1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

CORSO DI LAUREA IN
INFORMATICA APPLICATA
SCUOLA DI
SCIENZE TECNOLOGIE E FILOSOFIA DELL'INFORMAZIONE

Progetto Ingegneria del Software

Anno Accademico 2019-2020

Sessione Invernale

Tetris

Quattrone Cosimo Luca N° 286662

Agostini Riccardo N° 284966

INDICE

1. SPECIFICA DEL PROBLEMA

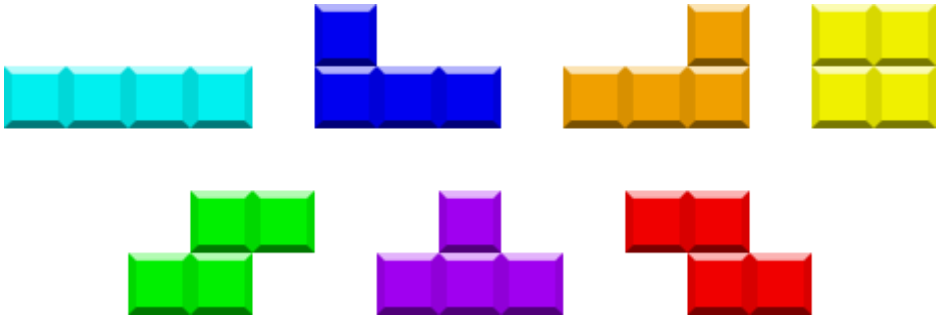
Si vuole realizzare un'applicazione desktop per il gioco Tetris.

L'applicazione dovrà avere un menu principale, e una schermata di gioco. Vengono assunte le regole del Tetris classico, per cui saranno disponibili 7 tetramini (o Figure):

- Tetramino I
- Tetramino L
- Tetramino J
- Tetramino T
- Tetramino O
- Tetramino S
- Tetramino Z

Di seguito, i tetramini raffigurati graficamente in ordine dall'altro verso il basso:

I – J – L – O – S – T – Z



All'inizio della partita comparirà un tetramino casuale nella parte alta dello schermo di gioco, questo col tempo scenderà fino ad arrivare al punto di non potersi più muovere, fatto ciò il tetramino si blocca ed in seguito ne viene generato uno nuovo che avrà lo stesso comportamento del precedente. Mentre il tetramino scende il giocatore potrà effettuare 4 azioni modificando la discesa della figura:

- Muovere la figura verso il basso in maniera più veloce
- Muovere la figura verso sinistra
- Muovere la figura verso destra
- Ruotare la figura di 90°

Una volta completata un'intera riga di tetramini essa verrà eliminata e conferirà un maggiore punteggio al giocatore.

La difficoltà del gioco sta nella velocità di discesa del tetramino, questa velocità viene gestita tramite livelli, più aumenta, più il tetramino scenderà velocemente lasciando meno spazio al giocatore per pensare e agire.

Oltre ai livelli il giocatore avrà un punteggio personale, questo verrà influenzato in due diversi modi:

- Aumenterà quando una figura viene bloccata
- Aumenterà quando viene completata una o più linee

Nel secondo caso l'aumento del punteggio è determinato dal numero di linee completate contemporaneamente, ad esempio se completiamo 4 linee contemporaneamente (In gergo questa azione è chiamata 'Tetris', ed è il massimo di linee che è possibile completare contemporaneamente) otterremo il massimo dei punti.

L'obbiettivo del gioco è quello di incastrare i pezzi nel miglior modo possibile, al fine di completare linee e ottenere il maggior punteggio.

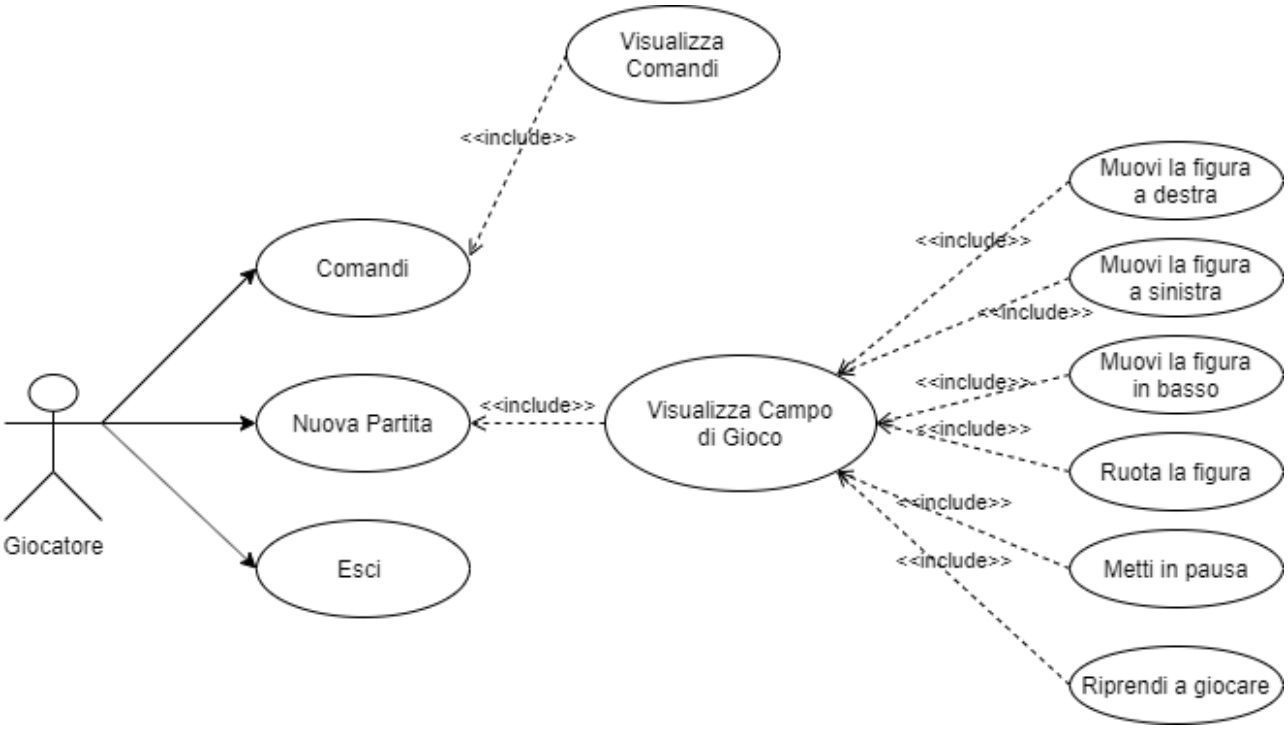
Il gioco termina se un tetramino appena generato è impossibilitato nel muoversi o ruotare.

2. SPECIFICA DEI REQUISITI

Di seguito vengono riportati i casi d'usi dell'applicativo che ci aiuteranno a comprendere al meglio le funzionalità del gioco.

Il diagramma dei casi d'uso evidenzia la partecipazione di un unico attore: il giocatore, il quale potrà scegliere inizialmente se uscire, visualizzare i comandi o iniziare una nuova partita. Una volta che il giocatore avrà selezionato 'nuova partita', il sistema mostrerà la griglia di gioco.

2.1 Diagramma dei casi d'uso



2.2 Relazioni e descrizione dei casi d'uso

Come prima cosa il giocatore avvierà l'applicazione.

- Apertura Applicazione

Si troverà davanti alla prima scelta, se cliccare su comandi, uscire o iniziare una nuova partita.

Caso d'uso	Cliccato sul bottone Esci
ID	#1
Attore	Giocatore
Precondizioni	Il giocatore lancia l'applicazione che mostra un panel contenente 3 bottoni.
Corso d'azione base	1) Il giocatore clicca sul bottone esci 2) L'applicazione si arresta
Post Condizioni	L'applicazione si arresta.
Percorso alternativo	/

Caso d'uso	Cliccato sul bottone Comandi
ID	#2
Attore	Giocatore
Precondizioni	Il giocatore lancia l'applicazione che mostra un panel contenente 3 bottoni.
Corso d'azione base	1) Il giocatore clicca sul bottone Comandi 2) L'applicazione mostra i comandi
Post Condizioni	Il giocatore può visualizzare un MessageBox contenente i comandi.
Percorso alternativo	/

Caso d'uso	Cliccato sul bottone Nuova Partita
ID	#3
Attore	Giocatore
Precondizioni	Il giocatore lancia l'applicazione che mostra un panel contenente 3 bottoni.
Corso d'azione base	1) Il giocatore clicca sul bottone Nuova Partita 2) L'applicazione mostra il campo di gioco
Post Condizioni	Il giocatore può iniziare a giocare.
Percorso alternativo	/

- Visualizza comandi

Se il giocatore sceglie di cliccare sul bottone comandi visualizzerà una MessageBox contenente i vari comandi e un bottone ‘OK’ che lo farà tornare al menu principale

Caso d’uso	Visualizza i Comandi
ID	#4
Attore	Giocatore
Precondizioni	#2
Corso d’azione base	1) Il giocatore visualizza i comandi 2) Il giocatore clicca il pulsante OK 3) Il giocatore visualizza di nuovo il menu principale
Post Condizioni	Il giocatore tornerà al menu principale.
Percorso alternativo	Il giocatore non clicca il pulsante OK, ma chiude il MessageBox.

- Visualizza Campo di Gioco

Se il giocatore sceglie di cliccare sul bottone Nuova partita verrà visualizzato il campo da gioco consentendo all’utente di interagire con l’interfaccia tramite le seguenti operazioni:

- Muovi la figura a destra
- Muovi la figura a sinistra
- Muovi la figura in basso
- Ruota la figura
- Metti in pausa
- Riprendi a giocare

Caso d’uso	Muovi la figura a destra
ID	#5
Attore	Giocatore
Precondizioni	#3
Corso d’azione base	1) Il giocatore clicca sulla freccia destra della tastiera 2) La figura si muove a destra
Post Condizioni	La figura si muove a destra di una unità.
Percorso alternativo	/

Caso d’uso	Muovi la figura a sinistra
ID	#6
Attore	Giocatore
Precondizioni	#3
Corso d’azione base	1) Il giocatore clicca sulla freccia sinistra della tastiera 2) La figura si muove a sinistra
Post Condizioni	La figura si muove a sinistra di una unità.
Percorso alternativo	/

Caso d’uso	Muovi la figura in basso
ID	#7
Attore	Giocatore
Precondizioni	#3
Corso d’azione base	1) Il giocatore clicca sulla freccia giù della tastiera 2) La figura si muove in basso più velocemente
Post Condizioni	La figura si muove a in basso di una unità. (Dipendente dal tick del timer)
Percorso alternativo	Aspettare il tick del timer e la figura effettuerà uno spostamento automatico verso il basso

Caso d’uso	Ruota la figura
ID	#8
Attore	Giocatore
Precondizioni	#3
Corso d’azione base	1) Il giocatore clicca sulla freccia su della tastiera 2) La figura ruota
Post Condizioni	La figura ruota di 90°.
Percorso alternativo	/

Caso d’uso	Metti in pausa
ID	#9
Attore	Giocatore
Precondizioni	#3 #10
Corso d’azione base	1) Il giocatore clicca sul pulsante pausa 2) Il gioco si ferma
Post Condizioni	Il gioco si ferma e compare il pulsante riprendi.
Percorso alternativo	/

Caso d’uso	Riprendi a giocare
ID	#10
Attore	Giocatore
Precondizioni	#3 #9
Corso d’azione base	1) Il giocatore clicca sul bottone Riprendi 2) Il gioco riprende
Post Condizioni	Il gioco riprende e compare il pulsante pausa.
Percorso alternativo	/

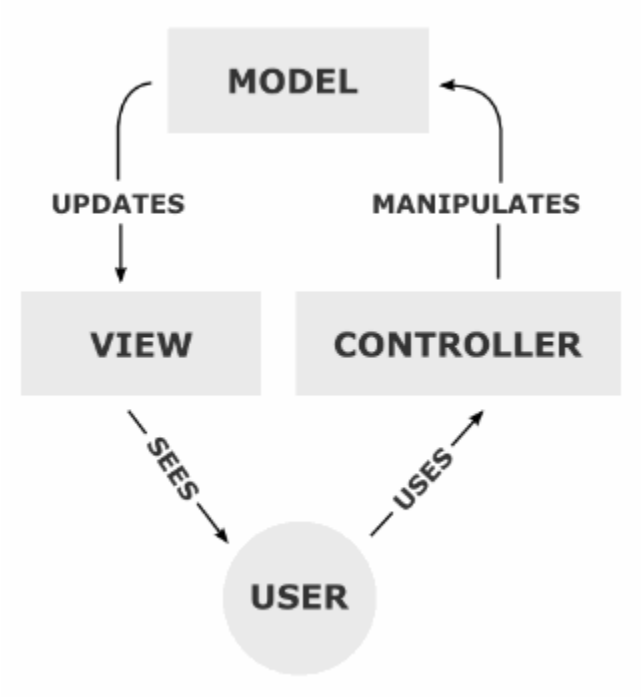
3. Analisi e progettazione

In questa sezione descriveremo l'analisi, le principali scelte di progettazione e di implementazione fatte per realizzare il gioco.

3.1 Scelte di progetto

3.1.1 Pattern Design

Dato che il problema posto presenta la necessità di un'interfaccia grafica (GUI) si è deciso di utilizzare il pattern MVC (Model – View – Controller).



Il pattern MVC è in grado di separare la logica di gestione dei dati dalla vista offerta all'utente.

Un pattern MVC efficace ha sempre 3 moduli fondamentali:

- **MODEL:** fornisce i metodi per accedere ai dati utili dell'applicazione.
- **VIEW:** visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti.
- **CONTROLLER:** riceve i comandi dell'utente e li attua modificando lo stato degli altri due componenti.

3.1.2 Gestione GUI

L'applicazione è incentrata in un'unica form, per lo switch tra Menu principale e griglia di gioco, vengono utilizzati i *panel*. Il menu principale è essenzialmente un *TableLayoutPanel*, mentre la griglia di gioco è appoggiata direttamente sulla form.

Per lo sviluppo del menu principale si è usato una *picture box* per introdurre il logo del Tetris, ed in seguito i 3 *button*: 'nuova partita', 'comandi', 'esci'.

Per ottenere l'impaginazione ottimale si è sfruttata la funzione del *TableLayoutPanel* che permette di modificare a piacimento numero e dimensione di righe e colonne.

(Con la percentuale si indica la dimensione del componente relativamente alle dimensioni attuali della form)

3 colonne distribuite in questo modo:

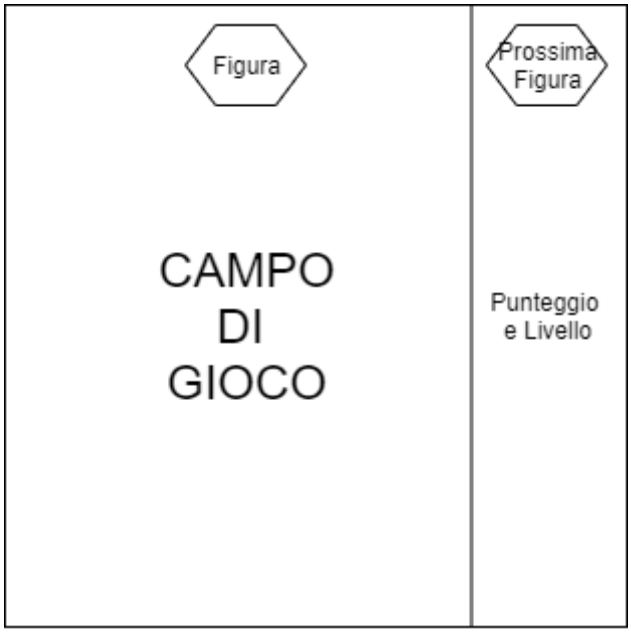
- colonna 1) 17.50%
- colonna 2) 65%
- colonna 3) 17.50%

5 righe distribuite in questo modo:

- riga 1) 60%
- riga 2) 10%
- riga 3) 10%
- riga 4) 10%
- riga 5) 10%

In base al bottone premuto nel menu verranno eseguite le seguenti azioni:

- Al click di ‘nuova partita’ verrà nascosto il *panel* e comincerà il gioco.
- Al click di ‘comandi’ si aprirà una *MessageBox* con i comandi del gioco.
- Al click di ‘esci’ la form si chiuderà.



Questa è l’idea concettuale iniziale dello sviluppo della GUI dell’applicazione.

Sulla parte destra vi è la griglia (o campo) di gioco, si è scelto di visualizzarlo sulla destra per facilitare le successive operazioni con le coordinate X Y.

Sul lato sinistro ci sarà una *Sidebar* che mostrerà la prossima figura, il livello attuale, il punteggio attuale e la *label* di pausa (e eventualmente la label riprendi).

Per le operazioni di pausa e riprendi, si è deciso di optare per due *label* (Modificandone l’aspetto grafico, facendoli assomigliare a dei bottoni).

Questo perché ogni volta che si esegue un click su un bottone esso ottiene il focus, per cui il *KeyEventHandler* smetterà di elaborare i tasti per muovere la figura, ma muoverà il focus fra i due bottoni.

La figura attuale è un’immagine ridisegnata in ogni istante nelle attuali coordinate, mentre una volta bloccata, non sarà più un’immagine unica, ma ogni suo blocco sarà un’immagine a sé. In seguito a una riga completata, i blocchi comporranno la griglia in maniera corretta.

Le immagini dei tetramini sono state scaricate da https://en.m.wikipedia.org/wiki/Main_Page

3.1.3 Gestione audio

L'applicazione dispone anche di una parte uditiva, sono infatti presenti 3 suoni che verranno riprodotti durante i 3 momenti ritenuti chiave del gioco:

- Quando una figura viene fissata
- Quando viene completata una linea
- Quando si perde (Game Over)

I suoni verranno memorizzati dalla classe *SoundPlayer* e riprodotti tramite il metodo *Play()*

Le risorse audio sono state scaricate da https://freesound.org/

3.1.4 Assunzioni e gestione progetto

Nella realizzazione del progetto, l'incapsulamento dei dati è stato tenuto in conto in maniera considerevole, praticamente quasi tutti i dati sensibili dell'applicazioni non risultano accessibili in maniera diretta, ma solo tramite metodi.

