Relazione progetto PMO 2023/2024

El Idrissi Adam 299083

Indice

1.1 Analisi dei requisiti

La specifica prevede la modellazione ed implementazione di un videogioco dello stile tower defense, dove il giocatore deve difendersi da orde di nemici tramite piazzamenti di unità in posizioni specifiche, con possibilità di potenziamento delle unità per resistere ad orde sempre piu' difficili e numerose tramite un sistema di risorse che costringono il giocatore a tenere conto di possibili errori/situazioni future.

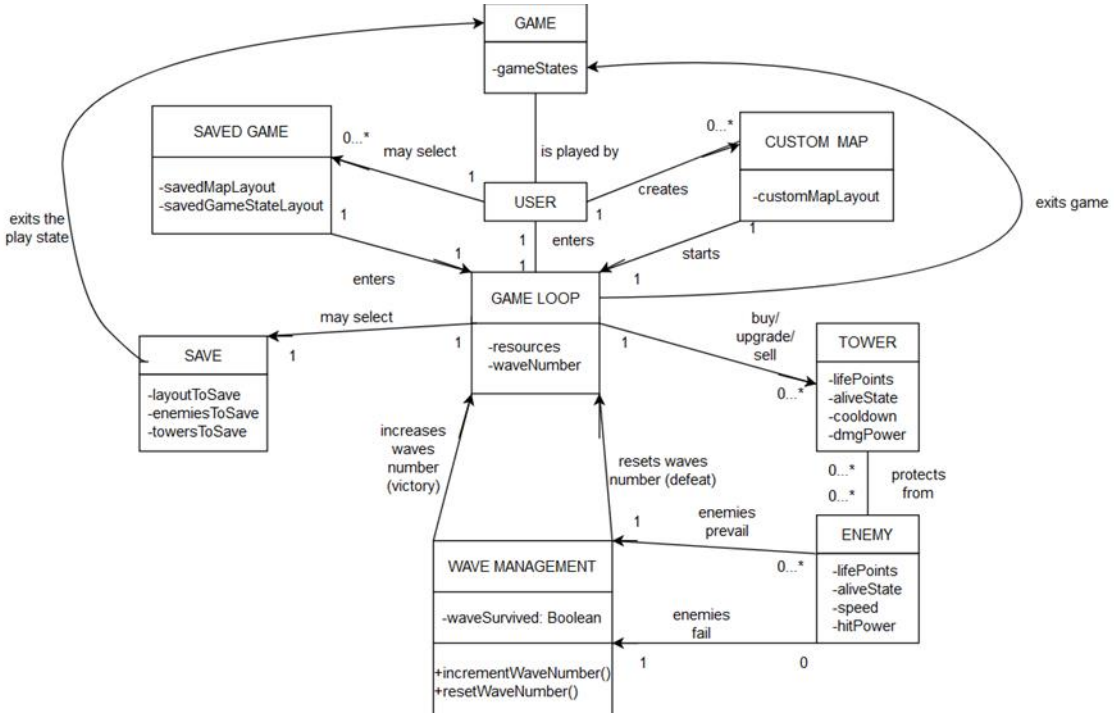
I requisiti principali del progetto sono:

- Differenti tipi di unità difensive, ognuna con proprie caratteristiche
- Differenti tipi di nemici, ognuno con proprie caratteristiche
- Possibilità di potenziare le unità tramite le risorse ottenute nella partita
- Ondate multiple con differente numero di nemici
- Possibilità di vendere le unità e recuperare le risorse spese
- Possibilità di curare/riparare le unità spendendo risorse

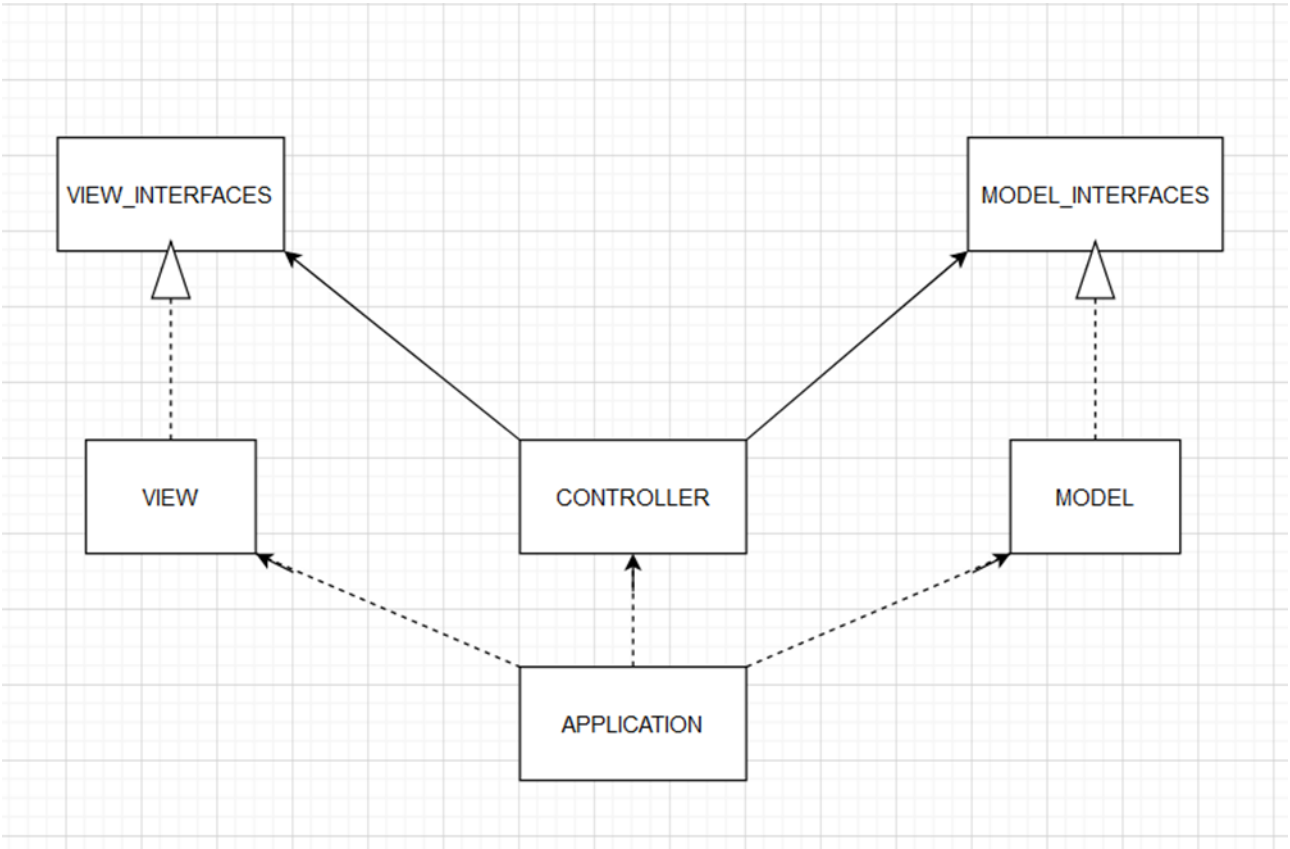
I requisiti secondari del progetto sono:

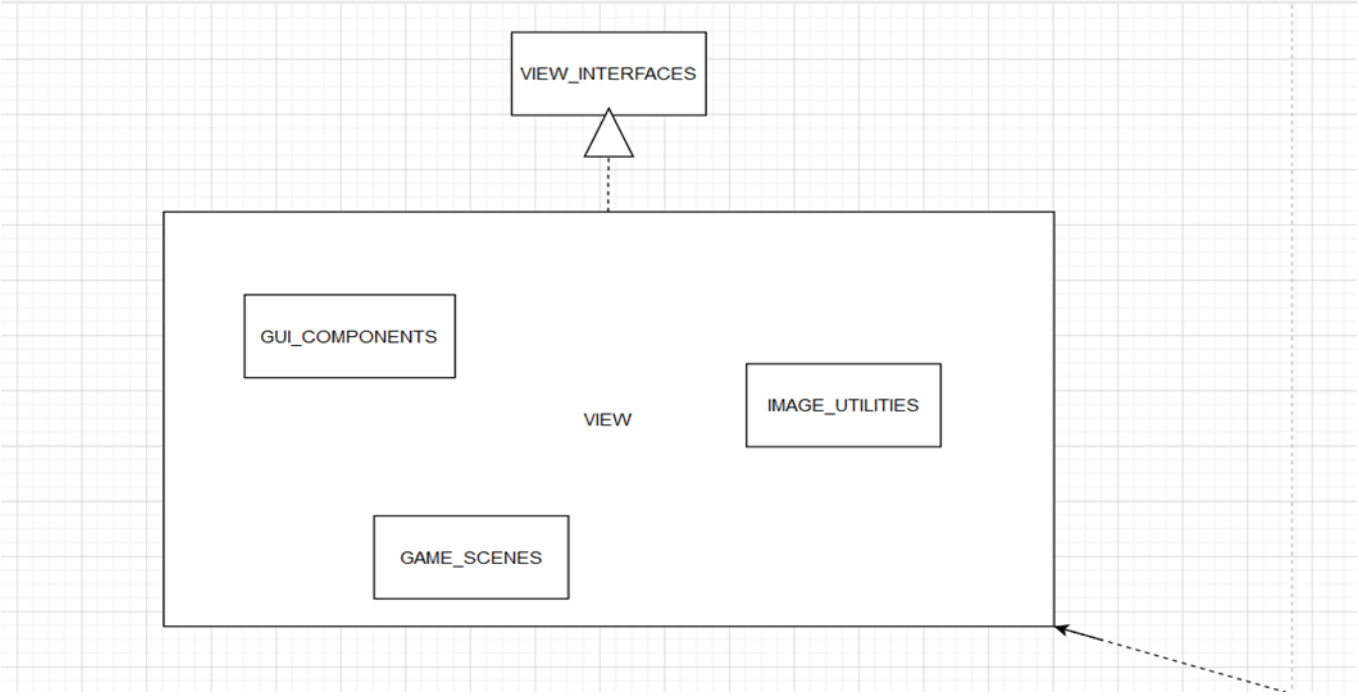
- Modalità di resistenza infinita (ondate ad oltranza)
- Possibilità di salvare e riprendere la partita in un secondo momento
- Possibilità di creare o generare nuove mappe a piacimento

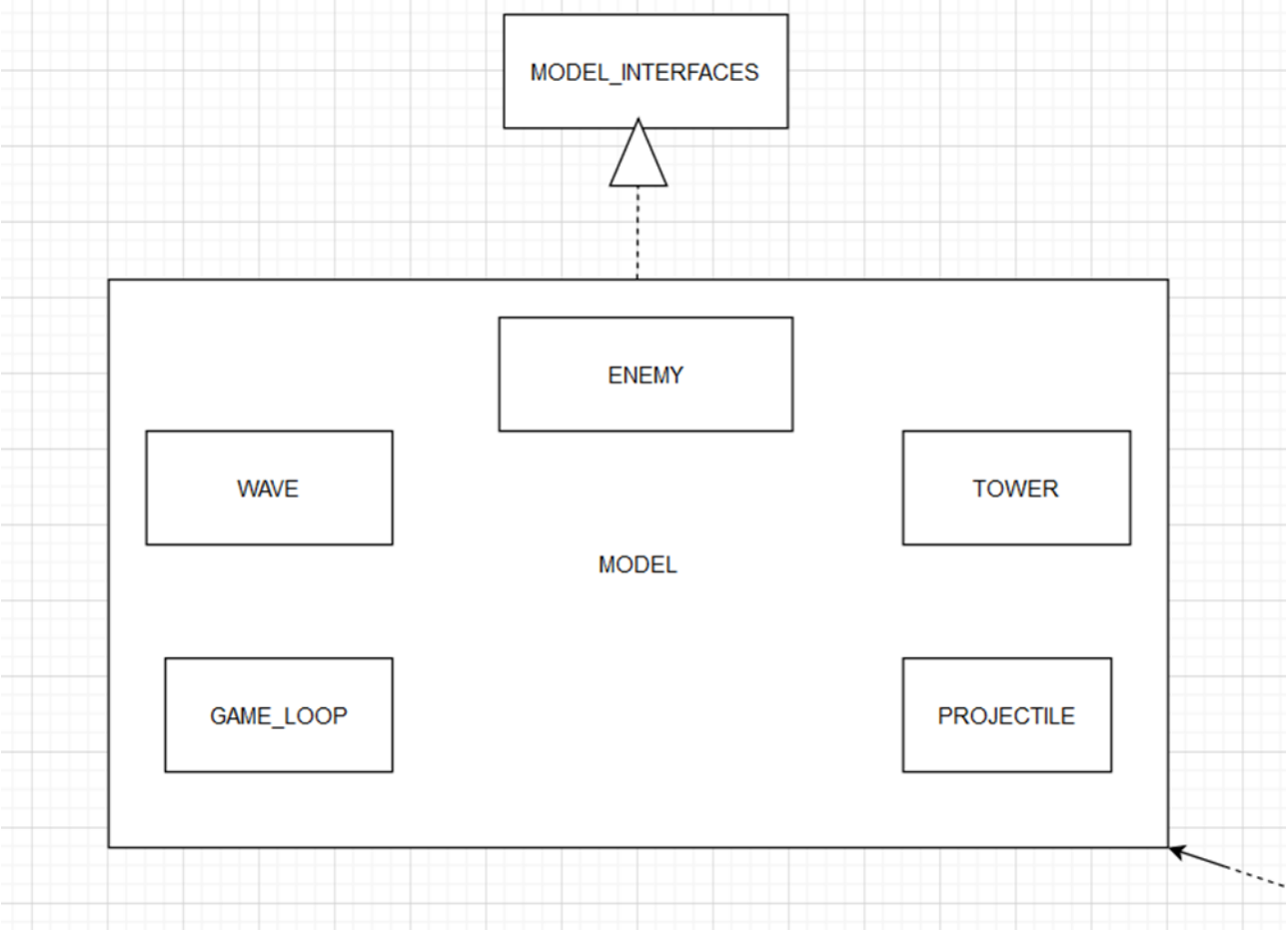
Prototipo del modello di dominio:

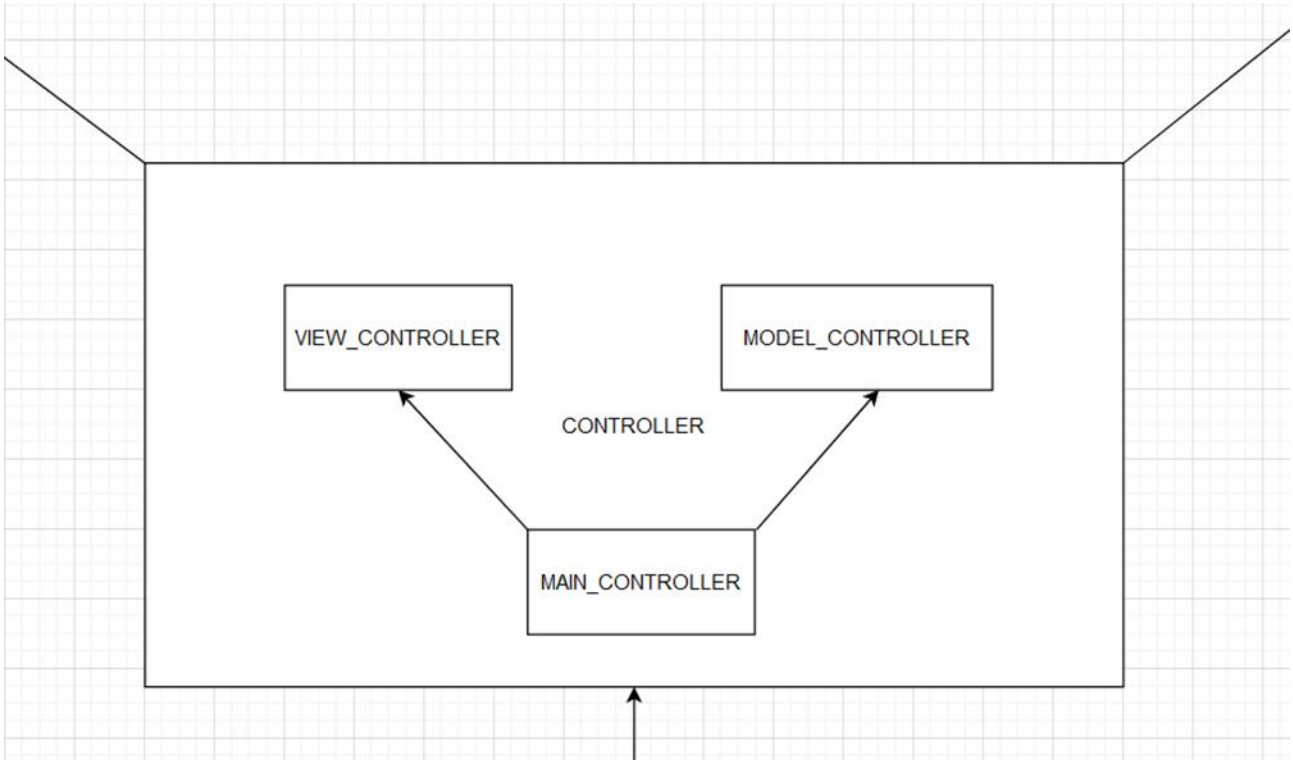


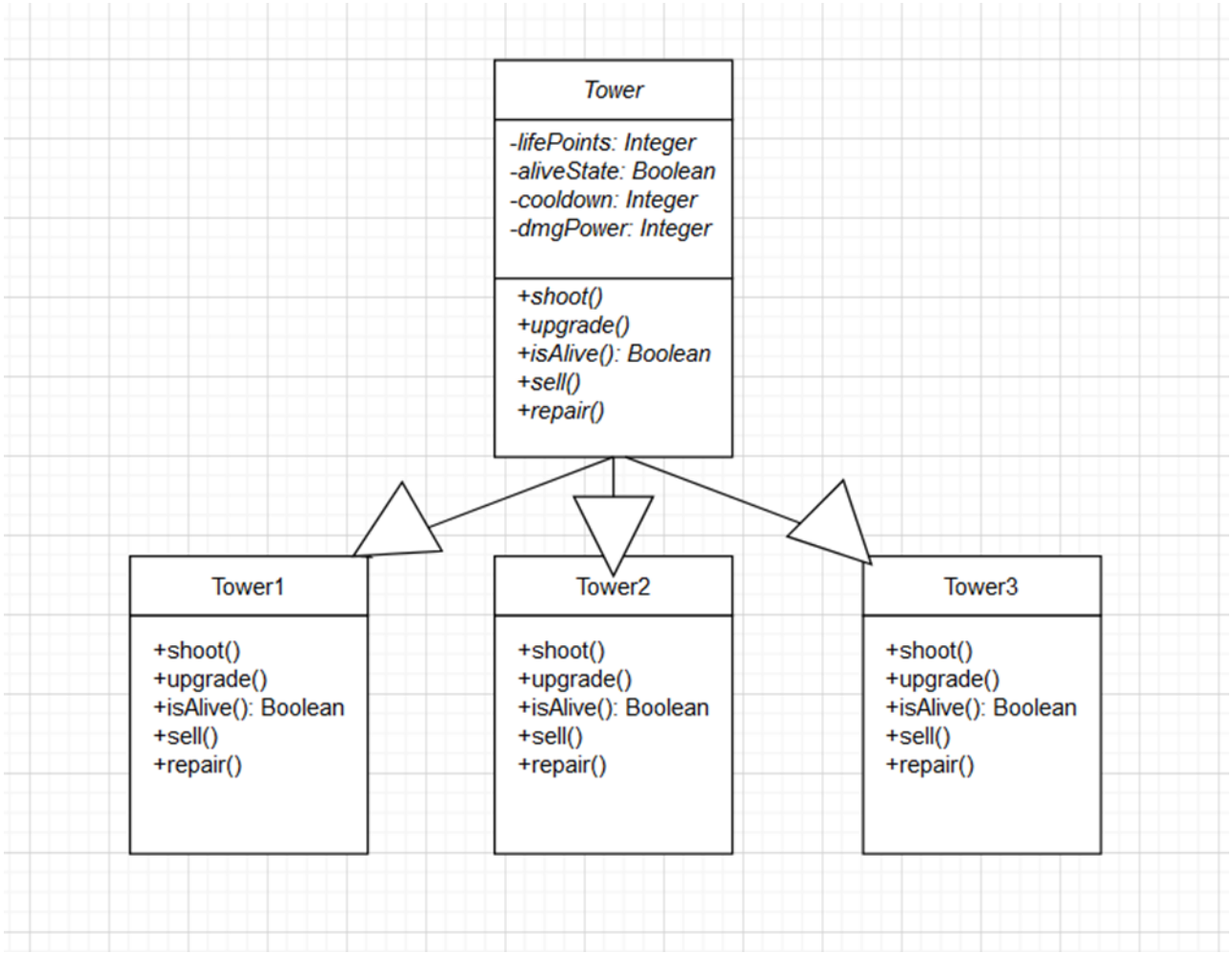
Ed i prototipi dei diagrammi delle componenti fondamentali:

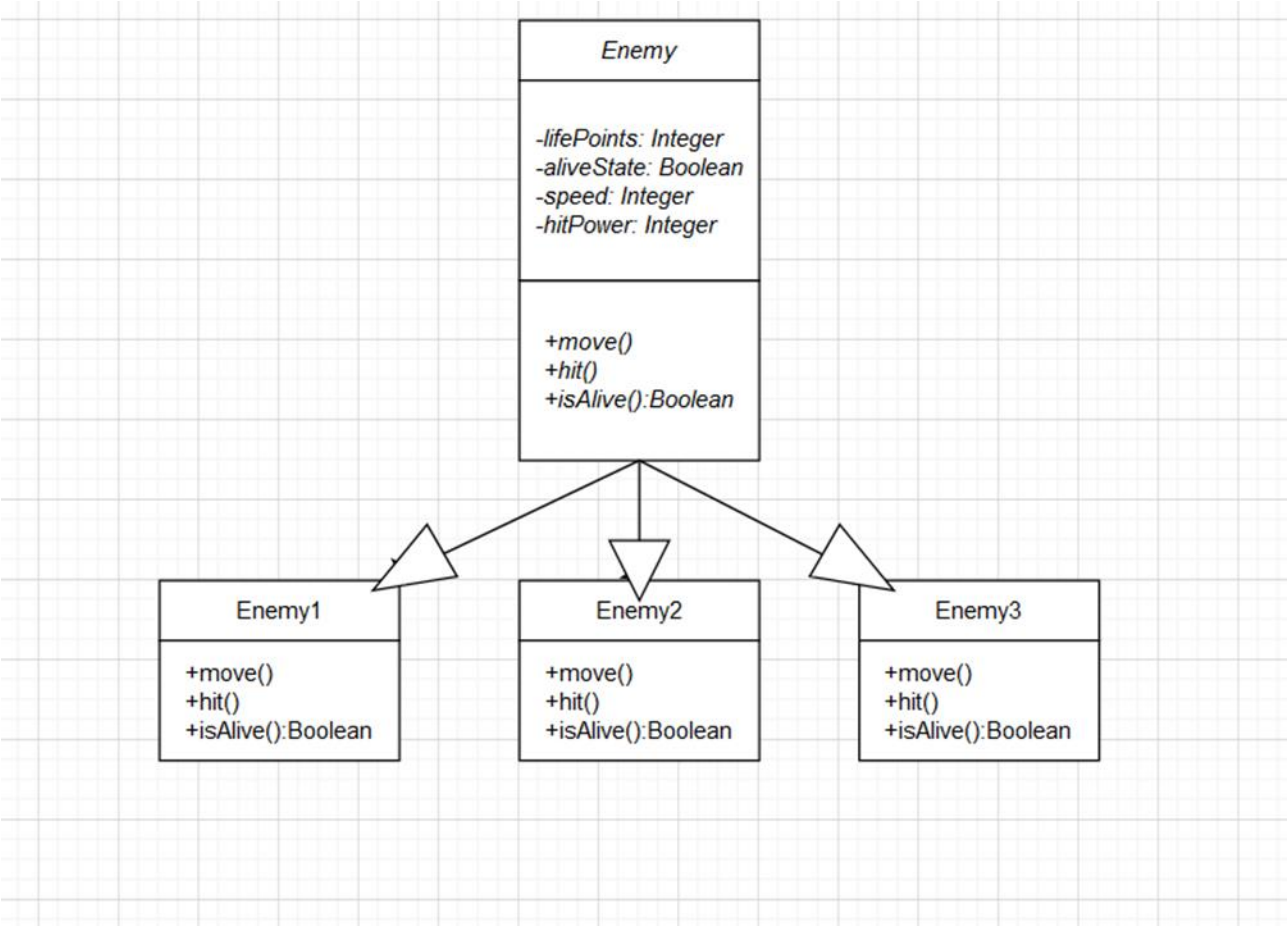



VIEW_INTERFACES
GUI_COMPONENTS
VIEW
IMAGE_UTILITIES
GAME_SCENES


MODEL_INTERFACES
ENEMY
WAVE
TOWER
MODEL
GAME_LOOP
PROJECTILE


VIEW_CONTROLLER
MODEL_CONTROLLER
CONTROLLER
MAIN_CONTROLLER


Tower
-lifePoints: Integer
-aliveState: Boolean
-cooldown: Integer
-dmgPower: Integer
+shoot()
+upgrade()
+isAlive(): Boolean
+sell()
+repair()
Tower1
+shoot()
+upgrade()
+isAlive(): Boolean
+sell()
+repair()
Tower2
+shoot()
+upgrade()
+isAlive(): Boolean
+sell()
+repair()
Tower3
+shoot()
+upgrade()
+isAlive(): Boolean
+sell()
+repair()



