Esercitazione guidata 1-3/4/2025

Sviluppo di un semplice videogioco

In questa esercitazione svilupperemo da zero un videogioco ispirato a Breakout [Atari, 1976] e Arkanoid [Taito, 1986]. Il laboratorio si articola in quattro parti, ciascuna ulteriormente suddivisa in tappe, come specificato nel seguito.

Ringraziamento: codice sviluppato da Luigi Rocca - IGAG-CNR

Parte 1: struttura di base del campo

Il campo di gioco consiste di una finestra 800x600. Il file `00_hello.cpp` fornisce il codice di base che istanzia tale finestra, le callback per gestirne il resize (con proporzione bloccata) e la chiusura e il loop di interazione.

1. Creiamo lo stato del programma e le primitive di base per racchetta a pallina:
   - aggiungiamo alle costanti globali `ball_radius = 10.0` e `paddle_size = {100.0, 16.0}` per fissare le dimensioni di pallina e racchetta
   - creiamo tre struct: una per `Paddle` la racchetta, una `Ball` per la pallina e una `State` per lo stato; `Paddle` e `Ball` incorporano dimensione e posizione delle rispettive shape, mentre `State` incorpora una racchetta e una pallina; tutte e tre le strutture forniscono un costruttore di default e un metodo di draw che prende in input la finestra.
   - nei costruttori di `Paddle` e `Ball` istanziamo le primitive ponendo la racchetta al margine inferiore della finestra, al centro; la pallina andrà posizionata sopra la racchetta, in posizione centrale; il costrutture di `State` per ora è vuoto. La funzione `draw` di `Paddle` usa la `draw` di `sf::RenderWindow` per disegnare un rettangolo nella posizione e con le dimensioni specificate nella struttura; La `draw` di `Ball` è analoga ma disegn un cerchio; la `draw` di `State` si limita a chiamare le due draw di cui sopra.
   - nel `main` dichiariamo una variabile di tipo `State` (che automaticmante implementa racchetta e pallina nelle posizioni e delle dimensioni desiderate) e aggiungiamo una chiamata a al suo metodo `draw` nella parte che disegna (tra `window.clear()` e `window.display()`).

   Il programma così creato visualizza semplicemente un rettangolo bianco con un pallino bianco sopra.

2. Aggiungiamo le texture:
   - alle costanti di programma aggiungiamo due stringhe coi path dei file contenenti le texture da usare per la pallina `texture_ball.png` e la racchetta `texture_paddle.png`

- estendiamo le struct `Ball` e `Paddle` con un nuovo campo `sf::Texture`; estendiamo di conseguenza i costruttori per assegnare la texture a ogni primitiva caricandola da file e i metodi `draw` per fare il set delle texture. La struct `State` non necessita cambiamenti.

Il programma così creato visualizza la stessa scena del precedente, ora utilizzando texture che danno una parvenza di tridimensionalità a racchetta e pallina.

3. Rendiamo l'uso delle texture indipendente da file incorporandole nel codice:
   - includiamo il file `textures.cpp` che contiene due array di byte (char) in cui sono contenute le due texture di cui sopra, codificate in esadecimale. Il contenuto di tali array è stato generato con il comando di sistema (sotto linux o MacOS) `xxd` che fa il dump di un file binario in esadecimale.
   - possiamo eliminare le due stringhe che specificano i path dei file di texture, ora non più necessarie
   - nei costruttori di `Ball` e `Paddle` cambiamo il comando che carica le texture con la versione che legge un array di esadecimali (overload del costruttore di `sf::Texture`).

   Il programma così creato visualizza la stessa cosa del precedente ma è autocontenuto, non richiedendo alcuna risorsa in input.

