Programmation Orientée Objets

Documentation - TETRIS

Étudiantes :

Julia Dias
Mateus Bastos Soares

Date :

January 28, 2026

ENSTA Paris
Année académique 2025/2026

1 Project Scope

1.1 General Objective

The Tetris project is an implementation of the classic Tetris game, written in C++. The project uses inheritance hierarchies, polymorphism, encapsulation, composition, and the state pattern design. It supports single-player, multiplayer local, and online multiplayer modes with network synchronization capabilities.

1.2 Minimal functional requirements

Implement the rules of the classic TETRIS game for one player:

- Generation and movement of the 7 standard pieces (I, O, T, L, J, Z, S).
- Control of lateral movements and rotations of the pieces (90°, 180°, 270°).
- Edge detection and collision detection with already positioned pieces.
- Removal of complete lines and scoring calculations based on the number of lines eliminated and the player's level.
- Build a graphical interface for user interaction

1.3 Extension Functional Requirements

1.3.1 Multiplayer Mode

- Allows two players, each on their own computer, to play through the same network.
- Allows two players to play locally on the same computer.

1.4 Tools utilized

- **Language:** C++
- **Graphics:** Raylib graphics library
- **Networking:** Windows Winsock2/ POSIX Unix
- **Threading:** Standard C++ `<thread>` and `<mutex>`

1.5 Implemented structure of the code

The main archives and class structures of the project are organized as follows:

- **Archive: models**
  Core data structures and entities of the game model. In addition to the `tetromino` pieces and movement rules, this module also includes synchronization utilities used by online controllers.

  - `ThreadQueue.h / .cpp`: implementation of a thread-safe queue for asynchronous communication between network and game logic threads.
    This class encapsulates a `std::queue<GamePacket>` protected by `std::mutex` and `std::condition_variable`, ensuring safe concurrent access.
  - `ThreadQueue.cpp`: it implements a thread-safe queue system used to manage asynchronous communication between different parts of the game, mainly in the online multiplayer mode.
  - `Tetrominos.h`: declaration of the `Points` struct and `tetromino` class, with private attributes (blocks, global position, type, rotation state) and public methods for movement, rotation, and querying state.
  - `Tetrominos.cpp`: implementation of the constructor (random piece generation), movement methods, clockwise rotation using SRS tables, and static kick tables `I_KICKS` and `JLSTZ_KICKS`.

- **Archive: controllers**
  Game logic and rules are encapsulated in controller classes that manipulate the model and expose a high-level API to the views and states.
  - `ControllerTetris.h / .cpp`: single-player Tetris controller. It owns the board (`boardCells`), the current and next `tetromino`, scoring and level progression, and provides methods such as `moveLeft()`, `moveRight()`, `rotate()`, `dropDown()`, `hardDown()` and `clearLines()`.
  - `ControllerServer.h / .cpp`: network controller that abstracts sockets (Windows and Unix), encapsulating server/client initialization, non-blocking mode, and sending/receiving a `GamePacket` structure that synchronizes board, score, lines, lives, game-over state and next tetromino for online play.

- **Archive: states**
  The different screens and game modes are implemented as concrete states that inherit from the common `IState` interface.
  - `IState.h`: abstract base class defining the interface for all states, with pure virtual methods `Enter()`, `Update()` and `Exit()`, and a concrete helper method `DrawButton()` reused by all derived states.
  - `StateMenu.h / .cpp`: main menu state. It displays animated background stars and menu buttons (single player, multiplayer, online, ranking). On user interaction, it returns a new concrete state via `std::unique_ptr<IState>` (e.g. `StateTetris`, `StateTetrisMultiplayer`, `StateTetrisOnline`, `StateRanking`).
  - `StateTetris.h / .cpp`: single-player game state. It composes a `ControllerTetris` and a `ViewerTetris*`, forwards user inputs to the controller, and is responsible for drawing the board, score and game-over panel.

