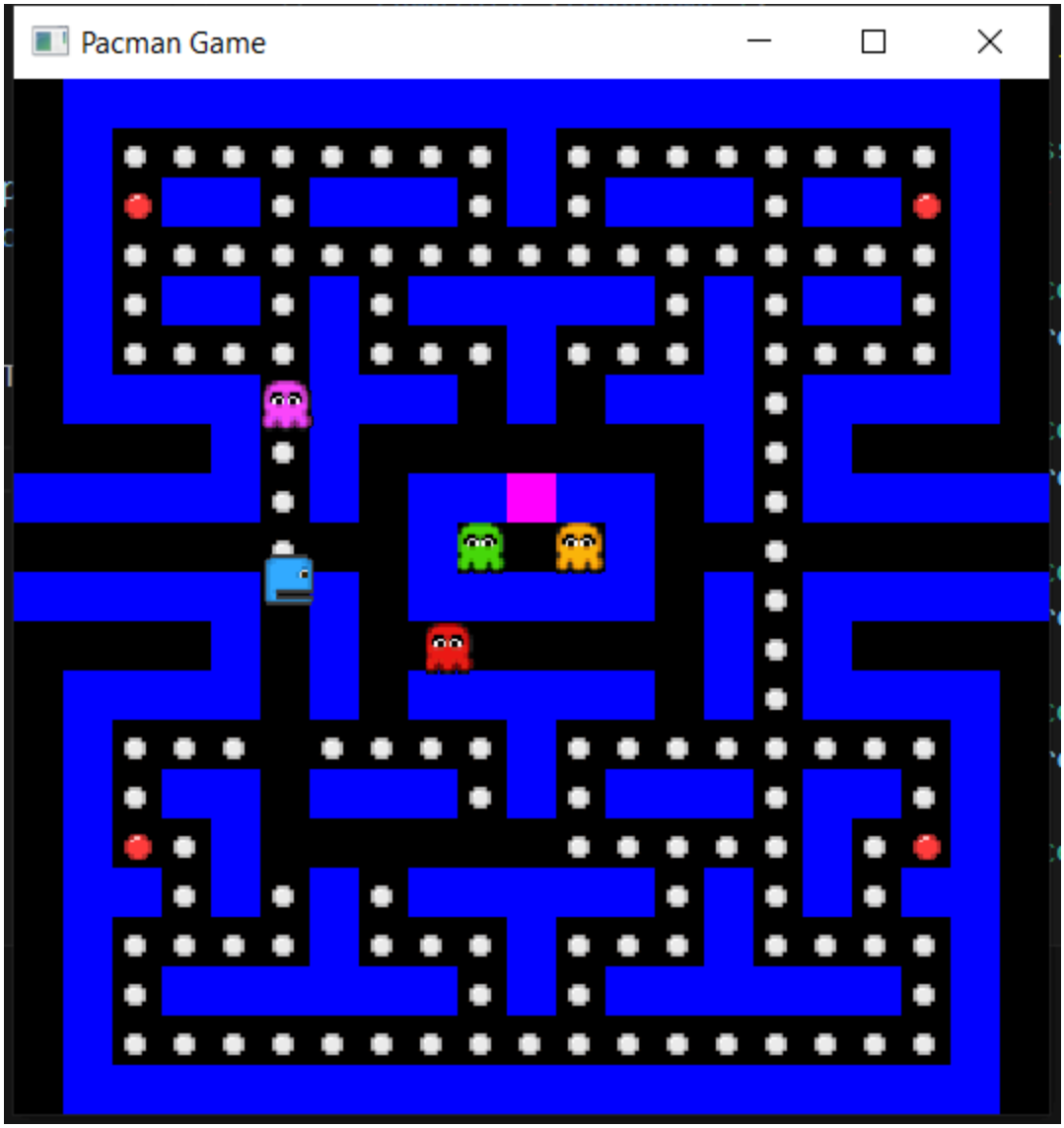
[Primo test del movimento dei fantasmi, zip backup: "movimentoPrimoFantasma.Zip"]

Per qualche motivo il fantasma si muoveva sempre seguendo quella freccia rossa disegnata ignorando la mia posizione, dopo qualche mezz'ora passata a cercare di sistemare ho interrotto la sessione e deciso che me ne sarei occupato un'altra volta.

Ripreso il lavoro, ho cercato di sistemare la funzione estraendo la scelta della direzione in **getGhostDirection()** e chiamandola solo quando fosse allineato alla griglia muovendolo sennò verso l'ultima direzione, ho inoltre messo la direzione di default a **RIGHT** invece che **NO_DIRECTION**, fatti questi cambiamenti il fantasma ha iniziato a muoversi, ma non ancora secondo il ragionamento desiderato. [movimentoRedSbagliato.zip]

Dopo qualche altro tentativo mi sono arreso ad usare qualche algoritmo di pathfinding online, in particolare ho valutato **A*** che però continua ad usare un'euristica e la **manhattanDistance** non mi convinceva visto che non teneva conto dei muri presenti sulla mappa.
Allora ho optato per un **BFS** che precomputa una mappa per scegliere la direzione.
Adesso finalmente **Red** (o blinky, non so ancora quale nome rendere ufficiale) si muove come desiderato! [movimentoGiustoRed.zip]
Prima di procedere con gli altri fantasmi ho ritenuto opportuno assicurarmi che tutto funzionasse davvero, in particolare rendere il fantasma in grado di muoversi correttamente tramite il **tunnel**, includerlo nei reset.
Innanzitutto ho aggiunto due if all'interno della **BFS** per tenere conto del **tunnel**, poi ho sistemato manageTunnel in modo da togliere l'appoggio al field **isTraversingTunnel** e dopo ho aggiunto il controller del tunnel al movimento dei fantasmi e li ho aggiunti alla funzione di reset.
A questo punto ho pensato che sia per il **gameplay** sia per il **testing** potrebbe essere comodo poter resettare la partita a piacimento, ho quindi deciso che il tasto **R** possa sempre servire a tale scopo e fatto le necessarie modifiche per permetterlo.