Parte 2: movimento di pallina e racchetta

1. Movimento della pallina: quando l'utente preme la barra spaziatrice, la pallina parte in una direzione predeterminata e prosegue in linea retta (per ora senza rimbalzi).
   - Aggiungiamo due variabili globali `float ball_speed` inizializzata a 300 e `sf::Angle ball_start_angle` inizializzata a -70 gradi, che utilizzeremo velocità e direzione di rilascio della pallina
   - Aggiungiamo a `Ball` due membri: un float per la velocità e un `sf::Angle` per la direzione di movimento della palla, che inizializziamo rispettivamente a `ball_speed` e `ball_start_angle` nel costruttore
   - Aggiungiamo a `Ball` un metodo `move` con un parametro float che indica un lasso di tempo; questo metodo aggiorna la posizione della palla secondo la sua posizione precedente, il tempo trascorso, la sua velocità e la sua direzione di moto (suggerimento per il calcolo dello spostamento: la classe `sf::Vector2f` fornisce un costruttore che prende in input un valore float e un angolo e genera il vettore della lunghezza di tale valore e diretto nella direzione dell'angolo, rispetto all'asse orizzontale)
   - Aggiungiamo a `State` un membro Booleano `start` che indica se la palla è in movimento o no; il costruttore inizializza tale membro a `false`; inoltre, aggiungiamo un metodo `update` che chiama il metodo

`move` della pallina solo se questa è in movimento

- Aggiungiamo alle `handle` una callback che gestisce l'evento `sf::Event::KeyPressed` e prende come parametro anche lo stato; questa funzione mette a `true` il membro `start` di `State` se è stata premuta la barra spaziatrice (`sf::Keyboard::Scancode::Space`), altrimenti non fa nulla. Aggiungiamo inoltre la callback generica 'template void handle (T& event, State& state) {}' per gestire tutti gli eventi non riconosciuti.
- Nel `main`: aggiungiamo ai parametri di `handleEvents` la lambda expression `[&state] (const auto& event) { handle (event, state); }`; aggiungiamo un timer `sf::Clock` e facciamo reset di questo timer a ogni ciclo del main loop;prima della sezione che aggiorna la grafica, chiamiamo il metodo `update` dello stato, passando come parametro il valore corrente del `Clock` in secondi.

Il programma così ottenuto fa partire la pallina alla pressione della barra spaziatrice; la pallina prosegue oltre i limiti della window.

2. Messa in pausa del gioco: consentiamo al giocatore di mettere in pausa il gioco e riprenderlo se viene premuta la barra spaziatrice; inoltre, il gioco va in pausa se la window perde il focus.

   - Nello stato: eliminiamo il membro `start` e introduciamo due nuovi membri Booleani `focus` e `pause` per ricordare rispettivamente se la window ha il focus e se il gioco è in pausa; nel costruttore, inizializziamo `focus` a `false` e `pause` a `true`; modifichiamo il metodo `update` di conseguenza, aggiornando la pallina solo se il gioco non è in pausa.
   - Modifichiamo la 'handle dell'evento `KeyPressed` in modo da fare toggle del membro `pause` dello stato ogni volta che viene premuta la barra spaziatrice.
   - Aggiungiamo le handle per gli eventi `FocusGained` e `FocusLost` che rispettivamente pongono il membro `focus` dello stato a `true` e `false`; la `FocusLost` inoltre pone il membro `pause` a `true` (nota: per riprendere il gioco dopo una pausa è sempre necessario premere la barra spaziatrice).

   Il programma così ottenut fa (ri)partire o arresta la pallina alla pressione della barra spaziatrice e la arresta se la finestra perde il focus.

3. Movimento della racchetta: permettiamo al giocatore di muovere la racchetta attraverso le frecce sinistra e destra quando il gioco non è in pausa.