  - `StateTetrisMultiplayer.h / .cpp`: local multiplayer state. It owns two `ControllerTetris` instances and two `ViewerTetris*`, handles independent controls for player 1 and player 2, and determines the winner based on their final scores and cleared lines.
  - `StateTetrisOnline.h / .cpp`: online multiplayer state. It composes two controllers (local and remote), two viewers and a `ControllerServer` instance, and manages the online phases through the `OnlinePhase` enum (role selection, waiting, IP input, playing). It periodically synchronizes local and remote state using the `GamePacket` structure.
  - `StateRanking.h / .cpp`: ranking state. It loads the best scores via `ScoreManager`, stores them in a `std::vector`, and draws a "Hall of Fame" panel with position, score and level, using special colors for the top three entries.

- **Archive: views**
  Rendering and user interface elements (e.g. `ViewerTetris`) are responsible for drawing the board and game information using Raylib, based on the data exposed by the controllers. These classes follow an observer-like relationship with the controllers, reading their state but not owning the game logic.

2 Game Screen

The Tetris game menu consists of four interactive rectangles, each representing a distinct game mode. The first three rectangles—Single Player, Local Multiplayer, and Online Multiplayer—initiate their respective gameplay modes upon selection, while the fourth displays the local ranking of recorded high scores.

Inside the game, the user will have four lifes represented by the heart in the down left of the screen. It has also the option of returning to the game menu in the up left. After the game is finished, it will be displayed the users results.

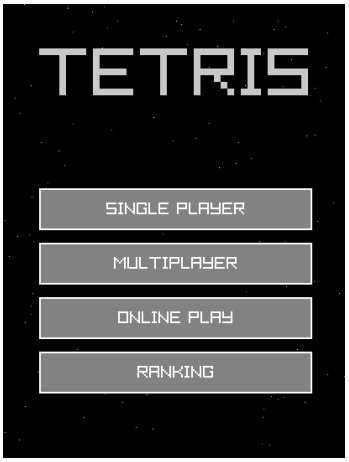

Figure 1: Game Menu

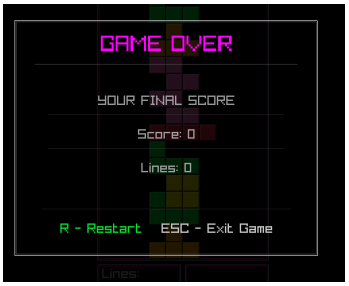

Figure 2: Losing Screen singleplayer gamemode

The singleplayer gamemode leads directly to the Tetris game. It can be played by using standard arrow keys to make the rotation and horizontal movements. The space key provides a hard drop.

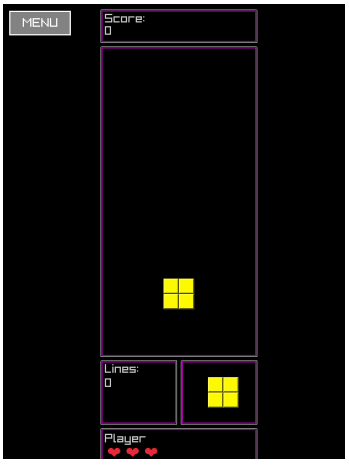

Figure 3: Singleplayer gamemode

The multiplayer gamemode leads directly to the Tetris game. It can be played by using standard arrow keys to make the rotation and horizontal movements. The zero key provides a hard drop. For the other player, "WASD" is responsible for the movements, while left shift provides a hard drop.

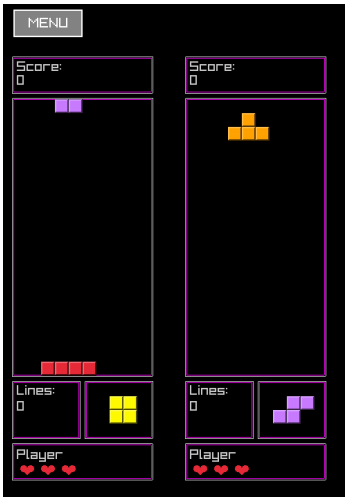

Figure 4: Multiplayer local gamemode

The user will have the option to chose to be the host or join a host. In the host screen, the player will wait for the host to join, while in the client a screen requiring the host ip is shown.