Per la gestione delle regole effettive del gioco si è optato per la creazione di una classe primaria astratta chiamata 'Figura' da cui ereditano le altre 7 effettive figure.

Ogni forma delle 7 figure è rappresentata da un array bidimensionale.

L'array identifica le coordinate di ogni figura, inoltre ogni classe avrà N forme dovute al numero di rotazioni possibili, questo perché la Figura_O avrà 0 possibili rotazioni, quindi una sola forma, mentre ad esempio la J avrà 4 forme disponibili.

Questo è un esempio di una forma della figura:

```
private int [,] _forma0 = new int [3, 2]
{
    {0, 7},
    {0, 7},
    {7, 7}
};
```

Come si può notare ogni figura avrà assegnato un numero per cui essere riconosciuta dalla griglia, che sarà sempre un array bidimensionale, ma con tutti i valori settati a 0. Numerando le varie figure è quindi possibile eseguire le future operazioni di *FissaFigura()* e *LineaCompletata()*.

In seguito si è deciso di creare una classe propria *ResourceNotFoundException,* che eredita appunto da *Exception,* per sopperire l'unico errore non gestibile dell'applicazione, il caricamento errato delle risorse.

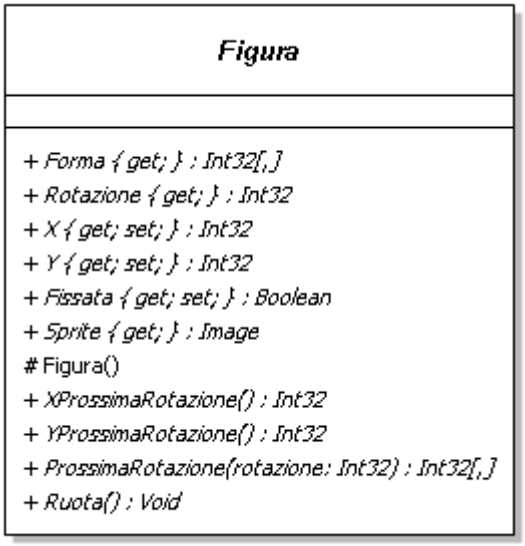
Alla fine della analisi della richiesta si è ottenuta la seguente tabella in cui suddividere le varie classi nel pattern MVC:

Model	Figura FiguraI FiguraJ FiguraL FiguraO FiguraS FiguraT FiguraZ EstrattoreFigura Terreno Gioco
View	TetrisForm
Controller	TetrisController

3.2 Classi

In seguito verranno descritte le classi utilizzate per lo sviluppo del progetto.

3.2.1 Figura



La classe *astratta* Figura contestualizza le basi di una figura.

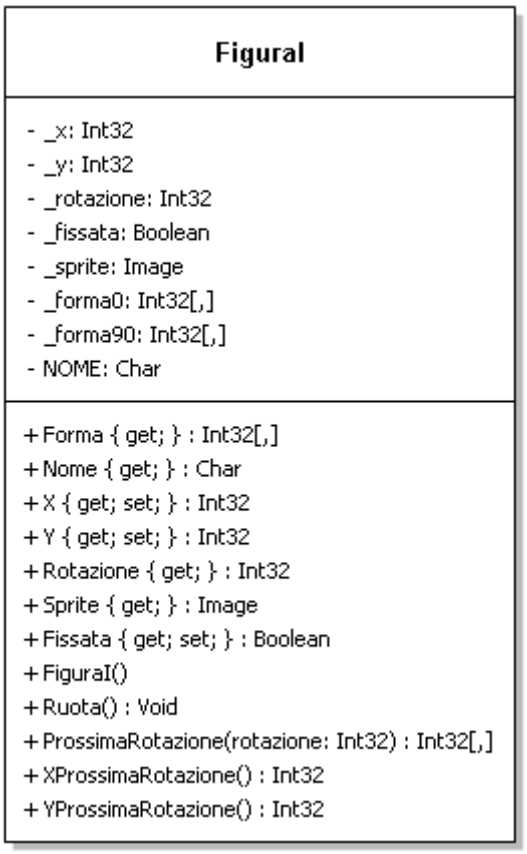
Ogni figura ereditaria dovrà avere la propria **forma**, il numero di **rotazione** attuale, questo numero può andare da 0, ovvero la forma iniziale, a 3(l'eventuale rotazione di 270°).

In seguito la propria coordinata **X**, la propria coordinata **Y**, il campo booleano **Fissata** e **l'immagine** assegnata.

Inoltre dovranno essere presenti i seguenti metodi:

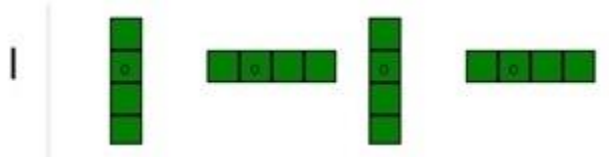
- **XProssimaRotazione**(): Restituisce la X della figura una volta eseguita la rotazione.
- **YProssimaRotazione**(): Restituisce la Y della figura una volta eseguita la rotazione.
- **ProssimaRotazione**(): Restituisce la forma della figura una volta effettuata la rotazione richiesta tramite parametro.
- **Ruota**(): Metodo che fa eseguire la rotazione alla figura.

3.2.2 FiguraI



La classe FiguraI eredita dalla classe **Figura**.

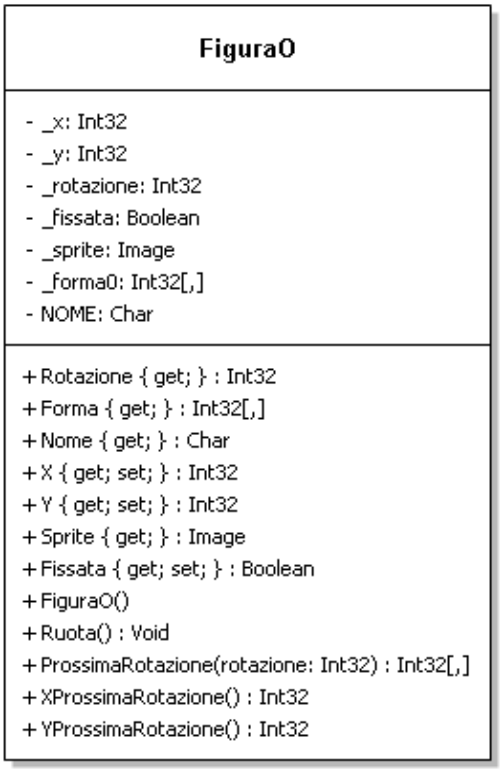
Avrà 2 **forme** totali: La prima in verticale e la seconda in orizzontale.



Inoltre la classe dispone di una costante **NOME**.

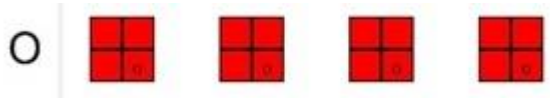
I metodi rimangono gli stessi della classe Figura, vengono solamente implementati i *get*/*set* degli attributi.

3.2.3 FiguraO



La classe FiguraO eredita dalla classe **Figura**.

Avrà una **forma** totale: essendo un quadrato non esegue delle rotazioni sensibili al fine del gioco.



Inoltre la classe dispone di una costante **NOME**.

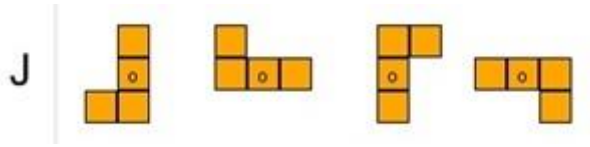
I metodi rimangono gli stessi della classe Figura, vengono solamente implementati i *get*/*set* degli attributi.

3.2.4 FiguraJ

FiguraJ
- _x: Int32 - _y: Int32 - _rotazione: Int32 - _fissata: Boolean - _sprite: Image - _forma0: Int32[,] - _forma90: Int32[,] - _forma180: Int32[,] - _forma270: Int32[,] - NOME: Char
+ Forma { get; } : Int32[,] + Sprite { get; } : Image + Nome { get; } : Char + X { get; set; } : Int32 + Y { get; set; } : Int32 + Rotazione { get; } : Int32 + Fissata { get; set; } : Boolean + FiguraJ() + Ruota() : Void + ProssimaRotazione(rotazione: Int32) : Int32[,] + XProssimaRotazione() : Int32 + YProssimaRotazione() : Int32

La classe FiguraJ eredita dalla classe **Figura**.

Avrà 4 **forme** totali: 0°, 90°, 180° e 270°.



Inoltre la classe dispone di una costante **NOME**.

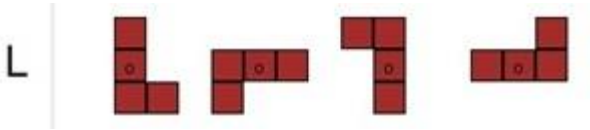
I metodi rimangono gli stessi della classe Figura, vengono solamente implementati i *get*/*set* degli attributi.

3.2.5 FiguraL

FiguraL
- _x: Int32 - _y: Int32 - _rotazione: Int32 - _fissata: Boolean - _sprite: Image - _forma0: Int32[,] - _forma90: Int32[,] - _forma180: Int32[,] - _forma270: Int32[,] - NOME: Char
+ Sprite { get; } : Image + Nome { get; } : Char + X { get; set; } : Int32 + Y { get; set; } : Int32 + Rotazione { get; } : Int32 + Forma { get; } : Int32[,] + Fissata { get; set; } : Boolean + FiguraL() + Ruota() : Void + ProssimaRotazione(rotazione: Int32) : Int32[,] + XProssimaRotazione() : Int32 + YProssimaRotazione() : Int32

La classe FiguraL eredita dalla classe **Figura**.

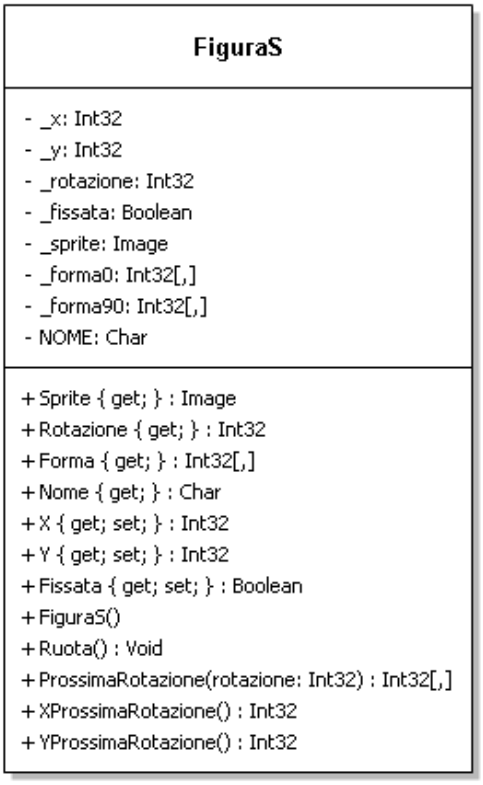
Avrà 4 **forme** totali: 0°, 90°, 180° e 270°.



Inoltre la classe dispone di una costante **NOME**.

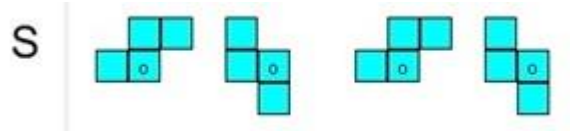
I metodi rimangono gli stessi della classe Figura, vengono solamente implementati i *get*/*set* degli attributi.

3.2.6 FiguraS



La classe FiguraS eredita dalla classe **Figura**.

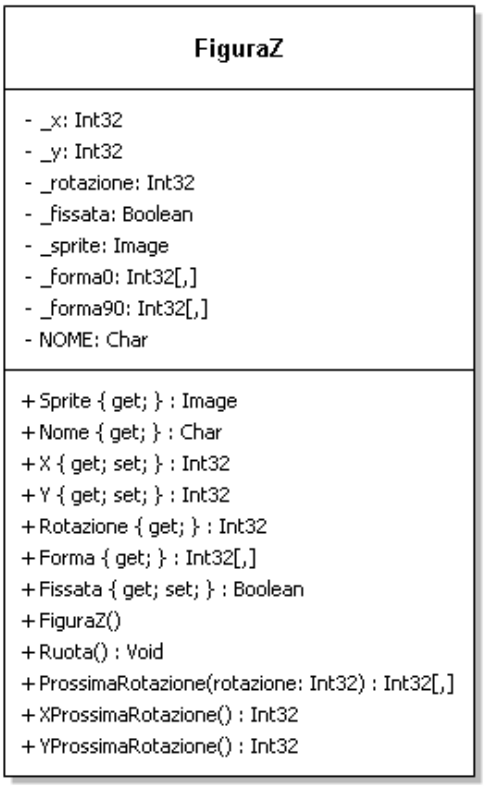
Avrà 2 **forme** totali: La prima in verticale e la seconda in orizzontale.



Inoltre la classe dispone di una costante **NOME**.

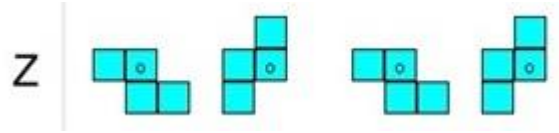
I metodi rimangono gli stessi della classe Figura, vengono solamente implementati i *get*/*set* degli attributi.

3.2.7 FiguraZ



La classe FiguraZ eredita dalla classe **Figura**.

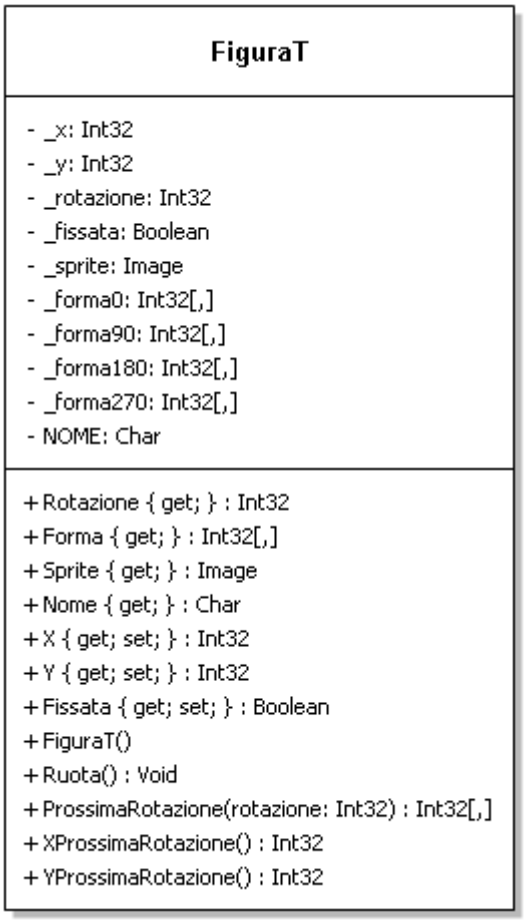
Avrà 2 **forme** totali: La prima in verticale e la seconda in orizzontale.



Inoltre la classe dispone di una costante **NOME**.

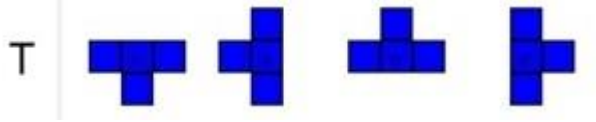
I metodi rimangono gli stessi della classe Figura, vengono solamente implementati i *get*/*set* degli attributi.

3.2.8 FiguraT



La classe FiguraT eredita dalla classe **Figura**.

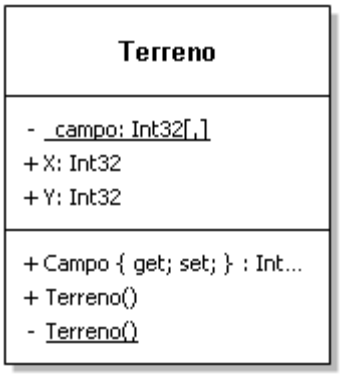
Avrà 4 **forme** totali: 0°, 90°, 180° e 270°.



Inoltre la classe dispone di una costante **NOME**.

I metodi rimangono gli stessi della classe Figura, vengono solamente implementati i *get*/*set* degli attributi.

3.2.9 Terreno

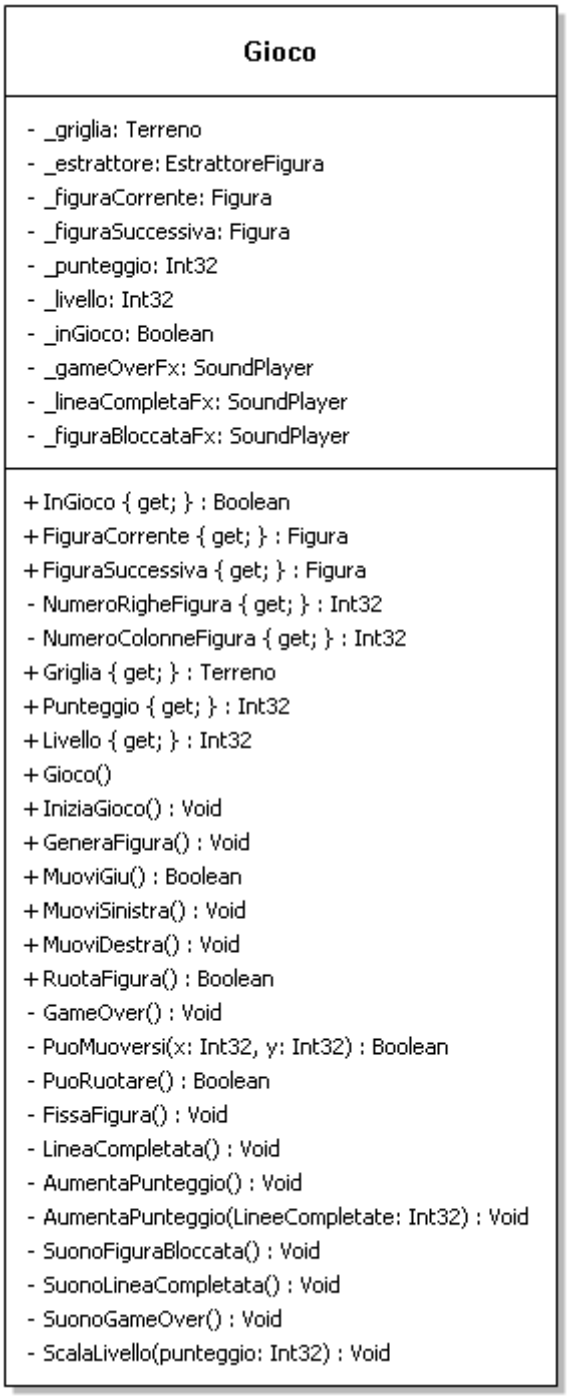


La classe terreno identifica la **griglia** di gioco.