   - Aggiungiamo alle costanti globali un `float paddle_speed` che inizializziamo a 400
   - Aggiungiamo a `Paddle` un membro `float speed` e due metodi `move_left` e `move_right` che prendono come parametro un `float` che rappresenta il tempo trascorso dall'ultimo movimento; tali metodi aggiornano la sola coordinata `x` della racchetta sottraendo

o sommando un offset calcolato in base al tempo trascorso e alla velocità
- Aggiungiamo a `State` due membri Booleani `move_paddle_left` e `move_paddle_right` che inizializziamo a `false` nel costruttore
- Aggiungiamo alla handle che gestisce l'evento `KeyPressed` il trattamento delle frecce, mettendo a `true` rispettivamente i membri `move_paddle_left` o `move_paddle_right` dello stato, secondo che sia premuta la freccia di sinistra o di destra; aggiungiamo la handle per trattare l'evento `KeyReleased`, che analogamente mette a `false` le stesse variabili quando viene rilasciata la rispettiva freccia.

Il programma così ottenuto aggiunge la gestione del movimento della racchetta attraverso le frecce sinistra e destra solo se il gioco non è in pausa.

Parte 3: Gestione dei rimbalzi

1. Gestione dei limiti del campo: facciamo in modo che pallina e racchetta possano muoversi solo entro i limiti del campo. Le regole adottate sono le seguenti:

   - se la racchetta supera il limite del campo a sinistra, corregggiamo la sua origine al valore x=0; se la supera a destra, correggiamo la sua origine a un valore calolato sottraendo la larghezza della racchetta al limite destro del campo
   - se la pallina supera il limite del campo a sinistra o destra, invertiamo la componente orizzontale del suo movimenti; se lo supera in alto o in basso, invertiamo la sua componente verticale.

   Procediamo attraerso i passi seguenti:

   - aggiungiamo a `State` due metodi: `field_limits` che gestisce le collisioni di racchetta e pallina coi limiti del campo e `collisions` che gestisce le collisioni in generale. Per il momento, il secondo si limiterà a chiamare il primo.
   - creiamo due funzioni `reflect_horizontal` e `reflect_vertical` che prendono in input una direzione specificata da un `sf::Angle` e restituiscono un `sf:Angle` corrispondente alla direzione in cui sia stata invertita rispettivamente la componente orizzontale o quella verticale della direzione (suggerimento: dentro la funzione usare il costruttore di `sf::Vector2f` che definisce un vettore attraverso modulo e angolo; dopo averne invertito una delle due componenti, usare il metodo di `sf::Vector2f` che restituisce l'angolo corrispondente alla direzione del vettore. In pratica, passiamo da rappresentazione in coordinate polari a rappresentazione in coordinate Cartesiane, cambiamo segno a x o y e poi torniamo in coordinate polari
   - nel metodo `field_limits` dello stato, constolliamo prima la posizione della racchetta e applichiamo l'eventuale correzione secondo le regole

specificate in precedenza; controlliamo poi la posizione della pallina e, se supera i limiti del campo, applichiamo la riflessione orizzontele o verticale secondo i casi
- alla fine del metodo `update` dello stato chiamiamo il metodo `collisions`.

Questa soluzione ha un problema che deriva dall'interazione tra la velocità della pallina e la frequenza dell'update: se la pallina si muove lentamente rispetto alla frequenza di update, una volta che ha superato i limiti del campo e viene calcolato il rimbalzo, l'update successivo potrebbe avvenire prima che la pallina sia rientrata completamente nel campo, portando a un ulteriore "falso rimbalzo". Questo innesca una serie di falsi rimbalzi che tengono la pallina ancorata al bordo del campo, finché non si verifica un update quando la pallina sia completamente dentro il campo.

2. Gestione robusta dei rimbalzi: rimediamo al problema precedente cambiando la logica del rimbalzo, in modo che la pallina rimbalzi solo se si sta muovendo verso l'esterno del campo:

   - modifichiamo le funzioni `reflect_horizontal` e `reflect_vertical` in modo che prendano in input un `sf::Vector2f`, spostando la conversione da coordinate polari a Cartesiane fuori della funzione, prima della chiamata
   - modifichiamo il metodo `field_limits` calcolando prima la direzione di movimento della pallina a partire dal suo angolo (`sf::Vector2f v (1.0, ball.angle);`) e utilizziamo le componenti x e y di tale vettore per controllare se la pallina si muove verso l'esterno del campo. Ad esempio, per il bordo sinistro, sostituiamo il controllo `if (ball.pos.x <= 0.0)` del codice precedente col controllo `if (ball.pos.x <= 0.0 && v.x < 0)`.