Sistemato queste cose ho creato una nuova funzione **manageEnemies()** per gestire tutti i fantasmi e ho implementato **Pinky**: il suo algoritmo è del tutto simile a quello di **Red** solo che con un target diverso: **Pinky** cerca infatti di posizionarsi davanti a Pacman, ovvero 4 caselle in avanti.
Dopo aver fatto un primo tentativo in cui applicavo direttamente questa logica mi sono accorto che il programma andava in crash quando mi avvicinavo ai bordi, ho allora capito che dovevo verificare il target calcolato fosse ammissibile.
Implementando questo accorgimento in una funzione a parte **getPinkTarget()** il movimento sembrava funzionare, per capire se scartare anche le posizioni occupate da muri ho giocato un po' a qualche versione di **pacman** trovata su internet e poi ho fatto dei test sul mio programma rendendo vera la velocità dei fantasmi.
Provando il codice sia gestendo il caso dei muri sia non, mi è sembrato che si muovesse in modo un po' più intelligente senza considerare i muri, ho quindi deciso di tenere questa versione del codice e ho salvato il backup come [movimentoPinky.zip]



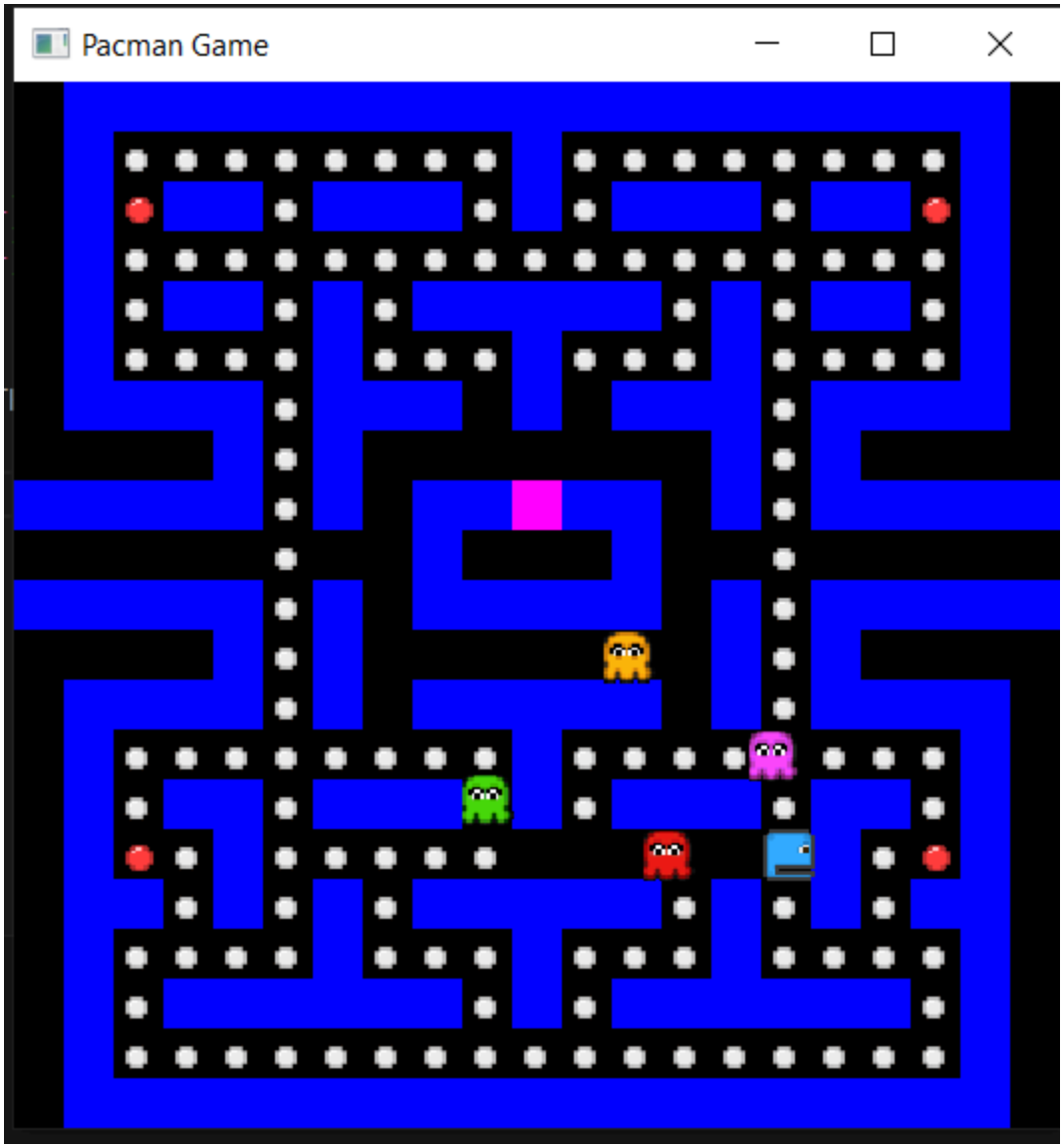
[Red e Pink funzionanti durante l'inseguimento]

Le iterazioni seguenti hanno avuto l'obiettivo di implementare anche Green e Yellow.
Il ragionamento alla base è che continuino ad usare **moveGhost()** passandogli un target diverso.
Nel caso di **Orange** questo target è pacman o una casella in basso a sinistra a seconda della vicinanza, in modo da simulare il suo comportamento timido che scappa quando troppo vicino a pacman. [movimentoOrange.zip]
Nel caso di **Green** invece è lievemente più complesso: essenzialmente si simula una coordinazione con **Red**, cercando di predire dove Pacman andrà per scappare da **Red** ed impostandolo come obiettivo. [movimentoGreen.zip]
Ho poi sistemato sia **Green** che **Pink** in modo da finalizzare l'inseguimento a pacman in caso siano sufficientemente vicini, per sistemare la situazione nel quale si tenevano sempre distanti.

A questo punto il gioco è effettivamente giocabile, rimangono solo da raffinare collisioni con i fantasmi ed il loro movimento e aggiungere il funzionamento dei SuperDot.

C) Raffinamento

L'obiettivo di questa fase è stata ripulire quanto fatto nell'ultima parte e rendere l'interazione con i fantasmi un po' più pulita.
La prima cosa che ho sistemato è stata la collisione con i fantasmi, essa infatti usava la stessa funzione **doCollide()** dei dot, che però erano statici e avevano maggiori problemi di intersezione fra loro ed efficienza, questo si manifestava rendendo talvolta possibile schivare i fantasmi passandogli attraverso a seconda dell'allineamento tra i personaggi e la griglia.
Ho quindi creato una nuova funzione **doesGhostCollide()** che invece di controllare le posizioni allineate alla griglia le controlla lì dove sono (.**getPosition()**) ed effettua dei confronti fra floating point, l'ho sostituita in **checkCollisions()** e si è subito sistemato il problema.
Ho notato che era leggermente troppo punitiva la detezione iniziale, di conseguenza ho creato una nuova costante che diminuisse leggermente il range di collisione per evitare che i fantasmi catturassero pacman attraverso gli angoli.
Poi ho deciso di migliorare gli algoritmi getTarget di **Pink** e **Green**, questi algoritmi infatti, dopo aver stabilito il target iniziale, verificavano se esso si trovasse fuori dai confini (per evitare errori e crash) e in quel caso resituivano direttamente il target di pacman, rendendo il loro movimento in quei frangenti uguale a quello di **Red**.
Questo metodo non mi soddisfaceva e differiva anche da quello originale, ho deciso allora di inserire una sottofunzione che in quel caso cercasse un punto alternativo laterale.
Ho anche spostato il controllo di validità di una casella in una funzione apposita e inserito come tipo di casella quella irraggiungibile (quelle vuote fuori dalla mappa), questo cambiamento ha sistemato il fatto che a volte il fantasma si fermasse quando pacman cercava di raggiungere una casella fuori dalla mappa.
Infine ho sistemato i controlli di getNextStepFromMap che non facevano funzionare il tunnel.