Quindi l'applicazione si basa sul processo di funzionamento dei videogiochi "tower defense", ovvero videogiochi in cui è necessario difendere la propria base/la propria zona d'interesse da masse di nemici, talvolta le masse di nemici sono composte da ondate di differenti tipi/numeri di nemici, inoltre in genere presentano un aspetto di strategia per il posizionamento tattico delle proprie difese o per il modo di gestirle, come cosa potenziare, in che modo gestire le risorse…

I requisiti principali correttamente implementati sono:

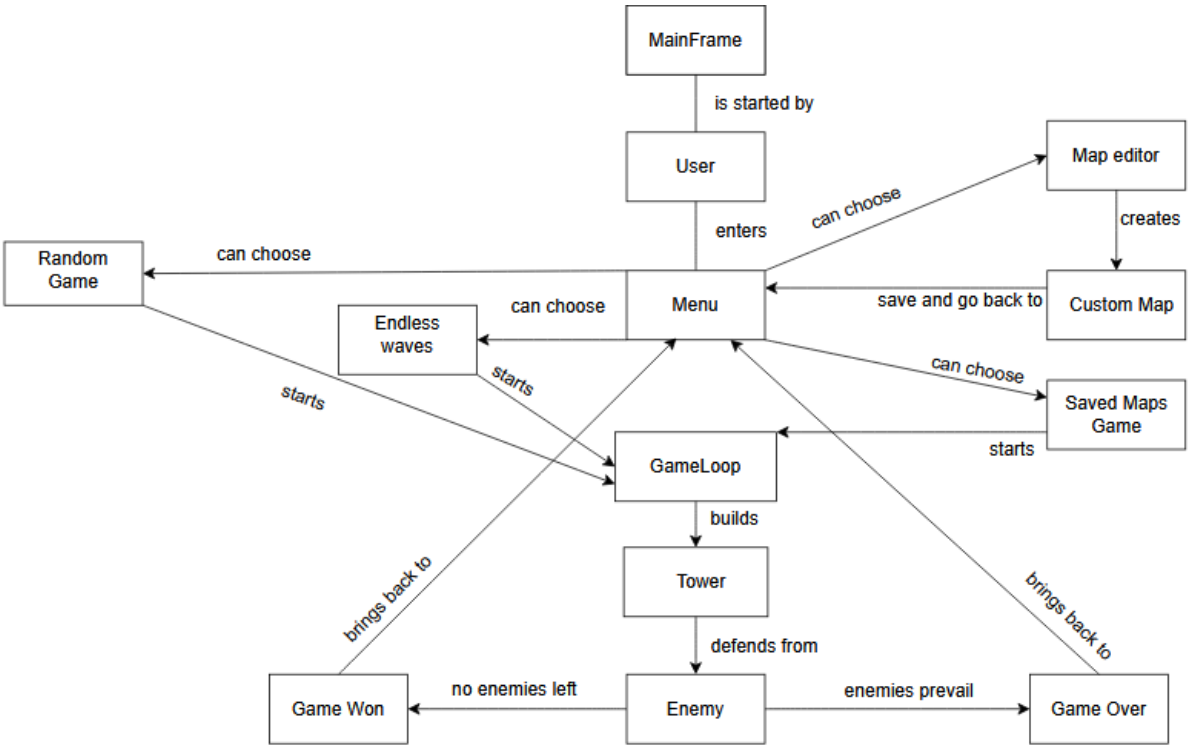
- Differenti tipi di unità difensive, ognuna con proprie caratteristiche
- Differenti tipi di nemici, ognuno con proprie caratteristiche
- Possibilità di potenziare le unità tramite le risorse ottenute nella partita
- Ondate multiple con differente numero di nemici (nella modalità ondate infinite)
- Possibilità di curare/riparare le unità spendendo risorse

I requisiti secondari correttamente implementati sono:

- Modalità di resistenza infinita (ondate ad oltranza)
- Possibilità di creare o generare nuove mappe a piacimento

1.2 Modello finale del dominio

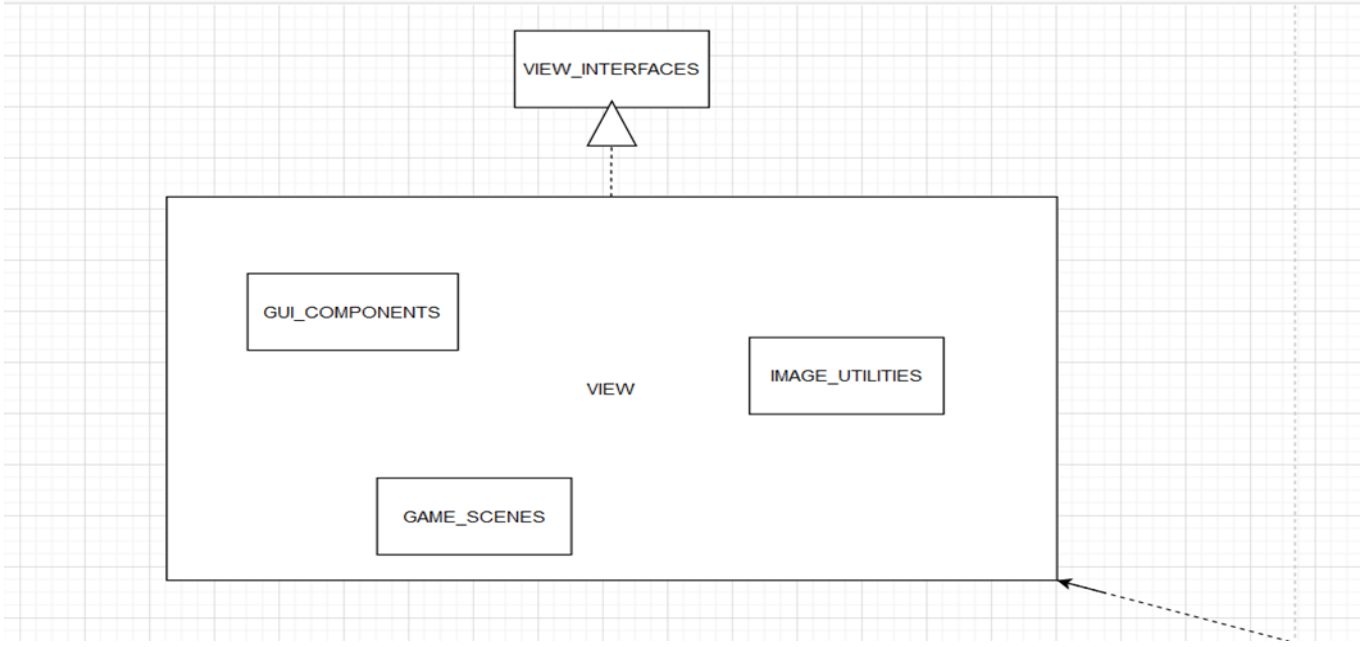
Tra la progettazione iniziale e lo sviluppo effettivo dell'applicazione, ci sono stati diversi cambiamenti alla struttura del modello, alcuni cambiamenti per comodità, altri per necessità, di seguito viene mostrato il nuovo modello del dominio:

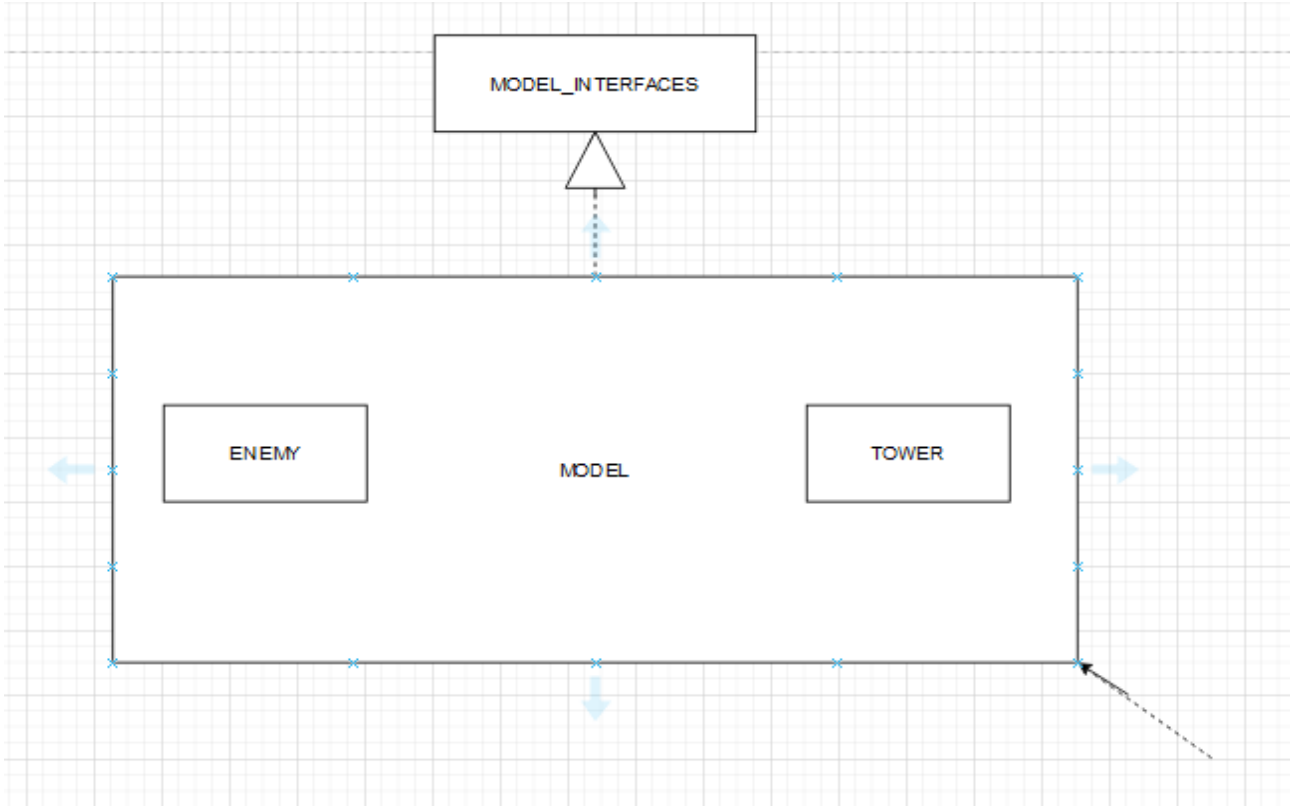


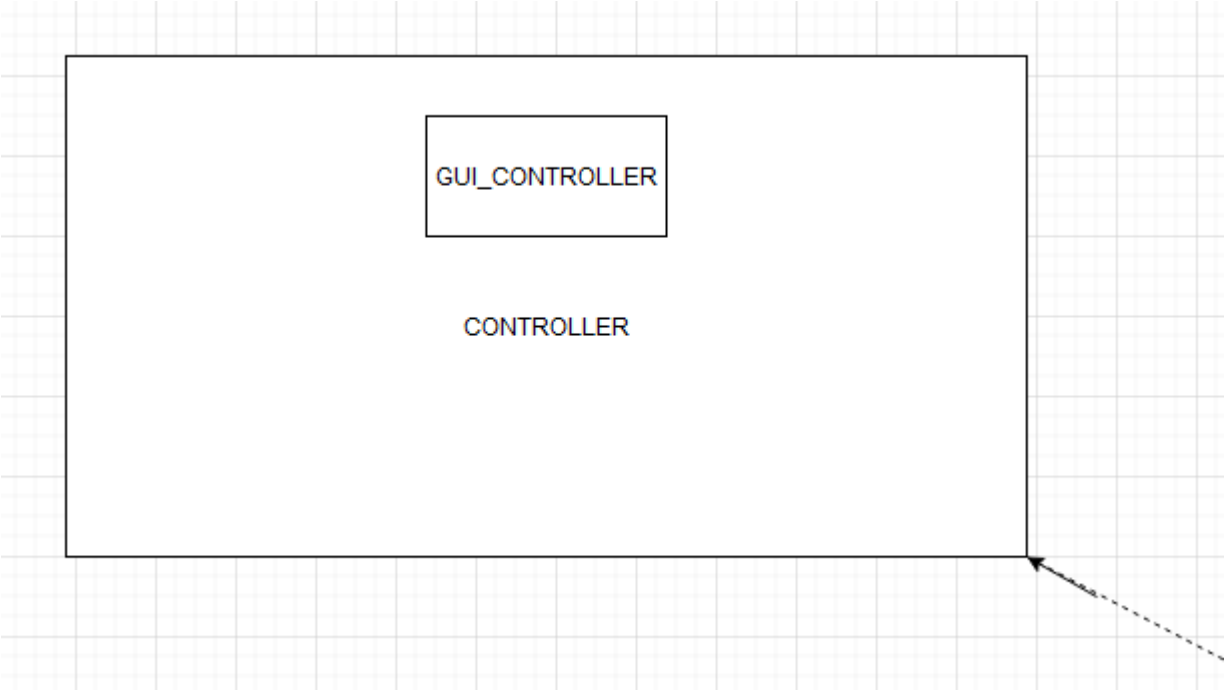
Il modello corrente rispecchia in modo piu' concreto il modello di dominio dell'applicazione sviluppata e mostra il flusso di svolgimento attraverso i macro-componenti, le funzionalità ed i dettagli dei micro-componenti verranno mostrati nella prossima sezione.

2.1 DESIGN ED ARCHITETTURA PROGETTO

Vengono mostrati di seguito i nuovi diagrammi dei componenti fondamentali:







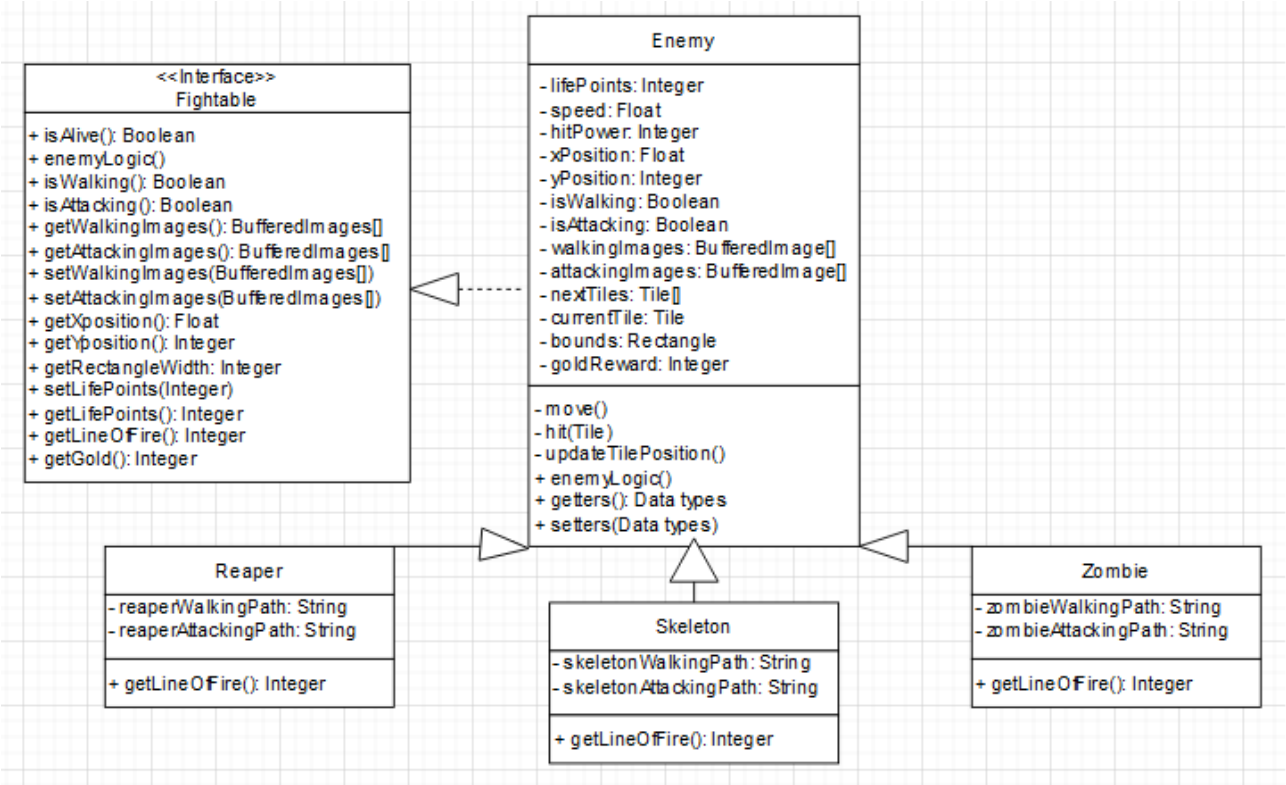
Come possibile notare dagli schemi qui mostrati, il design dell'applicazione è cambiato durante il processo di sviluppo, in particolare la sezione dei controller si è rimpicciolita mantenendo un solo controller per la GUI, mentre la view è rimasta simile all'idea di base ed il model si è ristretto a 2 super-componenti.

In seguito vengono descritti i dettagli delle singole componenti del progetto.

2.2 DETTAGLI MODEL

La sezione Model è composta da 2 superclassi principali che rappresentano le entità dei nemici e delle torri con la loro rispettiva logica, entrambe sono completate dalle proprie sottoclassi che consistono nei nemici e nelle torri vere e proprie, tutti con proprie animazioni/statistiche.

Di seguito il modello delle classi responsabili dell'entità "***nemici***":



Il funzionamento dei nemici è quindi gestito dalla superclasse *Enemy* che compone lo stampino principale dell'entità nemici, contiene tutti gli attributi principali ed i super-metodi che vengono poi ereditati dalle singole sottoclassi, queste ultime aggiungono solo 2 campi che descrivono il path per gli sprite dei frame per le animazioni di ogni tipo di nemico ed un metodo per restituire un valore chiamato *lineOfFire* che gestisce la condizione necessaria per i nemici ad essere attaccati dalle torri nella stessa riga, esse implementano l'interfaccia *Fightable*.

In particolare l'insieme dei campi dell'entità ***nemico*** è composta da:

- Un intero corrispondente ai punti vita

- Un float per la velocità di percorrenza della mappa del nemico

- Un intero per il danno che può arrecare alle torri

- Una posizione x in float per rendere il posizionamento dei nemici lungo la riga il piu' variegato possibile

- Una posizione y in interi per definire la linea della riga di percorrenza

- 2 booleani che definiscono lo status corrente dell'entità (se camminando o se attaccando)

- 2 array di immagini per i frames di animazione della camminata ed attacco

- Un array di tiles che permettono di definire la riga sulla quale l'entità si trova

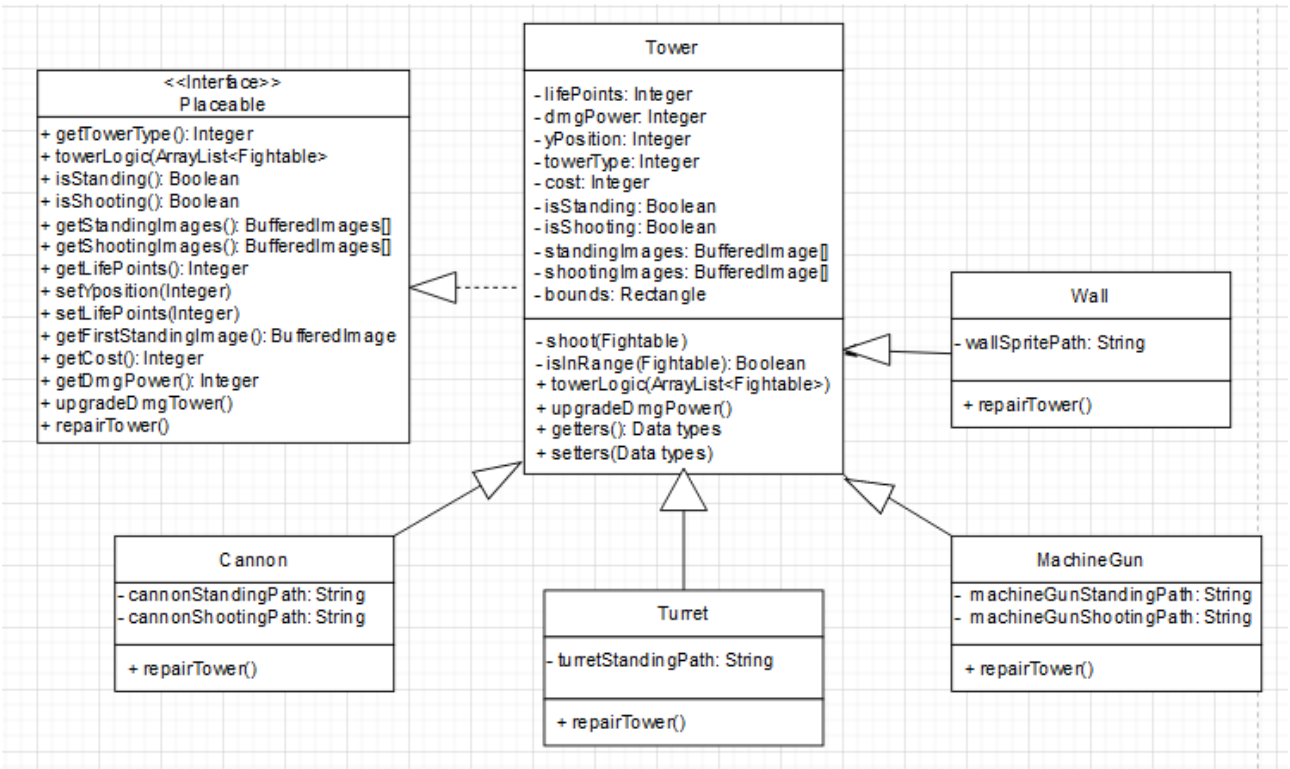
- Un tile corrente su cui il nemico si trova (per identificare la presenza di eventuali torri da abbattere)

- Un bounds che rappresenta il rettangolo di collisione o hitbox che contiene l'entità

- Un valore intero che corrisponde al numero di oro restituito in caso di sconfitta dell'entità in questione.

Il metodo fondamentale dell'entità ***nemico*** è il metodo *enemyLogic*() che descrive e gestisce il corretto funzionamento, tenendo in conto prima di tutto dello stato di vita dell'entità, e poi gestendo le situazioni di movimento grazie al sotto-metodo *move*() e quelle di attacco grazie al sotto-metodo *hit*(*Tile*), inoltre garantisce che una volta che il nemico sia arrivato al bordo della mappa, la partita termini con uno stato di Game Over.

Di seguito il modello delle classi responsabili dell'entità "***torri***":



Il funzionamento dei nemici è quindi gestito dalla superclasse *Tower* che compone lo stampino principale dell'entità torri, contiene tutti gli attributi principali ed i super-metodi che vengono poi ereditati dalle singole sottoclassi, queste ultime aggiungono solo 1 o al massimo 2 campi che descrivono il path per gli sprite dei frame per le animazioni di ogni tipo di torre ed un metodo per riparare la torre che deve essere specifico ad ogni sottoclasse per via del differente numero di punti vita *repairTower()*, esse implementano l'interfaccia *Placeable*.