L'attributo **campo** è un array bidimensionale statico con ogni valore settato a 0.

X e **Y** sono due costanti che identificano le dimensioni del campo di gioco.

3.2.10 Gioco



La classe gioco è la classe principale della parte Model.

In esse vengono svolte tutte le operazioni volte allo svolgimento della partita.

La classe gioco avrà una **griglia** di classe Terreno, un **estrattore** di classe EstrattoreFigura e la **figura corrente e successiva** rispettivamente di classe Figura.

Disporrà inoltre di un **punteggio**, di un **livello** e di un attributo booleano '**InGioco**' per verificare lo stato della partita.

L'ultima parte degli attributi della classe Gioco riguarda i suoni, è presente appunto il suono per la fine del gioco, per ogni volta che viene bloccata una figura e per ogni linea completata.

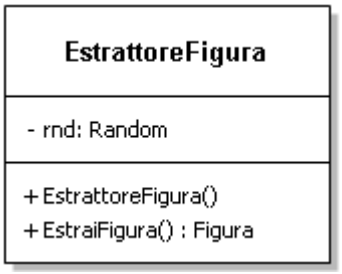
I metodi, oltre ai *get/set* dei rispettivi attributi sono i seguenti:

- **Gioco**() è il costruttore della classe e contiene al suo interno il metodo **IniziaGioco**() che inizializza la partita.
- **GeneraFigura**() è il metodo che assegna a figura corrente e figura successiva una oggetto di classe Figura generato casualmente.

- **Muovi[Giu, Sinistra, Destra]**(): Questi 3 metodi hanno la funzione di far muovere la figura nella relativa direzione.
- **RuotaFigura**(): Esegue la rotazione della figura.
- **GameOver**(): Gestisce la fine del gioco.
- **PuoMuoversi**(int x, int y): Controlla se la figura nella data posizione può effettuare un movimento (destra, sinistra, giu).
- **PuoRuotare**(): Controlla se la figura può eseguire una rotazione.

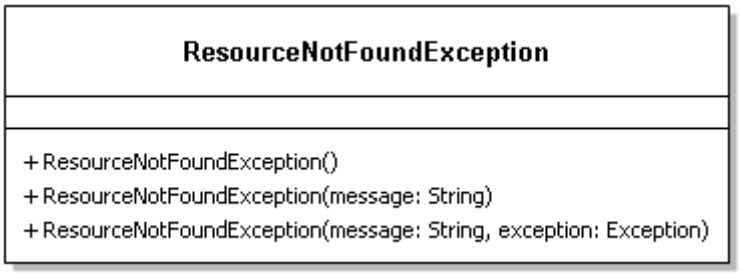
- **FissaFigura**(): Una volta impossibilita a ruotare o muoversi la figura verrà fissata alla griglia.
- **LineaCompletata**(): Controlla in tutta la griglia se sono state completate delle linee con le figure.
- **AumentaPunteggio**(): Aumenta il punteggio di 1 ogni volta che una figura viene fissata.
- **AumentaPunteggio**(int nLineeCompletate): Aumenta il punteggio in base al numero di linee completate, maggiore è il numero, maggiori punti si ricevono.
- **Suono** … (): Ha la singola funzione di riprodurre i 3 suoni disponibili.
- **ScalaLivello**(int punteggio): Controlla se si ha raggiunto il punteggio necessario per salire di livello.
- **NumeroRigheFigura**(): Ottiene il numero delle righe della figura attuale.
- **NumeroColonneFigura**(): Ottiene il numero delle colonne della figura attuale.

3.2.10 Estrattore



La classe estrattore ha come unica funzione l'estrazione casuale di una delle 7 figure presenti attraverso il metodo **EstraiFigura**().

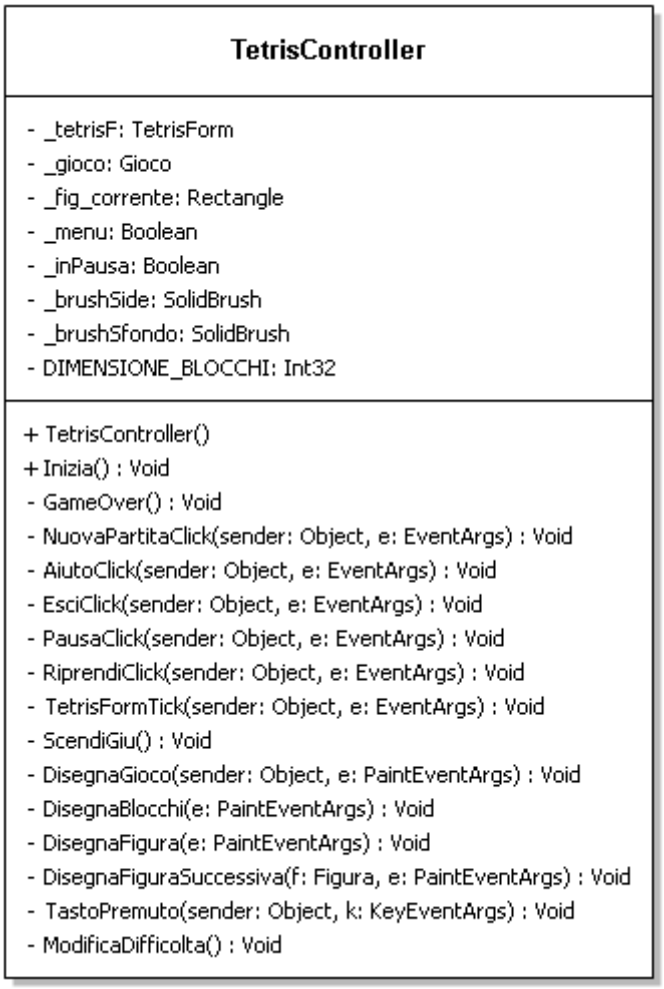
3.2.11 ResourceNotFoundException



Questa classe eredita da **Exception**.

È un'eccezione creata manualmente per sopperire l'unico errore non gestibile dell'applicazione, il caricamento errato delle risorse.

3.2.12 TetrisController



La classe TetrisController gestisce la parte grafica in riferimento allo svolgimento del gioco.

Ha come costante l'attributo **DIMENSIONE_BLOCCHI**, utilizzato nei metodi di disegno per stretchare l'immagine.

Conterrà due attributi appartenenti alle classi **TetrisForm** e **Gioco**, e un **Rectangle** che sarebbe effettivamente il contenitore grafico della figura attuale.

Infine abbiamo una variabile booleana per controllare che il gioco sia in pausa e due **brush** usati dai metodi di disegno.

I metodi oltre al costruttore e ai get/set sono i seguenti:

- **Inizia**(): Il metodo da invocare per iniziare l'applicazione.
- **GameOver**(): Gestione della fine della partita.
- **NuovaPartitaClick**(): Gestisce l'evento di click sul bottone nuova partita.
- **AiutoClick**(): Gestisce l'evento di click sul bottone Comandi.
- **EsciClick**(): Gestisce l'evento di click sul bottone Esci.
- **PausaClick**(): Gestisce l'evento di click sul bottone Pausa.
- **RiprendiClick**(): Gestisce l'evento di click sul bottone Riprendi.
- **TetrisFormTick**(): Gestisce l'evento del timer.
- **ScendiGiu**(): Gestisce la discesa della figura.
- **DisegnaGioco**(): Gestisce la parte grafica da disegnare.
- **DisegnaBlocchi**(): Disegna i blocchi delle varie figure fissate sullo schermo.
- **DisegnaFigura**(): Disegna la figura attuale.
- **DisegnaFiguraSuccessiva**(): Disegna la figura successiva nella Sidebar.

- **TastoPremuto**(): Gestisce la pressione dei tasti e la conseguente azione associata.
- **ModificaDifficolta**(): Aumenta la difficolta, diminuendo l'intervallo di tempo in cui il tick del timer si aggiorna.

(In tutti i metodi associati ai listener vengono omessi i parametri Object sender e Event… e, perché ritenuti superflui da motivare)

3.2.13 TetrisForm

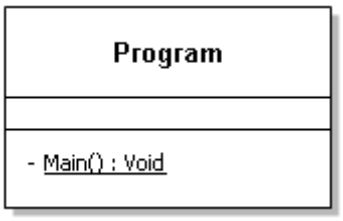
Questa è la classe che contiene le informazioni grafiche dell'applicazione.

Oltre ai **widget** grafici abbiamo i 3 **listener** *delegati* e il **timer**.

I metodi utilizzati sono i seguenti:

- tStart(): Abilita il timer.
- tStop(): Ferma il timer.
- tIntervallo(int intervallo): Modifica l'intervallo del timer con il parametro passato.

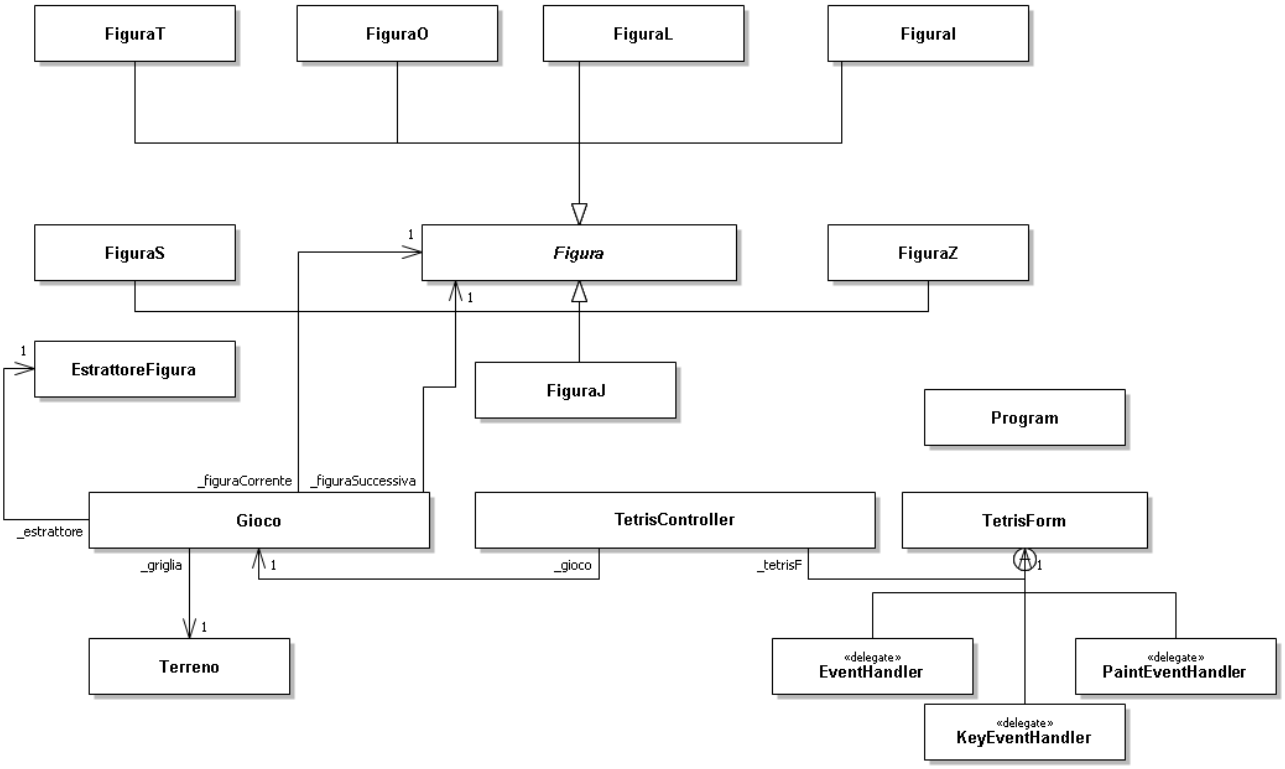
3.2.14 Program



Program è il punto di accesso e di avvio dell'applicazione. All'interno del suo metodo principale, andiamo ad istanziare un oggetto TetrisController. Che è il punto di ingresso dell'applicazione dal punto di vista del giocatore.

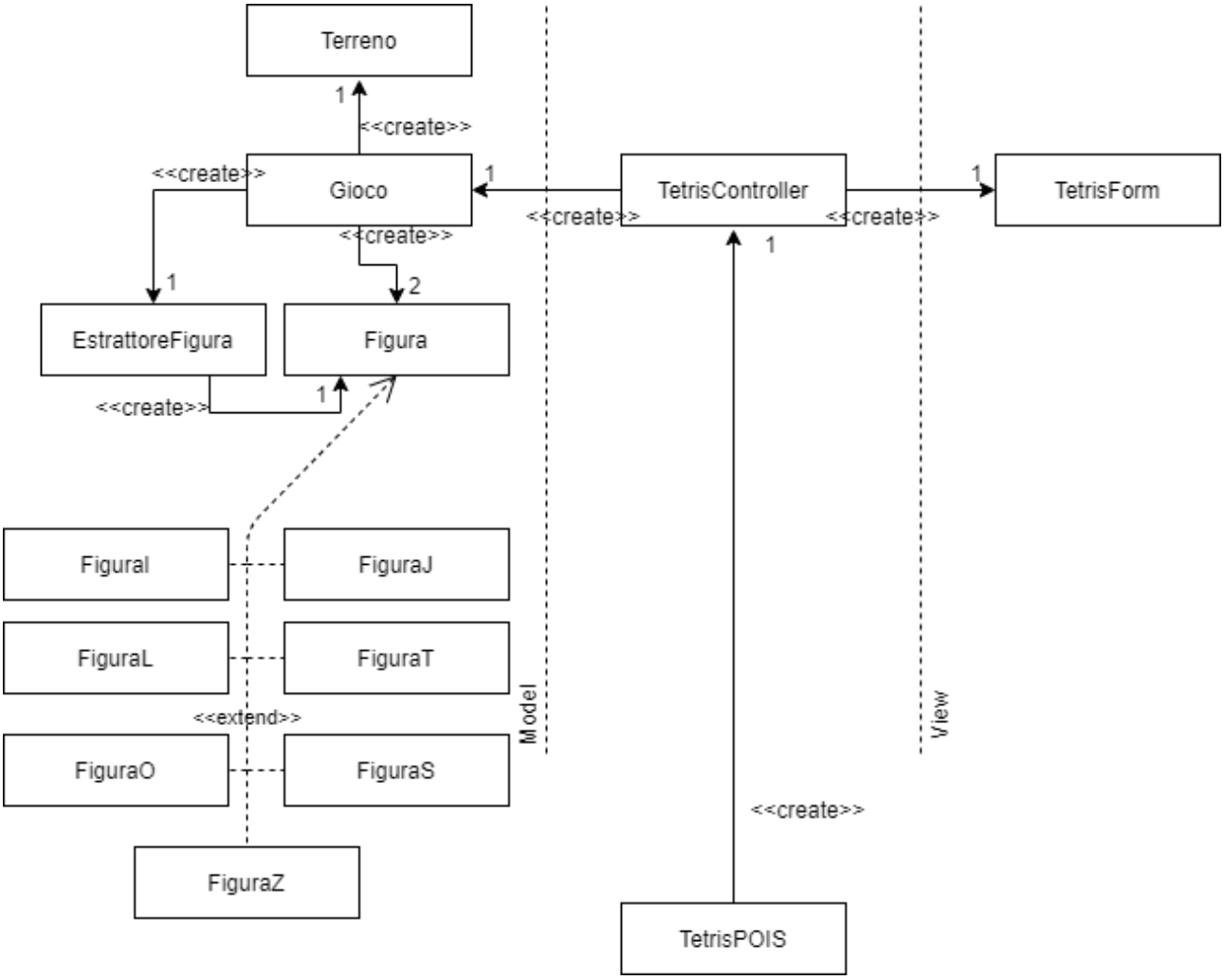
3.3 Diagramma UML

Una volta stabilite le varie relazioni presenti fra le classi otteniamo il seguente schema UML.



(Per difficoltà di visualizzazione riporto lo schema UML senza attributi e metodi, è comunque allegata l'immagine con lo schema UML intero)

3.4 Diagramma delle classi MVC



Questo diagramma mette in evidenza le relazioni che si creano con il pattern MVC.

Come si può notare il controller il gioco è la parte centrale del model, è lui che gestisce tutte le operazioni logiche sui dati.

TetrisController invece funge da intermediario tra il model e la view ovvero TetrisForm.

4. Implementazione

Di seguito viene riportato il codice dell'applicazione.