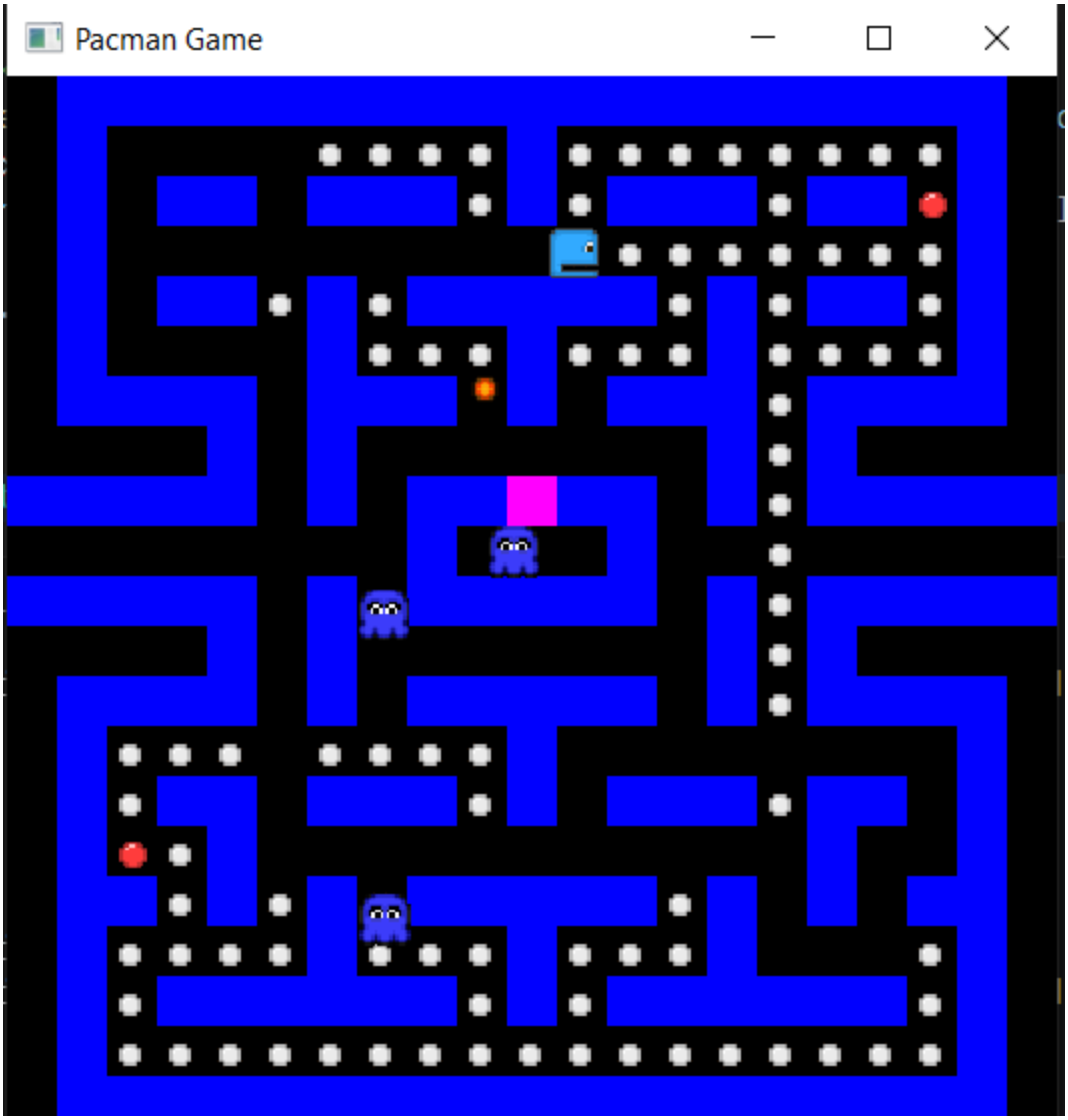


[Tutti i fantasmi in moto]
[raffinamentoFantasmi.zip]

VII) I super dot

Adesso che i fantasmi si muovono intelligentemente, il gioco è un po' troppo difficile!
È necessario dare a pacman il suo unico strumento di difesa: I super dot.
Per fare questo ho aggiunto un campo booleano a **state** isPacmanSuper e creato funzioni per assegnare questo stato (insieme al mood **FRIGHTENED** ai fanstmi) alla cattura di un super dot.
Per far durare l'effetto solo un tempo limitato (sei secondi) ho aggiunto un altro clock allo stato che viene resettato ogni volta che si raccoglie un super dot, una funzione **isSuper()** controlla poi se pacman è super quanto tempo è passato dall'ultimo reset e nel caso esso sia maggiore di 6 secondi, disabilita questo campo.

Per far scappare i fantasmi quando spaventati ho aggiunto due funzioni **moveScaredGhost()** e **chooseScaredDirection()**, chiamate poi all'interno di **manageEnemies()**, queste funzioni fanno si che se i fantasmi sono spaventati si muovono alla metà della velocità scegliendo una direzione casuale tra quelle valide ad ogni casella.
A questo punto mi sono chiesto come rappresentare la fuga una volta catturati, guardando il foglio di texture a disposizione ho pensato di rappresentare quelli che nel gioco originale son gli occhi dei fantasmi come un orb rosso, ho aggiunto allora lo stato **GhostMood::ORB**.
Ho aggiunto dei metodi alla struct **Ghost** in modo da poter gestire in modo sicuro il cambio di mood, così che una volta finito il tempo di invincibilità di pacman venga cambiato lo stato solo ai fantasmi che non stanno raggiungendo la loro casella di partenza.
Ho quindi gestito i nuovi casi di collisione (creando nuove sottofunzioni per rendere tutto ordinato e leggibile) e i nuovi passaggi di texture, facendo qualche test mi sono poi preso la libertà di cambiare la durata dello stato super rispetto al gioco originale: 10 secondi invece che 6, il gioco mi pare comunque sufficientemente difficile.
Infine per aggiustare la difficoltà del gioco ho fatto in modo che -come nella versione originale- **Orange** esca solo dopo che sono stati mangiati i primi 30 dots e **Green** dopo i primi 60.
Ho implementato ciò semplicemente controllando la grandezza del vettore contenente i dots nella funzione **manageEnemies()** (che è la nuova funzione che fa la prima gestione dei fantasmi) e a prescindere dal loro stato, se quel numero di dots non è ancora stato mangiato li fa muovere all'interno della loro base.