In particolare l'insieme dei campi dell'entità ***torre*** è composta da:

- Un intero corrispondente ai punti vita

- Un intero per il danno che può arrecare ai nemici

- Una posizione y in interi per definire la riga di posizionamento

- Un intero per definire il tipo della torre

- Un intero per il costo della torre

- 2 booleani che definiscono lo status corrente dell'entità (se normale o se sparando)

- 2 array di immagini per i frames di animazione della posizione normale e di attacco

- Un bounds che rappresenta il rettangolo di collisione o hitbox che contiene l'entità

Il metodo fondamentale dell'entità ***torre*** è il metodo *towerLogic*(*ArrayList*<*Fightable*>) che descrive e gestisce il corretto funzionamento, tenendo in conto la presenza di nemici sulla propria linea di tiro grazie al sotto-metodo *isInRange(Fightable)*, inoltre gestisce la condizione di standby o di attacco della torre stessa, se il nemico si trova in range ed è effettivamente ancora vivo, inizia la fase d'attacco modificando i booleani, perciò modificando il tipo di animazioni in uso corrente e danneggiando il nemico da abbattere tramite il metodo *shoot(Fightable)*.

Le componenti "Wave", "Projectile" e "GameLoop" sono state rimosse poiché la gestione delle ondate o "waves" viene gestita dalle classi della view come un parametro interno alla classe stessa e gestito come e quando serve, la gestione dei proiettili o "projectiles" viene gestita tramite la logica di danneggiamento delle torri e di ricevimento danno dei nemici, infine il gameLoop viene gestito dalla classe di gioco principale GameSceneBase e viene correttamente finalizzato nelle classi specializzate alle singole modalità di gioco, verrà spiegato di piu' a riguardo nella sezione dei dettagli della view.

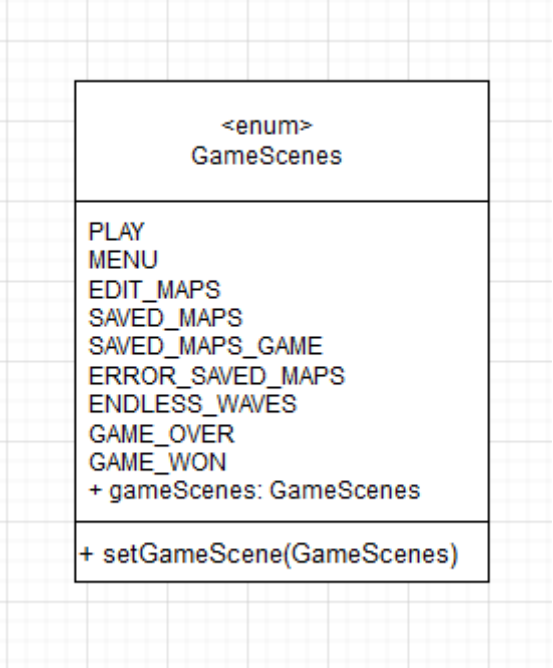
Il model non possiede piu' un controller apposito poiché la logica di entrambe le entità è autogestita dalle classi stesse.

2.3 DETTAGLI VIEW

La view è composta da 3 macro-componenti: “GameScenes” che si occupa della gestione delle varie scene di gioco (menu’, partita random, editor mappe…), “Gui components” (che si occupa delle componenti che formano la GUI nella sua interezza) e le “image utilities” (che consistono in metodi per la gestione delle immagini).

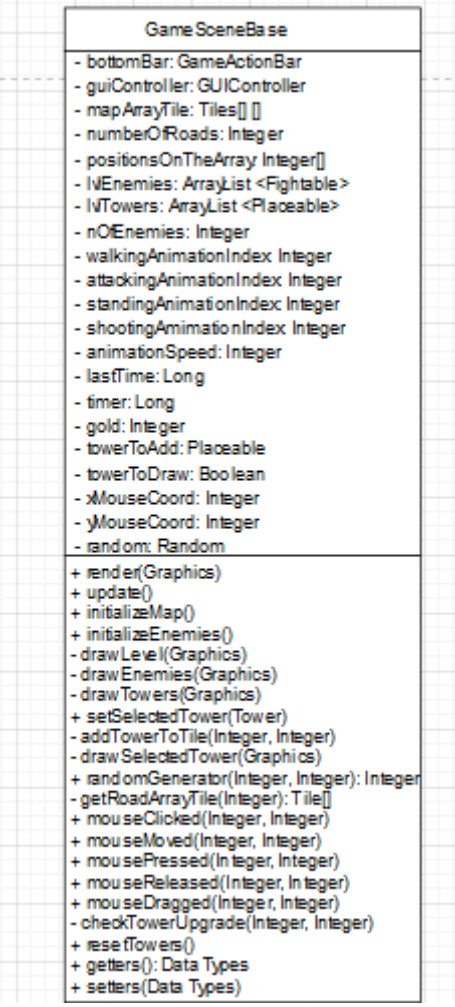
Di seguito vengono presentate le classi della macro-componente “***gameScenes”:***

Classe Enum “***GameScenes***”:



La classe ***GameScenes*** è una classe ENUM che definisce i vari stati del gioco, definendo inoltre lo stato corrente di gioco tramite una variabile statica e permettendo di modificarlo tramite un apposito metodo statico richiamabile dagli altri componenti in seguito. La classe è il fulcro della componente grafica in quanto essa definisce ciò che verrà renderizzato a tempo di esecuzione dalla classe responsabile del rendering, gestendo quindi il flusso del passaggio tra le scene del gioco.

Classe “***GameSceneBase***”:



La classe compone la componente fondamentale della scena di gioco vera e propria, essa contiene i campi e metodi fondamentali per il funzionamento delle sessioni di gioco, ogni sottoclasse provvede a specializzare questa classe per gli specifici casi di gioco. La classe è composta da:

- Una barra di azione sul bordo in basso della schermata

- Un oggetto di GUI controller che gestisce la generazione della mappa

- Un array 2d di tiles che compongono la mappa

- Un intero per definire il numero di strade sulla mappa

- Un array di interi per definire le posizioni delle colonne nelle quali ci saranno strade

- Una arrayList di nemici presenti nel livello

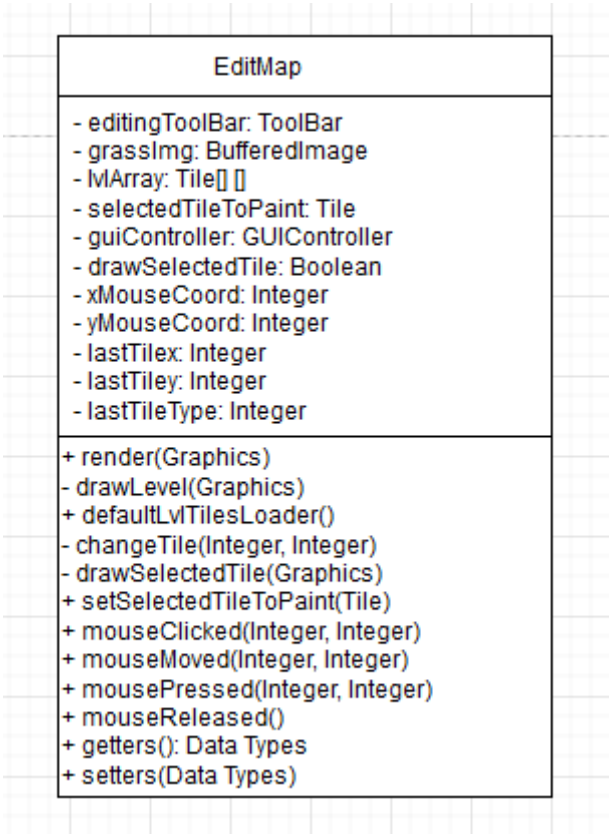
- Una arrayList di torri presenti nel livello

- Un intero per definire il numero di nemici nel livello

- Un intero per l'indice per le animazioni di camminata dei nemici
- Un intero per l'indice per le animazioni di attacco dei nemici
- Un intero per l'indice per le animazioni di posizione normale per le torri
- Un intero per l'indice per le animazioni di fuoco per le torri
- Un intero per definire la velocità delle animazioni
- 2 Long per le variabili di tempo
- Un intero per definire la risorsa di oro nella partita
- Un oggetto Placeable (torre) per definire l'eventuale torre da aggiungere
- Un booleano per definire la necessità di aggiungere una torre
- Un intero per la coordinata x del mouse
- Un intero per la coordinata y del mouse
- Un oggetto random per la creazione di valori random

I metodi principali della classe sono il metodo *render(Graphics)* che permette la renderizzazione dei componenti della classe stessa, il metodo *update*() che permette l'aggiornamento costante dei componenti costituenti l'ecosistema di gioco (lista nemici, lista torri, oro, animazioni…), *initializeMap()* viene implementata in ogni sottoclasse per creare la mappa di gioco, *initializeEnemies*() permette di inizializzare i nemici nella partita, *drawLevel*(*Graphics*), *drawEnemies*(*Graphics*) e *drawTowers*(*Graphics*) permettono di disegnare i nemici, il livello stesso e le torri presenti nel livello, *setSelectedTower*(*Tower*) permette di selezionare il tipo di torre da aggiungere, *addTowerToTile*(*Integer*, *Integer*) permette di effettivamente aggiungere la torre al tile scelto, *drawSelectedTileTower*(*Graphics*) permette di disegnare la preview della torre scelta per il tile, *randomGenerator*(*Integer*, *Integer*) permette di generare valori random, *getRoadArrayLine*(*Integer*) permette di ritornare l'intera riga corrispondente ad una strada, *mouseClicked*(*Integer*, *Integer*) gestisce la risposta dell'applicazione ai click in determinate posizioni, *mouseMoved*(*Integer*, *Integer*) gestisce la risposta dell'applicazione al passaggio del mouse in determinate posizioni, *mousePressed*(*Integer*, *Integer*) gestisce la risposta dell'applicazione alla pressione del mouse in determinate posizioni, *mouseReleased*() gestisce la risposta dell'applicazione al rilascio della pressione del mouse, *checkTowerUpgrade*(*Integer*, *Integer*) permette di controllare se il tile scelto contiene una torre per in caso ripararla o potenziarla, *resetTower*() permette di eliminare tutte le torri presenti nel livello.

Classe “***Edit Map***”:



La classe definisce il funzionamento dell’editor di mappe, dove il giocatore può creare layouts di mappe a piacimento sfruttando i 3 tipi di tiles terreno disponibili (terriccio, acqua e strada), gestendo di conseguenza anche il numero di spawn dei nemici, la mappa generata può poi essere salvata come un file txt, pronta per essere in caso provata in una partita vera e propria tramite l’apposita scena di gioco di cui verrà parlato in seguito.

La classe si compone di:

- Un oggetto EditingBar per gestire i pulsanti di scelta e salvataggio/ritorno al menu’

- Una immagine predefinita per poter generare automaticamente una mappa di default di terriccio

- Un array 2d di tiles che formano il livello

- Un oggetto tile che funge da riferimento al tile da cambiare in caso esso venga scelto dall’utente

- Un oggetto di GUI controller per l’inizializzazione dei tiles da cambiare

- Un booleano che definisce la flag di cambio del tile in caso esso venga scelto

- Un intero per il numero dell’ultimo tile evidenziato sull’asse x

- Un intero per il numero dell’ultimo tile evidenziato sull’asse y

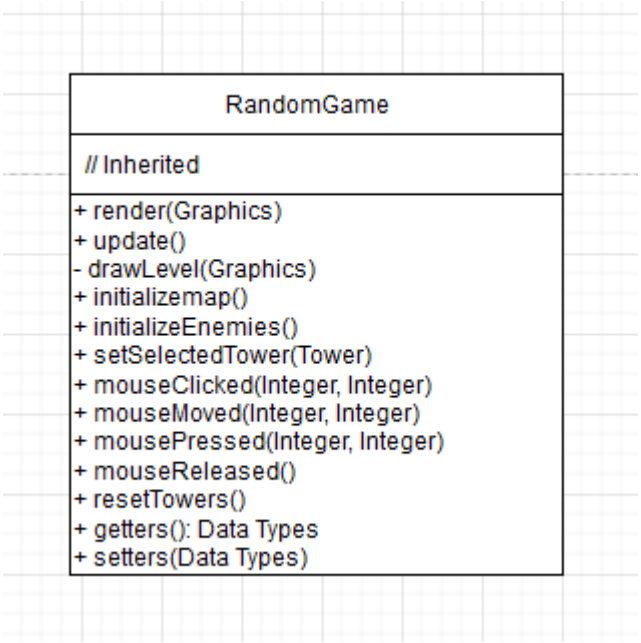
- Un intero per la coordinata x del mouse

- Un intero per la coordinata y del mouse

- Un intero per il tipo dell’ultimo tile evidenziato

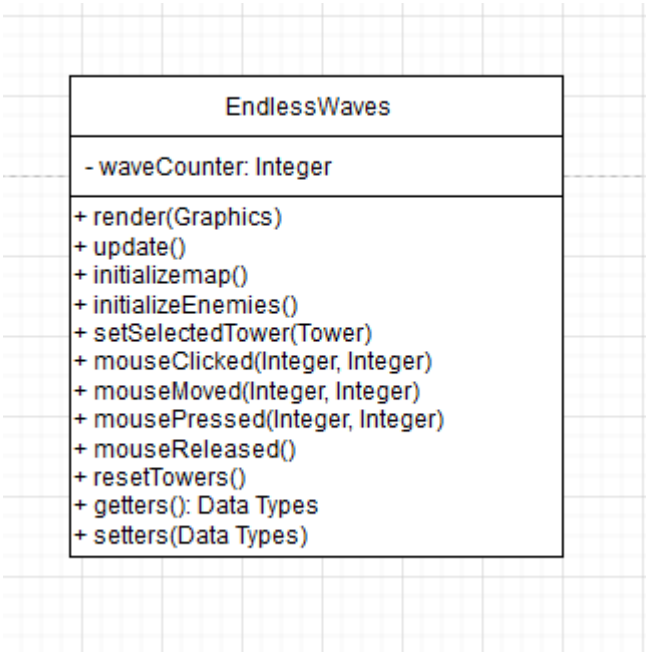
I metodi principali della classe sono il metodo *render*(*Graphics*) che gestisce i sotto-metodi di disegno componenti formando quindi il metodo di renderizzazione generale, il metodo *drawLevel*(*Graphics*) che disegna il livello di default con ogni tile definito come terriccio, *defaultLvlTilesLoader*() è responsabile del mettere a default (terriccio) tutti i tiles della mappa, *changeTile*(*Integer*, *Integer*) si occupa del cambiare il tile alla posizione definita in quello scelto dall'utente, *drawSelectedTile*(*Graphics*) si occupa di disegnare la preview del tile scelto per il cambio, *setSelectedTileToPaint*(*Tile*) si occupa del rendere possibile il cambio di un tile e di definire il tipo del tile che verrà inserito nella posizione scelta, mouseClicked(Integer, Integer) gestisce la risposta dell'applicazione ai click in determinate posizioni, *mouseMoved*(*Integer*, *Integer*) gestisce la risposta dell'applicazione al passaggio del mouse in determinate posizioni, *mousePressed*(*Integer*, *Integer*) gestisce la risposta dell'applicazione alla pressione del mouse in determinate posizioni, *mouseReleased*() gestisce la risposta dell'applicazione al rilascio della pressione del mouse.

Classe "***randomGame***":



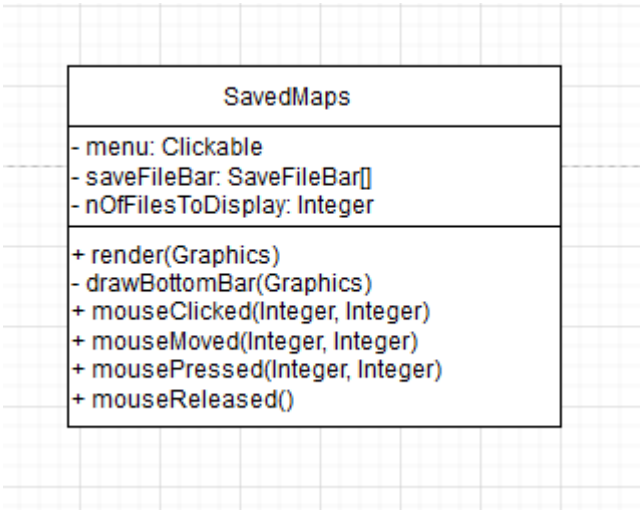
La classe eredita i propri campi dalla superclasse "***GameSceneBase***" ed implementa sia dei metodi che rimandano ai metodi della superclasse, sia delle implementazioni di metodi specifiche per la classe. In particolare questa classe definisce il comportamento della modalità di gioco "random", in cui il giocatore si trova ad affrontare un'ondata di nemici composta da un numero di nemici generato randomicamente, su una mappa generata anch'essa randomicamente e con un numero variabile di risorse. I metodi principali sono *update*() che gestisce l'aggiornamento delle condizioni come risorse, nemici e torri durante il gioco, *initializeMap*() gestisce la mappa che viene generata randomicamente, *initializeEnemies*() gestisce l'inizializzazione dei nemici e gli altri metodi vengono per gran parte ereditati dalla classe madre.

Classe “***EndlessWaves***”:



Anche questa classe eredita i metodi principali dalla classe madre “***GameSceneBase***”, in piu’ specializza alcuni metodi principali tra cui *update*() che come in precedenza si occupa di gestire l’aggiornamento delle condizioni del gioco, in questo caso aggiornando anche il fattore “ondata”, *initializeMap*() stavolta non genera una mappa casuale bensì sfrutta una mappa già generata appositamente per la modalità, dato che è una modalità ad ondate senza fine, non c’è bisogno di cambiare la mappa ad ogni partita, *resetTowers*() oltre che a resettare le torri, resetta anche altri valori come il contatore delle ondate ed il contatore delle risorse iniziali, il resto dei metodi è per gran parte ereditata dalla classe madre.

Classe “***Saved Maps***”:



La classe si occupa della gestione della selezione delle mappe custom salvate per giocare, essa mostrerà una serie di scelte che saranno composte dalle mappe precedentemente salvate, è composta in particolare da:

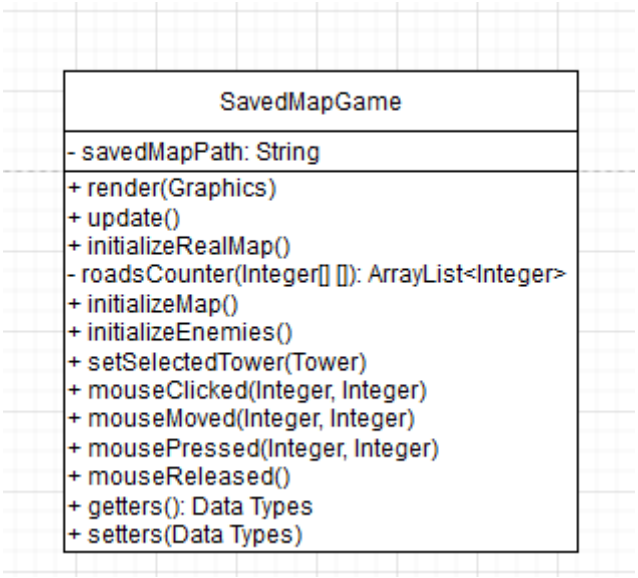
- Un oggetto menu' che permetterà il ritorno al menu' principale

- Un array di oggetti SaveFileBar che corrisponderanno alle barre di scelta del livello da caricare

- Un intero che corrisponderà al numero di livelli disponibili per essere caricati

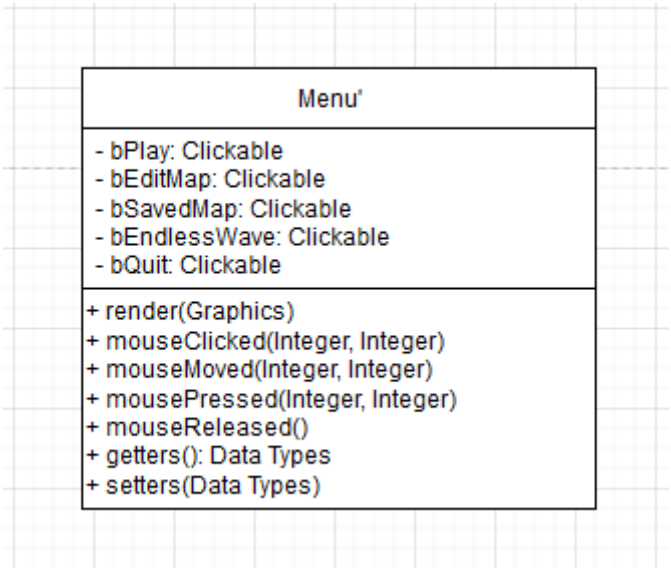
I metodi principali della classe sono *render*(*Graphics*) e *drawBottomBar*(*Graphics*) che si occupano della renderizzazione dei componenti della classe ed il metodo mouseClicked(Integer, Integer) che come per le precedenti classi gestisce la risposta dell'applicazione ai click in determinate posizioni, codesta classe consente di lanciare un'ulteriore classe ovvero la "***SavedMapGame***", che verrà mostrata di seguito.

Classe "***SavedMapGame***":



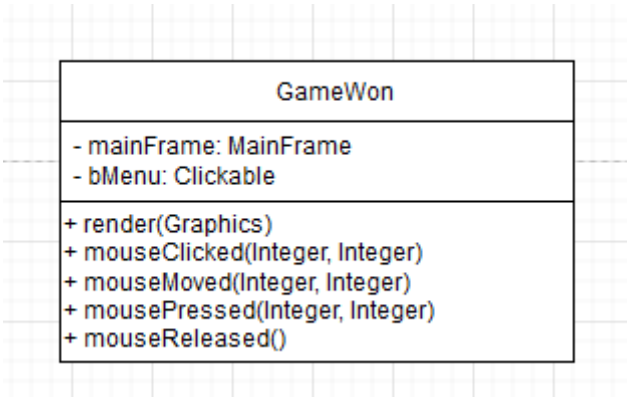
La classe definisce la modalità di gioco inizializzata dopo aver scelto una mappa custom dalla pagina di scelta precedentemente mostrata, in particolare possiede solo un campo non ereditato ovvero una stringa per contenere il path della mappa da caricare per la partita, i metodi principali invece sono *initializeRealMap*() che si occupa di prendere i dati della mappa custom scelta in precedenza ed utilizzarli per formare la mappa di gioco vera e propria, *roadCounter*(*Integer*[] []) che si occupa di analizzare il numero e la posizione delle strade nella mappa scelta, *initializeMap*() si occupa dell'inizializzazione predefinita della mappa a tempo di lancio dell'applicazione.

Classe “***Menu***”:



La classe rappresenta il menu’ che permette la scelta delle modalità, essa è la prima schermata che viene presentata al lancio dell’applicazione, in particolare contiene una serie di oggetti Clickable ovvero pulsanti che permettono la scelta delle varie modalità. I metodi principali sono *render*(Graphics) che come nelle classi precedenti si occupa del rendering dei componenti della schermata, la classe *mouseClicked*(*Integer*, *Integer*) si occupa come nelle classi precedenti di gestire la risposta dell’applicazione ai click in determinate posizioni.

Classe “***GameWon***”:



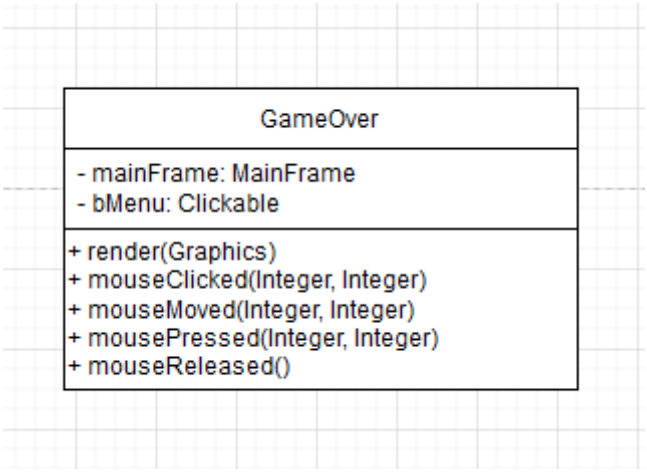
La classe comprende principalmente:

-Un oggetto MainFrame per riferimento

- Un oggetto Clickable ovvero un pulsante per tornare nel menu’

I metodi piu' importanti della classe sono *render*(*Graphics*) e *mouseClicked*(*Integer*, *Integer*) che come nelle classi precedenti si occupano rispettivamente della gestione della renderizzazione e della risposta del programma al click in determinate posizioni.