   A questo punto la pallina rimblaza sui quattro bordi del campo, ignorando l'interazione con la racchetta. Secondo dove poniamo il limite inferiore del campo, avremo rimbalzi all'altezza del limite superiore della racchetta o sul fondo della finestra (anche attraverso la racchetta).

3. Rimbalzo semplice sulla racchetta e restart del gioco: gestiamo l'interazione della pallina con la racchetta, in modo che se sul limite inferiore la pallina incontra la racchetta rimbalzi e se non la incontra sparisca sul fondo e il gioco venga riavviato dall'inizio.

   - Aggiungiamo a `Paddle` un metodo Booleano `hit` che prende in input una `Ball` e dice se questa interseca la racchetta. Questo metodo calcola la direzione di movimento della pallina e restituisce `true` se e solo se: la direzione punta verso il basso e la pallina interseca i limiti della racchetta: in verticale è sufficiente confrontare che l y della pallina abbia superato la quota di intersezione col fondo; in orizzontale è necessario fare il controllo rispetto ai limiti destro e sinistro della racchetta.

- Aggiungiamo a `State` un metodo `restart` che riporta il gioco alla situazione iniziale: pallina e racchetta al centro e gioco in pausa
- Modifichiamo il metodo `field_limits` in modo che quando la pallina incontra il limite inferiore della finestra venga chiamato il metodo `restart` invece di farla rimbalzare
- Modifichiamo il metodo `collisions` in modo che, dopo aver chiamato `field_limits`, controlli se c'è una collisione con la racchetta, attraverso il metodo `hit` di `Paddle` e, in tal caso, faccia rimbalzare la pallina verticalmente.

A questo punto abbiamo già un semplice gioco, in cui il giocatore si limita a cercare di mantenere la pallina in campo. [Possibile estensione: è facile aggiungere un meccanismo di conteggio punti, ad esempio ad ogni rimbalzo, e di gestione delle vite. La visualizzazione dei punti e delle vite richiede la modifica del metodo `draw` dello stato.]

4. Rimbalzo variabile sulla racchetta: per il momento il rimbalzo avviene secondo la legge di riflessione speculare. Lo rendiamo più simile al gioco originale facendo in modo che l'angolo di rimbalzo sia tanto più aperto quanto più la collisione avviene lontano dal centro della racchetta:

   - Aggiungiamo una costante globale `sf::Angle left_max_angle` che poniamo a -170 gradi per fissare un limite all'angolo di riflessione
   - Aggiungiamo a `Paddle` un metodo `void strike(Ball&)` per la gestione del rimbalzo; sostituiamo nella `State::collisions` le istruzioni di gestione della collisione con una chiamata a questo metodo.
   - Il metodo `strike` prima controlla se c'è una collisione (`hit`) con la pallina; in caso affermativo, calcola dapprima la distanza orizzontale tra i centri di pallina e racchetta e lo utilizza per calcolare l'angolo di uscita dopo il rimbalzo: questo sarà ottenuto calcolando l'interpolazione lineare tra `left_max_angle` e `right_max_angle`, quest'ultimo calcolato riflettendo in orizzontale `left_max_angle`. Per il calcolo dell'interpolazione lineare implementiamo una funzione helper `linear_interpolation` che prende tre valori: `v0` e `v1` che rappresentano i limiti dell'interpolazione e `t` compreso in [0,1] che è il parametro di interpolazione; per t=0 restituiamo `v0`, per t=1 restituiamo `v1` e per valori intermedi restituiamo un valore che varia linearmente con t tra tali estremi. Il valore corretto di `t` da passare alla funzione viene calcolato come distanza tra i centri di pallina e racchetta, normalizzata sull'ampiezza della racchetta più quela dela pallina, e aumentata di 0.5 per ricondurla all'intervallo [0,1]. L'angolo calcolato per interpolazione lineare viene utilizzato per aggiornare l'angolo di movimento della pallina.