[Fantasmi spaventati, Orange torna alla base, Green ancora inattivo]
[superDots.zip]

VIII) Animazioni

Il gioco si può vincere e perdere, tutte le interazioni funzionano… però manca qualcosa.
Certo, il gioco è troppo statico: servono delle **animazioni**!

Fin dall'inizio, guardando il foglio di texture trovato sapevo che sarebbe giunto questo momento, la buona notizia è che me lo immaginavo molto più doloroso di quanto fosse davvero.
Ho deciso di iniziare dalle cose "semplici": le **animazioni** dei **fantasmi**.
Per fare ordine ho deciso di spostare le costanti globali riguardanti texture in un file a parte vista l'imminente crescita di righe a ciò dedicate, poi ho preso il foglio di texture e segnato le coordinate corrispondenti ai vari stati del movimento dei fantasmi (LEFT, RIGHT, DOWN e UP)

ed in seguito ho creato per ogni fantasma una funzione che ritornasse l'animazione corretta (prendendo dalla sua struct la direzione).
Poi in un'altra funzione **animateGhost()** ho inserito una serie di if che controllano tipo di fantasma e stato e gli assegnano la texture corrispondente, questa funzione viene chiamata quattro volte da **animateEnemies()**, passandogli ogni volta un fantasma diverso.
Ero parecchio in dubbio su come dividere queste funzioni qui e credo che molti altri modi fossero di simile qualità, mi sono fermato a questo perché mi è parso il più intuitivo.

[Fantasmi in movimento animato]
[siMuovonoTutti.zip]

Per animare **pacman** ho proceduto in maniera analoga, ma questa volta con… 8 volte tante texture.

Dopo aver selezionato e testato i ritagli (con qualche lieve imperfezione forse, ma contare i pixel di imprecisione alla lunga è stucchevole), ho creato tre funzioni: **animatePacmanDirection(), animateSuperPacmanDirection()** e **animatePacmanMovement()**, le prime due impostano un nuovo campo **pacmanTextureVector** nella struct **state** ad un vector contenente i 4 frame dell'animazione della direzione in cui sta andando a seconda della stessa, mentre l'ultima funzione itera sui 4 frame basandosi sul clock **superClock** (che a differenza di **clock**, non viene resettato ogni iterazione del **Game Loop**).
Per regolare la velocità di animazione ho anche aggiunto due variabili globali in **textures.hpp**, una che dice semplicemente da quanti frame è composta l'animazione (4) e l'altra che regola la velocità, il compromesso migliore che ho trovato è 150 millisecondi.
Allegerei screenshot ma non credo l'animazione risulti molto, sappiate però che a mio parere è **molto bella.** [animazioniPacman.zip]

IX) Musica e suoni

Il gioco adesso sembra molto più vivo!
Però manca un ultimo tocco: il suono.

Vista la lunghezza del racconto fino ad adesso, qui cercherò di essere piuttosto breve.
Per implementare musica e suoni ho incluso la libreria **Audio** di SFML ed utilizzato le classi **sf::Sound** e **sf::Music**, esse sono lievemente diverse in quanto music carica il suono poco alla volta e permette una gestione più appropriata.
Ho aggiunto alle struct un **SoundManager** ed una sua istanza a **State**, in modo da avere un posto unico in cui gestire i suoni, inoltre per il cambio di musica ho creato la funzione **setMusic()**.
Il funzionamento di queste cose è proprio come ce lo si aspetta: si carica il brano musicale tramite path, durante le collisioni o i cambi di stato si chiama **state.soundManager.nomeSuono.play()** o si assegna la musica corretta tramite **setMusic()**.
Dopo aver scelto i suoni giusti e regolato un po' i volumi, il feel è migliorato notevolmente, anche se dal punto di vista del gameplay l'unica cosa che cambia è che adesso si ha un'idea migliore della durata dello stato super di **Pacman**.
[suoni.zip]

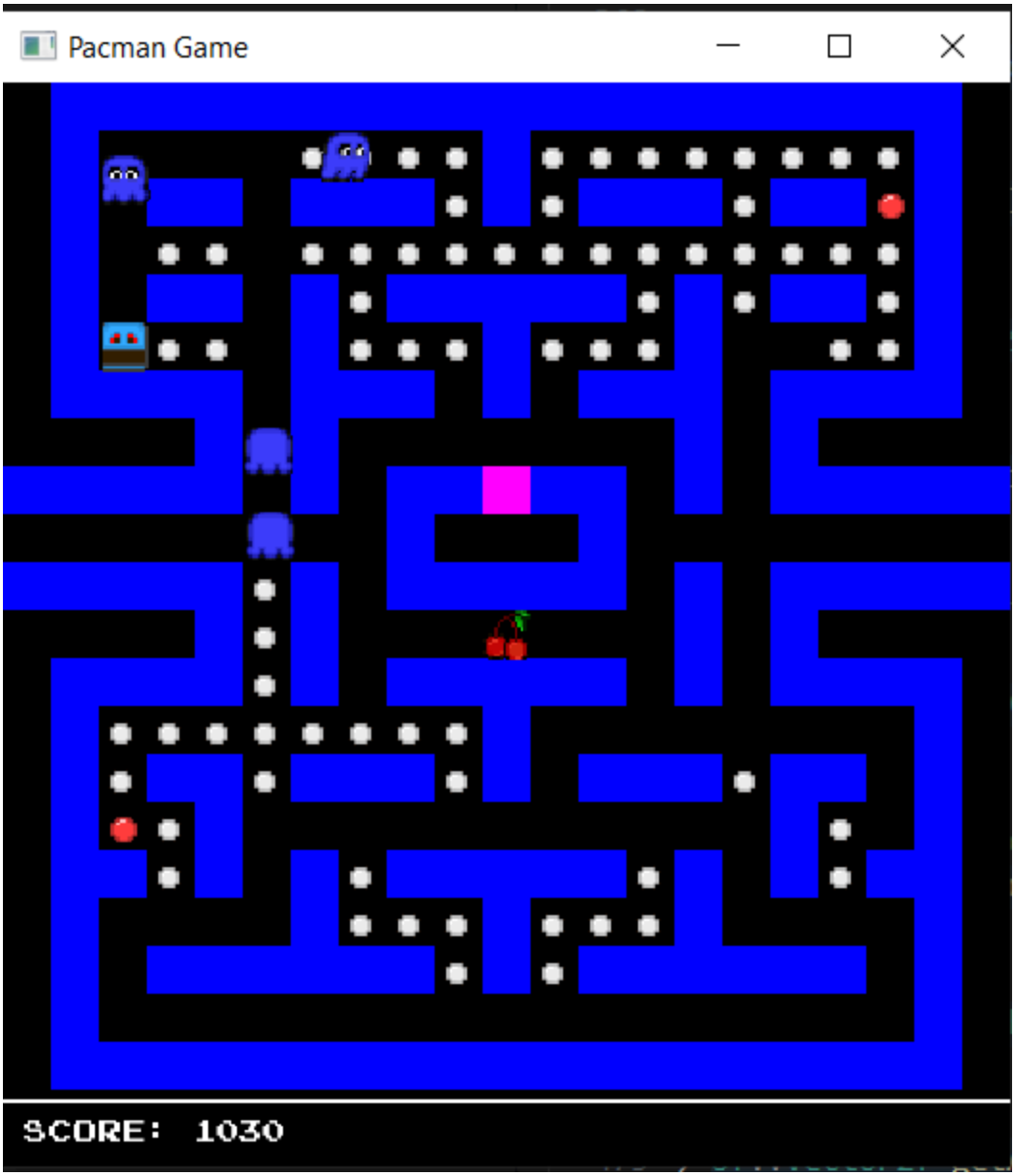
Ispirato dal libro **Clean Code** ho poi rivalutato le varie scelti di **struct** (paradigma **procedurale**) vs **class** (paradigma **OO**), da questa rivalutazione è emerso che la scelta di utilizzare struct per la maggior parte dei casi fosse la migliore visto che ci interessa poter manipolare i dati della struttura all'esterno e vogliamo poter aggiungere e modificare funzioni più facilmente rispetto agli oggetti stessi.
Questo ragionamento però non era vero per l'appena aggiunta **SoundManager,** i cui campi vengono acceduti quasi esclusivamente in fase di costruzione, mentre veniva spesso chiamato un loro metodo (di cui dall'esterno ci interessa solo dell'utilità), ho quindi deciso di trasformarla in