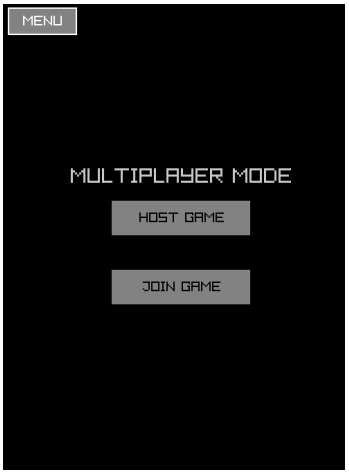

Figure 5: Online Menu

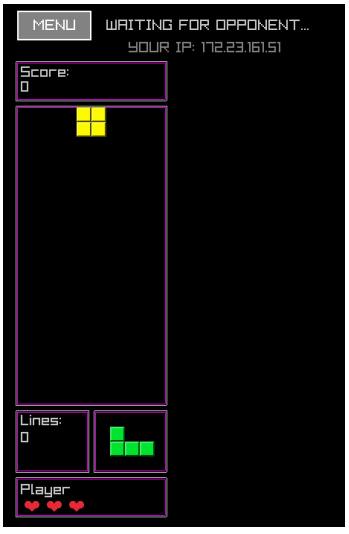

Figure 6: Host Screen

Figure 7: Client Screen

3 Code Architecture and Inheritance Hierarchy

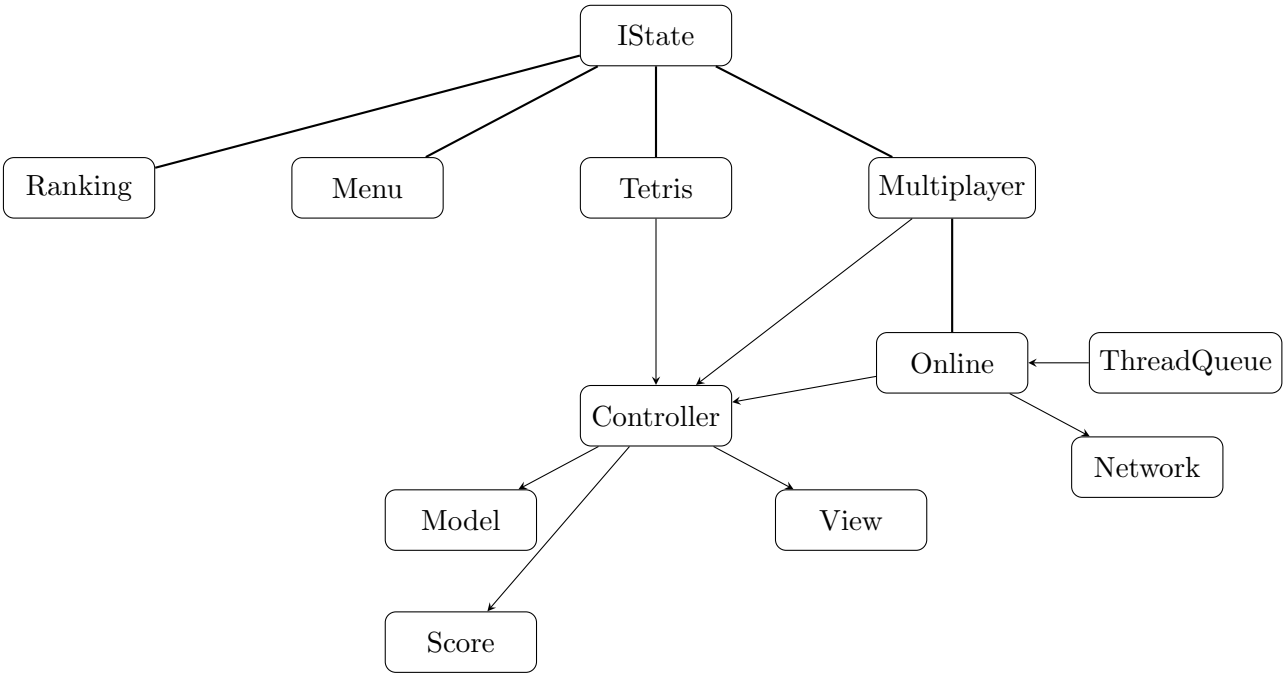

Figure 8: UML Diagram Classes

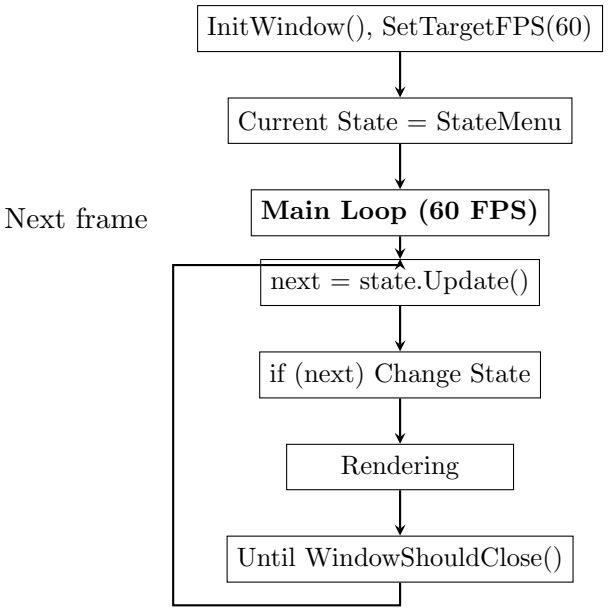

Figure 9: Simplified Main Loop Diagram

3.1 State Pattern Implementation

The project employs the **State Design Pattern** as its foundational architectural pattern, enabling management of different game modes via states.

3.1.1 Base State Interface

```
class IState {
public:
  virtual void Enter() = 0;
  virtual std::unique_ptr<IState> Update() = 0;
  virtual void Exit() = 0;

   { /* ... */ }
};
```

The `IState` abstract base class defines the contract for all game states:

- `Enter()`: Virtual method called when entering a state (initialization)
- `Update()`: Virtual method returning `std::unique_ptr<IState>` for state transitions
- `Exit()`: Virtual method called when leaving a state (cleanup)

3.1.2 Concrete State Implementations

All state classes inherit publicly from `IState` and override the three virtual methods:

State Class	Responsibility
`StateMenu`	Main menu with mode selection
`StateTetris`	Single-player gameplay
`StateRanking`	Hall of fame display
`StateTetrisMultiplayer`	Local 2-player mode
`StateTetrisOnline`	Network-based multiplayer

3.2 2.2 Game Model Hierarchy: Tetromino Class

The `tetromino` class represents individual pieces in the game.

3.2.1 Tetromino Structure

```
class tetromino {
private:
  Points blocks[4];              // Relative block positions