4.1 Terreno.cs

```
using System;

namespace TetrisPOIS.MVC.Model
{
    class Terreno
    {

        // Dimensioni della griglia
        public const int X = 10;
        public const int Y = 16;

        private static int[,] _campo = new int[Y, X];

        /// <summary>
        /// Costruttore
        /// </summary>
        public Terreno()
        {
            for (int i = 0; i < Y; i++)
                for (int j = 0; j < X; j++)
                    _campo[i, j] = 0;
        }

        /// <summary>
        /// Get/Set del campo di gioco
        /// </summary>
        public int[,] Campo
        {
            get { return _campo; }
            set { _campo = value; }
        }

    }
}
```

4.2 Figura

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TetrisPOIS
{
    public abstract class Figura
    {
        /// <summary>
        /// Get della forma
        /// </summary>
        public abstract int[,] Forma
        {
            get;
        }

        /// <summary>
        /// Get della rotazione
        /// </summary>
        public abstract int Rotazione
```

```
        {
            get;
        }

        /// <summary>
        /// Get/Set della coordinata X
        /// </summary>
        public abstract int X
        {
            get;
            set;
        }

        /// <summary>
        /// Get/Set della coordinata Y
        /// </summary>
        public abstract int Y
        {
            get;
            set;
        }

        /// <summary>
        /// Get/Set della proprietà "fissata"
        /// </summary>
        public abstract bool Fissata
        {
            get;
            set;
        }

        /// <summary>
        /// Get dell'immagine
        /// </summary>
        public abstract Image Sprite
        {
            get;
        }

        /// <summary>
        /// Metodo per identificare la coordinata X della prox rotazione
        /// </summary>
        /// <returns>
        /// La coordinata X della prox rotazione
        /// </returns>
        public abstract int XProssimaRotazione();

        /// <summary>
        /// Metodo per identificare la coordinata Y della prox rotazione
        /// </summary>
        /// <returns>
        /// La coordinata Y della prox rotazione
        /// </returns>
        public abstract int YProssimaRotazione();

        /// <summary>
        /// Metodo per identificare la forma della prossima rotazione
        /// </summary>
        /// <param name="rotazione">Rotazione che si vuole eseguire</param>
        /// <returns>
        /// La forma della figura ruotata
        /// </returns>
        public abstract int[,] ProssimaRotazione(int rotazione);

        /// <summary>
        /// Metodo per ruotare la figura
        /// </summary>
        public abstract void Ruota();
    }
}
```

4.3 FiguraI.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TetrisPOIS.MVC.Model
{
    class FiguraI : Figura
    {
        private const char NOME = 'I';
        private int _x, _y;
        private int _rotazione;
        private bool _fissata;
        private Image _sprite;

        // Forme della figura I
        private int[,] _forma0 = new int[4, 1]
        {
            {1},
            {1},
            {1},
            {1}
        };

        private int[,] _forma90 = new int[1, 4]
        {
            {1, 1, 1, 1}
        };


        /// <summary>
        /// Costruttore
        /// </summary>
        public FiguraI()
        {
            this._x = 4;
            this._y = 0;
            this._rotazione = 0;
            this._fissata = false;
            try
            {
                this._sprite = TetrisPOIS.Properties.Resources.Figura_I;
            }
            catch
            {
                new ResourceNotFoundException("Figura_I");
            }
        }

        /// <summary>
        /// Metodo per ruotare la figura
        /// </summary>
        public override void Ruota()
        {

            // Incremento la rotazione della figura
            _rotazione++;

            // La rotazione ha un ciclo che va da 0 a 3
            if (_rotazione > 3)
                _rotazione = 0;

            // Capovolgo l'immagine della figura
            _sprite.RotateFlip(RotateFlipType.Rotate90FlipNone);

            // Cambio le coordinate in base alla rotazione
            switch (_rotazione)
            {
```

```
            case 0:
                _y -= 2;
                _x++;
                break;
            case 1:
                _y++;
                _x--;
                break;
            case 2:
                _x += 2;
                _y--;
                break;
            case 3:
                _x -= 2;
                _y += 2;
                break;
        }
    }

    /// <summary>
    /// Metodo per identificare la forma della prossima rotazione
    /// </summary>
    /// <param name="rotazione">Rotazione che si vuole eseguire</param>
    /// <returns>
    /// La forma della figura ruotata
    /// </returns>
    public override int[,] ProssimaRotazione(int rotazione)
    {
        int[,] forma = null;

        rotazione++;
        if (rotazione > 3)
            rotazione = 0;

        switch (rotazione)
        {
            case 0:
                forma = _forma0;
                break;
            case 1:
                forma = _forma90;
                break;
            case 2:
                forma = _forma0;
                break;
            case 3:
                forma = _forma90;
                break;
        }

        return forma;
    }

    /// <summary>
    /// Metodo per identificare la coordinata X della prox rotazione
    /// </summary>
    /// <returns>
    /// La coordinata X della prox rotazione
    /// </returns>
    public override int XProssimaRotazione()
    {
        int xProx = _x;
        int rotazioneProx = _rotazione;

        rotazioneProx++;
        if (rotazioneProx > 3)
            rotazioneProx = 0;

        switch (rotazioneProx)
        {
            case 0:
                xProx++;
                break;
            case 1:
                xProx--;
```

```
            break;
        case 2:
            xProx += 2;
            break;
        case 3:
            xProx -= 2;
            break;
    }
    return xProx;
}

/// <summary>
/// Metodo per identificare la coordinata Y della prox rotazione
/// </summary>
/// <returns>
/// La coordinata Y della prox rotazione
/// </returns>
public override int YProssimaRotazione()
{
    int yProx = _y;
    int proxRotazione = _rotazione;

    proxRotazione++;
    if (proxRotazione > 3)
        proxRotazione = 0;

    switch (proxRotazione)
    {
        case 0:
            yProx -= 2;
            break;
        case 1:
            yProx++;
            break;
        case 2:
            yProx--;
            break;
        case 3:
            yProx += 2;
            break;
    }
    return yProx;
}

/// <summary>
/// Get della forma
/// </summary>
public override int[,] Forma
{
    get
    {
        int[,] forma = null;

        switch (_rotazione)
        {
            case 0:
                forma = _forma0;
                break;
            case 1:
                forma = _forma90;
                break;
            case 2:
                forma = _forma0;
                break;
            case 3:
                forma = _forma90;
                break;
        }
        return forma;
    }
}

/// <summary>
/// Get del nome
/// </summary>
```

```
        public char Nome
        {
            get{ return NOME; }
        }

        /// <summary>
        /// Get/Set della coordinata X
        /// </summary>
        public override int X
        {
            get { return _x; }
            set { _x = value; }
        }

        /// <summary>
        /// Get/Set della coordinata Y
        /// </summary>
        public override int Y
        {
            get { return _y; }
            set { _y = value; }
        }

        /// <summary>
        /// Get della rotazione
        /// </summary>
        public override int Rotazione
        {
            get { return _rotazione; }
        }

        /// <summary>
        /// Get dell'immagine
        /// </summary>
        public override Image Sprite
        {
            get { return _sprite; }
        }

        /// <summary>
        /// Get/Set della proprietà "fissata"
        /// </summary>
        public override bool Fissata
        {
            get { return _fissata; }
            set { _fissata = value; }
        }

    }
}
```

4.4 FiguraJ.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TetrisPOIS.MVC.Model
{
    class FiguraJ : Figura
    {
        private const char NOME = 'J';
        private int _x, _y;
        private int _rotazione;
        private bool _fissata;
        private Image _sprite;

        // Forme della figura
        private int[,] _forma0 = new int[3, 2]
```

```
{
    {0, 7},
    {0, 7},
    {7, 7}
};

private int[,] _forma90 = new int[2, 3]
{
    {7, 0, 0},
    {7, 7, 7}
};

private int[,] _forma180 = new int[3, 2]
{
    {7, 7},
    {7, 0},
    {7, 0}
};

private int[,] _forma270 = new int[2, 3]
{
    {7, 7, 7},
    {0, 0, 7}
};

/// <summary>
/// Costruttore
/// </summary>
public FiguraJ()
{

    this._x = 4;
    this._y = 0;
    this._rotazione = 0;
    this._fissata = false;
    try
    {
        this._sprite = TetrisPOIS.Properties.Resources.Figura_J;
    }
    catch
    {
        new ResourceNotFoundException("Figura_J");
    }
}

/// <summary>
/// Metodo per ruotare la figura
/// </summary>
public override void Ruota()
{
    _rotazione++;
    if (_rotazione > 3)
        _rotazione = 0;
    _sprite.RotateFlip(RotateFlipType.Rotate90FlipNone);

    switch (_rotazione)
    {
        case 0:
            _y--;
            break;
        case 1:
            break;
        case 2:
            _x++;
            break;
        case 3:
            _y++;
            _x--;
            break;
    }
}

/// <summary>
/// Metodo per identificare la forma della prossima rotazione
/// </summary>
```

```
/// <param name="rotazione">Rotazione che si vuole eseguire</param>
/// <returns>
/// La forma della figura ruotata
/// </returns>
public override int[,] ProssimaRotazione(int rotazione)
{
    int[,] forma = null;

    rotazione++;
    if (rotazione > 3)
        rotazione = 0;

    switch (rotazione)
    {
        case 0:
            forma = _forma0;
            break;
        case 1:
            forma = _forma90;
            break;
        case 2:
            forma = _forma180;
            break;
        case 3:
            forma = _forma270;
            break;
    }

    return forma;
}

/// <summary>
/// Metodo per identificare la coordinata X della prox rotazione
/// </summary>
/// <returns>
/// La coordinata X della prox rotazione
/// </returns>
public override int XProssimaRotazione()
{
    int xProx = _x;
    int rotazioneProx = _rotazione;

    rotazioneProx++;
    if (rotazioneProx > 3)
        rotazioneProx = 0;

    switch (rotazioneProx)
    {
        case 0:
            break;
        case 1:
            break;
        case 2:
            xProx++;
            break;
        case 3:
            xProx--;
            break;
    }
    return xProx;
}

/// <summary>
/// Metodo per identificare la coordinata Y della prox rotazione
/// </summary>
/// <returns>
/// La coordinata Y della prox rotazione
/// </returns>
public override int YProssimaRotazione()
{
    int yProx = _y;
    int rotazioneProx = _rotazione;

    rotazioneProx++;
    if (rotazioneProx > 3)
```

```
            rotazioneProx = 0;

        switch (rotazioneProx)
        {
            case 0:
                yProx--;
                break;
            case 1:
                break;
            case 2:
                break;
            case 3:
                yProx++;
                break;
        }
        return yProx;
    }

    /// <summary>
    /// Get della forma
    /// </summary>
    public override int[,] Forma
    {
        get
        {
            int[,] forma = null;

            switch (_rotazione)
            {
                case 0:
                    forma = _forma0;
                    break;
                case 1:
                    forma = _forma90;
                    break;
                case 2:
                    forma = _forma180;
                    break;
                case 3:
                    forma = _forma270;
                    break;
            }
            return forma;
        }
    }

    /// <summary>
    /// Get dell'immagine
    /// </summary>
    public override Image Sprite
    {
        get { return _sprite; }
    }

    /// <summary>
    /// Get del nome
    /// </summary>
    public char Nome
    {
        get { return NOME; }
    }

    /// <summary>
    /// Get/Set della coordinata X
    /// </summary>
    public override int X
    {
        get { return _x; }
        set { _x = value; }
    }

    /// <summary>
    /// Get/Set della coordinata Y
    /// </summary>
    public override int Y
```

```
        {
            get { return _y; }
            set { _y = value; }
        }

        /// <summary>
        /// Get della rotazione
        /// </summary>
        public override int Rotazione
        {
            get { return _rotazione; }
        }

        /// <summary>
        /// Get/Set della proprietà "fissata"
        /// </summary>
        public override bool Fissata
        {
            get { return _fissata; }
            set { _fissata = value; }
        }
    }
}
```

4.5 FiguraL.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TetrisPOIS.MVC.Model
{
    class FiguraL : Figura
    {
        private const char NOME = 'L';
        private int _x, _y;
        private int _rotazione;
        private bool _fissata;
        private Image _sprite;

        // Forme della figura
        private int[,] _forma0 = new int[3, 2]
        {
            {6, 0},
            {6, 0},
            {6, 6}
        };

        private int[,] _forma90 = new int[2, 3]
        {
            {6, 6, 6},
            {6, 0, 0}
        };

        private int[,] _forma180 = new int[3, 2]
        {
            {6, 6},
            {0, 6},
            {0, 6}
        };

        private int[,] _forma270 = new int[2, 3]
        {
            {0, 0, 6},
            {6, 6, 6}
        };
```

```
/// <summary>
/// Costruttore
/// </summary>
public FiguraL()
{
    this._x = 4;
    this._y = 0;
    this._rotazione = 0;
    this._fissata = false;
    try
    {
        this._sprite = TetrisPOIS.Properties.Resources.Figura_L;
    }
    catch
    {
        new ResourceNotFoundException("Figura_L");
    }
}

/// <summary>
/// Metodo per ruotare la figura
/// </summary>
public override void Ruota()
{
    _rotazione++;
    if (_rotazione > 3)
        _rotazione = 0;
    _sprite.RotateFlip(RotateFlipType.Rotate90FlipNone);

    switch (_rotazione)
    {
        case 0:
            _x++;
            break;
        case 1:
            _y++;
            _x--;
            break;
        case 2:
            _y--;
            break;
        case 3:
            break;
    }
}

/// <summary>
/// Metodo per identificare la forma della prossima rotazione
/// </summary>
/// <param name="rotazione">Rotazione che si vuole eseguire</param>
/// <returns>
/// La forma della figura ruotata
/// </returns>
public override int[,] ProssimaRotazione(int rotazione)
{
    int[,] forma = null;

    rotazione++;
    if (rotazione > 3)
        rotazione = 0;

    switch (rotazione)
    {
        case 0:
            forma = _forma0;
            break;
        case 1:
            forma = _forma90;
            break;
        case 2:
            forma = _forma180;
            break;
        case 3:
            forma = _forma270;
```

```
            break;
    }
    return forma;
}

/// <summary>
/// Metodo per identificare la coordinata X della prox rotazione
/// </summary>
/// <returns>
/// La coordinata X della prox rotazione
/// </returns>
public override int XProssimaRotazione()
{
    int xProx = _x;

    int rotazioneProx = this._rotazione;

    rotazioneProx++;
    if (rotazioneProx > 3)
       rotazioneProx = 0;

    switch (rotazioneProx)
    {
        case 0:
            xProx++;
            break;
        case 1:
            xProx--;
            break;
        case 2:
            break;
        case 3:
            break;
    }
    return xProx;
}

/// <summary>
/// Metodo per identificare la coordinata Y della prox rotazione
/// </summary>
/// <returns>
/// La coordinata Y della prox rotazione
/// </returns>
public override int YProssimaRotazione()
{
    int yProx = _y;
    int rotazioneProx = this._rotazione;

    rotazioneProx++;
    if (rotazioneProx > 3)
        rotazioneProx = 0;

    switch (rotazioneProx)
    {
        case 0:
            break;
        case 1:
            yProx++;
            break;
        case 2:
            yProx--;
            break;
        case 3:
            break;
    }
    return yProx;
}

/// <summary>
/// Get dell'immagine
/// </summary>
public override Image Sprite
{
    get { return _sprite; }
}
```

```
        /// <summary>
        /// Get del nome
        /// </summary>
        public char Nome
        {
            get { return NOME; }
        }

        /// <summary>
        /// Get/Set della coordinata X
        /// </summary>
        public override int X
        {
            get { return _x; }
            set { _x = value; }
        }

        /// <summary>
        /// Get/Set della coordinata Y
        /// </summary>
        public override int Y
        {
            get { return _y; }
            set { _y = value; }
        }

        /// <summary>
        /// Get della rotazione
        /// </summary>
        public override int Rotazione
        {
            get { return _rotazione; }
        }

        /// <summary>
        /// Get della forma
        /// </summary>
        public override int[,] Forma
        {
            get
            {
                int[,] forma = null;

                switch (_rotazione)
                {
                    case 0:
                        forma = _forma0;
                        break;
                    case 1:
                        forma = _forma90;
                        break;
                    case 2:
                        forma = _forma180;
                        break;
                    case 3:
                        forma = _forma270;
                        break;
                }
                return forma;

            }
        }

        /// <summary>
        /// Get/Set della proprietà "fissata"
        /// </summary>
        public override bool Fissata
        {
            get { return _fissata; }
            set { _fissata = value; }
        }
    }
}
```

4.6 FiguraO.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TetrisPOIS.MVC.Model
{
    class FiguraO : Figura
    {
        private const char NOME = 'O';
        private int _x, _y;
        private int _rotazione;
        private bool _fissata;
        private Image _sprite;

        // Forma della figura
        private int[,] _forma0 = new int[2, 2]
        {
            {2, 2},
            {2, 2}
        };


        /// <summary>
        /// Costruttore
        /// </summary>
        public FiguraO()
        {
            this._x = 4;
            this._y = 0;
            this._rotazione = 0;
            this._fissata = false;
            try
            {
                this._sprite = TetrisPOIS.Properties.Resources.Figura_O;
            }
            catch
            {
                new ResourceNotFoundException("Figura_O");
            }
        }

        /// <summary>
        /// Metodo per ruotare la figura
        /// </summary>
        public override void Ruota()
        {
            _rotazione = 0;
        }

        /// <summary>
        /// Get della rotazione
        /// </summary>
        public override int Rotazione
        {
            get { return _rotazione; }
        }

        /// <summary>
        /// Get della forma
        /// </summary>
        public override int[,] Forma
        {
            get { return _forma0; }
        }

        /// <summary>
        /// Metodo per identificare la forma della prossima rotazione
```

```
        /// </summary>
        /// <param name="rotazione">Rotazione che si vuole eseguire</param>
        /// <returns>
        /// La forma della figura ruotata
        /// </returns>
        public override int[,] ProssimaRotazione(int rotazione)
        {
            return _forma0;
        }

        /// <summary>
        /// Metodo per identificare la coordinata X della prox rotazione
        /// </summary>
        /// <returns>
        /// La coordinata X della prox rotazione
        /// </returns>
        public override int XProssimaRotazione()
        {
            return _x;
        }