una classe aggiungendo metodi per interagire con i vari suoni e uno per cambiare la musica di sfondo, ottenuto spostando all'interno della classe la funzione **setMusic()**.

X) Il punteggio

Il gioco funziona e ha un ottimo **feel** però solo sopravvivere non è sufficientemente divertente: serve un punteggio.
L'idea alla base è piuttosto semplice: restando fedeli ai valori originali di pacman, aggiungiamo una variabile mostrata sempre a schermo che incrementa quando vengono catturati **dot** o **fantasmi**.
Per realizzarlo ho aggiunto il campo **score** allo **State** ed un enum per i diversi incrementi, nei diversi punti in cui si controllano le collisioni ho incrementato lo **score** dei punti indicati dagli enum, mentre per i **fantasmi** ho tenuto in conto anche una seconda variabile **killStreak** che tiene conto di quanti fantasmi di fila sono stati catturati.
Entrambe queste variabili si resettano insieme al resto del gioco, con la **killStreak** che si resetta anche al termine della durata di una super.
Per mostrare il punteggio ho prima dovuto aggiustare nelle costanti globali le dimensioni della finestra per lasciargli spazio e poi aggiunto una funzione **drawScore()** chiamata da **drawAll**() che scrive il valore -con il font già in uso per i **menù**- in fondo allo schermo.
Ho scelto di metterlo in fondo in modo da non dover interferire da come viene disegnato il resto della mappa, inoltre stilisticamente mi piace.
Per delimitare la zona ho inoltre disegnato una linea bianca utilizzando una semplice **RectangleShape.**
[punteggio.zip]

La prima immagine che quando ho pensato al "**punteggio** in pacman" mi è venuta in mente è quella della **ciliegina** al centro della mappa, non appena ho reso funzionante il punteggio ho quindi deciso di aggiungerla.
L'idea alla base qui è di avere il **frutto** che compare quando si sono appena raccolti 70 o 100 punti e renderlo disponibile per poco tempo grazie ad un **clock**, garantendo 300 punti al giocatore se riesce a raccoglierlo.
Per implementarla ho dovuto aggiungere a **State** un **fruitClock** e una nuova struct **Fruit** che di base è poco più di un wrapper di **sprite**: l' unica informazione aggiuntiva che contiene è un campo booleano **isActive.**
Tralasciando la definizione delle varie costanti, le funzioni che ho dovuto aggiungere sono state **spawnFruit()** e **checkFruitDespawn()** per attivare e disattivare il frutto, mentre per la seconda nella funzione si verificano già le condizioni la prima viene chiamata da **manageDots()** quando si catturano il settantesimo ed il centesimo **dot**.
Ho poi aggiunto una funzione **fruitCollision()** per controllare e gestire la collisione con il frutto, facendola poi chiamare da **checkCollision()**.



[Punteggio funzionante e frutta in campo]

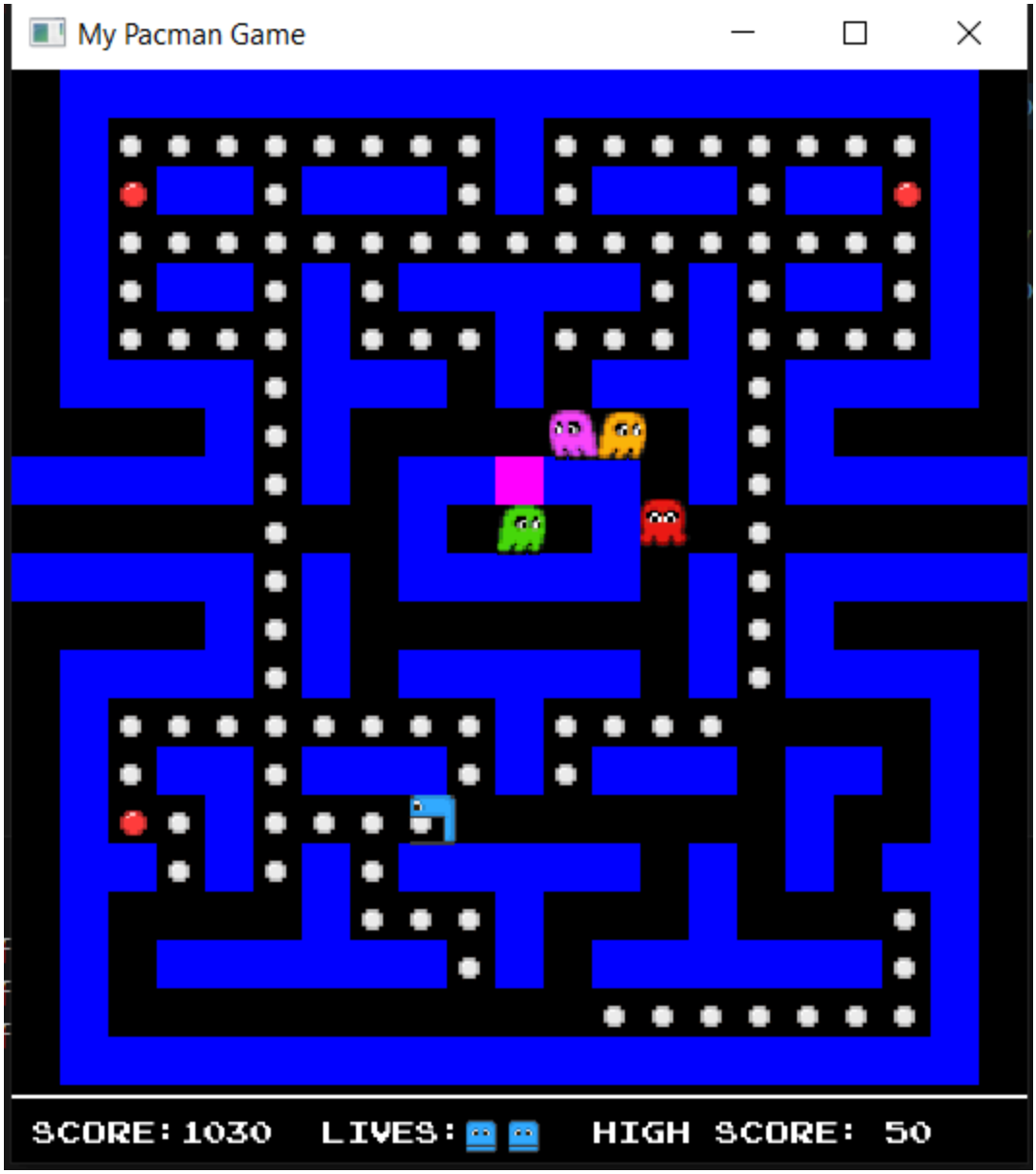
[frutta.zip]