  Points positionToBoard;        // Global board position
  int type;                      // 0–6 (I, J, L, O, S, T, Z)
  int rotationState;             // 0–3 rotation states

  static const std::vector<Points> I_KICKS[4][4];
  static const std::vector<Points> JLSTZ_KICKS[4][4];

public:
  tetromino();
  void moveTetromino(int dx, int dy);
  void rotateTetrominoCW();
  const Points* getBlock() const;
  Points getGlobalPosition() const;
};
```

- **Encapsulation:** All member variables are private; access through public methods ensures data integrity
- **Static arrays:** SRS (Super Rotation System) data stored as static members for memory efficiency
- **Grid representation:** 7 distinct tetromino types (I, J, L, O, S, T, Z) defined via switch statement in constructor

To ensure a good rotation system as well as well-behaved moves next to walls, the project implements the Super Rotation System (SRS) using static matrices.

4 Detailed OOP Features

4.1 Inheritance and Polymorphism

```
class StateTetris : public IState {
public:
  void Enter() override;              // Virtual method override
  std::unique_ptr<IState> Update() override;
  void Exit() override;

private:
  ControllerTetris controllerTetris;
  ViewerTetris* viewer;
  std::thread controllerThread;     // Threading support
};
```

The base class defines pure virtual methods that are overridden by its derived classes. In this example, StateTetris inherits from IState to create a game state that describes the behavior of the TETRIS game in local play. This is defined in other states, such as multiplayer online and local, to describe how the game will behave in each case.