        /// <summary>
        /// Metodo per identificare la coordinata Y della prox rotazione
        /// </summary>
        /// <returns>
        /// La coordinata Y della prox rotazione
        /// </returns>
        public override int YProssimaRotazione()
        {
            return _y;
        }

        /// <summary>
        /// Get del nome
        /// </summary>
        public char Nome
        {
            get { return NOME; }
        }

        /// <summary>
        /// Get/Set della coordinata X
        /// </summary>
        public override int X
        {
            get { return _x; }
            set { _x = value; }
        }

        /// <summary>
        /// Get/Set della coordinata Y
        /// </summary>
        public override int Y
        {
            get { return _y; }
            set { _y = value; }
        }

        /// <summary>
        /// Get dell'immagine
        /// </summary>
        public override Image Sprite
        {
            get { return _sprite; }
        }

        /// <summary>
        /// Get/Set della proprietà "fissata"
        /// </summary>
        public override bool Fissata
        {
            get { return _fissata; }
            set { _fissata = value; }
        }
    }
```

```
}
```

4.7 FiguraS.cs

```
using System.Drawing;

namespace TetrisPOIS.MVC.Model
{
    class FiguraS : Figura
    {
        private const char NOME = 'S';
        private int _x, _y;
        private int _rotazione;
        private bool _fissata;
        private Image _sprite;

        // Forme della figura
        private int[,] _forma0 = new int[2, 3]
{
            {0, 4, 4},
            {4, 4, 0}
};

        private int[,] _forma90 = new int[3, 2]
        {
            {4, 0},
            {4, 4},
            {0, 4}
        };



        /// <summary>
        /// Costruttore
        /// </summary>
        public FiguraS()
        {
            this._x = 3;
            this._y = 0;
            this._rotazione = 0;
            this._fissata = false;
            try
            {
                this._sprite = TetrisPOIS.Properties.Resources.Figura_S;
            }
            catch
            {
                new ResourceNotFoundException("Figura_S");
            }
        }

        /// <summary>
        /// Metodo per ruotare la figura
        /// </summary>
        public override void Ruota()
        {

            _rotazione++;
            if (_rotazione > 3)
                _rotazione = 0;
            _sprite.RotateFlip(RotateFlipType.Rotate90FlipNone);

            switch (_rotazione)
            {
                case 0:
                    break;
                case 1:
                    _x++;
                    break;
                case 2:
                    _x--;
                    _y++;
```

```
            break;
        case 3:
            _y--;
            break;
    }

}

/// <summary>
/// Metodo per identificare la forma della prossima rotazione
/// </summary>
/// <param name="rotazione">Rotazione che si vuole eseguire</param>
/// <returns>
/// La forma della figura ruotata
/// </returns>
public override int[,] ProssimaRotazione(int rotazione)
{
    int[,] forma = null;

    rotazione++;
    if (rotazione > 3)
        rotazione = 0;

    switch (rotazione)
    {
        case 0:
            forma = _forma0;
            break;
        case 1:
            forma = _forma90;
            break;
        case 2:
            forma = _forma0;
            break;
        case 3:
            forma = _forma90;
            break;
    }

    return forma;
}

/// <summary>
/// Metodo per identificare la coordinata X della prox rotazione
/// </summary>
/// <returns>
/// La coordinata X della prox rotazione
/// </returns>
public override int XProssimaRotazione()
{
    int xProx = _x;
    int rotazioneProx = _rotazione;

    rotazioneProx++;
    if (rotazioneProx > 3)
        rotazioneProx = 0;

    switch (rotazioneProx)
    {
        case 0:
            break;
        case 1:
            xProx++;
            break;
        case 2:
            xProx--;
            break;
        case 3:
            break;
    }
    return xProx;
}

/// <summary>
/// Metodo per identificare la coordinata Y della prox rotazione
```

```
/// </summary>
/// <returns>
/// La coordinata Y della prox rotazione
/// </returns>
public override int YProssimaRotazione()
{
    int yProx = _y;
    int rotazioneProx = _rotazione;

    rotazioneProx++;
    if (rotazioneProx > 3)
        rotazioneProx = 0;

    switch (rotazioneProx)
    {
        case 0:
            break;
        case 1:
            break;
        case 2:
            yProx++;
            break;
        case 3:
            yProx--;
            break;
    }
    return yProx;
}

/// <summary>
/// Get dell'immagine
/// </summary>
public override Image Sprite
{
    get { return _sprite; }
}

/// <summary>
/// Get della rotazione
/// </summary>
public override int Rotazione
{
    get { return _rotazione; }
}

/// <summary>
/// Get della forma
/// </summary>
public override int[,] Forma
{
    get
    {
        int[,] forma = null;

        switch (_rotazione)
        {
            case 0:
                forma = _forma0;
                break;
            case 1:
                forma = _forma90;
                break;
            case 2:
                forma = _forma0;
                break;
            case 3:
                forma = _forma90;
                break;
        }
        return forma;

    }
}

/// <summary>
```

```
        /// Get del nome
        /// </summary>
        public char Nome
        {
            get { return NOME; }
        }

        /// <summary>
        /// Get/Set della coordinata X
        /// </summary>
        public override int X
        {
            get { return _x; }
            set { _x = value; }
        }

        /// <summary>
        /// Get/Set della coordinata Y
        /// </summary>
        public override int Y
        {
            get { return _y; }
            set { _y = value; }
        }

        /// <summary>
        /// Get/Set della proprietà "fissata"
        /// </summary>
        public override bool Fissata
        {
            get { return _fissata; }
            set { _fissata = value; }
        }
    }
}
```

4.8 FiguraT.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TetrisPOIS.MVC.Model
{
    class FiguraT : Figura
    {
        private const char NOME = 'T';
        private int _x, _y;
        private int _rotazione;
        private bool _fissata;
        private Image _sprite;

        // Forme della figura
        private int[,] _forma0 = new int[2, 3]
        {
            {0, 3, 0},
            {3, 3, 3}
        };

        private int[,] _forma90 = new int[3, 2]
        {
            {3, 0},
            {3, 3},
            {3, 0}
        };

        private int[,] _forma180 = new int[2, 3]
        {
            {3, 3, 3},
```

```
    {0, 3, 0}
};

private int[,] _forma270 = new int[3, 2]
{
    {0, 3},
    {3, 3},
    {0, 3}
};


/// <summary>
/// Costruttore
/// </summary>
public FiguraT()
{
    this._x = 3;
    this._y = 0;
    this._rotazione = 0;
    this._fissata = false;
    try
    {
        this._sprite = TetrisPOIS.Properties.Resources.Figura_T;
    }
    catch
    {
        new ResourceNotFoundException("Figura_T");
    }
}

/// <summary>
/// Metodo per ruotare la figura
/// </summary>
public override void Ruota()
{
    _rotazione++;
    if (_rotazione > 3)
        _rotazione = 0;
    _sprite.RotateFlip(RotateFlipType.Rotate90FlipNone);

    switch (_rotazione)
    {
        case 0:
            break;
        case 1:
            _x++;
            break;
        case 2:
            _x--;
            _y++;
            break;
        case 3:
            _y--;
            break;
    }
}

/// <summary>
/// Metodo per identificare la forma della prossima rotazione
/// </summary>
/// <param name="rotazione">Rotazione che si vuole eseguire</param>
/// <returns>
/// La forma della figura ruotata
/// </returns>
public override int[,] ProssimaRotazione(int rotazione)
{
    int[,] forma = null;

    rotazione++;
    if (rotazione > 3)
        rotazione = 0;

    switch (rotazione)
    {
        case 0:
```

```
            forma = _forma0;
            break;
        case 1:
            forma = _forma90;
            break;
        case 2:
            forma = _forma180;
            break;
        case 3:
            forma = _forma270;
            break;
    }

    return forma;
}

/// <summary>
/// Metodo per identificare la coordinata X della prox rotazione
/// </summary>
/// <returns>
/// La coordinata X della prox rotazione
/// </returns>
public override int XProssimaRotazione()
{
    int xProx = _x;
    int rotazioneProx = _rotazione;

    rotazioneProx++;
    if (rotazioneProx > 3)
        rotazioneProx = 0;

    switch (rotazioneProx)
    {
        case 0:
            break;
        case 1:
            xProx++;
            break;
        case 2:
            xProx--;
            break;
        case 3:
            break;
    }
    return xProx;
}

/// <summary>
/// Metodo per identificare la coordinata Y della prox rotazione
/// </summary>
/// <returns>
/// La coordinata Y della prox rotazione
/// </returns>
public override int YProssimaRotazione()
{
    int yProx = _y;
    int rotazioneProx = _rotazione;

    rotazioneProx++;
    if (rotazioneProx > 3)
        rotazioneProx = 0;

    switch (rotazioneProx)
    {
        case 0:
            break;
        case 1:
            break;
        case 2:
            yProx++;
            break;
        case 3:
            yProx--;
            break;
    }
```

```
    return yProx;
}

/// <summary>
/// Get della rotazione
/// </summary>
public override int Rotazione
{
    get { return _rotazione; }
}

/// <summary>
/// Get della forma
/// </summary>
public override int[,] Forma
{
    get
    {
        int[,] forma = null;

        switch (_rotazione)
        {
            case 0:
                forma = _forma0;
                break;
            case 1:
                forma = _forma90;
                break;
            case 2:
                forma = _forma180;
                break;
            case 3:
                forma = _forma270;
                break;
        }
        return forma;
    }
}

/// <summary>
/// Get dell'immagine
/// </summary>
public override Image Sprite
{
    get { return _sprite; }
}

/// <summary>
/// Get del nome
/// </summary>
public char Nome
{
    get { return NOME; }
}

/// <summary>
/// Get/Set della coordinata X
/// </summary>
public override int X
{
    get { return _x; }
    set { _x = value; }
}

/// <summary>
/// Get/Set della coordinata Y
/// </summary>
public override int Y
{
    get { return _y; }
    set {  _y = value; }
}

/// <summary>
/// Get/Set della proprietà "fissata"
```

```
        /// </summary>
        public override bool Fissata
        {
            get { return _fissata; }
            set { _fissata = value; }
        }
    }
}
```

4.9 FiguraZ.cs

```
using System.Drawing;

namespace TetrisPOIS.MVC.Model
{
    class FiguraZ : Figura
    {
        private const char NOME = 'Z';
        private int _x, _y;
        private int _rotazione;
        private bool _fissata;
        private Image _sprite;

        // Forme della figura
        private int[,] _forma0 = new int[2, 3]
{
            {5, 5, 0},
            {0, 5, 5}
};

        private int[,] _forma90 = new int[3, 2]
        {
            {0, 5},
            {5, 5},
            {5, 0}
        };

        /// <summary>
        /// Costruttore
        /// </summary>
        public FiguraZ()
        {
            this._x = 3;
            this._y = 0;
            this._rotazione = 0;
            this._fissata = false;
            try
            {
                this._sprite = TetrisPOIS.Properties.Resources.Figura_Z;
            }
            catch
            {
                new ResourceNotFoundException("Figura_Z");
            }

        }

        /// <summary>
        /// Metodo per ruotare la figura
        /// </summary>
        public override void Ruota()
        {
            _rotazione++;
            if (_rotazione > 3)
                _rotazione = 0;
            _sprite.RotateFlip(RotateFlipType.Rotate90FlipNone);

            switch (_rotazione)
            {
                case 0:
                    break;
                case 1:
```

```
            _x++;
            break;
        case 2:
            _x--;
            _y++;
            break;
        case 3:
            _y--;
            break;
    }

}

/// <summary>
/// Metodo per identificare la forma della prossima rotazione
/// </summary>
/// <param name="rotazione">Rotazione che si vuole eseguire</param>
/// <returns>
/// La forma della figura ruotata
/// </returns>
public override int[,] ProssimaRotazione(int rotazione)
{
    int[,] forma = null;

    rotazione++;
    if (rotazione > 3)
        rotazione = 0;

    switch (rotazione)
    {
        case 0:
            forma = _forma0;
            break;
        case 1:
            forma = _forma90;
            break;
        case 2:
            forma = _forma0;
            break;
        case 3:
            forma = _forma90;
            break;
    }

    return forma;
}

/// <summary>
/// Metodo per identificare la coordinata X della prox rotazione
/// </summary>
/// <returns>
/// La coordinata X della prox rotazione
/// </returns>
public override int XProssimaRotazione()
{
    int xProx = _x;
    int rotazioneProx = _rotazione;

    rotazioneProx++;
    if (rotazioneProx > 3)
        rotazioneProx = 0;

    switch (rotazioneProx)
    {
        case 0:
            break;
        case 1:
            xProx++;
            break;
        case 2:
            xProx--;
            break;
        case 3:
            break;
    }
```

```
    return xProx;
}

/// <summary>
/// Metodo per identificare la coordinata Y della prox rotazione
/// </summary>
/// <returns>
/// La coordinata Y della prox rotazione
/// </returns>
public override int YProssimaRotazione()
{
    int yProx = _y;
    int rotazioneProx = _rotazione;

    rotazioneProx++;
    if (rotazioneProx > 3)
        rotazioneProx = 0;

    switch (rotazioneProx)
    {
        case 0:
            break;
        case 1:
            break;
        case 2:
            yProx++;
            break;
        case 3:
            yProx--;
            break;
    }
    return yProx;
}

/// <summary>
/// Get dell'immagine
/// </summary>
public override Image Sprite
{
    get { return _sprite; }
}

/// <summary>
/// Get del nome
/// </summary>
public char Nome
{
    get { return NOME; }
}

/// <summary>
/// Get/Set della coordinata X
/// </summary>
public override int X
{
    get { return _x; }
    set { _x = value; }
}

/// <summary>
/// Get/Set della coordinata Y
/// </summary>
public override int Y
{
    get { return _y; }
    set { _y = value; }
}

/// <summary>
/// Get della rotazione
/// </summary>
public override int Rotazione
{
    get { return _rotazione; }
}
```

```
        /// <summary>
        /// Get della forma
        /// </summary>
        public override int[,] Forma
        {
            get
            {
                int[,] forma = null;

                switch (_rotazione)
                {
                    case 0:
                        forma = _forma0;
                        break;
                    case 1:
                        forma = _forma90;
                        break;
                    case 2:
                        forma = _forma0;
                        break;
                    case 3:
                        forma = _forma90;
                        break;
                }
                return forma;

            }
        }

        /// <summary>
        /// Get/Set della proprietà "fissata"
        /// </summary>
        public override bool Fissata
        {
            get { return _fissata; }
            set { _fissata = value; }
        }
    }
}
```

4.10EstrattoreFigura.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TetrisPOIS.MVC.Model
{
    class EstrattoreFigura
    {
        private Random rnd = new Random();

        public Figura EstraiFigura()
        {
            Figura figuraCorrente = null;

            switch (rnd.Next(0, 7))
            {
                case 0:
                    figuraCorrente = new FiguraI();
                    break;
                case 1:
                    figuraCorrente = new FiguraO();
                    break;
                case 2:
                    figuraCorrente = new FiguraT();
                    break;
                case 3:
                    figuraCorrente = new FiguraS();
```

```
                    break;
                case 4:
                    figuraCorrente = new FiguraZ();
                    break;
                case 5:
                    figuraCorrente = new FiguraL();
                    break;
                case 6:
                    figuraCorrente = new FiguraJ();
                    break;
            }

            return figuraCorrente;
        }
    }
}
```

4.11Gioco.cs

```
using System;
using System.Diagnostics;
using System.Media;
using TetrisPOIS.MVC;
using TetrisPOIS.MVC.Model;

namespace TetrisPOIS
{
    /// <summary>
    /// Classe che contiene il codice effettivo del gioco
    /// </summary>
    class Gioco
    {
        // Elementi di gioco
        private Terreno _griglia;
        private EstrattoreFigura _estrattore = new EstrattoreFigura();
        private Figura _figuraCorrente = null, _figuraSuccessiva = null;
        private int _punteggio;
        private int _livello;
        private bool _inGioco = false;

        // Elementi sonori
        private SoundPlayer _gameOverFx;
        private SoundPlayer _lineaCompletaFx;
        private SoundPlayer _figuraBloccataFx;


        /// <summary>
        /// Costruttore
        /// </summary>
                public Gioco()
        {
            IniziaGioco();

            // Gestione caricamento immagini
            try
            {
                this._gameOverFx = new SoundPlayer(TetrisPOIS.Properties.Resources.game_over);
            }
            catch
            {
                new ResourceNotFoundException("Suono_GameOver");
            }
            try
            {
                this._lineaCompletaFx = new SoundPlayer(TetrisPOIS.Properties.Resources.linea_completa);
            }
            catch
            {
                new ResourceNotFoundException("Suono_LineaCompleta");
            }
            try
            {
```

```
                this._figuraBloccataFx = new SoundPlayer(TetrisPOIS.Properties.Resources.figura_bloccata);
            }
            catch
            {
                new ResourceNotFoundException("Suono_FiguraBloccata");
            }
        }

        /// <summary>
        /// Metodo per iniziare una partita
        /// </summary>
        public void IniziaGioco()
        {
            this._inGioco = true;
            this._griglia = new Terreno();
            this._punteggio = 0;
            this._livello = 1;

            GeneraFigura();
        }

        /// <summary>
        /// Metodo per generare una nuova figura
        /// e assegnarne una nuova a quella successiva
        /// </summary>
        public void GeneraFigura()
        {
            if (_inGioco)
            {
                if ((_figuraCorrente == null) && (_figuraSuccessiva == null))
                {
                    // Estraggo tutti e due le figure a inizio partita
                    _figuraCorrente = _estrattore.EstraiFigura();
                    _figuraSuccessiva = _estrattore.EstraiFigura();
                }
                else
                {
                    // A partita avviata switch delle figure e ne estraggo una successiva
                    _figuraCorrente = _figuraSuccessiva;
                    _figuraSuccessiva = _estrattore.EstraiFigura();
                }
            }