Ma che senso ha vedere il punteggio se non si ha il confronto con il massimo raggiunto? Chiaramente, serve avere un riferimento per l' **highscore**.

Per ottenerlo ho aggiunto un campo **highscore** a state e le righe necessarie per disegnarlo sulla riga di quanto fatto prima, per salvarlo su **file** ho incluso <fstream> e creato due funzioni **saveHighscore()** e **loadHighscore()** per le interazioni con il **file**, queste funzioni vengono chiamate in **setGameOver()** e **setGameWon()** (quest'ultima funzione mancava e l'ho aggiunta spostando le linee per impostare lo stato di gioco.

XI) Le vite

Il gioco funziona, l'**interfaccia** tiene conto dei **punti** ma…
Ehy, è parecchio **difficile** questo gioco!
Nei **playtest** fatti da me e persone strette risulta che effettivamente raggiungere la schermata di vittoria non è per niente semplice ed essendo un parere così diffuso non potevo semplicemente dire che "siamo scarsi".
A confermare questo parere, dal confronto tra il mio gioco è il classico pacman emerge una mancanza: **le vite.**
Avere **tre tentativi** invece di uno solo è il tocco che manca per poter divertirsi appieno con questo progetto e le ultime modifiche fatte all'**interfaccia** generano proprio uno spazio perfetto in cui inserirle.
L'implementazione è abbastanza intuitiva: Ho aggiunto un campo intero **lives** a **State** che indica le vite correnti, una funzione **drawLives()** che disegni il numero giusto di icone di pacman più piccole in fondo allo schermo per indicare le vite rimaste e poi ho ritoccato la gestione delle collisioni con i fantasmi.
Ho infatti sostituito il **gameOver** diretto al contatto con i fantasmi con una funzione **damagePacman()** che diminuisca le vite o se esse sono esaurite cambi lo stato di gioco.
Al fine di poter far ripartire la partita correttamente però, è stato necessario creare una nuova funzione **respawn()** che esegua solo alcune delle line di **resetGame()**, in modo da poter riposizionare i **personaggi** mantenendo però il resto del gioco nello stato attuale.



[Highscore e una vita consumata]
[highScore_vite.zip]

XII) Pulizie di primavera

Okay, sembra davvero di essere quasi arrivati alla fine.
Nel corso dello sviluppo di questo codice ho sempre cercato di scrivere codice molto pulito effettuando refactoring prima e dopo ogni nuova implementazione, tuttavia temo di aver preso qualche scorciatoia ogni tanto, magari per finire il task corrente prima di mettere la mia testa su altro.
Di conseguenza, siamo arrivati al momento delle **pulizie.**

La prima cosa che ho sistemato è stata proprio riguardo **restartGame**() e **respawn**() trattate nello scorso capitolo: sono praticamente dei ri-costruttori eppure sono lì da soli tra le funzioni, ciò è **brutto**.
La loro casa appropriata era chiaramente dentro a **State**, così li ho spostati lì dentro.
Nota bene: Molte, forse quasi tutte le funzioni che ho scritto potevano essere considerati dei **metodi** delle strutture di gioco, ma questo significherebbe un cambio totale di paradigma di programmazione che -fra l'altro- differirebbe da quanto insegnato durante **FCG**, di conseguenza mi sono votato a spostare solo le funzioni che davvero aveva più senso mettere all'interno di una struttura, ricordando il principio che avere **strutture** (paradigma **procedurale**) invece che **classi** (paradigma **OO**) rende più facile l'accesso ai dati e l'aggiunta/modifica di funzioni.
Ometterò varie modifiche minori da questo racconto ormai troppo lungo, fra le cose che vale la pena menzionare c'è però lo spostamento delle linee che muovevano pacman fuori dal main: non so perchè avessi così tanta paura a spostarle .

```
93  int main() {
94      sf::RenderWindow window(sf::VideoMode({WINDOW_WIDTH, WINDOW_HEIGHT}), "My Pacman Game");
95      sf::Texture mainTexture;
96      if (!mainTexture.loadFromFile("src/progetto/RisorseProgetto/pacmanSprites.png"))  return -1;
97
98      window.setFramerateLimit(FRAME_RATE);
99
100     State state(mainTexture);
101     state.soundManager.setMusic(MENU_THEME);
102
103     //  ---   GAME LOOP   ---   //
104     while (window.isOpen()) {
105         window.handleEvents(
106             [&window](const sf::Event::Closed&) { handle_close(window); },
107             [&window](const sf::Event::Resized& event) { handle_resize(event, window); },
108             [&state, &window](const auto& event) { handle(event, state); }
109         );
110
111         window.clear();
112
113         if(state.gameState != GameState::PLAYING) {
114             drawMenu(window, state);
115
116             window.display();
117             continue; //If the game is not playing, skip the rest of the loop
118         }
119
120         float deltaTime = state.clock.restart().asSeconds();
121         managePacman(state, deltaTime);
122         manageEnemies(state, deltaTime);
123
124         drawAll(window, state);
125         window.display();
126     }
127
128     return 0;
129 }
```

Adesso finalmente il **main** sta in un'unica inquadratura.