5. Rimbalzo sui bordi della racchetta: quando il rimbalzo avviene sul bordo esterno della racchetta, la palla deve rimbalzare verso l'alto solo se il suo centro non ha ancora superato la y del centro della racchetta. Aggiungiamo istruzioni in fondo al metodo `strike`, in modo che se la pallina si trova

già oltre la racchetta, allora il rimbalzo avvenga verso il basso (riflessione verticale). In questo caso, la pallina proseguirà fuori dal campo e il gioco andrà in restart).

[Variante: A partire da questo punto, è possibile aggiungere una seconda racchetta dal lato opposto del campo e gestire l'uscita della pallina anche da quel lato per implementare il gioco Pong [Atari, 1972] a due giocatori. La seconda racchetta può utilizzare la stessa classe della prima per il disegno, mentre richiede qualche modifica per le regole di rimbalzo. Dovranno essere utilizzati altri tasti per il suo controllo nella GUI. Nota: nel gioco originale le racchette erano disposte sui lati sinistro e destro e si muovevano verticalmente. Non è difficile adattare il programma a questo scenario, ma richiede qualche cambiamento.]

Parte 4: Gestione del muro

1. Struttura dati e disegno del muro: dobbiamo aggiungere al campo di gioco un muro di mattoni rappresentati da rettangoli. Durante il gioco, il muro verrà abbattuto un mattone alla volta dalle collisioni tra la pallina e i mattoni. Abbiamo quindi bisogno di una struttura dati in grado di ricordare quali sono in ogni istante i mattoni presenti nel muro e che ci possa consentire di calcolare le collisioni.

   - Il muro sarà posizionato in modo da lasciare ai suoi lati due corrodoi sottili, sufficientemente larghi da permettere alla pallina di entrarci, e avrà dietro un corridoio piuttosto ampio. Definiamo alcune costanti globali per stabilire: posizione (50, 100) e dimensioni globali (750,150) del muro, numero di mattoni in orizzontale e in verticale, colore dei mattoni e del loro bordo e spessore del bordo
   - Definiamo una `struct Block` per rappresentare un singolo mattone e una `struct Wall` per rappresentare l'intero muro.
   - `Block` ha due campi `sf::Vector2f` per memorizzare posizione e dimensioni del mattone e un cambpo `bool` per indicare se il mattone è intatto o è stato abbattuto; ha un costruttore esplicito che riceve in input posizione e dimensione e inizializza il Booleano a true (mattone intatto); ha inoltre un metodo `draw` che, se il mattone è intatto, disegna un rettangolo della dimensione e nella posizione specificata, utilizzando colori e spessore del bordo come specificato dalle costanti di programma.
   - `Wall` ha un solo campo `std::vector<Block>` per mantenere il muro, un costruttore di default e un metodo `draw`. Quest'ultimo si limita a ciclare sul contenuto del vector e chiamare il metodo `draw` di ciascun blocco. Il costruttore deve prima calolare la dimensione del singolo mattone in base alle costanti di programma (area occupata dal muro e numero di mattoni nelle due dimensioni), quindi istanziare i mattoni di tale dimensione nelle posizioni opportune (ciclando su righe e colonne) e inserendoli nel `vector`.

- Aggiungiamo un membro `Wall` allo stato (notiamo che il costruttore di default dello stato chiama il costruttore di default di `Wall`, istanziando quindi la struttura dati secondo le specifiche date dalle costanti di programma)
- Modifichiamo il metodo `draw` dello stato aggiungendo una chiamata all'omonimo metodo di `Wall`.