        }

        /// <summary>
        /// Metodo per muovere verso il basso la
        /// figura corrente
        /// </summary>
        public bool MuoviGiu()
        {
            if (_inGioco)
            {
                // Controllo che la figura possa ancora muoversi altrimenti la fisso nella griglia
                if (((FiguraCorrente.Y + NumeroRigheFigura) < _griglia.Campo.GetLength(0)) &&
(PuoMuoversi(FiguraCorrente.X, FiguraCorrente.Y + 1)))
                {
                    FiguraCorrente.Y++;
                }
                else
                {
                    FissaFigura();
                    FiguraCorrente.Fissata = true;
                    LineaCompletata();
                }
            }

            // Se la seconda riga è occupata da un'altra figura allora sarà impossibile muoversi e sarà game
over
            if ((_griglia.Campo[1, 3] != 0) || (_griglia.Campo[1, 4] != 0) || (_griglia.Campo[1, 5] != 0))
            {
                GameOver();
            }

            return FiguraCorrente.Fissata;
```

```
} // DOPO WHITE BOX

/// <summary>
/// Metodo per muovere verso sinistra la
/// figura corrente
/// </summary>
public void MuoviSinistra()
{
    if (_inGioco)
    {
        // Controllo che la figura non sia al bordo della griglia
        if (FiguraCorrente.X > 0)
        {
            if (PuoMuoversi(FiguraCorrente.X - 1, FiguraCorrente.Y))
            {
                FiguraCorrente.X--;
            }
        }
    }
}

/// <summary>
/// Metodo per muovere verso destra la
/// figura corrente
/// </summary>
public void MuoviDestra()
{
    if (_inGioco)
    {
        // Controllo che la figura non sia al bordo della griglia
        if ((FiguraCorrente.X + NumeroColonneFigura) < _griglia.Campo.GetLength(1))
        {
            if (PuoMuoversi(FiguraCorrente.X + 1, FiguraCorrente.Y))
            {
                FiguraCorrente.X++;
            }
        }
    }
}

/// <summary>
/// Metodo per ruotare la figura corrente
/// </summary>
/// <returns>
/// Se la figura ha effettuato una rotazione o meno
/// </returns>
public bool RuotaFigura()
{
    bool fatto = false;

    if (_inGioco)
    {
        if (PuoRuotare())
        {
            _figuraCorrente.Ruota();
            fatto = true;
        }
    }
    return fatto;
}

/// <summary>
/// Metodo per far finire una partita
/// </summary>
private void GameOver()
{
    this._inGioco = false;

    // Riproduco FX
    try
    {
        SuonoGameOver();
    }
    catch
    {
```

```
            new ResourceNotFoundException("Suono_GameOver");
        }
    }

    /// <summary>
    /// Metodo per accertarsi che la figura corrente
    /// possa muoversi in una determinata direzione
    /// (basso, sx, dx)
    /// </summary>
    /// <returns>
    /// Se la figura puo muoversi
    /// </returns>
    private bool PuoMuoversi(int x, int y)
    {
        bool puoMuoversi = true;

        // Creo la matrice temporanea
        int[,] tempGriglia = _griglia.Campo;

        // Controllo se la figura ha spazio per muoversi
        for (int i = y, righe = 0; i < (y + NumeroRigheFigura); i++, righe++)
        {
            for (int j = x, colonne = 0; j < (x + NumeroColonneFigura); j++, colonne++)
            {
                if ((tempGriglia[i, j] != 0) && (FiguraCorrente.Forma[righe, colonne] != 0))
                {
                    puoMuoversi = false;
                }

            }

        }

        return puoMuoversi;
    }

    /// <summary>
    /// Metodo per accertarsi che la figura corrente
    /// possa ruotare.
    /// </summary>
    /// <returns>
    /// Se la figura puo ruotare
    /// </returns>
    private bool PuoRuotare()
    {
        bool puoruotare = true;

        // Creo una matrice temporanea della figura a rotazione effetuata
        int[,] temp = _figuraCorrente.ProssimaRotazione(_figuraCorrente.Rotazione);

        // Ottengo le cordinate a rotazione effettuata
        int XProx = _figuraCorrente.XProssimaRotazione();
        int YProx = _figuraCorrente.YProssimaRotazione();

        // Controllo che la figura non esca dai bordi della griglia
        if ((XProx + temp.GetLength(1)) > _griglia.Campo.GetLength(1))
        {
            puoruotare = false;
        }
        else if ((YProx + temp.GetLength(0)) > _griglia.Campo.GetLength(0))
        {
            puoruotare = false;
        }
        else if (XProx < 0)
        {
            puoruotare = false;
        }
        else
        {
            // Controllo che non si sovrapponga a un'altra figura
            for (int i = YProx; i < (YProx + temp.GetLength(0)); i++)
            {
                for (int j = XProx; j < (XProx + temp.GetLength(1)); j++)
                {
                    if (_griglia.Campo[i, j] != 0)
```

```
                        {
                            puoruotare = false;
                        }
                    }
                }
            }

            return puoruotare;
        }

        /// <summary>
        /// Metodo per "fissare" una figura alla griglia di gioco
        /// </summary>
        private void FissaFigura()
        {
            // Identifico le coordinate attuali della figura nella griglia, scorro tutta la figura
            // e assegno alla griglia i valori della figura corrente
            for (int i = FiguraCorrente.Y, righe = 0; i < (FiguraCorrente.Y + NumeroRigheFigura); i++,
righe++)
            {
                for (int j = FiguraCorrente.X, colonne = 0; j < (FiguraCorrente.X + NumeroColonneFigura);
j++, colonne++)
                {
                    if (FiguraCorrente.Forma[righe, colonne] != 0)
                    {
                        _griglia.Campo[i, j] = FiguraCorrente.Forma[righe, colonne];
                    }
                }
            }

            // Riproduco FX
            try
            {
                SuonoFiguraBloccata();
            }
            catch
            {
                new ResourceNotFoundException("Suono_FiguraBloccata");
            }

            // Aumento punteggio e controllo il livello
            AumentaPunteggio();
            ScalaLivello(_punteggio);

        }

        /// <summary>
        /// Metodo che controllo se è stata completata una o più linee
        /// </summary>
        private void LineaCompletata()
        {
            int nLineeCompletate = 0;

            for(int i = 0; i < _griglia.Campo.GetLength(0); i++)
            {
                bool lineaCompletata = true;

                // Controllo riga per riga se ci sono delle linee completate
                for(int j = 0; (j < _griglia.Campo.GetLength(1)) && (lineaCompletata == true); j++)
                {
                    if(_griglia.Campo[i,j] == 0)
                    {
                        lineaCompletata = false;
                    }
                }

                if (lineaCompletata)
                {
                    nLineeCompletate++;

                    for (int y = i; y > 0; y--)
                    {
                        for (int j = 0; j < _griglia.Campo.GetLength(1); j++)
                        {
                            // La riga della griglia sarà uguale a quella sopra
```

```
                        _griglia.Campo[y, j] = _griglia.Campo[y - 1, j];
                            _griglia.Campo[0, j] = 0;
                    }
                }
            }
        }

        if(nLineeCompletate > 0)
        {
            // Aumento punteggio in base alle righe completate, controllo il livello e riproduco suono
            try
            {
                SuonoLineaCompletata();
            }
            catch
            {
                new ResourceNotFoundException("Suono_LineaCompletata");
            }

            AumentaPunteggio(nLineeCompletate);
            ScalaLivello(_punteggio);
        }

    }

    /// <summary>
    /// Metodo che gestisce l'aumento statico del punteggio
    /// </summary>
    private void AumentaPunteggio()
    {
        _punteggio += 1;
    }

    /// <summary>
    /// Metodo che sfrutta l'overloading per gestire l'aumento del punteggio in base alle linee
completate
    /// </summary>
    /// <param name="LineeCompletate">Numero di linee completate</param>
    private void AumentaPunteggio(int LineeCompletate)
    {
        switch (LineeCompletate)
        {
            case 1:
                _punteggio += 10;
                break;
            case 2:
                _punteggio += 30;
                break;
            case 3:
                _punteggio += 60;
                break;
            case 4:
                _punteggio += 100;
                break;
        }
    }

    /// <summary>
    /// Riproduce il suono di una figura che si blocca
    /// </summary>
    private void SuonoFiguraBloccata()
    {
        _figuraBloccataFx.Play();
    }

    /// <summary>
    /// Riproduce il suono di una linea completata
    /// </summary>
    private void SuonoLineaCompletata()
    {
        _lineaCompletaFx.Play();
    }

    /// <summary>
    /// Riproduce il suono del game over
```

```
/// </summary>
private void SuonoGameOver()
{
    _gameOverFx.Play();
}

/// <summary>
/// Aumenta il livello in base al punteggio raggiunto
/// </summary>
/// <param name="punteggio">Punteggio attuale del giocatore</param>
private void ScalaLivello(int punteggio)
{
    if (punteggio > 100)
    {
        _livello = 2;
    }
    if (punteggio > 200)
    {
        _livello = 3;
    }
    if (punteggio > 300)
    {
        _livello = 4;
    }
    if (punteggio > 400)
    {
        _livello = 5;
    }

}

/// <summary>
/// Get dell'attributo InGioco
/// </summary>
public bool InGioco
{
    get { return _inGioco; }
}

/// <summary>
/// Get della figura corrente
/// </summary>
public Figura FiguraCorrente
{
    get { return _figuraCorrente; }
}

/// <summary>
/// Get della figura successiva
/// </summary>
public Figura FiguraSuccessiva
{
    get { return _figuraSuccessiva; }
}

/// <summary>
/// Get del numero di righe della figura corrente
/// </summary>
private int NumeroRigheFigura
{
    get { return _figuraCorrente.Forma.GetLength(0); }
}

/// <summary>
/// Get del numero di colonne della figura corrente
/// </summary>
private int NumeroColonneFigura
{
    get { return _figuraCorrente.Forma.GetLength(1); }
}

/// <summary>
/// Get della griglia di gioco
/// </summary>
public Terreno Griglia
```

```
        {
            get { return this._griglia; }
        }

        /// <summary>
        /// Get del punteggio della partita attuale
        /// </summary>
        public int Punteggio
        {
            get { return _punteggio; }
        }

        /// <summary>
        /// Get del livello
        /// </summary>
        public int Livello
        {
            get { return _livello; }
        }

    }
}
```

4.12TetrisController.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace TetrisPOIS.MVC.Controller
{
    /// <summary>
    /// Classe controller, integra la parte grafica con il codice effettivo del gioco
    /// </summary>
    class TetrisController
    {
        // Costante utilizzata per lo stretch dell'immagine
        private const int DIMENSIONE_BLOCCHI = 25;

        // Classi che il controller gestirà
        private TetrisForm _tetrisF;
        private Gioco _gioco;

        // Elementi operazionali
        private Rectangle _fig_corrente;
        private bool _menu = true;
        private bool _inPausa = false;

        // Elementi grafici
        private SolidBrush _brushSide = new SolidBrush(Color.DarkBlue);
        private SolidBrush _brushSfondo = new SolidBrush(Color.DarkBlue);

        /// <summary>
        /// Costruttore
        /// </summary>
        public TetrisController()
        {
            this._tetrisF = new TetrisForm();
            _tetrisF.KeyDown += new KeyEventHandler(TastoPremuto);
            _tetrisF.Paint += new PaintEventHandler(DisegnaGioco);
            _tetrisF.T.Tick += new System.EventHandler(TetrisFormTick);
            _tetrisF.NuovaPartita.Click += new System.EventHandler(NuovaPartitaClick);
            _tetrisF.Esci.Click += new System.EventHandler(EsciClick);
            _tetrisF.Aiuto.Click += new System.EventHandler(AiutoClick);
            _tetrisF.Pausa.Click += new System.EventHandler(PausaClick);
            _tetrisF.Riprendi.Click += new System.EventHandler(RiprendiClick);
        }

        /// <summary>
        /// Metodo che mostra all'utente la form
        /// </summary>
        public void Inizia()
        {
```

```
            _tetrisF.ShowDialog();
        }

        /// <summary>
        /// Gestisce la fine del gioco
        /// </summary>
        private void GameOver()
        {
            _tetrisF.tStop();
            // Utilizzo una message box per la fine della partita
            string messaggio = "\tPurtroppo hai perso! Hai totalizzato " + _gioco.Punteggio + " punti";
            string titolo = "Game Over";
            DialogResult risposta = MessageBox.Show(messaggio, titolo, MessageBoxButtons.OK,
MessageBoxIcon.Error);

            if (risposta == System.Windows.Forms.DialogResult.OK)
            {
                _tetrisF.Close();
            }
            else
            {
                throw new Exception("Errore nel finire la partita!");
            }

        }

        /// <summary>
        /// Gestione del click del mouse sul bottone Nuova Partita
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void NuovaPartitaClick(object sender, EventArgs e)
        {
            _gioco = new Gioco();
            _menu = false;
            _tetrisF.tStart();
            _tetrisF.MenuPrincipale.Hide();
        }

        /// <summary>
        /// Gestione del click del mouse sul bottone Comandi
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void AiutoClick(object sender, EventArgs e)
        {
            MessageBox.Show("\tFRECCIA SU: Ruota la figura \r\n\tFRECCIA GIU: Aumenta velocita discesa della
figura \r\n\tFRECCIA DESTRA: Muove la figura a destra \r\n\tFRECCIA SINISTRA: Muove la figura a sinistra",
                            "COMANDI",
                            MessageBoxButtons.OK,
                            MessageBoxIcon.Information);
        }

        /// <summary>
        /// Gestione del click del mouse sul bottone Esci
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void EsciClick(object sender, EventArgs e)
        {
            _tetrisF.Close();
        }

        /// <summary>
        /// Metodo per mettere in pausa la partita
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void PausaClick(object sender, EventArgs e)
        {
            _inPausa = true;
            _tetrisF.Riprendi.Show();
            _tetrisF.Pausa.Hide();
            _tetrisF.tStop();
        }
```

```
/// <summary>
/// Metodo per riprendere la partita
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void RiprendiClick(object sender, EventArgs e)
{
    _inPausa = false;
    _tetrisF.Riprendi.Hide();
    _tetrisF.Pausa.Show();
    _tetrisF.tStart();
}


/// <summary>
/// Metodo che gestisce gli eventi che si verificano in un dato intervallo
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void TetrisFormTick(object sender, System.EventArgs e)
{
    if (_gioco.InGioco == false)
    {
        GameOver();
    }
    else if (_gioco.InGioco)
    {
        ScendiGiu();
    }

}

/// <summary>
/// Fa scendere di un blocco la figura
/// </summary>
private void ScendiGiu()
{
    if (_gioco.InGioco)
    {
        if(_gioco.FiguraCorrente.Fissata == false)
        {
            _gioco.MuoviGiu();
            if(_gioco.FiguraCorrente.Fissata == true)
            {
                _tetrisF.Invalidate();
                _gioco.GeneraFigura();
            }
        }
    }

    // Aggiorno il livello e il punteggio nella GUI
    _tetrisF.Punteggio.Text = "Punteggio = " + _gioco.Punteggio.ToString();
    _tetrisF.Livello.Text = "Livello = " + _gioco.Livello.ToString();

    // Aumento la difficolta
    ModificaDifficolta();

    _tetrisF.Invalidate(_fig_corrente);
    _fig_corrente.Y += DIMENSIONE_BLOCCHI;
    _tetrisF.Invalidate(_fig_corrente);
    _tetrisF.Update();

}

/// <summary>
/// Metodo che gestisce la parte di visualizzazione grafica
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void DisegnaGioco(object sender, PaintEventArgs e)
{
    if (!_menu)
    {
        e.Graphics.FillRectangle(_brushSide, new Rectangle(250, 0, 125, 400));
```

```
                DisegnaBlocchi(e);
                DisegnaFigura(e);
                DisegnaFiguraSuccessiva(_gioco.FiguraSuccessiva, e);
            }

        }

        /// <summary>
        /// Gestisce la visualizzazione dei blocchi fissati alla griglia
        /// </summary>
        /// <param name="e"></param>
        private void DisegnaBlocchi(PaintEventArgs e)
        {
            for (int i = 0; i < _gioco.Griglia.Campo.GetLength(0); i++)
            {
                for (int j = 0; j < _gioco.Griglia.Campo.GetLength(1); j++)
                {
                    switch (_gioco.Griglia.Campo[i, j])
                    {
                        case 1:
                            try
                            {
                                e.Graphics.DrawImage(TetrisPOIS.Properties.Resources.Blocco_I, j *
DIMENSIONE_BLOCCHI, i * DIMENSIONE_BLOCCHI);
                            }
                            catch
                            {
                                new ResourceNotFoundException("Blocco_I");
                            }

                            break;
                        case 2:
                            try
                            {
                                e.Graphics.DrawImage(TetrisPOIS.Properties.Resources.Blocco_O, j *
DIMENSIONE_BLOCCHI, i * DIMENSIONE_BLOCCHI);
                            }
                            catch
                            {
                                new ResourceNotFoundException("Blocco_O");
                            }

                           break;
                        case 3:
                            try
                            {
                                e.Graphics.DrawImage(TetrisPOIS.Properties.Resources.Blocco_T, j *
DIMENSIONE_BLOCCHI, i * DIMENSIONE_BLOCCHI);
                            }
                            catch
                            {
                                new ResourceNotFoundException("Blocco_T");
                            }

                           break;
                        case 4:
                            try
                            {
                                e.Graphics.DrawImage(TetrisPOIS.Properties.Resources.Blocco_S, j *
DIMENSIONE_BLOCCHI, i * DIMENSIONE_BLOCCHI);
                            }
                            catch
                            {
                                new ResourceNotFoundException("Blocco_S");
                            }

                           break;
                        case 5:
                            try
                            {
                                e.Graphics.DrawImage(TetrisPOIS.Properties.Resources.Blocco_Z, j *
DIMENSIONE_BLOCCHI, i * DIMENSIONE_BLOCCHI);
                            }
                            catch
                            {
```

```
                                new ResourceNotFoundException("Blocco_Z");
                            }

                            break;
                        case 6:
                            try
                            {
                                e.Graphics.DrawImage(TetrisPOIS.Properties.Resources.Blocco_L, j *
DIMENSIONE_BLOCCHI, i * DIMENSIONE_BLOCCHI);
                            }
                            catch
                            {
                                new ResourceNotFoundException("Blocco_L");
                            }

                            break;
                        case 7:
                            try
                            {
                                e.Graphics.DrawImage(TetrisPOIS.Properties.Resources.Blocco_J, j *
DIMENSIONE_BLOCCHI, i * DIMENSIONE_BLOCCHI);
                            }
                            catch
                            {
                                new ResourceNotFoundException("Blocco_J");
                            }

                            break;
                    }
                }
            }
        }

        /// <summary>
        /// Disegna l'attuale figura
        /// </summary>
        /// <param name="e"></param>
        private void DisegnaFigura(PaintEventArgs e)
        {
            _fig_corrente = new Rectangle(_gioco.FiguraCorrente.X * DIMENSIONE_BLOCCHI,
                                          _gioco.FiguraCorrente.Y * DIMENSIONE_BLOCCHI,
                                          _gioco.FiguraCorrente.Sprite.Width,
                                          _gioco.FiguraCorrente.Sprite.Height);
            try
            {
                e.Graphics.DrawImage(_gioco.FiguraCorrente.Sprite, _fig_corrente);
            }
            catch
            {
                new ResourceNotFoundException("Sprite_FiguraCorrente");
            }

        }

        /// <summary>
        /// Disegna la figura successiva
        /// </summary>
        /// <param name="f"></param>
        /// <param name="e"></param>
        private void DisegnaFiguraSuccessiva(Figura f, PaintEventArgs e)
        {
            e.Graphics.FillRectangle(_brushSfondo, new Rectangle(280, 50, 30, 30));
            try
            {
                e.Graphics.DrawImage(f.Sprite, 290, 50);
            }
            catch
            {
                new ResourceNotFoundException("Sprite_Figura");
            }


        }

        /// <summary>
```

```
        /// Metodo che gestisce l'input da tastiera
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="k"></param>
        private void TastoPremuto(object sender, KeyEventArgs k)
        {
            if (!_inPausa)
            {
                switch (k.KeyData)
                {
                    case (Keys.Up):
                        _gioco.RuotaFigura();

                        // Attraverso invalidate() elimino dalla gestione del timer il rettangolo in cui è
l'immagine
                        // attualmente e il rettangolo in cui comparirà l'immagine ruotata, eseguendo infine
l'update()
                        _tetrisF.Invalidate(new Rectangle(_gioco.FiguraCorrente.X * DIMENSIONE_BLOCCHI,
                                                          _gioco.FiguraCorrente.Y * DIMENSIONE_BLOCCHI,
                                                          _gioco.FiguraCorrente.Sprite.Width,
                                                          _gioco.FiguraCorrente.Sprite.Height));
                        _tetrisF.Invalidate(_fig_corrente);
                        _tetrisF.Update();
                        break;
                    case (Keys.Down):
                        ScendiGiu();
                        break;
                    case (Keys.Left):
                        _tetrisF.Invalidate(new Rectangle((_gioco.FiguraCorrente.X - 1) * DIMENSIONE_BLOCCHI,
                                                          _gioco.FiguraCorrente.Y * DIMENSIONE_BLOCCHI,
                                                          _gioco.FiguraCorrente.Sprite.Width,
                                                          _gioco.FiguraCorrente.Sprite.Height));
                        _gioco.MuoviSinistra();
                        _tetrisF.Invalidate(_fig_corrente);
                        _tetrisF.Update();
                        break;
                    case (Keys.Right):
                        _tetrisF.Invalidate(new Rectangle((_gioco.FiguraCorrente.X + 1) * DIMENSIONE_BLOCCHI,
                                                          _gioco.FiguraCorrente.Y * DIMENSIONE_BLOCCHI,
                                                          _gioco.FiguraCorrente.Sprite.Width,
                                                          _gioco.FiguraCorrente.Sprite.Height));
                        _gioco.MuoviDestra();
                        _tetrisF.Invalidate(_fig_corrente);
                        _tetrisF.Update();
                        break;
                }
            }
            else
            {
                k.Handled = true;
            }

        }

        /// <summary>
        /// Modifica la velocita con cui scendono i blocchi
        /// </summary>
        private void ModificaDifficolta()
        {

            switch (_gioco.Livello)
            {
                case 2:
                    _tetrisF.tIntervallo(800);
                    break;
                case 3:
                    _tetrisF.tIntervallo(600);
                    break;
                case 4:
                    _tetrisF.tIntervallo(500);
                    break;
                case 5:
                    _tetrisF.tIntervallo(400);
                    break;
            }
```

```
        }

    }

}
```