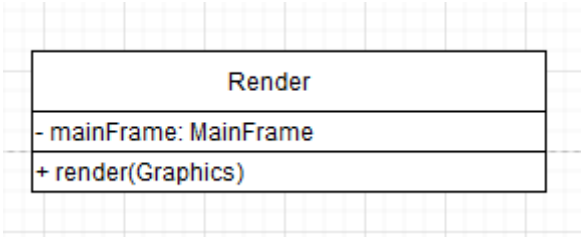
Classe "***GameOver***":



La classe presenta un oggetto MainFrame come riferimento ed un oggetto Clickable ovvero un pulsante per tornare nel menu' principale.

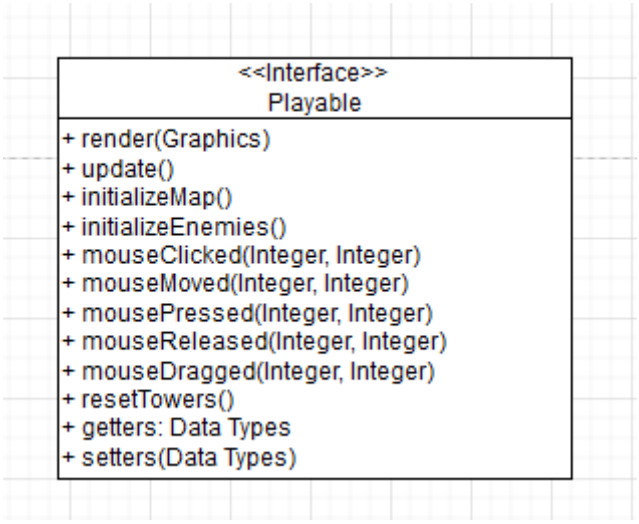
I metodi hanno un funzionamento pressoché identico alla classe precedente "***GameWon***", non presenta notevoli differenze.

Classe "***Render***":

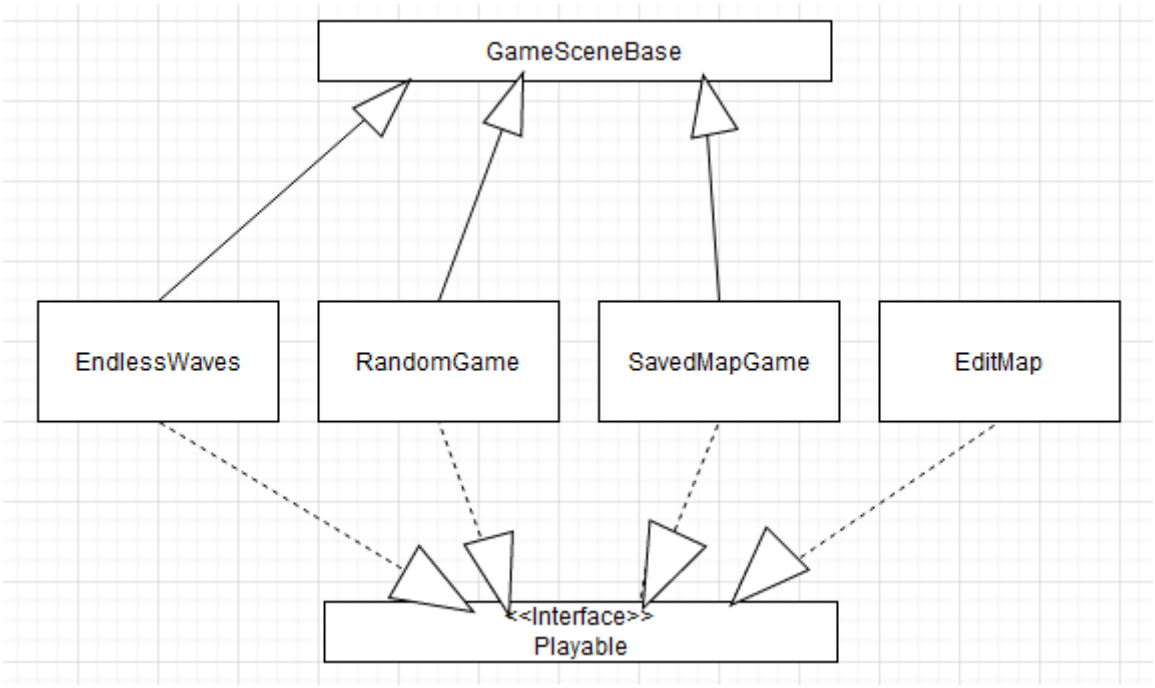


La classe contiene solo un oggetto MainFrame come riferimento ed il metodo principale è *render*(*Graphics*) che in questo caso sfrutta la variabile statica della classe GameScenes per definire quale scena del gioco renderizzare, la variabile è inizializzata alla scena menu' ma può essere cambiata dagli altri componenti, cambiando inoltre di conseguenza anche l'oggetto della renderizzazione di questa stessa classe.

L’interfaccia implementata da diverse di queste classi è l’interfaccia “***Playable***”:

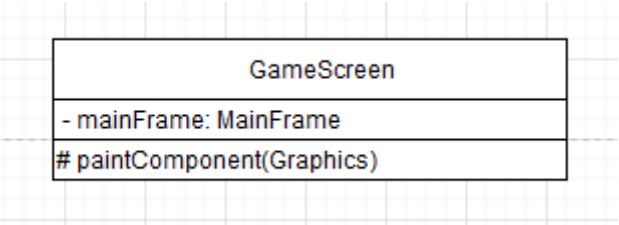


Lo schema della gerarchia dell’eredita/implementa di questo macro-componente è:



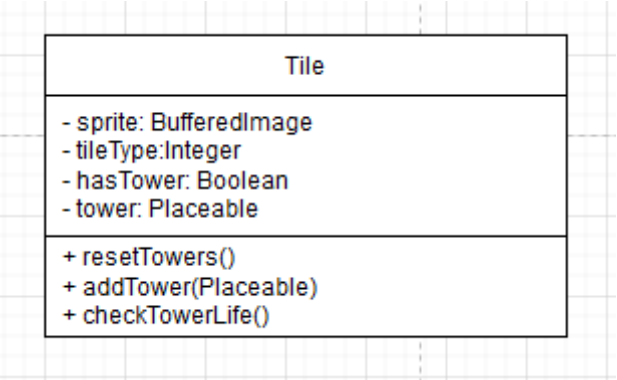
Di seguito vengono presentate le classi della macro-componente "***guiComponents":***

Classe "***GameScreen***":



La classe "GameScreen" è un'estensione della classe JPanel, contiene un oggetto di riferimento MainFrame ed un solo metodo *protected* ovvero *paintComponent*(*Graphics*), ereditato ma con un override che permette di renderizzare costantemente tutti i componenti necessari.

Classe "***Tile***":



La classe è formata da:

- Un'immagine che definisce lo sprite che verrà assegnato all'oggetto durante l'inizializzazione

- Un intero che definisce il tipo del tile

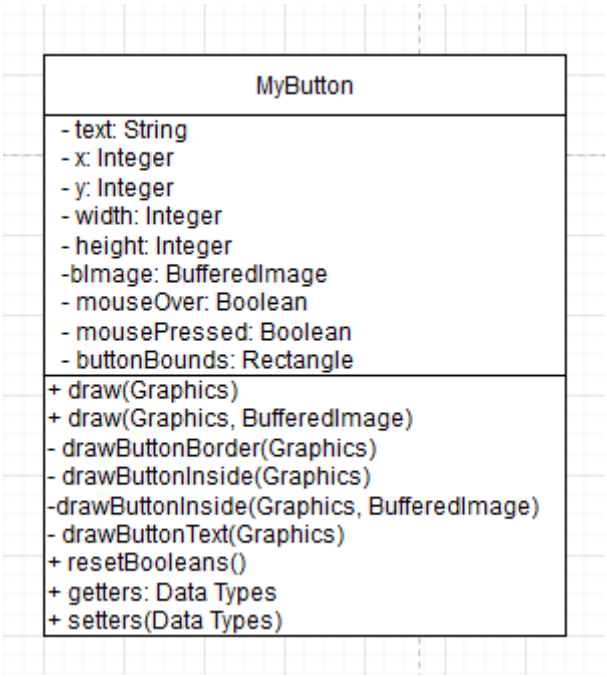
- Un booleano che definisce se il tile contiene una torre

- Un oggetto Placeable (Torre) che gestisce l'inizializzazione e gestione dell'eventuale torre nel tile

I metodi principali sono *resetTower*() che eliminano la torre nel tile inizializzando ad una condizione di base, *addTower*(*Placeable*) permette di aggiungere ed inizializzare la torre scelta dall'utente nel tile, *checkTowerLife*() permette di controllare lo stato della torre e definire se essa ha ancora punti vita o se deve essere eliminata.

Come intuibile, la classe ***Tile*** compone il mattone fondamentale dell'intera grafica del progetto, ovvero definisce il terreno di gioco ma anche la possibilità del piazzamento di torri e la logica dietro di esse.

Classe “***MyButton***”:



La classe ***myButton*** consiste in:

- Una stringa che definisce il testo contenuto nel pulsante

- Un intero che corrisponde alla posizione x del pulsante

- Un intero che corrisponde alla posizione y del pulsante

- Un intero che corrisponde alla larghezza del pulsante

- Un intero che corrisponde all’altezza del pulsante

- Un’immagine che corrisponde all’eventuale immagine del pulsante (ad esempio scelta torri)

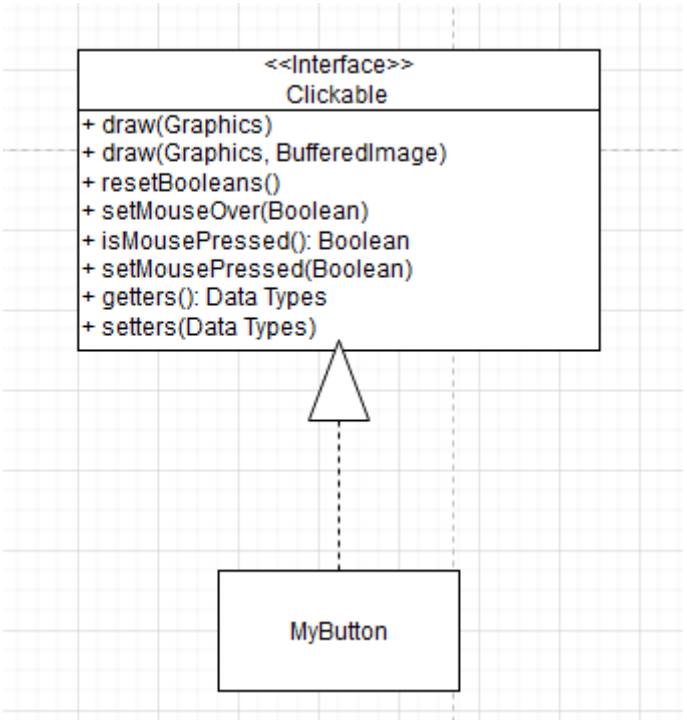
- Un booleano per definire la presenza del mouse sopra al pulsante

- Un booleano per definire la pressione del mouse sopra al pulsante

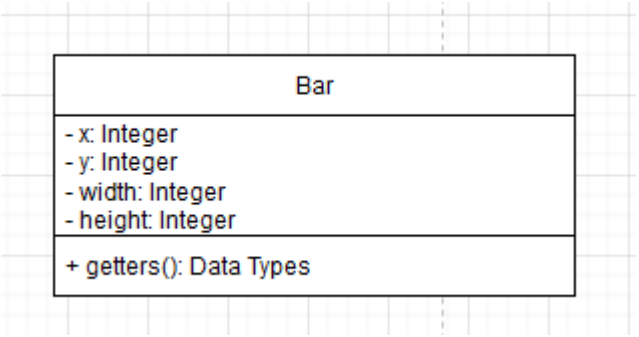
- Un oggetto Rectangle per definire il contorno del pulsante (per capire se il mouse è dentro quel contorno)

I metodi principali della classe sono *draw*(*Graphics*) e la sua variante *draw*(*Graphics*, *BufferedImage*) che si occupano di disegnare il pulsante ed i suoi componenti tramite i sotto-metodi *drawButtonBorder*(*Graphics*), *drawButtonInside*(*Graphics*), la sua variante *drawButtonInside*(*Graphics*, *BufferedImage*), *drawButtonText*(*Graphics*) e *resetBooleans* che permette di disattivare i booleani per la presenza e pressione del mouse.

La classe ***MyButton*** implementa l'interfaccia "***Clickable***":



Classe "***Bar***":



La classe consiste in:

- Un intero per la posizione x della barra

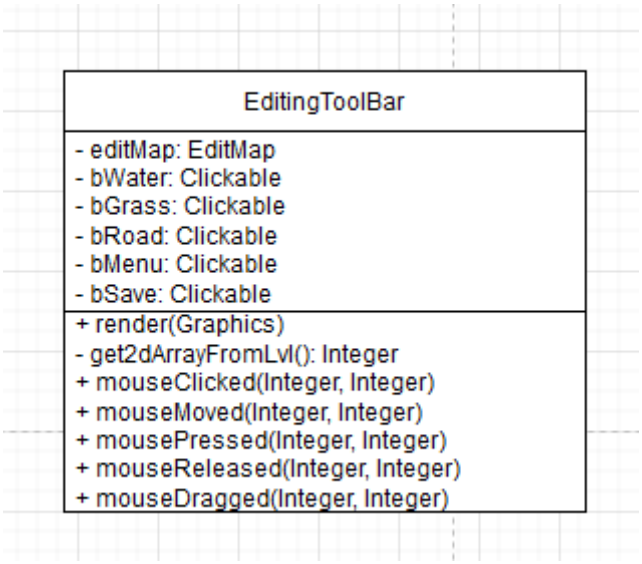
- Un intero per la posizione y della barra

- Un intero per la larghezza della barra

- Un intero per l'altezza della barra

I metodi della classe consistono in getters per ottenere i valori dei campi.

La classe "***EditingToolBar***":



La classe consiste in:

- Un oggetto di riferimento EditMap

- Un oggetto Clickable per il pulsante del tile di acqua

- Un oggetto Clickable per il pulsante del tile di terriccio

- Un oggetto Clickable per il pulsante del tile di strada

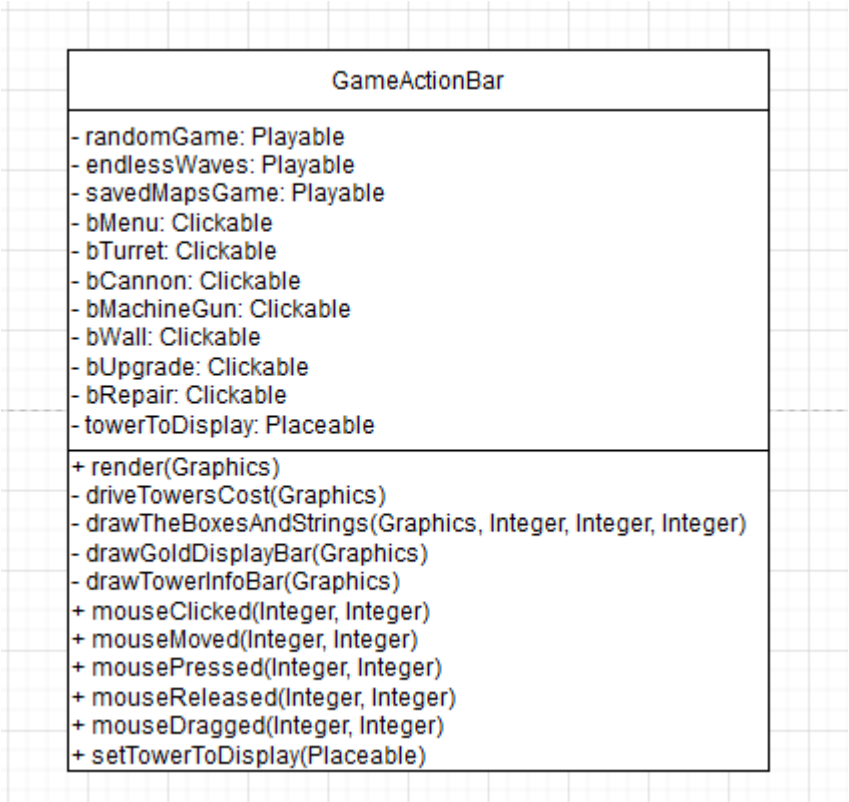
- Un oggetto Clickable per il pulsante menu

- Un oggetto Clickable per il pulsante salva

I metodi principali sono *render*(*Graphics*) che si occupa di renderizzare i componenti della pagina, *get2dArrayFromLvl*() che ottiene l'array di interi 2d dalla mappa per salvarlo come un file e renderlo disponibile per una partita, *mouseClicked*(*Integer*, *Integer*) e gli altri metodi simili hanno lo stesso funzionamento che hanno nelle classi precedenti.

La classe forma la barra che viene usata nella scena di editing map per scegliere i tiles con cui modificare la mappa o salvarla e tornare al menu'.

La classe “***GameActionBar***”:



La classe si compone di:

-Un oggetto di riferimento Playable che corrisponde al randomGame

- Un oggetto di riferimento Playable che corrisponde all’endlessWaves

- Un oggetto di riferimento Playable che corrisponde al savedMapsGame

- Un oggetto Clickable che corrisponde al pulsante menu’

- Un oggetto Clickable che corrisponde al pulsante di scelta della torre turret

- Un oggetto Clickable che corrisponde al pulsante di scelta della torre cannon

- Un oggetto Clickable che corrisponde al pulsante di scelta della torre machineGun

- Un oggetto Clickable che corrisponde al pulsante di scelta della torre wall

- Un oggetto Clickable che corrisponde al pulsante di scelta del potenziamento torre

- Un oggetto Clickable che corrisponde al pulsante di scelta della riparazione torre

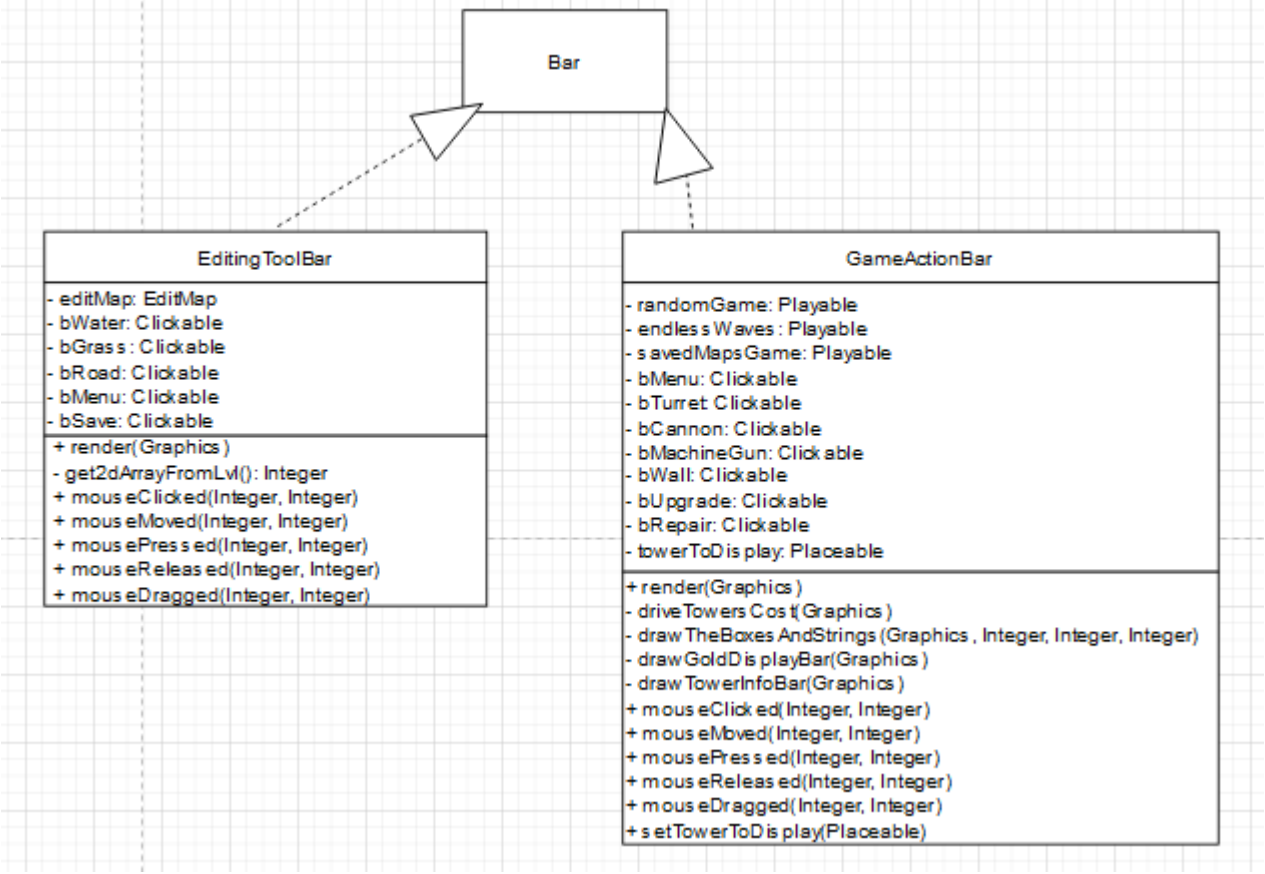
- Un oggetto di riferimento Placeable che definisce la torre da mostrare nella barra

I metodi principali della classe sono *render*(*Graphics*) che come nelle precedenti classi si occupa di renderizzare i necessari componenti, *drawTowersCost*(*Graphics*) che si occupa di disegnare i costi

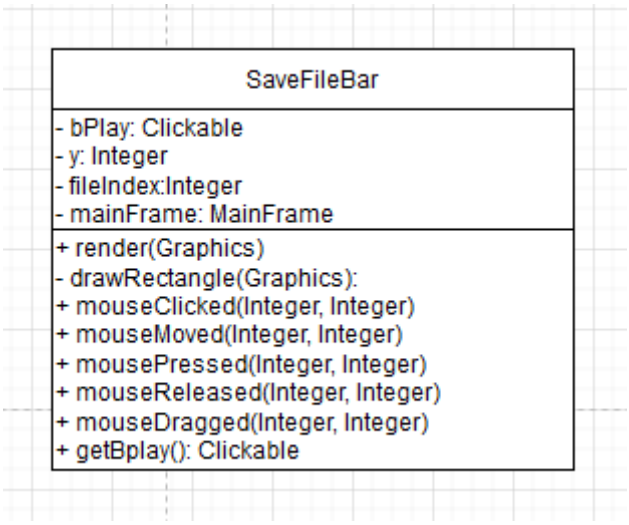
delle torri sopra ai loro rispettivi pulsanti, *drawTheBoxesAndStrings*(*Graphics*, *Integer*, *Integer*, *Integer*) che si tratta in realtà del sotto-metodo utilizzato dal metodo precedente per il disegno dei costi delle torri, *drawGoldDisplayBar*(*Graphics*) che si occupa di disegnare la barra che mostra l'oro disponibile nella partita, *drawTowerInfoBar*(*Graphics*) che mostra le informazioni della torre selezionata dall'utente, mouseClicked(Integer, Integer) e gli altri metodi simili hanno lo stesso compito che avevano nelle classi precedenti, *setTowerToDisplay*(*Placeable*) si occupa di inizializzare la torre scelta dal giocatore da mostrare.