Notiamo che a questo punto il gioco si comporta in modo inconsistente. Il muro viene visualizzato solo come componente pittorica, ma non ha alcun ruolo: la pallina lo attraversa senza interferire con esso. Notiamo anche che la pallina attraversa il muro passandoci "dietro": questo è dovuto al fatto che la `State::draw` disegna prima la pallina e dopo il muro e le primitive disegnate dopo oscurano quelle disegnate prima. Possiamo disegnare la pallina dopo il muro per farcela passare "sopra" oppure definire il colore dei mattoni semi-trasparente (diminuendo il valore della componente alpha del colore) per vedere la pallina in trasparenza quando ci passa dietro.

2. Interazione tra pallina e muro: per completare il nostro gioco dobbiamo gestire le interazioni tra pallina e muro. Ogni volta che la pallina collide con un mattone, rimbalza e il mattone viene distrutto.

   - Aggiungiamo a `Block` i metodi `bool is_inside (sf::Vector2f)` - che prese le coordinate di un punto dice se questo si trova all'interno del mattone - e `void hit(Ball&)` - che gestisce la potenziale collisione di un mattone con la pallina; in caso di collisione, tale metodo predispone il rimbalzo della pallina e la distruzione del mattone
   - Aggiungiamo a `Wall` il metodo `void hit(Ball&)` che cicla su tutti i mattoni intatti del muro, verificando le collisioni con la pallina mediante il metodo `Block::hit` specificato al punto precedente. **Nota:** possiamo permetterci questo ciclo esaustivo per semplicità, perché il numero di mattoni è relativamente piccolo e la macchina molto veloce. Tuttavia, si tratta di un metodo di collision detection molto inefficiente. Nel caso specifico, poiché il muro ha una struttura a griglia regolare, conoscendo la posizione e le dimensioni dei mattoni (come costruiti all'inizio) sarebbe possibile verificare se siano intatti i soli mattoni la cui area si sovrappone a quella della pallina, ignorando tutti gli altri. Per scene più complicate è invece necessario ricorrere a strutture dati che possano restringere in campo degli oggetti con in quali si hanno potenziali collisioni. La collision detection rappresenta un'area importante del calcolo geometrico per la grafica ed è un argomento molto esteso.
   - Il metodo `Block::is_inside` si implementa semplicemente confrontando le coordinate del punto con i limiti del rettangolo corrispondente al mattone.
   - Nel metodo `Block::hit` calcoliamo prima i quattro punti che stanno sugli estremi superiore, inferiore, sinistro e destro della pallina. Per ciascuno di questi punti, vediamo se è contenuto nel mattone e se

la direzione della pallina è rivolta verso l'interno del mattone; in caso positivo, aggiorniamo la direzione della pallina, analogamente a quanto fatto in precedenza per i limiti del campo, e poniamo `intact = false`
- Aggiungiamo a `State::collisions` una chiamata alla `Wall::hit` per il relativo membro dello stato.

A questo punto abbiamo un gioco con un comportamento consistente e analogo a quello dei giochi *arcade* a cui si ispira.

Possibili estensioni

- Gestione del punteggio e delle vite (con visualizzazione dinamica delle variazioni utilizzando `sf::Text`)
- Utilizzo di texture anche sui mattoni per migliorare la grafica; eventuale colorazione dipendente dalla posizione del mattone
- Lancio della pallina in una direzione iniziale scelta dal giocatore
- Variazione della velocita e/o dimensione della pallina al variare del punteggio
- Gestione dei livelli: questo può richiedere una ristrutturazione del codice perché livelli diversi possono richiedere dimensioni dei mattoni e numeri diversi, velocità e dimensioni della pallina diverse, ecc. Può essere conveniente creare una struttura dati che memorizza tutti i parametri che caratterizzano un livello e mantenere un array di tali strutture numerato sui livelli.
- Aggiunta di ostacoli fissi e/o agenti autonomi dinamici, come oggetti che si muovono dal muro verso il basso e devono essere intercettati o evitati dalla racchetta
- Ecc.