4.13TetrisForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace TetrisPOIS
{
    /// <summary>
    /// Contiene i metodi della parte grafica
    /// </summary>
    public partial class TetrisForm : Form
    {
        // Listener
        private delegate void KeyEventHandler(object sender, KeyEventArgs e);
        private delegate void PaintEventHandler(object sender, PaintEventArgs e);
        private delegate void EventHandler(object sender, System.EventArgs e);

        // Timer
        private Timer _t;

        /// <summary>
        /// Costruttore
        /// </summary>
        public TetrisForm()
        {
            InitializeComponent();
            this._t = new Timer();
            Inizializza();

        }

        /// <summary>
        /// Inizializza il timer
        /// </summary>
        public void Inizializza()
        {
            this._t.Enabled = false;
            tIntervallo(1000);
        }

        /// <summary>
        /// Get del timer
        /// </summary>
        public Timer T
        {
            get { return _t; }
        }

        /// <summary>
        /// Permette al timer di partire/ripartire
        /// </summary>
        public void tStart()
        {
            this._t.Enabled = true;
        }

        /// <summary>
        /// Fa stoppare il timer
        /// </summary>
        public void tStop()
```

```
        {
            this._t.Enabled = false;
        }

        /// <summary>
        /// Setta l'intervallo di t a un dato valore
        /// </summary>
        /// <param name="valore">Il valore che assumera l'intervallo</param>
        public void tIntervallo(int valore)
        {
            _t.Interval = valore;
        }

    }
}
```

4.14TetrisForm.Designer.cs

```
namespace TetrisPOIS
{
    using System;

    partial class TetrisForm

    {
        /// <summary>
        /// Variabile di progettazione necessaria.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Pulire le risorse in uso.
        /// </summary>
        /// <param name="disposing">ha valore true se le risorse gestite devono essere eliminate, false in
caso contrario.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Codice generato da Progettazione Windows Form

        /// <summary>
        /// Metodo necessario per il supporto della finestra di progettazione. Non modificare
        /// il contenuto del metodo con l'editor di codice.
        /// </summary>
        private void InitializeComponent()
        {
            this.Punteggio = new System.Windows.Forms.Label();
            this.Livello = new System.Windows.Forms.Label();
            this.MenuPrincipale = new System.Windows.Forms.TableLayoutPanel();
            this.Logo = new System.Windows.Forms.PictureBox();
            this.NuovaPartita = new System.Windows.Forms.Button();
            this.Esci = new System.Windows.Forms.Button();
            this.Aiuto = new System.Windows.Forms.Button();
            this.ProssimaFigura = new System.Windows.Forms.Label();
            this.Pausa = new System.Windows.Forms.Label();
            this.Riprendi = new System.Windows.Forms.Label();
            this.MenuPrincipale.SuspendLayout();
            ((System.ComponentModel.ISupportInitialize)(this.Logo)).BeginInit();
            this.SuspendLayout();
            //
            // Punteggio
            //
            this.Punteggio.AutoSize = true;
            this.Punteggio.BackColor = System.Drawing.Color.DarkBlue;
            this.Punteggio.Font = new System.Drawing.Font("Snap ITC", 9F, System.Drawing.FontStyle.Regular,
System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.Punteggio.ForeColor = System.Drawing.Color.Gold;
```

```
            this.Punteggio.Location = new System.Drawing.Point(252, 212);
            this.Punteggio.Name = "Punteggio";
            this.Punteggio.Size = new System.Drawing.Size(107, 17);
            this.Punteggio.TabIndex = 0;
            this.Punteggio.Text = "Punteggio = 0";
            //
            // Livello
            //
            this.Livello.AutoSize = true;
            this.Livello.BackColor = System.Drawing.Color.DarkBlue;
            this.Livello.Font = new System.Drawing.Font("Snap ITC", 11.25F, System.Drawing.FontStyle.Bold,
System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.Livello.ForeColor = System.Drawing.Color.Gold;
            this.Livello.Location = new System.Drawing.Point(251, 175);
            this.Livello.Name = "Livello";
            this.Livello.Size = new System.Drawing.Size(113, 19);
            this.Livello.TabIndex = 1;
            this.Livello.Text = "Livello = 1";
            //
            // MenuPrincipale
            //
            this.MenuPrincipale.AccessibleRole = System.Windows.Forms.AccessibleRole.MenuBar;
            this.MenuPrincipale.BackColor = System.Drawing.Color.DarkBlue;
            this.MenuPrincipale.ColumnCount = 3;
            this.MenuPrincipale.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent, 17.3913F));
            this.MenuPrincipale.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent, 65.2174F));
            this.MenuPrincipale.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent, 17.3913F));
            this.MenuPrincipale.Controls.Add(this.Logo, 1, 0);
            this.MenuPrincipale.Controls.Add(this.NuovaPartita, 1, 1);
            this.MenuPrincipale.Controls.Add(this.Esci, 1, 3);
            this.MenuPrincipale.Controls.Add(this.Aiuto, 1, 2);
            this.MenuPrincipale.Location = new System.Drawing.Point(-4, 0);
            this.MenuPrincipale.Name = "MenuPrincipale";
            this.MenuPrincipale.RowCount = 5;
            this.MenuPrincipale.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Percent, 60F));
            this.MenuPrincipale.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Percent, 10F));
            this.MenuPrincipale.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Percent, 10F));
            this.MenuPrincipale.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Percent, 10F));
            this.MenuPrincipale.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Percent, 10F));
            this.MenuPrincipale.Size = new System.Drawing.Size(379, 408);
            this.MenuPrincipale.TabIndex = 2;

            //
            // Logo
            //
            this.Logo.Image = global::TetrisPOIS.Properties.Resources.Logo;
            this.Logo.Location = new System.Drawing.Point(68, 3);
            this.Logo.Name = "Logo";
            this.Logo.Size = new System.Drawing.Size(241, 211);
            this.Logo.SizeMode = System.Windows.Forms.PictureBoxSizeMode.StretchImage;
            this.Logo.TabIndex = 0;
            this.Logo.TabStop = false;
            //
            // NuovaPartita
            //
            this.NuovaPartita.BackColor = System.Drawing.Color.Black;
            this.NuovaPartita.BackgroundImageLayout = System.Windows.Forms.ImageLayout.Center;
            this.NuovaPartita.FlatStyle = System.Windows.Forms.FlatStyle.Flat;
            this.NuovaPartita.Font = new System.Drawing.Font("Snap ITC", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.NuovaPartita.ForeColor = System.Drawing.Color.Gold;
            this.NuovaPartita.Location = new System.Drawing.Point(68, 247);
            this.NuovaPartita.Name = "NuovaPartita";
            this.NuovaPartita.Size = new System.Drawing.Size(241, 34);
            this.NuovaPartita.TabIndex = 1;
            this.NuovaPartita.Text = "Nuova Partita";
            this.NuovaPartita.UseVisualStyleBackColor = false;
```

```
            //
            // Esci
            //
            this.Esci.Anchor = ((System.Windows.Forms.AnchorStyles)((((System.Windows.Forms.AnchorStyles.Top
| System.Windows.Forms.AnchorStyles.Bottom)
            | System.Windows.Forms.AnchorStyles.Left)
            | System.Windows.Forms.AnchorStyles.Right)));
            this.Esci.BackColor = System.Drawing.Color.Black;
            this.Esci.FlatStyle = System.Windows.Forms.FlatStyle.Flat;
            this.Esci.Font = new System.Drawing.Font("Snap ITC", 9.75F, System.Drawing.FontStyle.Regular,
System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.Esci.ForeColor = System.Drawing.Color.Gold;
            this.Esci.Location = new System.Drawing.Point(68, 327);
            this.Esci.Name = "Esci";
            this.Esci.Size = new System.Drawing.Size(241, 34);
            this.Esci.TabIndex = 3;
            this.Esci.Text = "Esci";
            this.Esci.UseVisualStyleBackColor = false;
            //
            // Aiuto
            //
            this.Aiuto.BackColor = System.Drawing.Color.Black;
            this.Aiuto.BackgroundImageLayout = System.Windows.Forms.ImageLayout.Center;
            this.Aiuto.FlatStyle = System.Windows.Forms.FlatStyle.Flat;
            this.Aiuto.Font = new System.Drawing.Font("Snap ITC", 12F, System.Drawing.FontStyle.Regular,
System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.Aiuto.ForeColor = System.Drawing.Color.Gold;
            this.Aiuto.Location = new System.Drawing.Point(68, 287);
            this.Aiuto.Name = "Aiuto";
            this.Aiuto.Size = new System.Drawing.Size(241, 34);
            this.Aiuto.TabIndex = 4;
            this.Aiuto.Text = "Comandi";
            this.Aiuto.UseVisualStyleBackColor = false;
            //
            // ProssimaFigura
            //
            this.ProssimaFigura.AutoSize = true;
            this.ProssimaFigura.BackColor = System.Drawing.Color.DarkBlue;
            this.ProssimaFigura.Font = new System.Drawing.Font("Snap ITC", 8.25F,
System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.ProssimaFigura.ForeColor = System.Drawing.Color.Gold;
            this.ProssimaFigura.Location = new System.Drawing.Point(252, 19);
            this.ProssimaFigura.Name = "ProssimaFigura";
            this.ProssimaFigura.Size = new System.Drawing.Size(123, 15);
            this.ProssimaFigura.TabIndex = 3;
            this.ProssimaFigura.Text = "Prossima figura";
            //
            // Pausa
            //
            this.Pausa.AutoSize = true;
            this.Pausa.BackColor = System.Drawing.Color.DarkBlue;
            this.Pausa.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
            this.Pausa.Font = new System.Drawing.Font("Snap ITC", 14.25F, System.Drawing.FontStyle.Regular,
System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.Pausa.ForeColor = System.Drawing.Color.Gold;
            this.Pausa.Location = new System.Drawing.Point(285, 248);
            this.Pausa.Name = "Pausa";
            this.Pausa.Size = new System.Drawing.Size(79, 27);
            this.Pausa.TabIndex = 7;
            this.Pausa.Text = "Pausa";
            //
            // Riprendi
            //
            this.Riprendi.AutoSize = true;
            this.Riprendi.BackColor = System.Drawing.Color.DarkBlue;
            this.Riprendi.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
            this.Riprendi.Font = new System.Drawing.Font("Snap ITC", 12.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.Riprendi.ForeColor = System.Drawing.Color.Gold;
            this.Riprendi.Location = new System.Drawing.Point(268, 287);
            this.Riprendi.Name = "Riprendi";
            this.Riprendi.Size = new System.Drawing.Size(96, 25);
            this.Riprendi.TabIndex = 8;
            this.Riprendi.Text = "Riprendi";
            this.Riprendi.Visible = false;
```

```
            //
            // TetrisForm
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.BackColor = System.Drawing.SystemColors.ActiveCaptionText;
            this.ClientSize = new System.Drawing.Size(376, 401);
            this.Controls.Add(this.MenuPrincipale);
            this.Controls.Add(this.Livello);
            this.Controls.Add(this.Punteggio);
            this.Controls.Add(this.ProssimaFigura);
            this.Controls.Add(this.Pausa);
            this.Controls.Add(this.Riprendi);
            this.KeyPreview = true;
            this.Margin = new System.Windows.Forms.Padding(2);
            this.MaximizeBox = false;
            this.MaximumSize = new System.Drawing.Size(392, 440);
            this.MinimumSize = new System.Drawing.Size(392, 440);
            this.Name = "TetrisForm";
            this.Text = "Tetris";
            this.MenuPrincipale.ResumeLayout(false);
            ((System.ComponentModel.ISupportInitialize)(this.Logo)).EndInit();
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion

        public System.Windows.Forms.Label Punteggio;
        public System.Windows.Forms.Label Livello;
        public System.Windows.Forms.TableLayoutPanel MenuPrincipale;
        private System.Windows.Forms.PictureBox Logo;
        public System.Windows.Forms.Button NuovaPartita;
        public System.Windows.Forms.Button Esci;
        public System.Windows.Forms.Label ProssimaFigura;
        public System.Windows.Forms.Button Aiuto;
        public System.Windows.Forms.Label Pausa;
        public System.Windows.Forms.Label Riprendi;
    }
}
```

4.15 ResourceNotFoundException.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TetrisPOIS.MVC
{
    class ResourceNotFoundException : Exception
    {
        public ResourceNotFoundException()
        {
        }

        public ResourceNotFoundException(string message)
            : base(String.Format("La risorsa: " + message + "non è stata trovata!"))
        {

        }

        public ResourceNotFoundException(string message, Exception exception)
            : base(message, exception)
        {

        }
    }
}
```