Nel rileggere il codice ho ridotto di 0.5 l'**epsylon** di imprecisione permessa nel confronto fra float per **isAlignedToGrid**(), il che ha risolto uno dei due principali -lievi ma riscontrabili- problemi presenti nel gioco: le collisioni con i fantasmi troppo imprecise.
Risolto uno, ho deciso di affrontare anche l'altro: Ogni tanto i fantasmi parevano bloccarsi per un po' prima di attraversare il tunnel invertendo ripetutamente direzione.
Indagando sulla sorgente del problema mi sono ricordato della mia funzione apposita per evitare i cambi repentini di direzione **getPossibileDirections()** e mi sono chiesto perchè in tal caso non funzionasse.
La risposta era semplice: non veniva chiamata sempre, in particolare in **getNextStepFromMap()** non poteva essere chiamata a causa di una differenza di tipi.
Ho allora creato al volo la versione adatta al tipo **getPossibileIntDirections()** e applicata alla funzione in questione, questo ha risolto il **bug** dell'attraversamento del tunnel.
[fixTunnel.zip]

Altre migliorie significative ma che non spiego nel dettaglio sono state:
Aggiunta di: initalizeDot(), enableSuper(), disableSuper(), killGhost().
Spostamento calcolo pacmanCords da argomento di manageDots a prima riga della stessa (prima era fuori in quanto veniva utilizzato anche per i fantasmi)
Riordino funzioni per consentire un flusso di lettura più scorrevole.
[funzioniRiordinatel.zip]

XII) I livelli

Avendo completato il gioco in lieve anticipo rispetto alla scadenza del progetto, ho pensato di aggiungere i **livelli** per completezza.
L'idea base è piuttosto semplice: un campo intero all'interno di **State** che ci dica a che livello siamo, del testo a schermo che ci comunichi tale informazione, il **gameWon** che incrementi tale livello e lievi modifiche nelle variabili durante il gioco nei livelli successivi.
Ho iniziato dalle cose visibili e più facili da testare: Ho aggiunto il campo e disegnato la scritta, per quest'ultima ho dovuto pensare un po' a dove metterla visto che la scoreboard era già troppo affollata (ho deciso anche di ridurre la dimensione del testo per renderlo leggibile).
Non gradendo l'aspetto della scoreboard allungata ho pensato di scrivere il livello corrente semplicemente in cima **sopra al muro blu**, un luogo non fastidioso che sia ben visibile senza dover complicare l'interfaccia, ho aggiunto la funzione **drawLevel()** e tutto è subito andato per il verso giusto.
Dopo ho aggiunto la scritta per i livello nei menù, già che stavo lavorando fra i menù ho deciso di aggiustare posizioni, dimensioni e aggiungere anche i ringraziamenti: il tutto si è concretizzato con -oltre alle solite variabili globali- le funzioni **drawThanks()** e **drawNewLevel().**
Per i modificatori mi sono documentato online e ho valutato diverse opzioni usate i diversi cloni di pacman, alla fine ho optato per creare delle formule io.

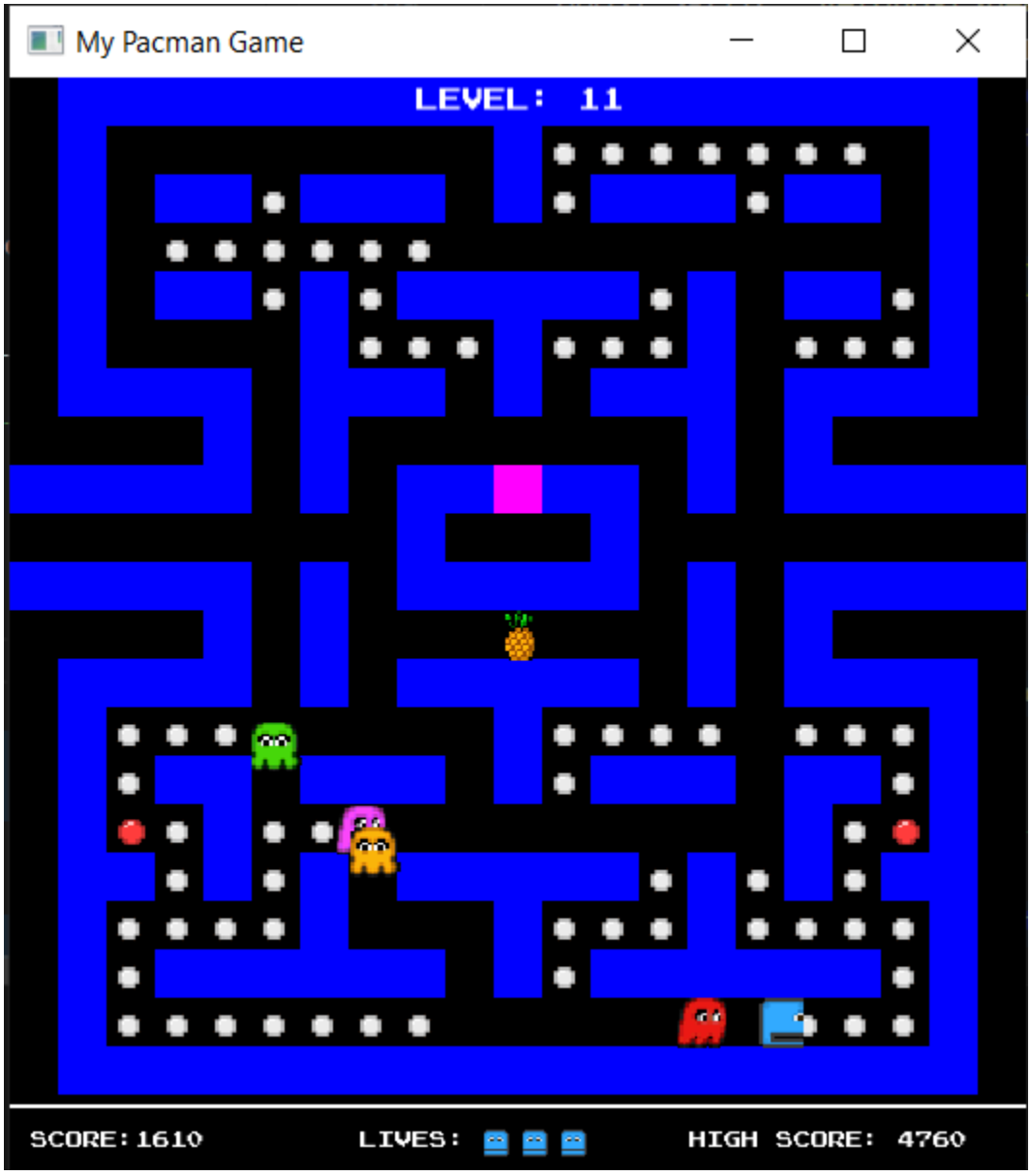
L’obbiettivo è che i fantasmi diventino sempre più veloci e la super duri sempre meno, ho allora impostato due funzioni **getGhostSpeed**() e **getSuperTime**() per ottenere questo risultato, in particolare avendo la **velocità dei fantasmi** che raggiunga quella di pacman al livello 5 e che acceleri sempre fino ad arrivare a 91 al livello 21 (livello “hardcore” ideale, oltre al quale non diventa più difficile), mentre la **durata della super** diminuisce di 0.42 secondi a livello fino ad arrivare a 0 dal 21 in poi.
Per impostare nel gioco queste modifiche ho aggiunto una funzione **startGame**() chiamata all’avvio del gioco tramite tasto **Enter**, questa funzione resetta il gioco mantenendo il livello e poi applica le modifiche tramite **levelSettings**().
Alcune versioni del gioco - tra cui credo quella originale- al salire dei livelli cambiano anche il movimento dei fantasmi mandandoli talvolta verso un angolo della mappa, l’idea non mi piaceva tanto di conseguenza ho deciso di **scartarla**.
Allego screenshot per chiarezza.

```
void levelSettings(State &state){
    GHOST_SPEED = getGhostSpeed(state.level);
    SCARED_SPEED = GHOST_SPEED / 2;
    SUPER_TIME = getSuperTime(state.level);
}

float getGhostSpeed(int level) {
    if(level >= 1 && level <= 5){ //Reach 80 in level 5, gradually increasing speed
        return 70.0f + level * 2.0f;
    }
    if(level >5 && level <= 20){   //Reach 90 in level 20, gradually increasing speed
        return 80.0f + level * 0.5f;
    }
    return 91.0f;
}

float getSuperTime(int level) {
    //Supertime drecreases with the level, reaching 1 second in level 20
    //After 21 it stays at 0 seconds
    if(level > 21) return 0.0f;
    return SUPER_TIME - (level - 1) * 0.42f;
}
```