La classe compone la barra da cui scegliere le torri da inserire ed i dettagli della partita come risorse o dati delle torri stesse

Entrambe le classi ***EditingToolBar*** e ***GameActionBar*** sono sottoclassi della classe ***Bar:***



Classe “***SaveFileBar***”:



La classe è costituita da:

- Un oggetto Clickable che corrisponde al pulsante play

- Un intero per la coordinata y

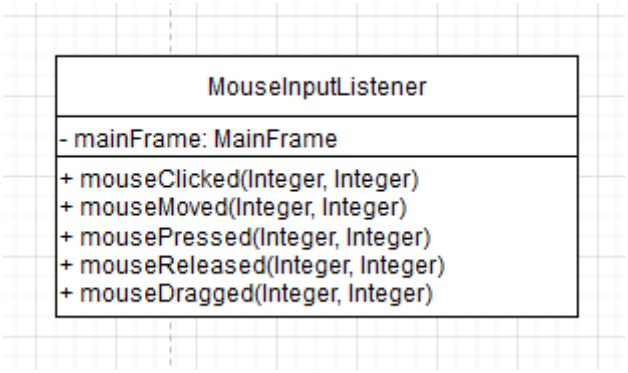
- Un intero per l’indice del file da scegliere

- Un oggetto di riferimento mainFrame

I metodi principali sono *render*(*Graphics*) che si occupa della renderizzazione dei componenti, *drawRectangle*(*Graphics*) è il sottometodo per disegnare il rettangolo in cui verrà mostrato il file da scegliere ed il pulsante per giocarlo, *mouseClicked*(*Integer*, *Integer*) e gli altri metodi simili hanno lo stesso compito di quelli nelle precedenti classi.

La classe compone la barra che rappresenta una possibilità di file da giocare tra le mappe custom, per ogni mappa custom verrà generata una barra di questo tipo.

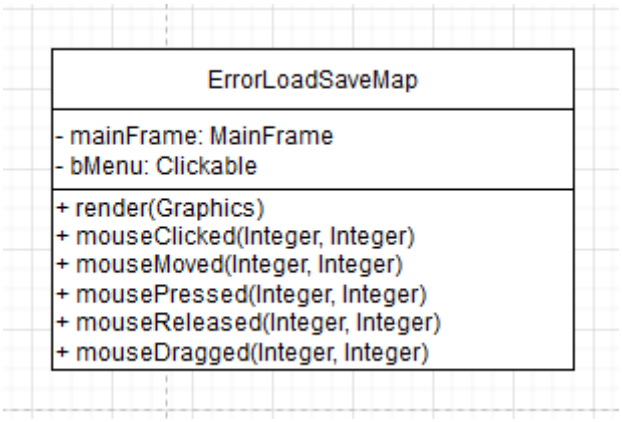
Classe “***MouseInputListener***”:



La classe è composta da un oggetto di riferimento mainFrame, inoltre i metodi principali sono i *mouseClicked*(*Integer*, *Integer*) e simili che definiscono i comportamenti da attuare dal programma basandosi sul valore della variabile GameScenes.gameScenes ovvero dalla scena attualmente in esecuzione.

La classe è fondamentale per la gestione della risposta del programma ai click dell'utente.

Classe "***LoadSaveMap***":



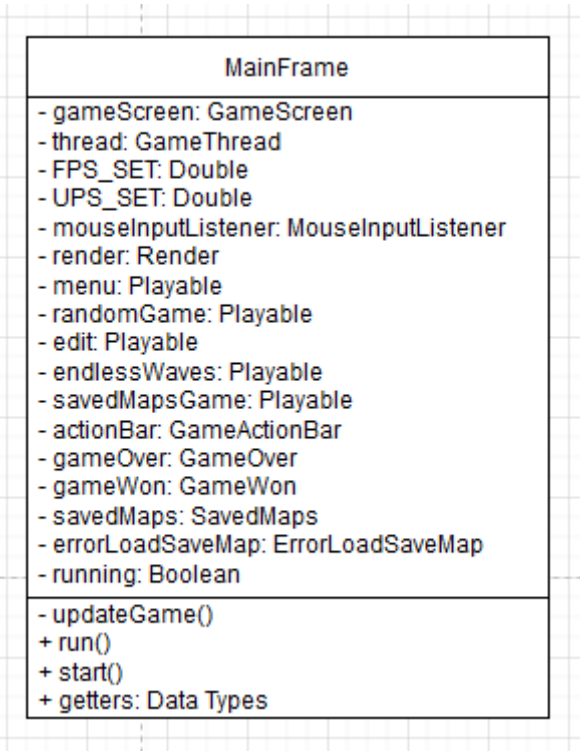
La classe si compone di:

- Un oggetto di riferimento mainFrame

- Un oggetto Clickable che rappresenta il pulsante menu

I metodi principali sono *render*(*Graphics*) che renderizza i componenti ed i metodi *mouseClicked*(*Integer*, *Integer*) e simili, che anche qui hanno gli stessi compiti di quelli nelle classi precedenti.

Classe “***MainFrame***”:



La classe è composta da:

-Un oggetto riferimento di GameScreen

- Un oggetto GameThread che consente di utilizzare un thread per lo svolgimento del programma

- Un final double per FPS_SET che consente di mantenere stabile il numero di FPS

- Un final double per UPS_SET che consente di mantenere stabile il numero di UPS (updates)

- Un oggetto di MouseInputListener che implementa l’ascolto dei click e delle posizioni del mouse

- Un oggetto render che consiste nel riferimento principale per renderizzare tutti i componenti

- Un oggetto Playable che definisce la gameScene menu

- Un oggetto Playable che definisce la gameScene randomGame

- Un oggetto Playable che definisce la gameScene edit

- Un oggetto Playable che definisce la gameScene endlessWaves

- Un oggetto Playable che definisce la gameScene savedMapGame

- Un oggetto GameActionBar che definisce la barra di gioco da cui scegliere le torri da aggiungere

- Un oggetto GameOver che definisce la scena in caso la partita venga persa

- Un oggetto GameWon che definisce la scena in caso la partita venga vinta

- Un oggetto savedMaps che definisce la schermata di scelta tra le mappe custom precedentemente salvate

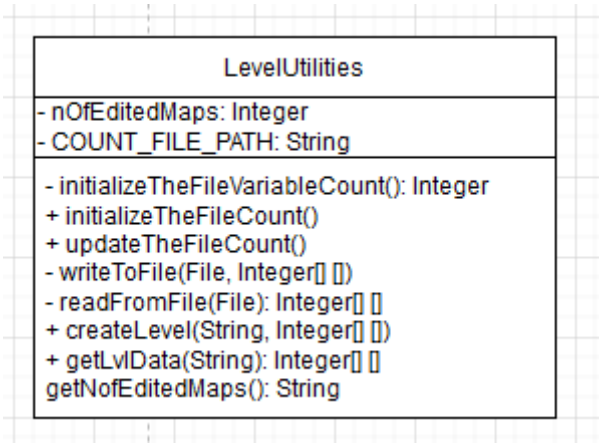
- Un oggetto ErrorLoadSaveMap che definisce la schermata di errore quando la mappa non è giocabile o quando la mappa è giocabile ed è stata vinta

- Un booleano che rappresenta lo stato di attività per il thread

I metodi principali sono *updateGame*() che si occupa degli update delle varie componenti a seconda della scena attualmente in uso, *run*() fa partire il gameloop e quindi anche gli updates e lo scorrimento generale e *start*() permette l'inizializzazione del thread.

Di seguito vengono presentate le classi della macro-componente "***imageUtilities":***

Classe "***LevelUtilities***":



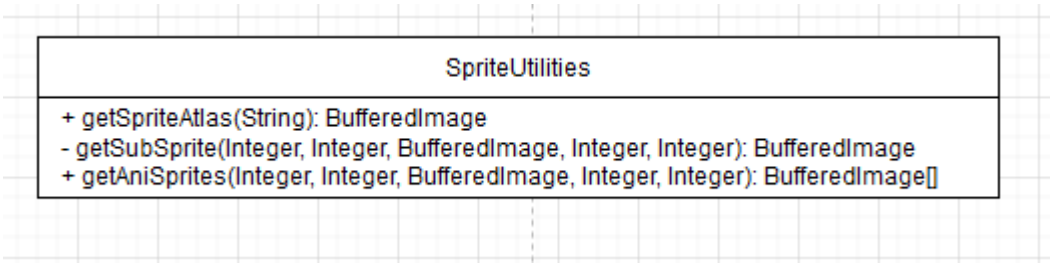
La classe è composta da:

- Un intero che corrisponde al numero di mappe editate

- Una stringa corrispondente al path del file dove è conservato il numero di mappe editate

I metodi principali della classe sono *initializeTheFileVariableCount*() che legge il numero di file editati dall'apposito documento ed inizializza la variabile descritta in precedenza, essa viene chiamata all'inizio dell'inizializzazione della classe stessa, *initializeTheFileCount*() ha lo stesso compito della classe precedente ma viene utilizzata dalle altre classi, *updateTheFileCount*() permette di aggiornare il numero di mappe editate salvato nel documento apposito, *writeToFile*(*File*, *Integer*[] []) permette di scrivere un array 2d di interi ovvero la mappa su un nuovo documento, *readFromFile*(*File*) invece ritorna un array 2d di interi che corrisponde alla mappa salvata sul documento, *createLevel*(*String*, *Integer*[] []) permette di creare un nuovo file e scrivere in esso la mappa che si intende salvare.

Classe "***SpriteUtilities***":

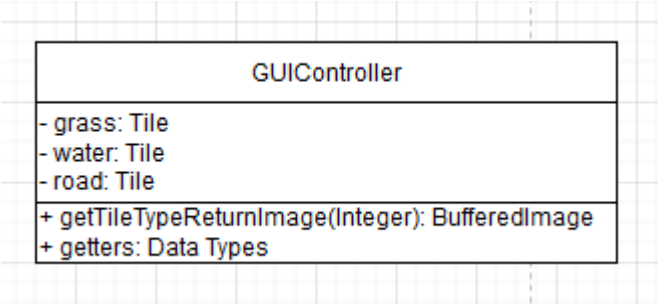


La classe è considerata una utility class quindi contiene solo metodi statici, in particolare *getSpriteAtlas*(*String*) che restituisce l'immagine al path richiesto, *getSubSprite*(*Integer*, *Integer*, *BufferedImage*, *Integer*, *Integer*) restituisce una sotto-imagine dal path richiesto, *getAniSprites*(*Integer*, *Integer*, *BufferedImage*, *Integer*, *Integer*) restituisce una serie di immagini (i frames per le animazioni).

2.4 DETTAGLI CONTROLLER

Di seguito viene mostrato l'unico componente **Controller**:

Classe "***GUIController"***:



La classe è composta da 3 oggetti di tipo Tile fondamentali che determinano la generazione e/o modifica della mappa, in particolare il metodo piu' importante è getTileTypeReturnImage(Integer) che come intuibile dal nome, restituisce un'immagine a seconda del parametro passato al metodo stesso.

Questo è l'unico controller implementato e favorisce una manipolazione semplificata e veloce dei tiles del terreno per le mappe.

2.5 Scelte progettuali, eventuali criticità di design

Come evidenziato in precedenza, il progetto ha avuto diversi cambiamenti rispetto al piano iniziale, alcuni dovuti ad esigenze di sviluppo mentre altri dovuti a semplici comodità, di seguito vengono esposte le scelte di progetto e le criticità piu' importanti:

- Per la GUI viene utilizzata la classe Graphics invece che componenti già generati e predefiniti, l'idea dietro questa scelta di progetto è puramente di comodità ed ha permesso di capire meglio come funzionano gli eventi dietro le quinte, anche per quanto riguarda la risposta dell'applicazione all'interazione con i componenti stessi.
- L'idea delle ondate multiple per ogni modalità di gioco è stata scartata e rimpiazzata con un'idea di gioco piu' legata alle singole ondate. Dato che per avere piu' ondate basta selezionare la modalità ad ondate infinite, avere multiple ondate anche nelle altre modalità risulterebbe in una ridondanza non da poco.
- Il passaggio da un modello MVC comune con un controller principale (nel caso della pianificazione di base, con 2 sotto-controller per i macro-componenti e gestiti da un controller maggiore) ad un modello non piu' molto assimilato a questo concetto di modello è stata una criticità e di conseguenza scelta definita nel corso dello sviluppo, in particolare si è preferito, dove possibile, preferire una logica dei componenti autogestita o nei casi estremi, gestita da pochi altri componenti senza l'intervento di controllori esterni che avrebbero avuto bisogno di codice e metodi non indifferenti (andando per cui contro all'idea di base di un controller snello). Per comodità è stato comunque conservato il GUIController che favorisce l'inizializzazione, modifica e mantenimento dei componenti Tile dell'interfaccia grafica, il controller risulta molto snello a differenza di come sarebbero usciti quelli per gli altri macro-componenti, per cui non è risultato in alcun modo invasivo o ridondante.
- La possibilità di vendere le unità è stata messa da parte poiché ritenuta un aiuto troppo grande, per cui si è optato per una visione piu' "Spietata" dove occorre pianificare bene le proprie mosse invece di poterle correggere quando si vuole con il passare del tempo.
- La possibilità di salvare lo stato di una partita e continuarla in un secondo momento è stata una criticità non indifferente sia per il fatto che risulterebbe controproducente per la necessità del giocatore di decidere in fretta e con precisione le proprie mosse, per cui si è optato per un'idea piu' "al momento" per il flusso di gioco, senza interruzioni e riprese.
- La decisione di incorporare aspetti come la gestione delle ondate, le risorse ed il gameloop non come componenti a sé stanti, bensì come parti delle classi a cui fanno riferimento, deriva dal fatto che si è notato nel corso dello sviluppo che questi aspetti potevano essere semplificati e non necessitavano intere classi, risparmiando codice eventualmente ridondante o non strettamente necessario.
- Non sono state implementate unità di testing jUnit dato che i test sono stati effettuati manualmente quando necessario per poter creare appositamente bug o situazioni favorevoli ad errori a tempo d'esecuzione o causati dall'utente.

3.1 Inizio, progresso e termine del processo di sviluppo

Il processo di sviluppo dell'applicazione è iniziato con una semplice implementazione della classe Enemy che all'inizio veniva pensata come astratta, poi convertita in concreta per semplice comodità, successivamente il focus è passato sul provare a disegnare qualche componente per testare le prime componenti grafiche, in seguito si è passato di nuovo ad implementare parti del Model tramite il continuo dell'implementazione per le classi inerenti i nemici. Come è possibile notare il flusso di lavoro si è spostato tra elementi del Model e della View costantemente, questo poiché l'idea era di gestire entrambe autonomamente le une dalle altre ma comunque già possibilmente funzionanti l'una al passo con l'altra. Infatti in seguito si è passato ad implementare il menu' con le sue funzionalità e possibilità di accedere alla scena di "partita random", la quale non era ancora del tutto rifinita ma dava comunque la possibilità di testare e debuggare costantemente. Lo sviluppo basilare delle scene della view è piu' o meno andato di pari passo con lo sviluppo delle classi dei nemici e della loro logica, successivamente durante la rifinitura delle classi view, sono state implementate anche le classi model per le torri e la loro logica. In seguito la logica delle animazioni viene terminata e viene generata la "Final Commit", ovvero una versione non completa ma stabile e con tutte le funzionalità di base, il tutto ai fini di testing e per iniziare a definire i bordi definitivi del progetto, questa versione è stata poi completata tramite la rifinitura della logica di gioco tra cui dello spawn ed animazioni/danno dei nemici ed anche delle torri, gli ultimi step sono stati aggiungere la possibilità di scegliere e caricare una mappa salvata per poterci avviare una partita come quella random, la logica per vedere le info delle torri e la possibilità di ripararle e potenziarle con le risorse. L'ultimo commit contiene la versione definitiva e stabile del progetto.

3.2 Considerazioni finali sullo sviluppo

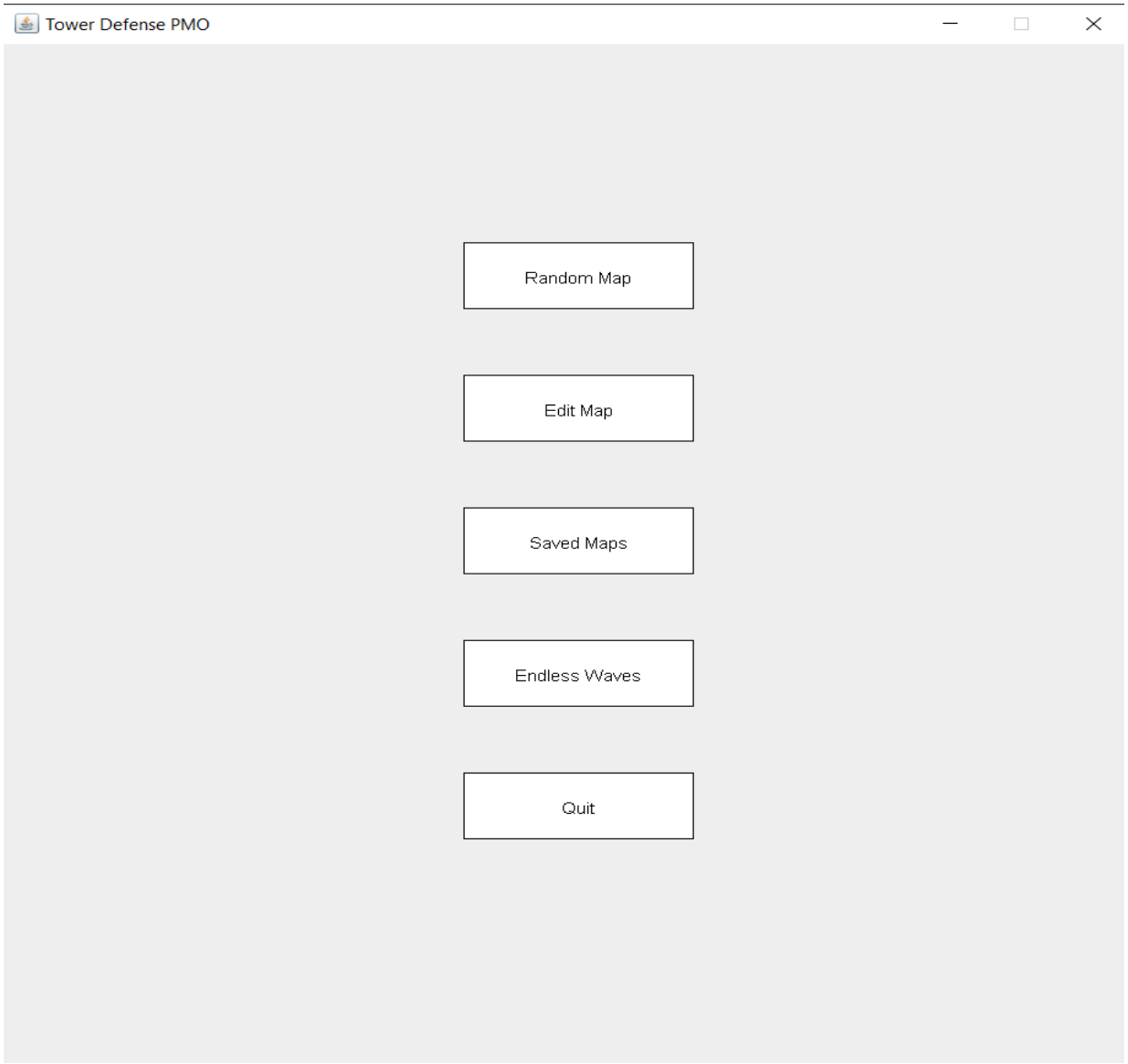
In definitiva lo sviluppo non ha avuto intoppi notevoli, probabilmente le parti leggermente piu' impegnative sono state la gestione delle animazioni e la gestione parallela tra la view ed il model nella loro interezza. Tuttavia non sono state presenti estreme criticità o problematiche insormontabili, per quanto possibile ove presente una problematica di tipo strutturale, si è sempre cercato di implementare un'alternativa piu' o meno valida. Considerato questo progetto come prima esperienza in questo campo di sviluppo, il risultato non è affatto deludente, i requisiti principali sono stati per gran parte soddisfatti e non sono presenti problematiche gravi che possono rendere l'esperienza dell'utente sgradevole, ovviamente il progetto non ha caratteristiche straordinarie o rivoluzionarie in quanto si basa su un semplice concetto, ma comunque porta a termine gli obbiettivi prefissati. L'esperienza maturata grazie a questo progetto ha sicuramente conferito uno strato in piu' di sicurezza nell'uso del linguaggio e rappresenta un buon punto di inizio per progetti in caso anche maggiori.

4.1 Info sulle modalità

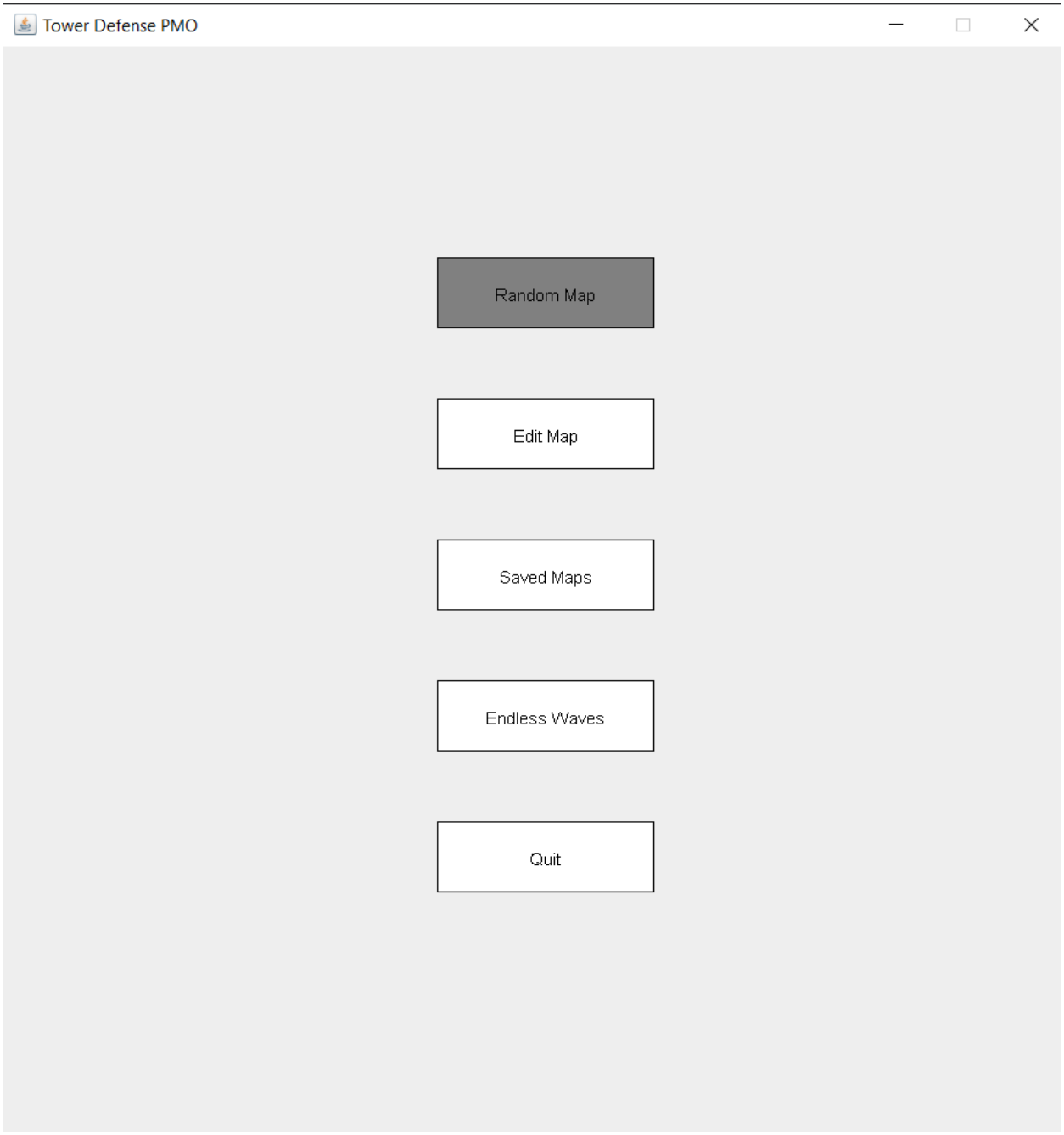
L'applicazione prevede diverse modalità di gioco, la piu' classica ed intuitiva è la modalità partita random o "**RandomGame**", la quale genera a caso la mappa ed il numero di nemici, fornendo all'utente un numero di risorse proporzionato al numero di nemici presenti nel livello.