4.16Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;
using TetrisPOIS.MVC.Controller;

namespace TetrisPOIS
{
    static class Program
    {
        /// <summary>
        /// Punto di ingresso principale dell'applicazione.
        /// </summary>
        [STAThread]
        static void Main()
        {
            TetrisController gioco = new TetrisController();
            gioco.Inizia();
        }
    }
}
```

5. TEST

La fase di testing è essenziale nello sviluppo di un software. È tuttavia molto difficile riuscire a creare un'applicazione al 100% corretta.

Il gioco sviluppato è stato corretto durante la sua scrittura, e apparentemente non vi sono problemi nel suo utilizzo, in questa fase stresseremo l'applicazione per vedere gli eventuali risultati.

5.1 White-Box Test

Il white-box test, o anche detto il test strutturale, è un particolare tipo di test che permette di rilevare errori all'interno dell'applicazione sfruttando la conoscenza del codice per ricavare da esso dati di test che consentano di percorrere tutto il programma.

L'unica criticità in dubbio riguardo l'applicazione è sul metodo **LineaCompletata**().

Per eseguire un test sul metodo viene creato un codice apposito al fine di provare varie impostazioni per testarne la correttezza, anche in condizioni surreali, questo perché nel codice che scriveremo arriveremo quasi a riempire l'intera griglia di blocchi (Cosa impossibile nello svolgimento di una partita normale), per testare il comportamento dell'applicazione anche in casi estremamente rari.

Per l'ambiente di Testing è stato creata una nuova soluzione apposita, in cui sono presenti le seguenti classi:

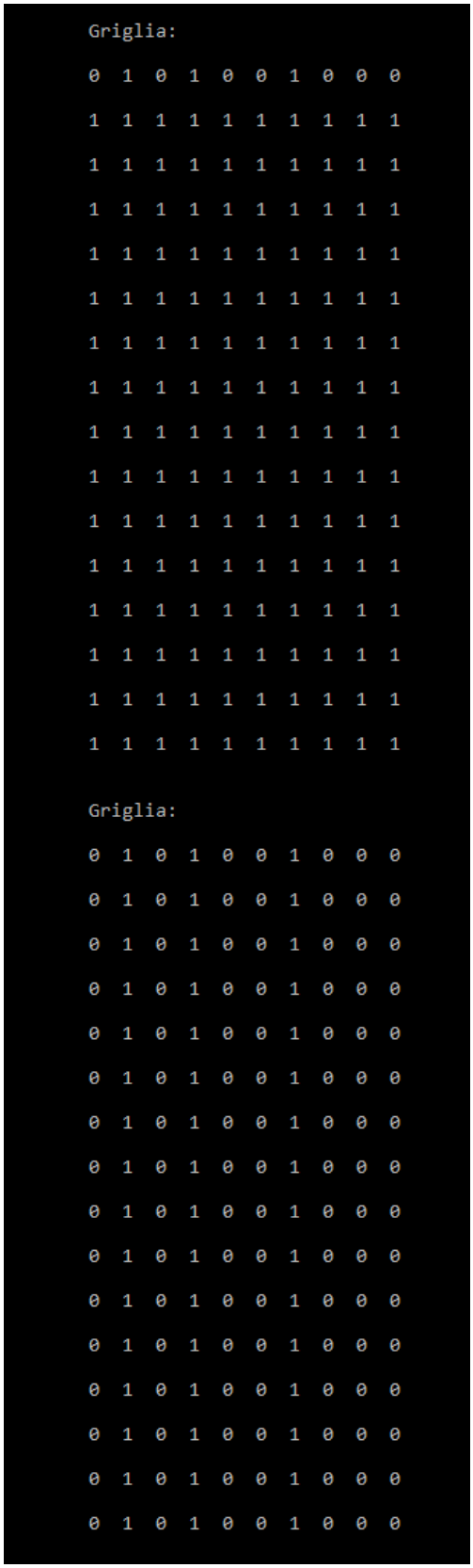
- TestWB
- Gioco
- Terreno

Il codice delle due classi provenienti dal progetto effettivo è stato semplificato e ottimizzato considerato il test da eseguire.

Per la classe Test WB invece viene scritto ed eseguito il seguente codice:

```
class TestWB
    {
        private Gioco _gioco;

        public TestWB()
        {
            this._gioco = new Gioco();
        }

        public void StampaGriglia()
        {
            System.Console.Out.WriteLine("\r\n\t Griglia: \r\n");
            for (int i = 0; i < 16; i++)
            {
                System.Console.Write("\t " + _gioco.Griglia.Campo[i, 0] + " ");
                System.Console.Write(" " + _gioco.Griglia.Campo[i, 1] + " ");
                System.Console.Write(" " + _gioco.Griglia.Campo[i, 2] + " ");
                System.Console.Write(" " + _gioco.Griglia.Campo[i, 3] + " ");
                System.Console.Write(" " + _gioco.Griglia.Campo[i, 4] + " ");
                System.Console.Write(" " + _gioco.Griglia.Campo[i, 5] + " ");
                System.Console.Write(" " + _gioco.Griglia.Campo[i, 6] + " ");
                System.Console.Write(" " + _gioco.Griglia.Campo[i, 7] + " ");
                System.Console.Write(" " + _gioco.Griglia.Campo[i, 8] + " ");
                System.Console.Write(" " + _gioco.Griglia.Campo[i, 9]);


                System.Console.Out.WriteLine("\r\n");
            }
        }

        public void Test()
        {
            for (int i = 15; i >= 0; i--)
            {
                for (int j = 0; j < 10; j++)
                {

                    if(i == 0)
                    {
                        _gioco.Griglia.Campo[i, 1] = 1;
                        _gioco.Griglia.Campo[i, 3] = 1;
                        _gioco.Griglia.Campo[i, 6] = 1;
                    }
                    else
                    {
                        _gioco.Griglia.Campo[i, j] = 1;
                    }
                }
            }

            StampaGriglia();

            _gioco.LineaCompletata();

            StampaGriglia();

        }
    }
```

Ecco il risultato del test effettuato:



Confrontando i risultati, salta subito all'occhio un problema trascurato in precedenza:

```
if (lineaCompletata)
{
    nLineeCompletate++;

    for (int y = i; y > 0; y--)
    {
        for (int j = 0; j < _griglia.Campo.GetLength(1); j++)
        {
            // La riga della griglia sarà uguale a quella sopra
            _griglia.Campo[y, j] = _griglia.Campo[y - 1, j];
        }
    }
}
```

Completata una linea, i blocchi in essa scompaiono, quindi si attiva un effetto a cascata che porta ogni campo della griglia a scendere di una posizione, innescato questo movimento, basta poco a pensare a cosa succederebbe se ad essere completata fosse la seconda linea.

Ebbene, anche se è veramente difficile che accada, se dovesse verificarsi questo evento otterremmo che la prima linea, non potendo prendere i valori da un'altra ipotetica linea superiore, rimarrebbe invariata, sporcando inoltre le altre linee completate.

Per risolvere l'errore si cercherà di trovare un modo per impostare in maniera standard la prima linea a 0, dopo aver effettuato lo scambio della linea.

L’errore è stato risolto in questo modo:

```
if (lineaCompletata)
{
    nLineeCompletate++;

    for (int y = i; y > 0; y--)
    {
        for (int j = 0; j < _griglia.Campo.GetLength(1); j++)
        {
            // La riga della griglia sarà uguale a quella sopra
            _griglia.Campo[y, j] = _griglia.Campo[y - 1, j];
            _griglia.Campo[0, j] = 0;
        }
    }
}
```

Riprovando a effettuare il test e il risultato ottenuto conferma la risoluzione del problema:

```
Griglia:
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  1  0  1  0  0  1  0  0  0
```

5.2 Black-Box Test

Questa seconda tipologia di test si basa sull'utilizzo dell'applicativo, si testa il programma inserendo input e osservando il valore degli output, non tenendo conto del codice. Questo tipo di testing è maggiormente sfruttato nella fase finale dello sviluppo software, per poter evidenziare questi problemi:

- Errori nella GUI
- Errori nella performance
- Errori di terminazione o crash
- Valutazione Casi d'uso

5.2.1 Errori nella GUI

Non sono stati individuati errori nell'interfaccia grafica.

5.2.2 Errori nella performance/ terminazione o crash

Non sono stati individuati errori.

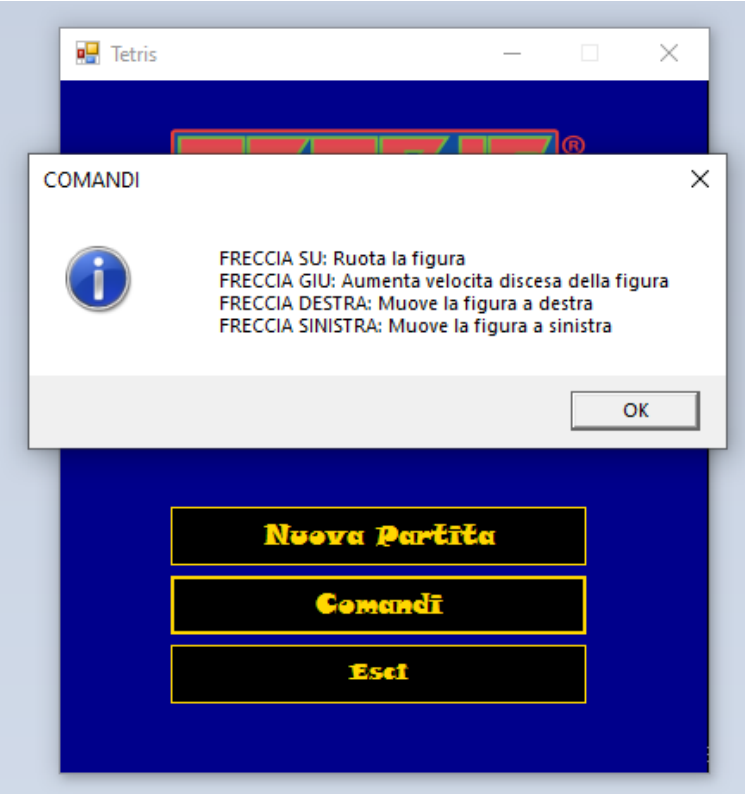
5.2.3 Validazione Casi D'uso

- Validazione Caso ID #1 #2 #3

All'apertura dell'applicazione abbiamo davanti i 3 bottoni, a cui ognuno è collegata la corrispettiva azione.

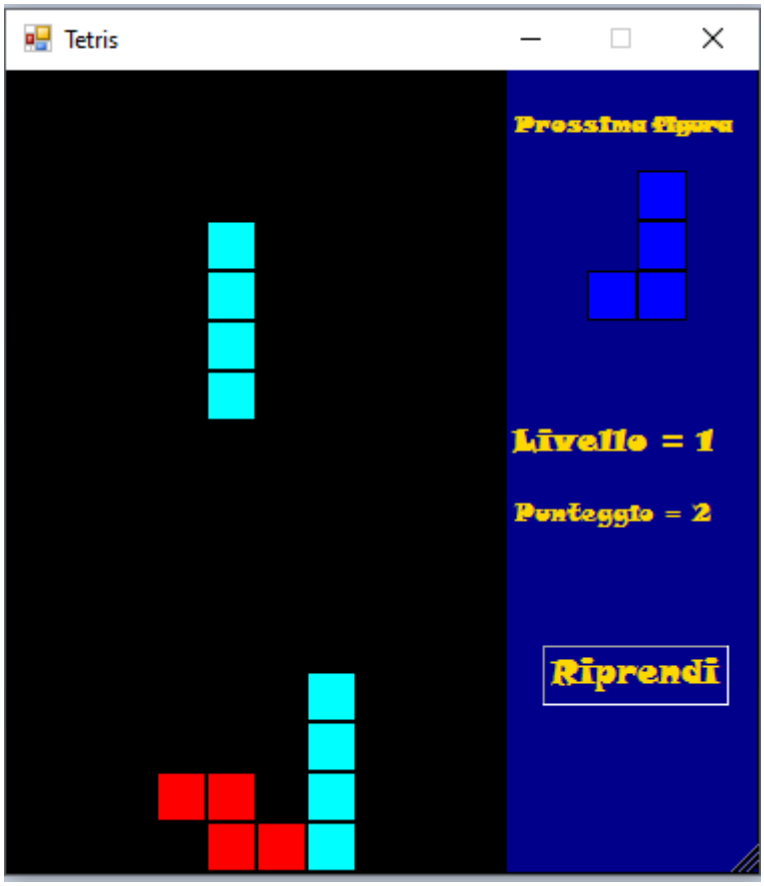
- Validazione Caso ID #4

Una volta cliccato il bottone comandi verranno visualizzati attraverso MessageBox i comandi del gioco.

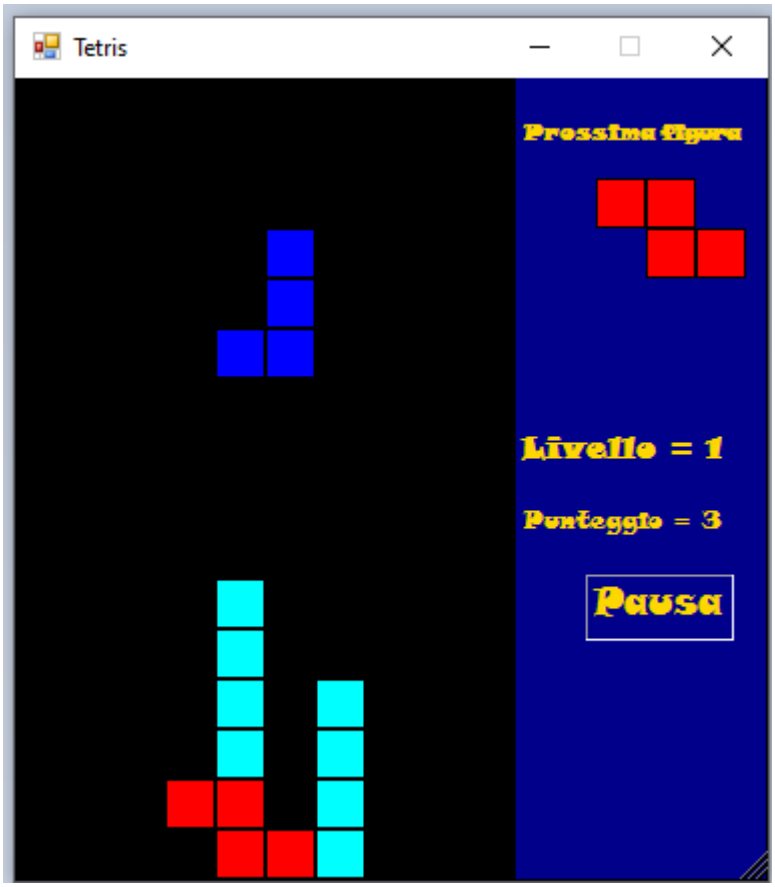


- Validazione Caso ID #9 #10

Dopo aver cliccato il tasto Pausa il gioco si blocca e compare il tasto riprendi.



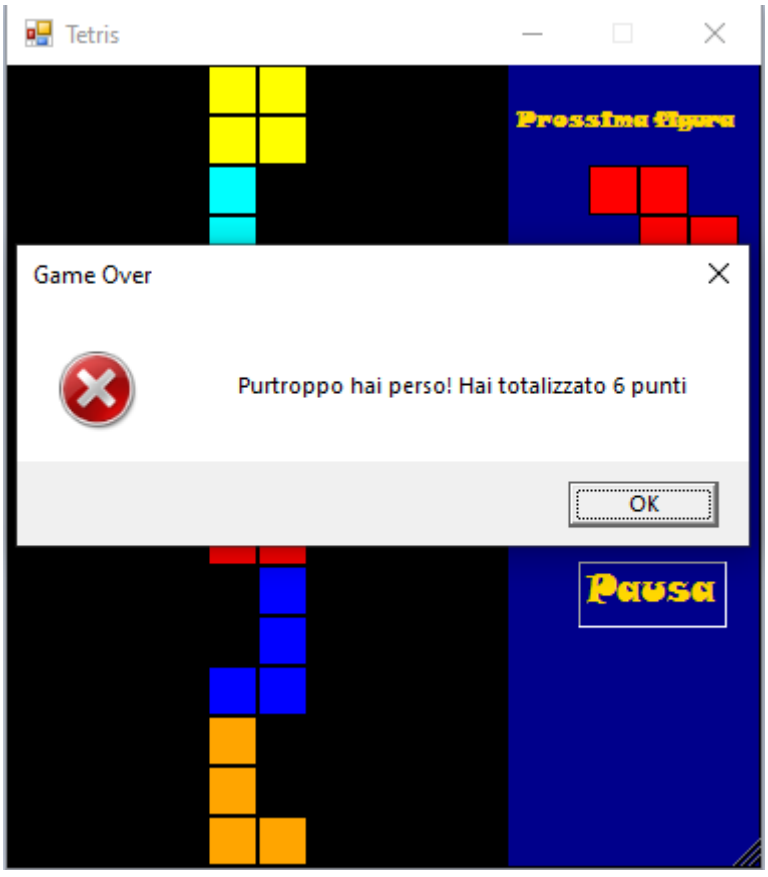
Dopo aver cliccato il tasto Riprendi il gioco riprende e ricompare il tasto pausa.



- Validazione Caso ID #11

Dopo aver generato l'evento GameOver, il giocatore non potrà più continuare la partita.

Dopo aver chiuso il MessageBox o aver cliccato su ok l'applicazione si chiuderà automaticamente.



6. COMPILAZIONE ED ESECUZIONE

L'ambiente di sviluppo utilizzato è Visual Studio 2017 Community.

Sarà possibile compilare il gioco seguendo queste istruzioni:

1. Scaricare dalla repository GitHub i file del progetto
2. Doppio click sul file con estensione .csproj (Microsoft Visual Studio Solution) ed attendere il caricamento del progetto.
3. Cliccare sul menu a tendina Compila → Compila soluzione

Dopo aver terminato la compilazione sarà possibile eseguire l'applicativo raggiungendo il file eseguibile navigando nel percorso:

TetrisPOIS/bin/Release

Ed in seguito doppio click sull'eseguibile *TetrisPOIS.exe*

Il software è stato collaudato nei seguenti hardware:

Sistema Operativo	RAM	CORE	Architettura
Windows 8.1	2 GB	4	32 Bit
Windows 10	4 GB	4	32 Bit
Windows 10	16 GB	4	64 Bit

L'applicazione utilizza molta RAM nelle architetture a 32 bit, raggiunge dei picchi durante il controllo del completamento delle linee.

Nei sistemi collaudati non presenta comunque problemi.

Il consiglio è di giocare con un sistema con almeno 2 GB di RAM per evitare fenomeni di freezing dell'immagine o Frame-Lag.