[livelli.zip]

A questo punto ho ritoccato anche il funzionamento dei **frutti**: più si va avanti, più i frutti danno punti e cambiano di aspetto.
Ho operato in modo del tutto simile a prima, andando prima a sezionare le texture per i nuovi frutti e poi a creare due funzioni **getFruitPoints()** e **getFruitTextureRect()** che dato il livello ritornano l’ammontare di **punti** del frutto o la sua **texture**, la prima viene chiamata al momento

della **raccolta** del frutto per l'assegnazione dei punti, la seconda al momento del **passaggio** di livello in **levelSettings()** per impostare la texture corretta.
Come ultimo accorgimento, ho aggiunto a **startGame()** due linee per tenere non solo il **livello** quando si iniziano partite successive ad una vittoria ma anche il **punteggio**.



[Interfaccia con il livello 11 e ananas in gioco]
[diversaFrutta.zip]

Infine mi sono dedicato ad un po' di refactoring post aggiunta, salto le modifiche meno importanti ma mi soffermo su come ho cambiato la gestione del **restart** del gioco.
Prima era gestito in ogni caso da **restartGame()** che reimpostava tutti i campi della struttura al loro valore inziare, questa gestione ha iniziato a smettere di andare bene con l' aggiunta di **vite** e **highScore** ed è ulteriormente peggiorata con i **livelli**.

Infatti in questo punto (quasi finale) dello sviluppo del gioco ci sono effettivamente due possibili situazioni di ripartenza: Il **reset completo**, desiderato nel caso si **perda** la partita o si resetti volutamente premendo **R** e il **reset parziale,** desiderato nel caso si completi il livello e ammesso nel caso si inizi la prima partita in cui **livello**, **vite** e **punteggio** vengono mantenute.
Il secondo caso prima era gestito brutalmente salvandosi i valori delle tre variabili prima di effettuare il reset e poi reimpostarle, però non era fluida l' assegnazione di questo funzionamento solo ai casi di reset parziale.
Per risolvere questa cosa ho aggiunto un ulteriore metodo a State ovvero **setNewLevel()** che è essenzialmente un reset che non reimposta quelle tre variabili, a questo punto quando si preme **Enter** per uscire dalla schermata di **GameOver** viene chiamato il reset totale della partita, mentre altrimenti viene chiamato il reset parziale.
Sono ancora indeciso se cambiare i nomi di queste due funzioni in **partialReset()** e **totalReset()** che specificano meglio cosa avviene al loro interno ma meno il loro utilizzo.

Conclusione

Sviluppare Pacman **mi è piaciuto**, affrontare da solo un progetto unicamente sullo sviluppo software mi ha confermato ancora una volta che **sono nel posto giusto** e che scrivere codice è una delle cose che più mi piace, fra le linee di vsCode mi sento a **casa**.
Usare SFML all'inizio è stato **spaventoso**: tante cose nuove dall'intuzione non facile e pressochè nessun'aiuto su come usarle, tuttavia dopo aver sperimentato e manipolato i vari elementi alla base mi sono sentito veramente **padrone del mio progetto**, libero di sfruttare i pochi pilastri alla base per dare vita a cose più complesse.
Su **SFML** ogni nuova piccola feature è un **problema** da affrontare, ma nonostante sia impegnativo è s**empre possibile** trovare una strada con gli strumenti appresi, mentre su **Unity** -altro supporto per videogiochi con cui avevo esperienza, anche se **Engine** invece che **libreria**- le piccole aggiunte sono pressochè immediate e facili da regolare, ma le aggiunte più corpose richiedono di imparare un nuovo frammento dell'immenso framework.
Secondo me questi due approcci **diversi** hanno entrambi il loro fascino, credo SFML -nonstante non sia competitivo in se per se- mi abbia dato l'idea di cosa significhi sviluppare con un Engine un po' più basso livello o meno attrezzato (tipo **Godot**).
I punti più critici della libreria sono:

- Gli **aggiustamenti** precisi, per il quale non c'è un'interfaccia grafica o un modo ottimale di testare (ad esempio il numero migliore da assegnare per costanti di velocità o dimensioni di sprite).
- Ciò che esula dalla programmazione: come installazione, build o linking statico *(per fare una build eseguibile da chi non ha installati i tool)*, in queste cose è presente poca documentazione e non di facile comprensione.

Per quello che è l'obbiettivo del progetto, credo mi abbia preso un po' più tempo del "dovuto" ma mi abbia ripagato con **molta soddisfazione**.
Il fascino di iniziare veramente da **zero** ed arrivare ad avere un proprio gioco funzionante e fatto secondo i propri gusti e con le proprie scelte per me è stato tanto.