4.2 Encapsulation and Access Control

```
class ControllerTetris {
private:
  int boardCells[20][10];           // Board state - hidden
  int score, linesCleared, level; // Game state - private
  float dropTimer, dropInterval;  // Timing - private
  std::mutex currentTetrominoMutex; // Thread safety

  ScoreManager scoreManager;        // File I/O encapsulated

public:
  // Read access via const getters
  int getScore() const { return score; }
  int getLinesCleared() const { return linesCleared; }
  int getCell(int row, int col) const { return boardCells[row][col]; }

  // Controlled write access via methods
  void moveLeft();  // Validates collision before moving
  void rotate();    // Applies SRS kicks, validates rotation
  void dropDown();  // Updates board and score atomically
};
```

Encapsulation provides invariant protection by ensuring the board state cannot be corrupted externally and can only be modified through validated methods. In this implementation, the classes have their getters and setters according to their need to access private members to keep their data integrity. The ControllerTetris class, for example, encapsulates the boardCells[20][10] array and game state variables within private members, exposing controlled access through public methods such as moveLeft(), rotate(), and dropDown()

5 Networking and Socket Programming

The "ControllerServer" implements a cross-platform socket abstraction that encapsulates differences between Windows (Winsock2) and Unix/Linux (POSIX sockets). On top of that, a producer-consumer logic was implemented to handle the dynamics of sending and receiving packets. In this way, using threads and mutexes makes the networking system more robust.

Regarding the GamePacket structure, a fixed-size state container was implemented, designed for binary transmission over TCP sockets. Containing the complete board (board[20][10]), next tetromino, cleared lines, score, lives, and game-over state, it enables full game state synchronization between clients, enabling the client and server to communicate while the game is happening.

At last, the SetNonBlocking() method configures sockets for non-blocking operation, essential for maintaining 60 FPS in the game loop. On Windows, ioctlsocket(FIONBIO) and on Unix fcntl(O_NONBLOCK) allow recv() to return immediately instead of blocking.

6 Game Logic and Collision Detection

6.1 5.1 Collision Detection

The collision detection system implements complete validation for all tetrominoes simultaneously. "checkCollision(tetromino t, int dy)" verifies boundary limits and board occupancy for vertical movements, while checkCollisionLateral(t, int dx) validates horizontal movements. Detection considers relative offset (blocks[i].x + pos.x) and proposed movement (dy, dx), rejecting invalid actions before state modification, ensuring board consistency across all game modes.

6.2 5.2 Line Clearing and Scoring

The line-clearing algorithm implements multiple-line removal with downward shifting of lines above. For each full line detected, it executes a downward shift (boardCells[k][j] = boardCells[k-1][j]) and decrements the index to reexamine the new line in position. The scoring system follows the instructions proposed in the class documentation with exponential speed progression (dropInterval = 0.7 * pow(0.85, level)).

7 Extensibility and Future Improvements

7.1 Adding New Game Modes

New game modes can be implemented in the TETRIS game. For example, minigames and varying times of tetrominoes moves and rotation would be an example of extensibility related to new game modes.

7.2 Improving Network Architecture

The binary transmission protocol using fixed-size GamePacket structures provides performance but lacks flexibility. Adopting JSON serialization would enable future protocol extensions and easier debugging. Additionally, implementing a proper message queuing system with dedicated worker threads would improve responsiveness under high-latency conditions.

8 Conclusion

The POO Tetris project exemplifies C++ OOP design with:

- **Inheritance:** 5 concrete states inheriting from the IState abstract interface
- **Polymorphism:** Virtual methods enabling dynamic state transitions and behavior
- **Encapsulation:** Private members with controlled public interfaces
- **Composition:** Controllers, viewers, and servers composed within state classes
- **Network Programming:** Cross-platform socket abstraction with non-blocking I/O
- **Design Patterns:** State, Strategy, Observer, and Template Method patterns

The codebase demonstrates some of the subjects learned in the classroom. Although it needs improvements, the result is a functional Tetris game with different features, such as online and local multiplayer, as well as a local game.