L'utente deve quindi piazzare le torri difensive nelle postazioni idonee (in qualunque posizione a patto che siano sui tile Road) per difendersi dai nemici ed in caso di vittoria, verrà mostrata una schermata di vittoria, in caso contrario, una schermata di sconfitta, in entrambi i casi, la mappa successiva sarà comunque differente dalla precedente data la natura casuale della generazione della mappa, ciò implica teoricamente una rigiocabilità senza limiti senza possibilità di ritentare il livello in caso di sconfitta. Di seguito una serie di esempi:

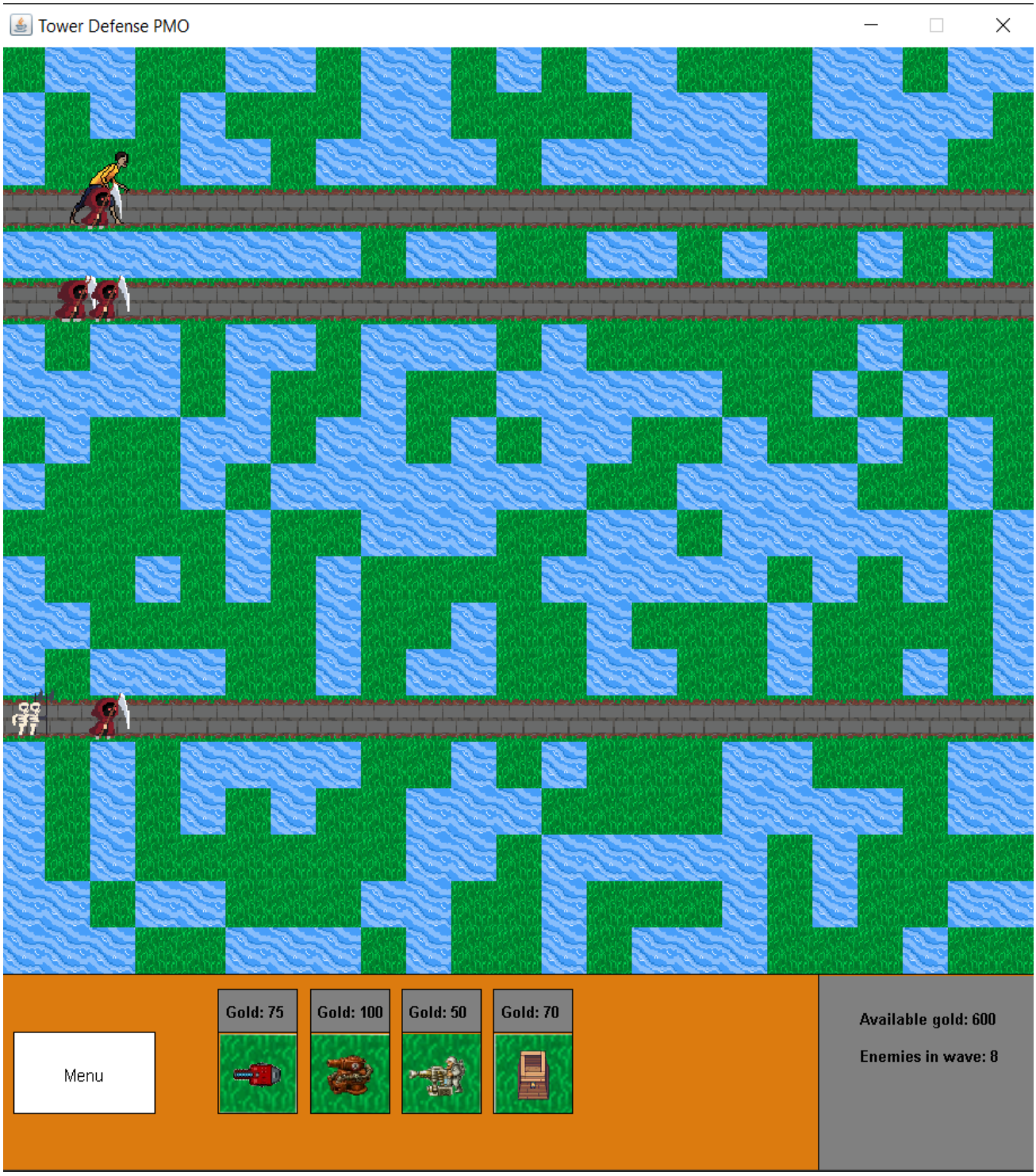
(Menu' principale)



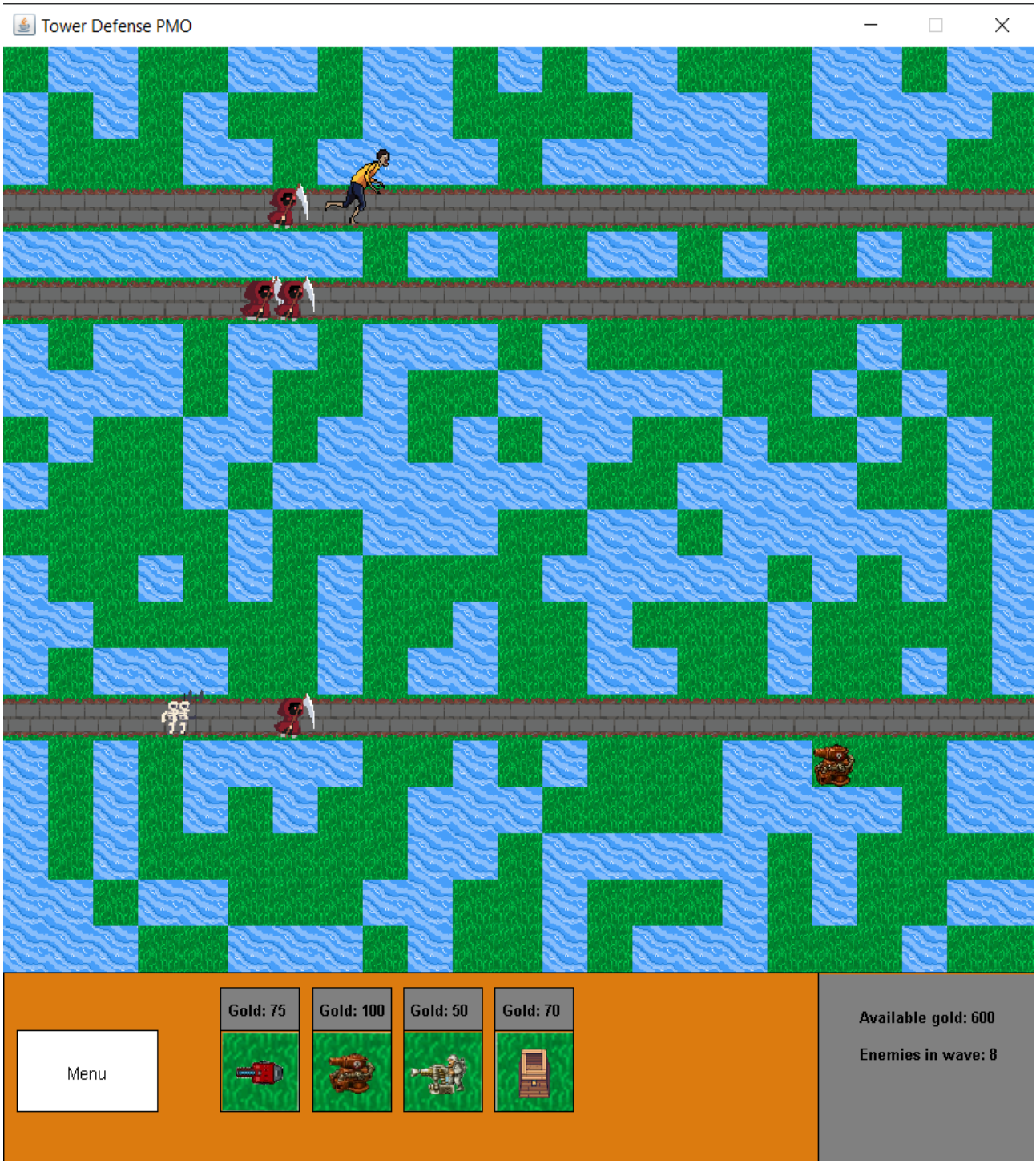
(Scelta della modalità **RandomGame**)



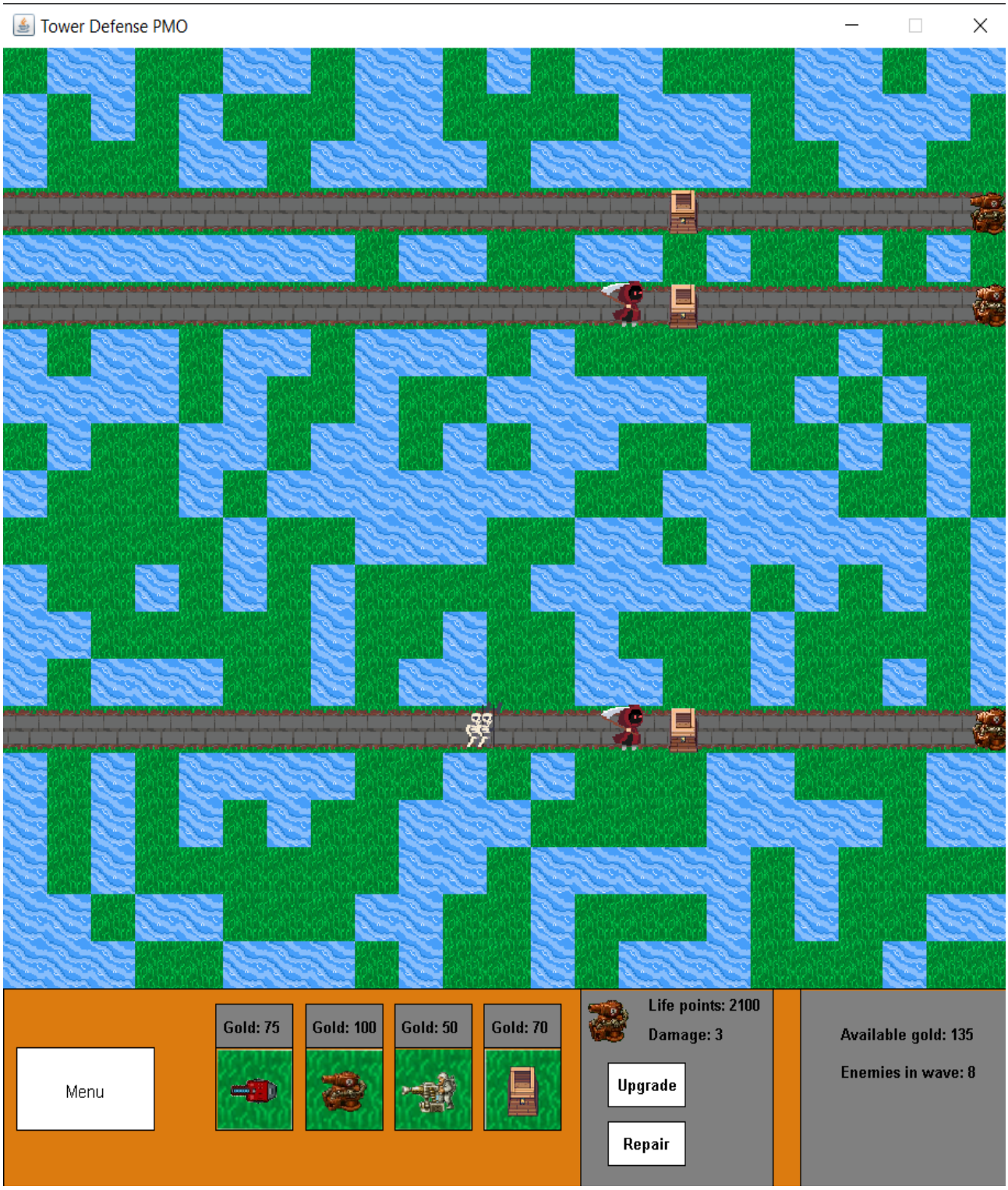
(Inizio della partita random)



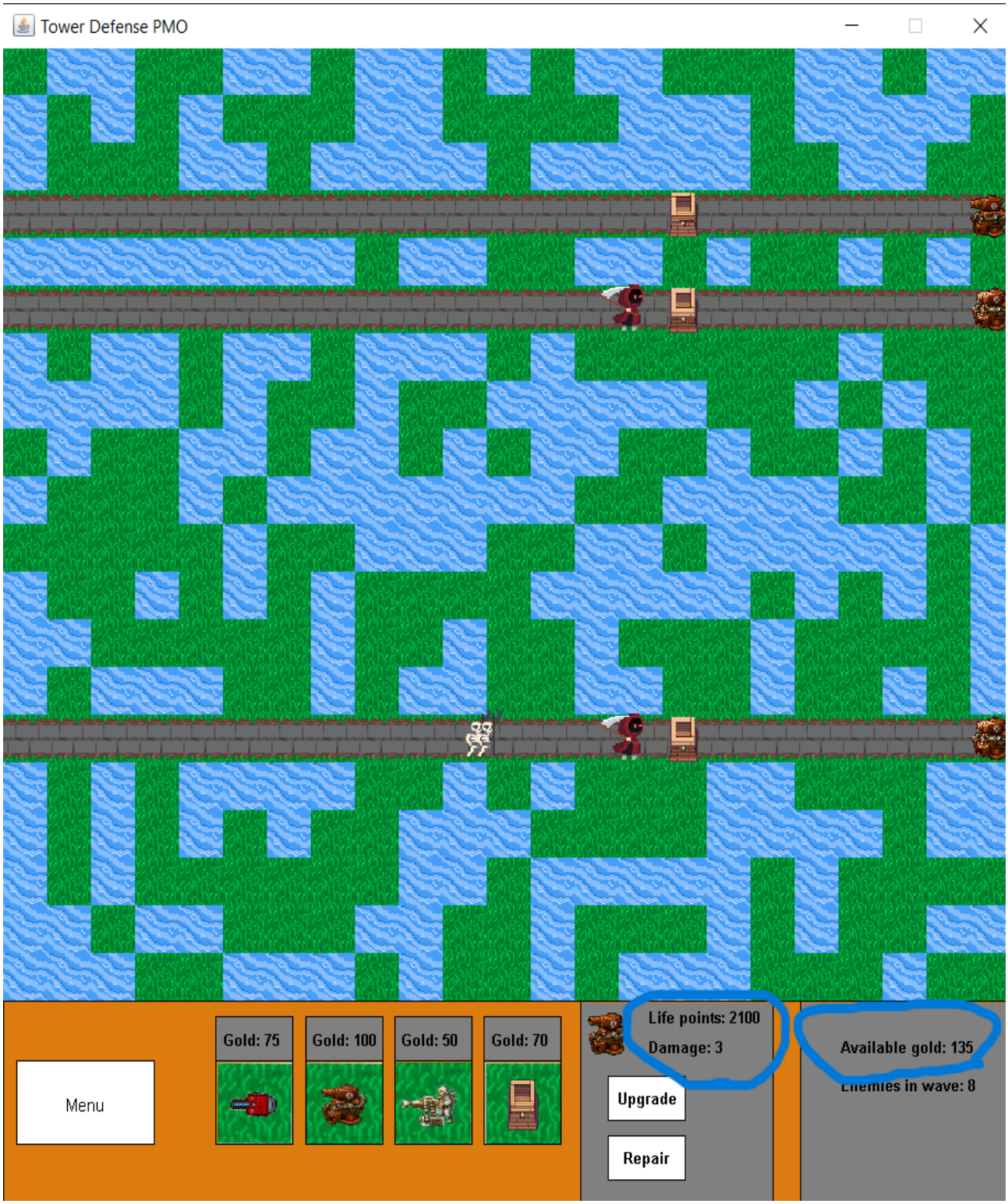
(Selezione di una torre)



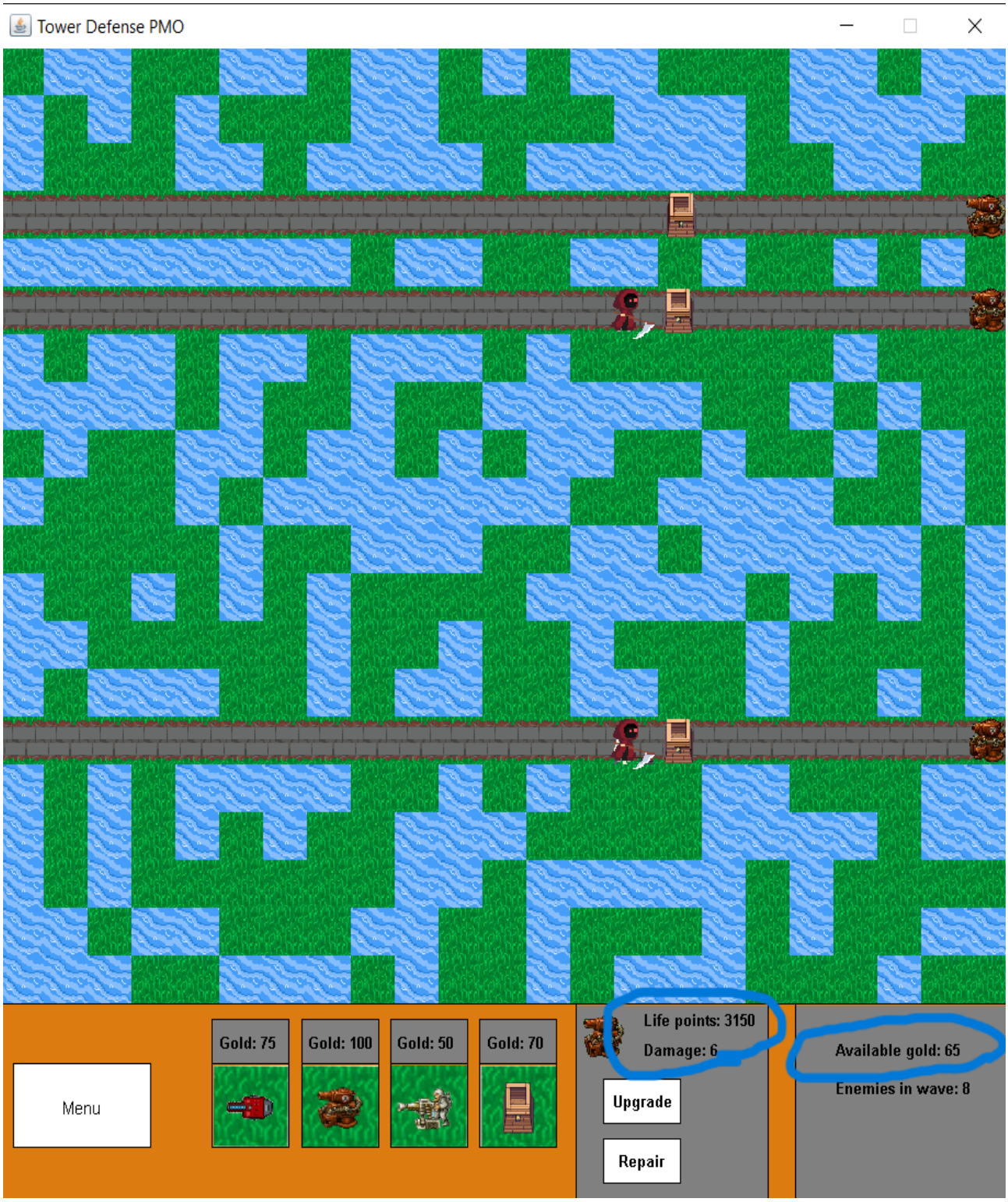
(Piazzamento torri)



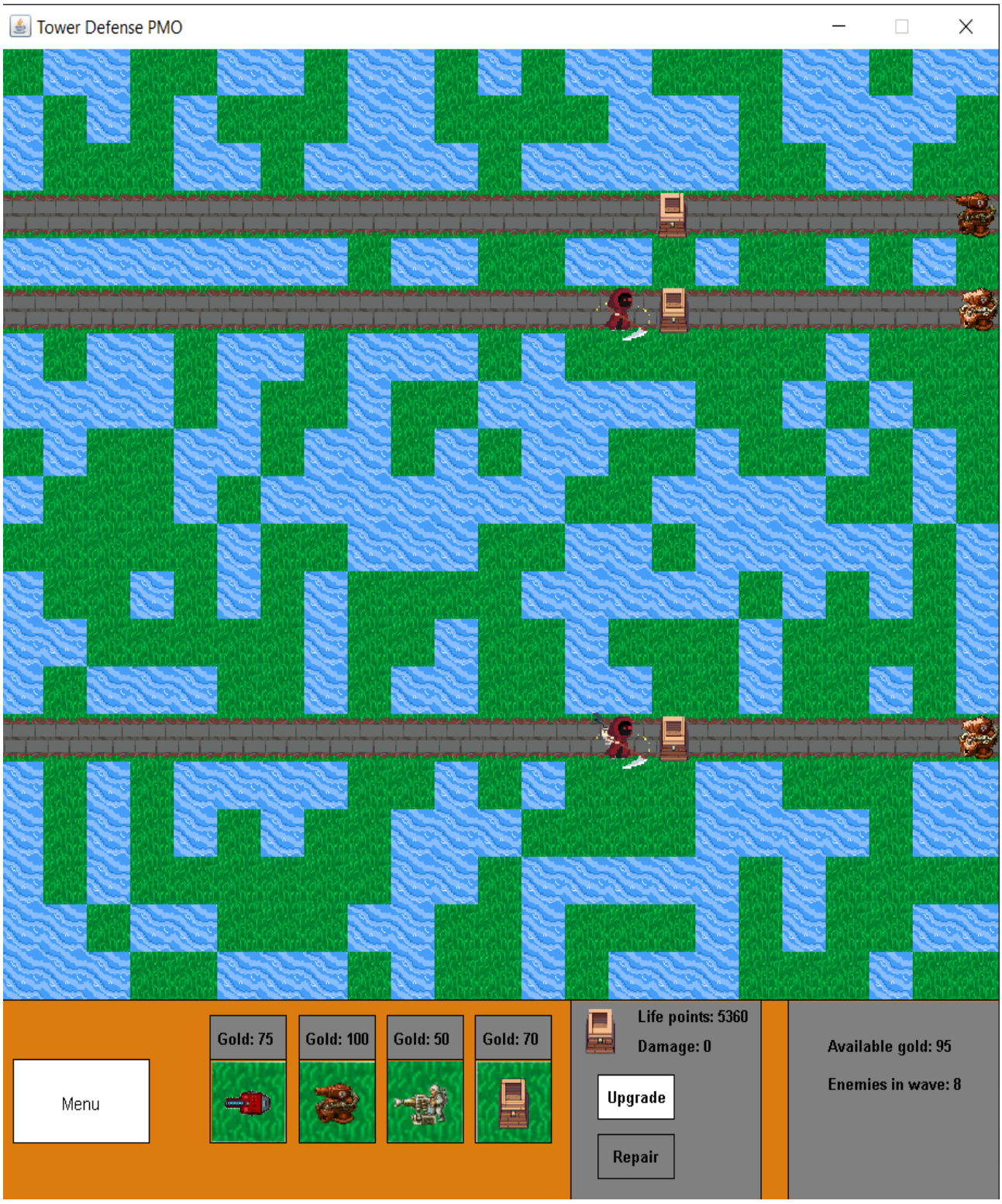
(Potenziamento torre - prima -)



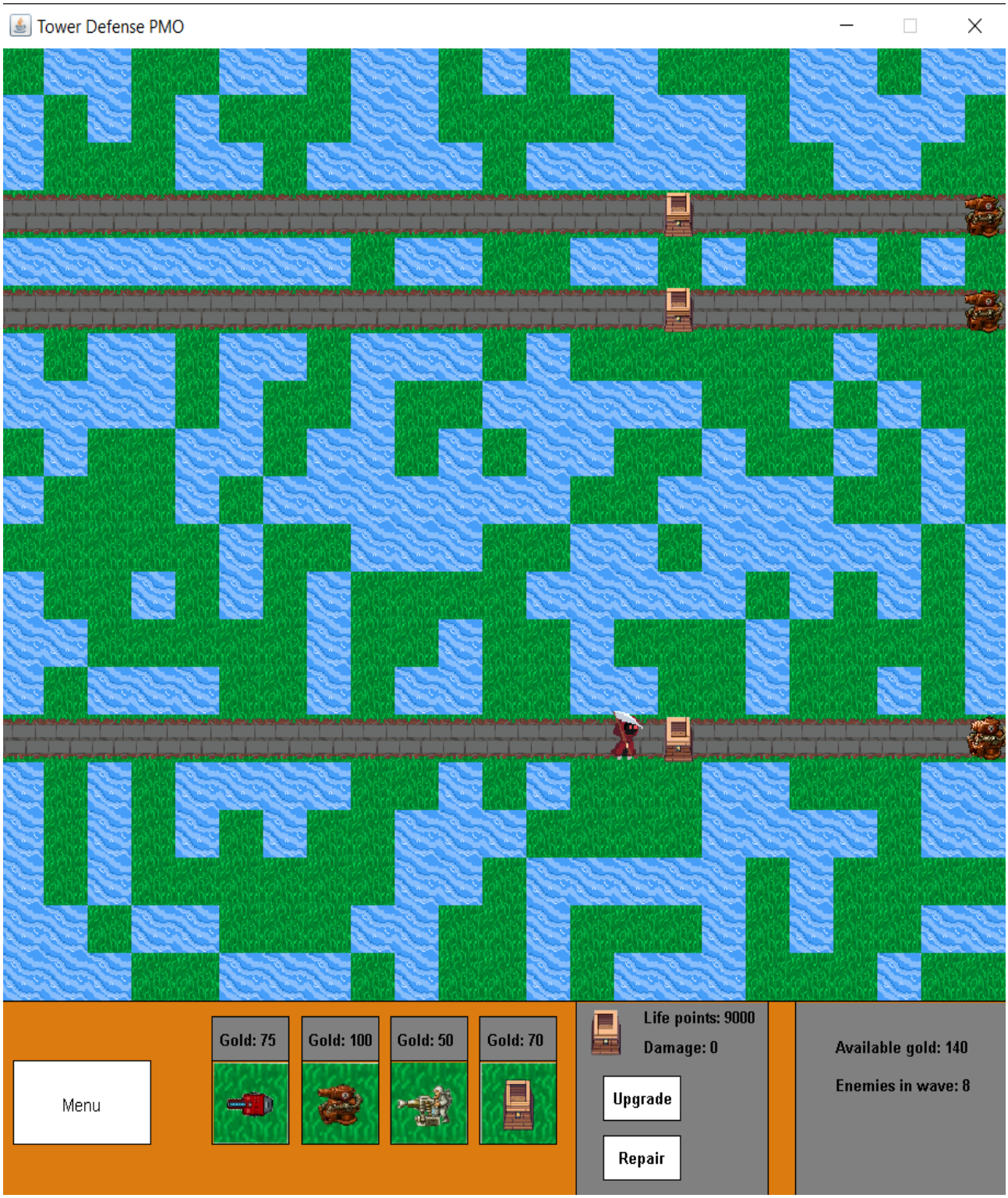
(Potenziamento torre - dopo -)



(Riparazione torre - prima -)



(Riparazione torre - dopo -)

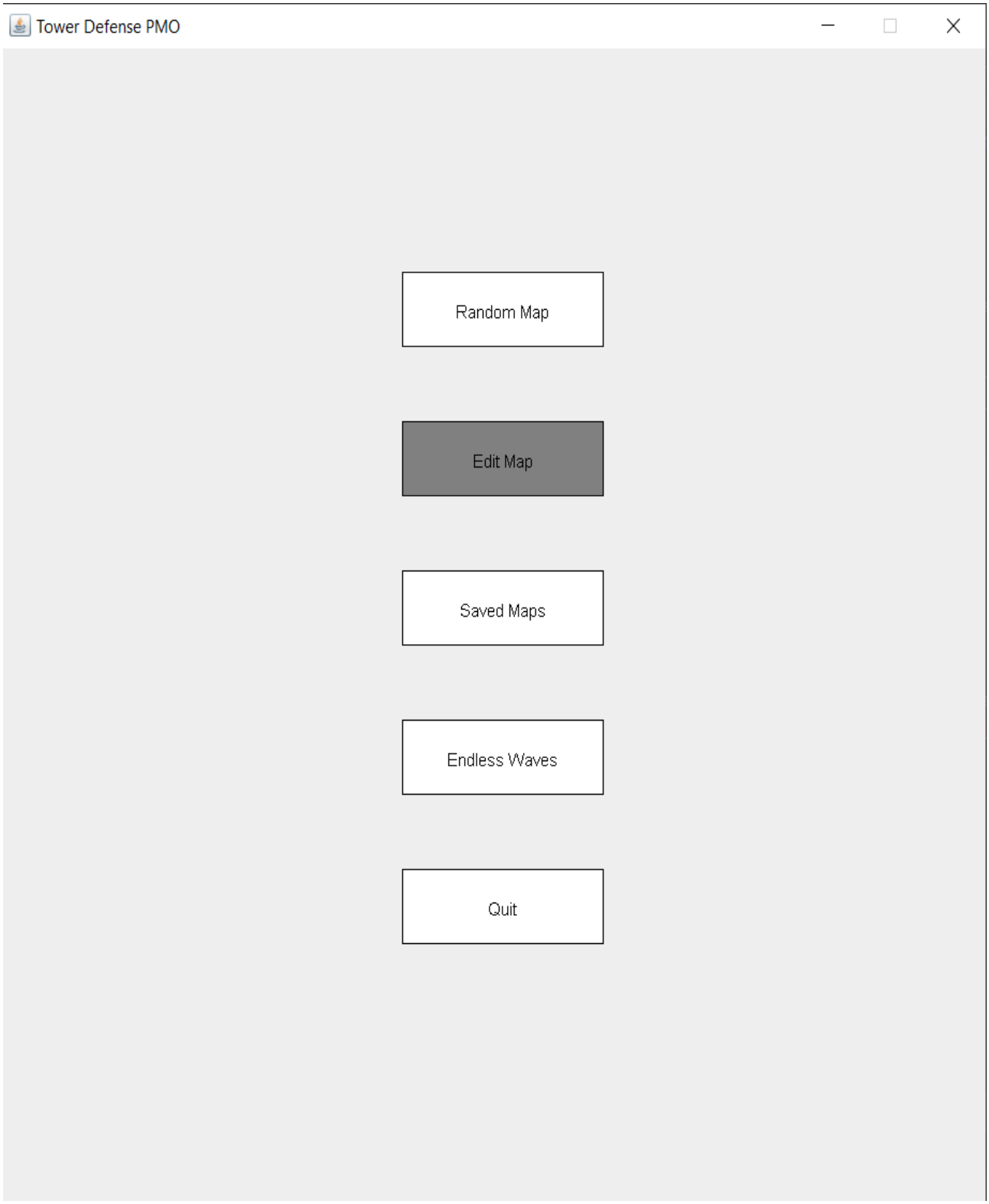


(Scenario di vittoria)

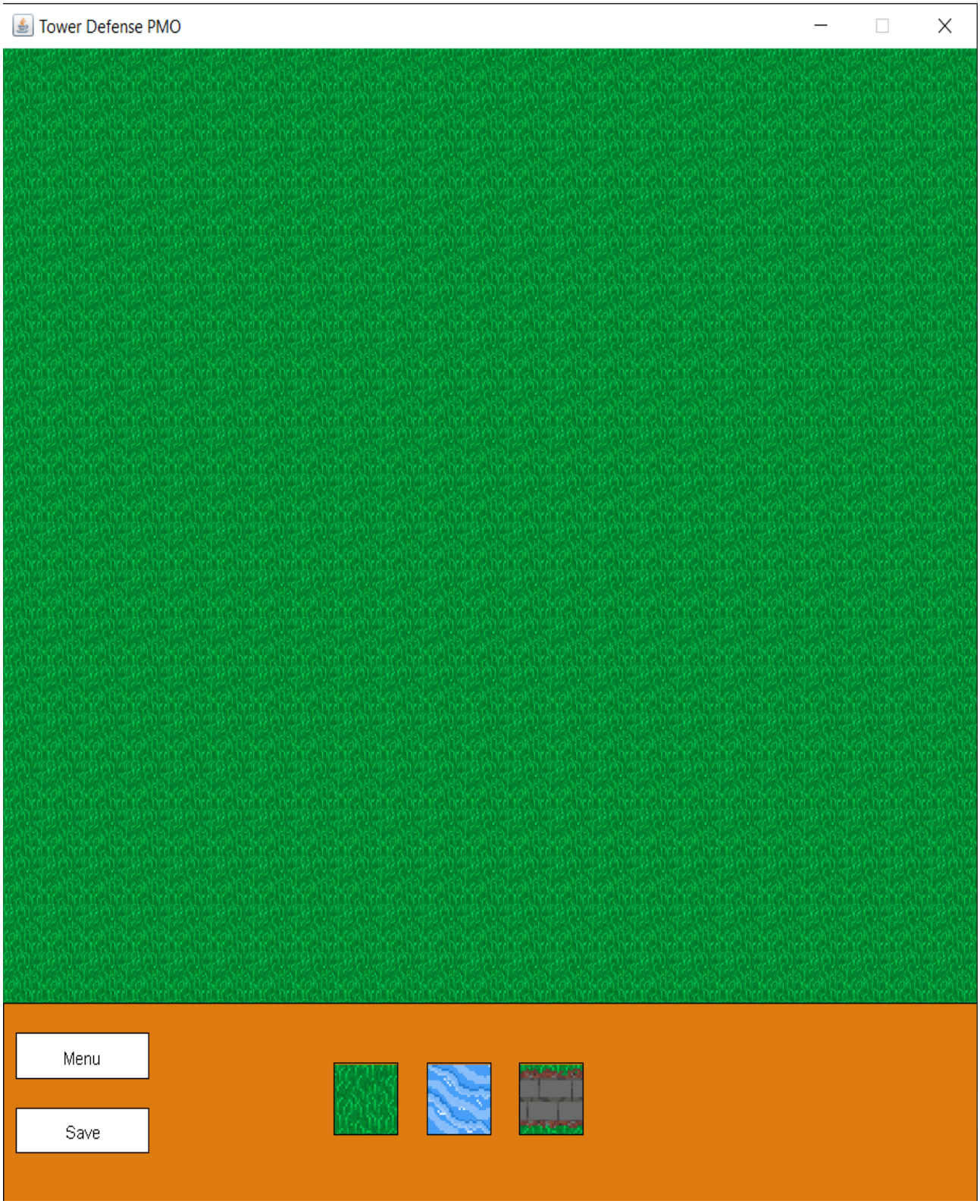
(Scenario di sconfitta)


Tower Defense PMO
YOU HAVE LOST! ;-;
Menu

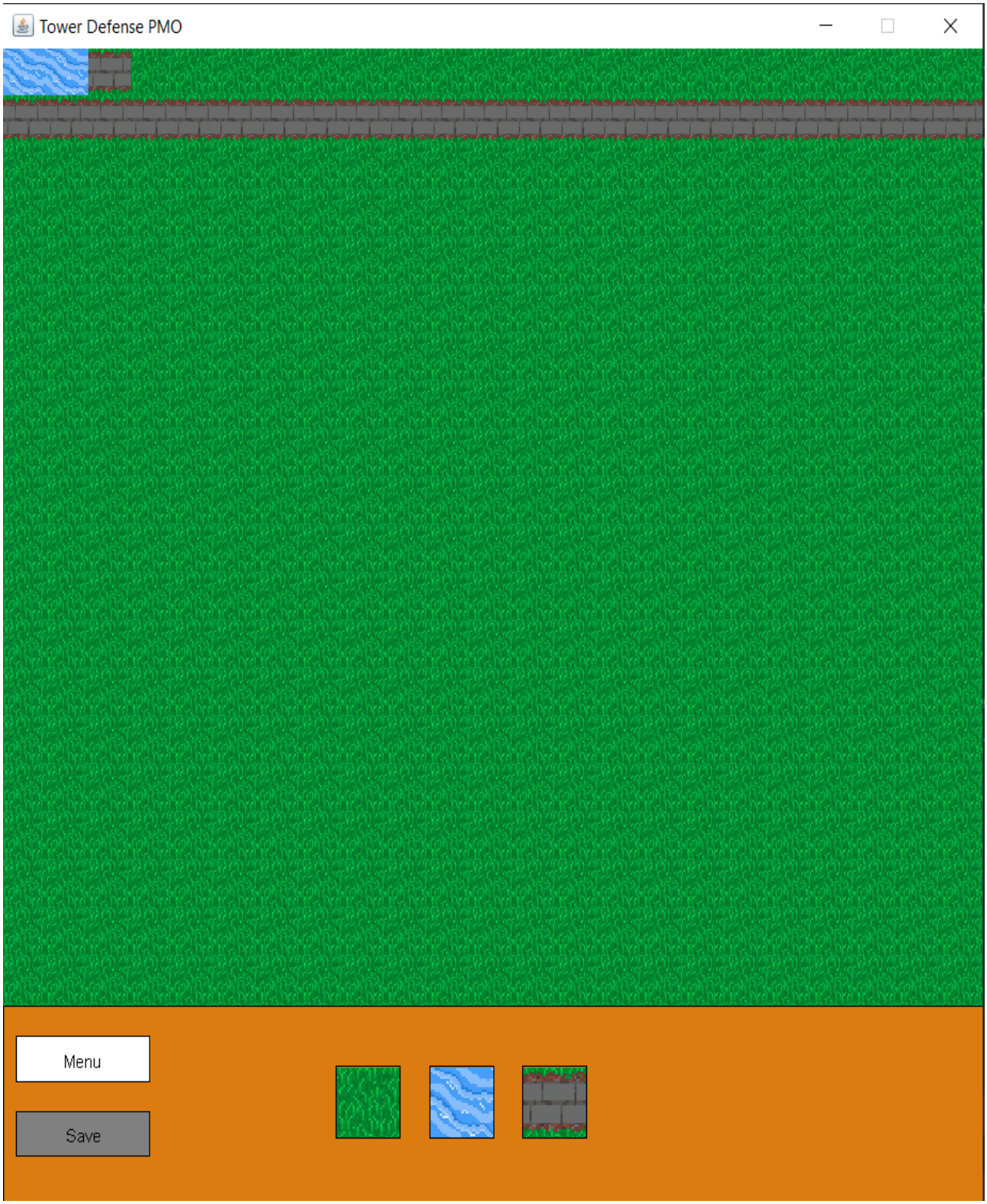
(Scelta della modalità **EditMap**)



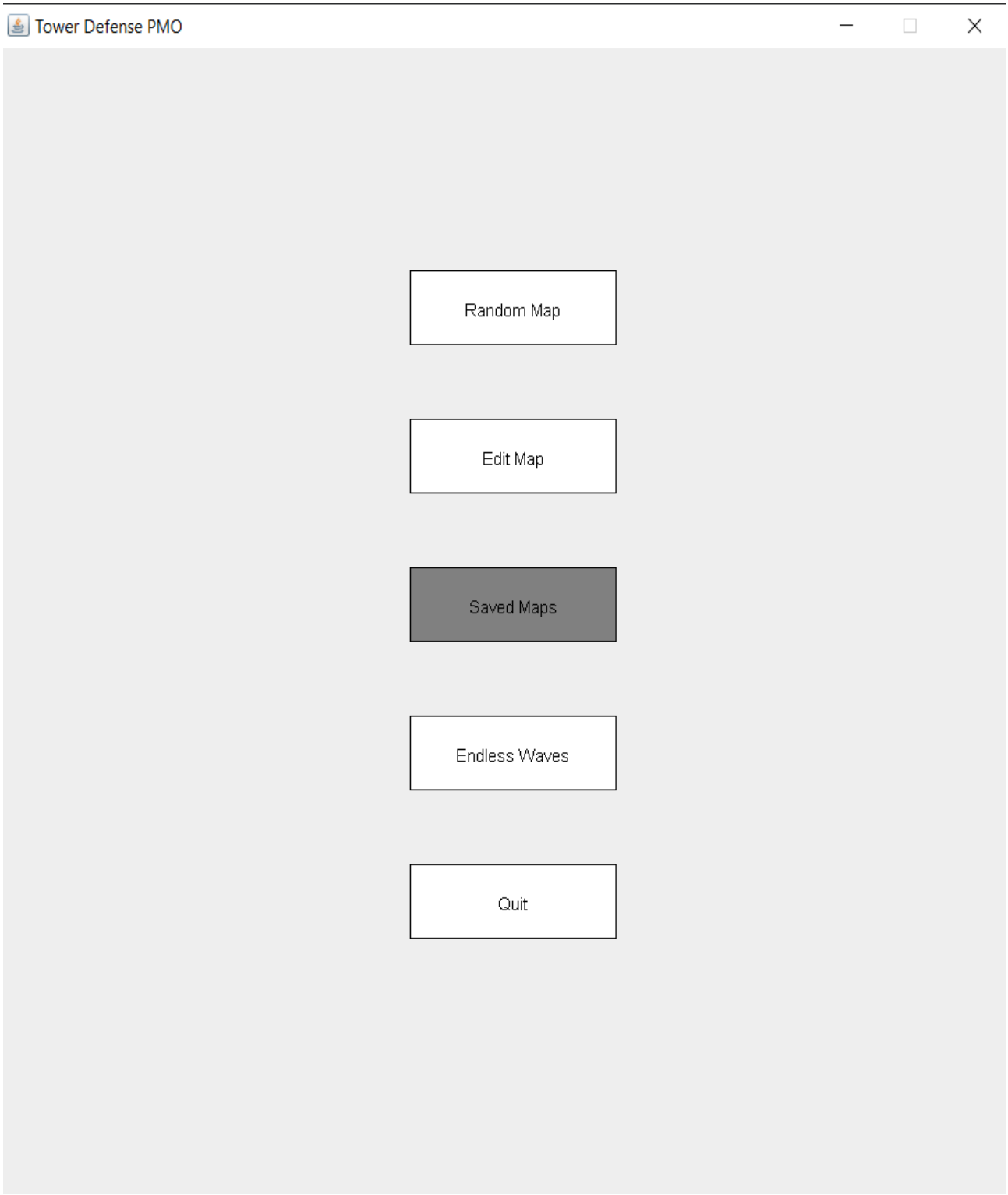
(Mappa di default per l’edit)



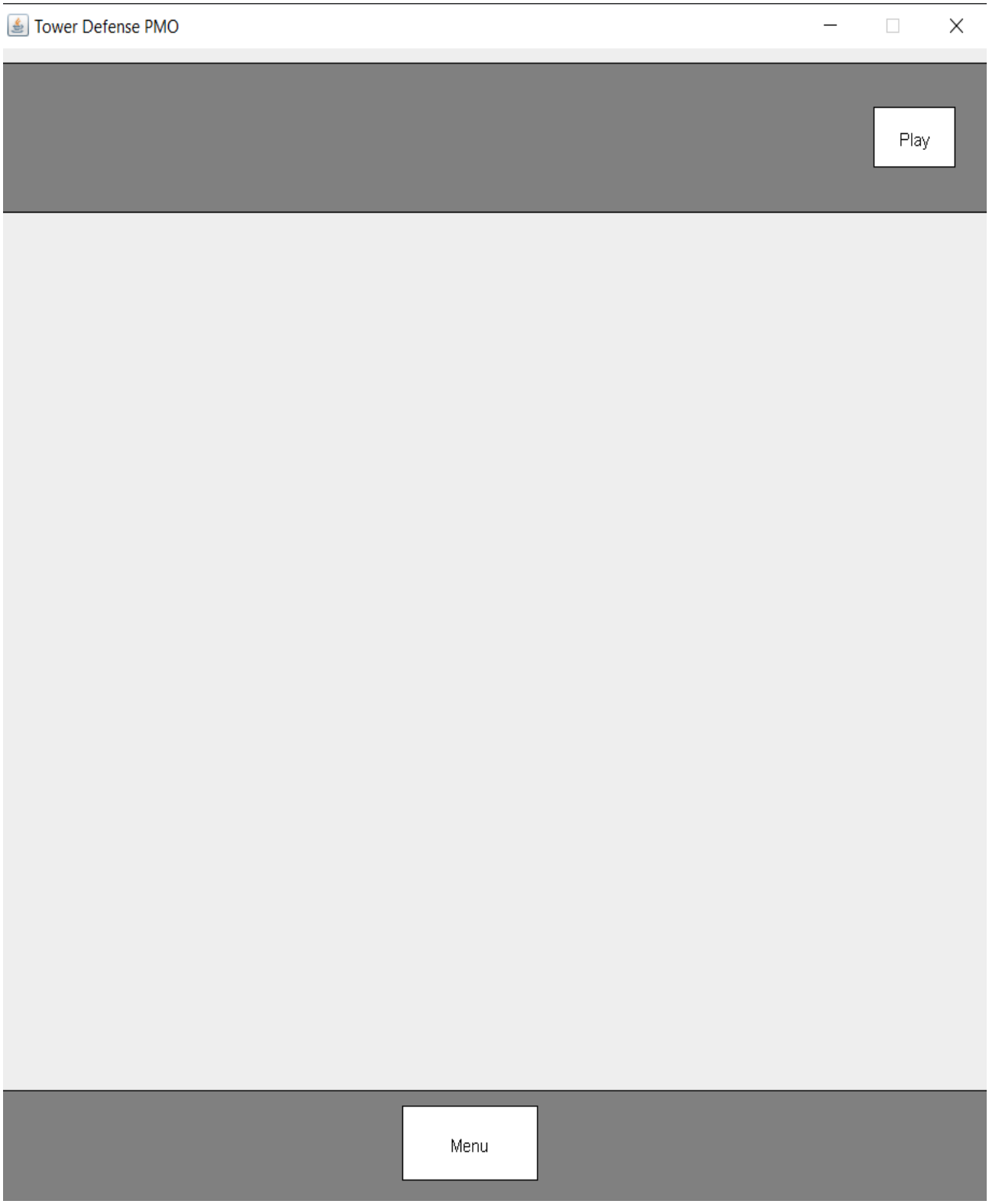
(Salvataggio mappa editata)



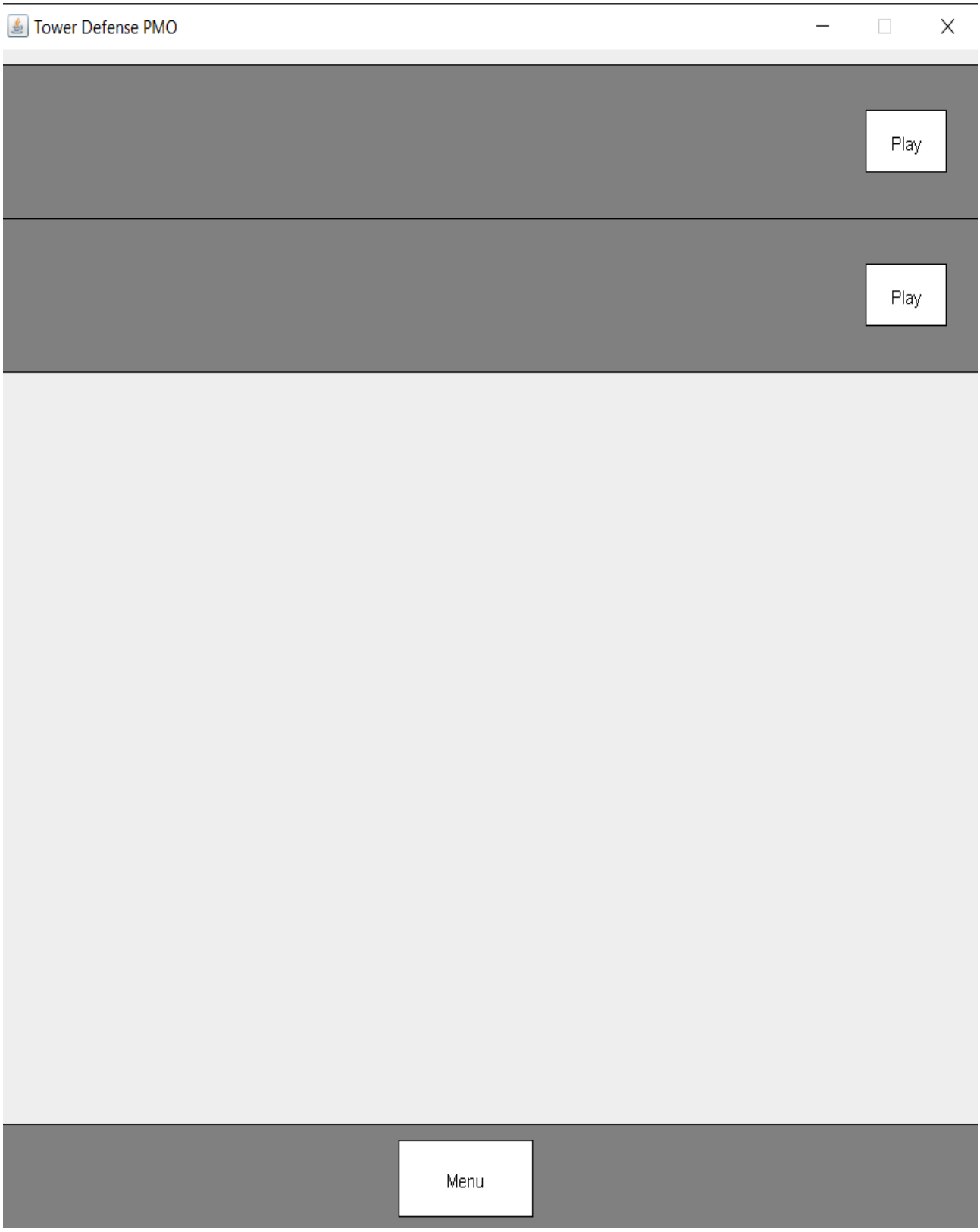
(Scelta modalità per la selezione delle mappe custom)



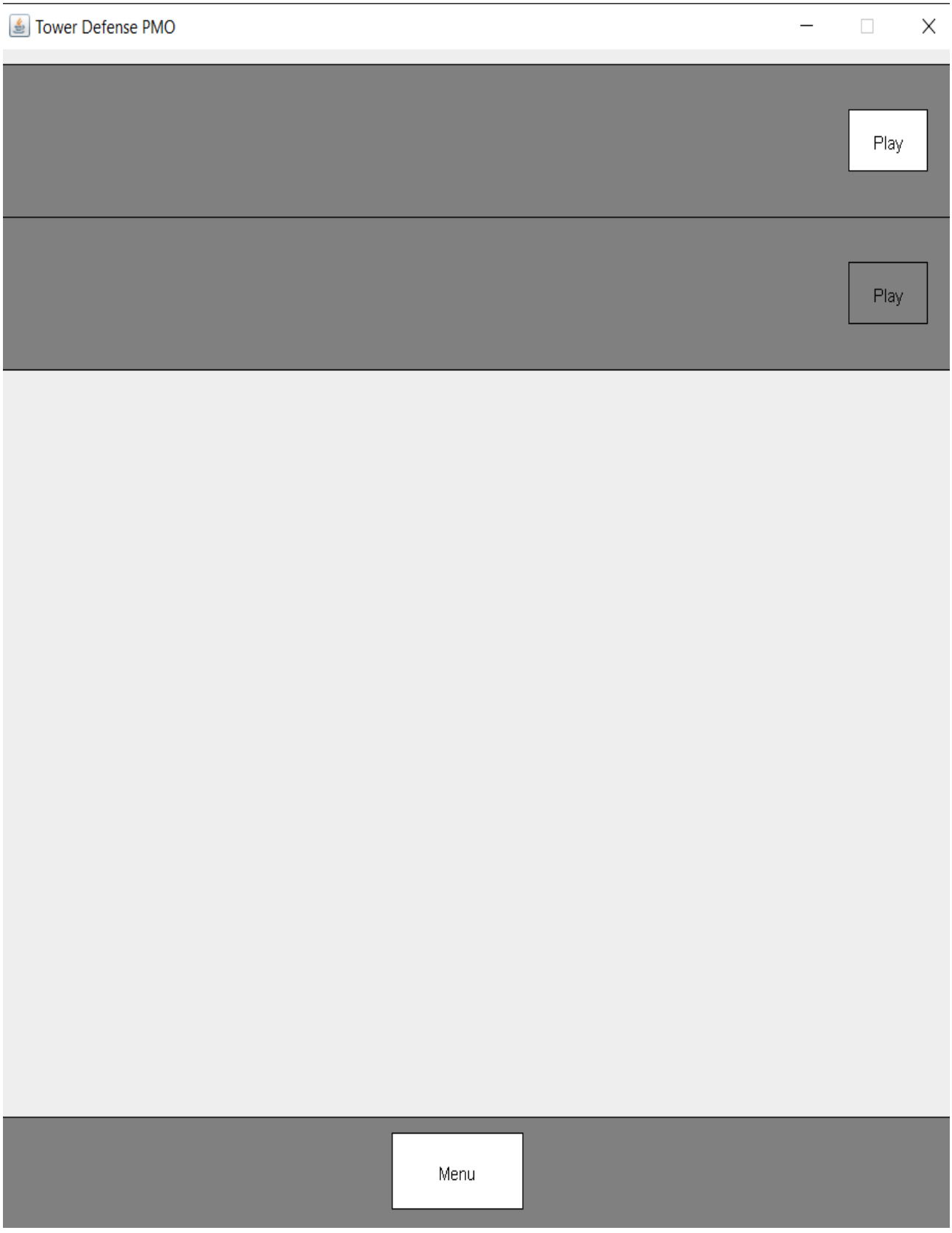
(Prima del riavvio dell’applicazione (necessario per aggiornare la lista di mappe disponibili))



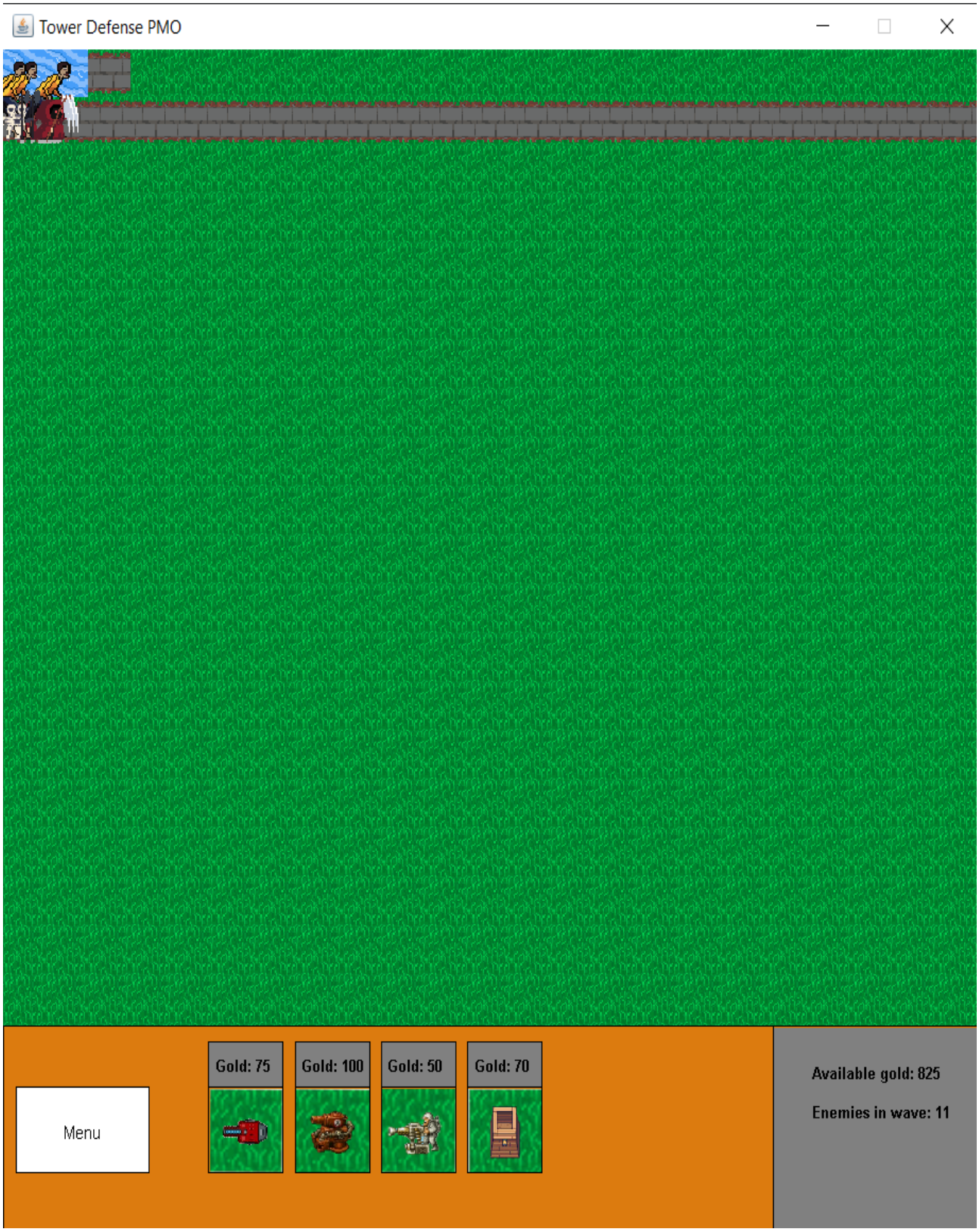
(Dopo aver riavviato)



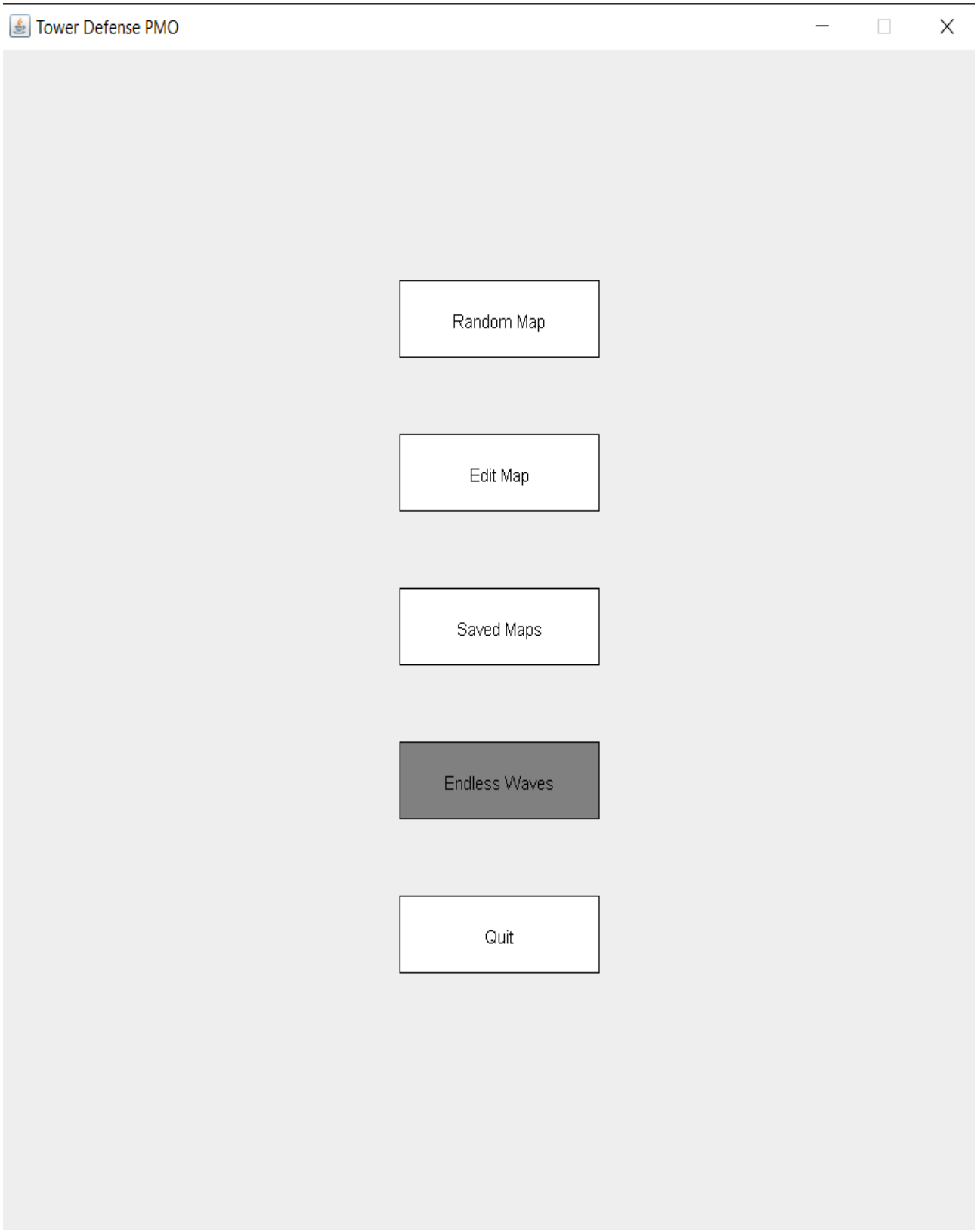
(Selezione livello appena creato e salvato)



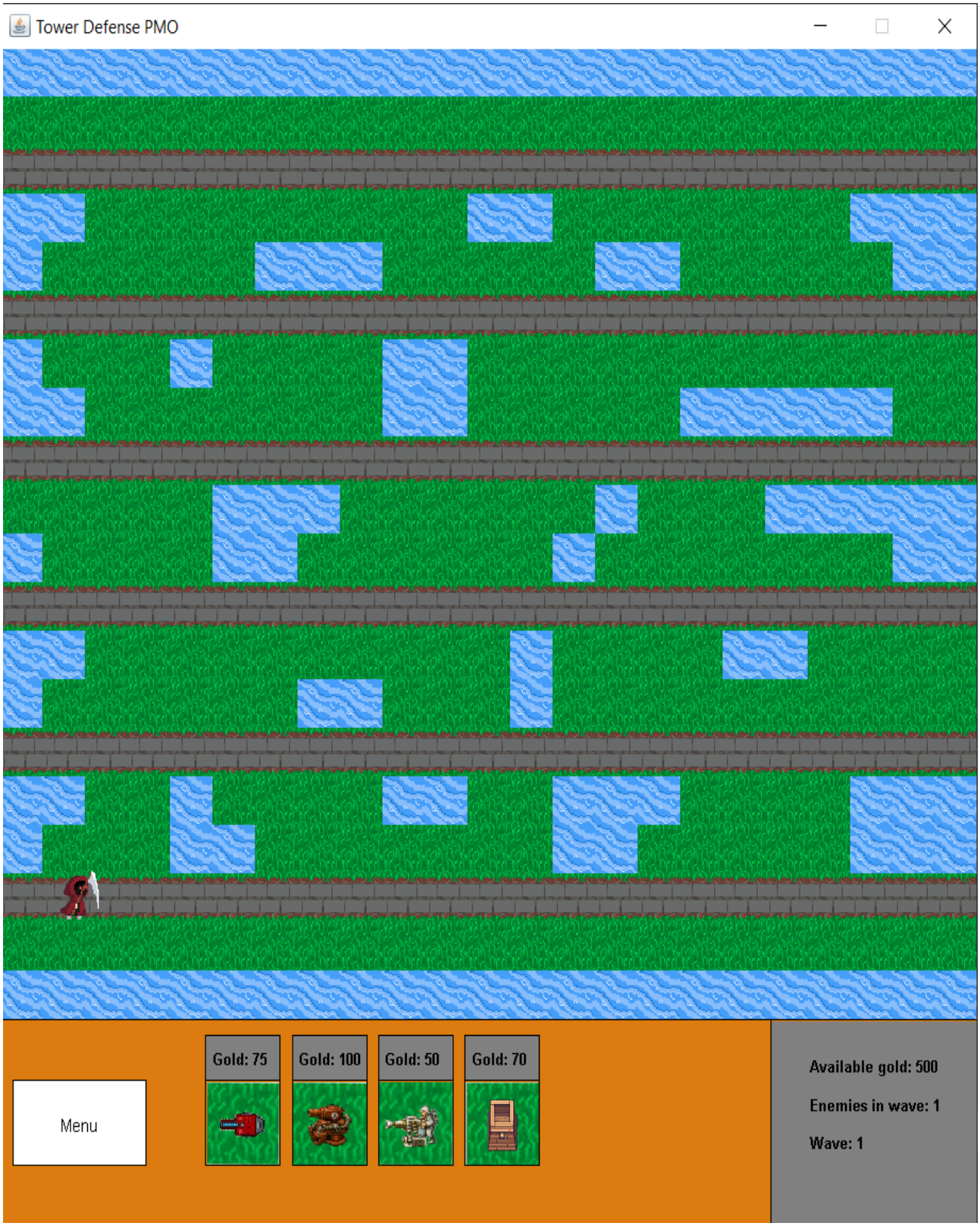
(Mappa avviata normalmente)



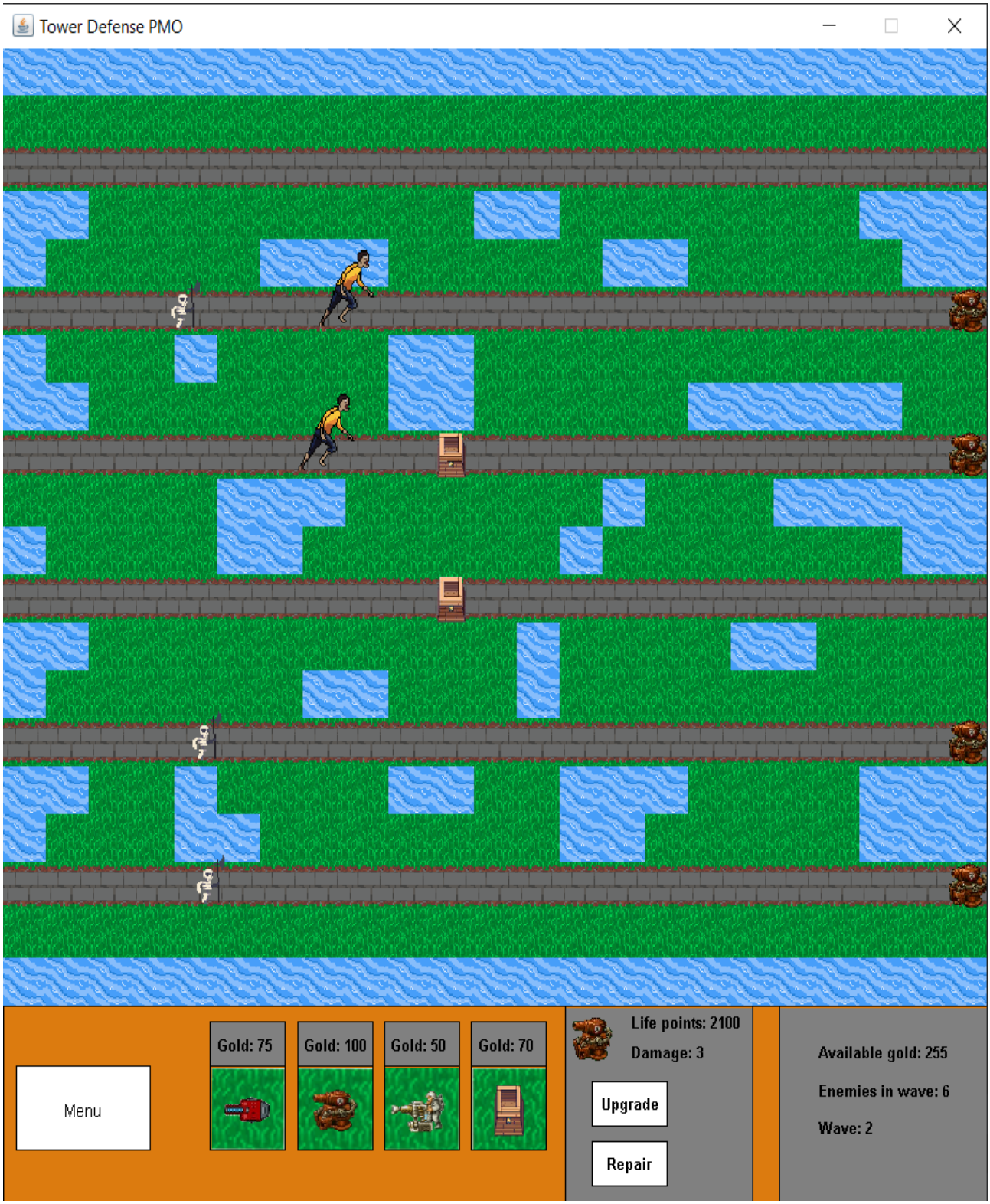
(Selezione modalità ondate ad oltranza)



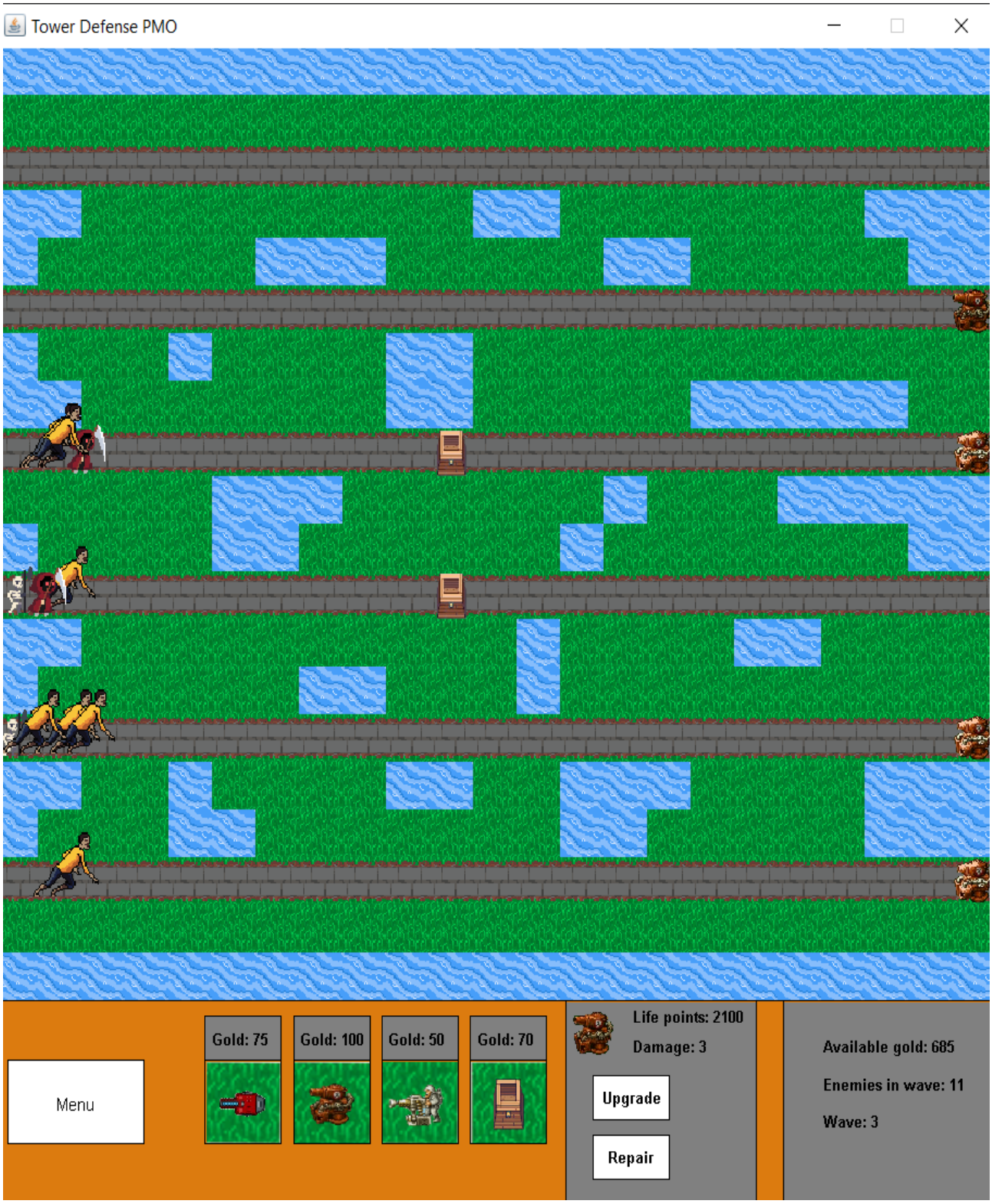
(Prima ondata)



(Seconda ondata)



(Terza ondata)



4.2 Possibili implementazioni future

Il progetto lascia aperte diverse possibilità future, in caso di aggiunta di una o piu' nuove modalità di gioco basterà estendere la classe base di gioco GameSceneBase ed overridare alcuni metodi per fare in modo che essi implementino i nuovi funzionamenti e le nuove logiche.

Per aggiungere nuove unità basterà estendere le classi nemici o torri a seconda della necessità.

Il progetto è quindi piuttosto flessibile a possibili cambiamenti ed implementazioni future.

4.3 Crediti

Per le sorgenti dei frames di animazione si ringraziano craftpix.net, itch.io, sprite-resource.com, kaaringaming.com, tutti i diritti ai rispettivi proprietari, l'utilizzo degli sprite e delle immagini sono solo a scopo di studio e non per scopi commerciali.