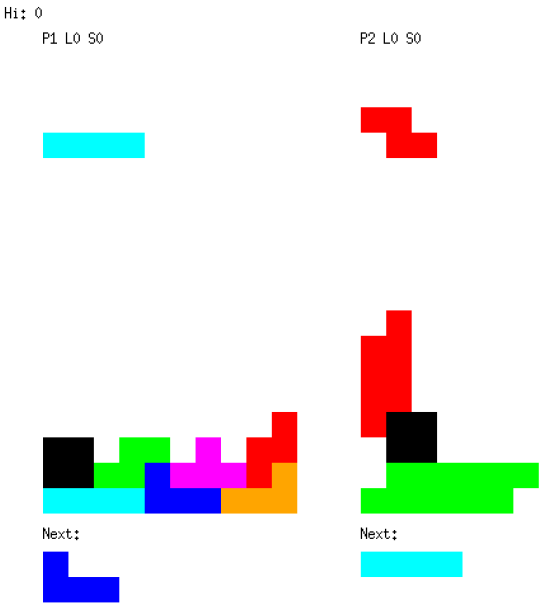
B i q u a d r i s

Full Documentation
CS246 - F2025



Team

Kartheek Chinta	kchinta@uwaterloo.ca
Khush Patel	k497pate@uwaterloo.ca
Harrison Yu	h52yu@uwaterloo.ca

December 1st, 2025

1 Introduction

Biquadris is a two-player competitive variant of the classic puzzle game Tetris. The game features two independent 11×15 boards where players take turns dropping tetris blocks (four-cell pieces) onto their respective boards. Unlike traditional Tetris, Biquadris is turn-based rather than real-time, allowing players unlimited time to strategically position each block.

Victory is achieved when an opponent can no longer place their next block on the board, while scoring rewards both line clears and complete block elimination based on the difficulty level at which blocks were generated.

2 Overview

This implementation is structured around eleven core classes organized in a modular architecture using C++20 modules. The system follows an MVC-style design where Game acts as the controller, Player and Board form the model layer, and Display classes handle the view.

Eleven core classes:
(a) Game - Central controller orchestrating gameplay
(b) Player - Manages individual player state
(c) Board - Manages the 18×11 grid of cells
(d) Block - Abstract base for tetris pieces
(e) Position - Tracks positions for individual block pieces
(e) Cell - Represents individual board positions
(f) Level - Abstract base for block generation strategies
(g) Display - Abstract base for rendering (TextDisplay/GraphicsDisplay)
(h) Score - Calculates and tracks player scores
(i) CommandInterpreter - This is crucial for the game's operation. It parses user input, handles command abbreviations, multiplier prefixes, and executes commands.
(j) HighScoreManager - Manages persistence of high scores to file, which is a required feature of the game.

3 Updated UML

UML Legend

Black Diamond (*—): Composition
Strong ownership. The parent fully manages the lifetime of the child. If the parent is destroyed, the child is destroyed.

White Diamond (o—): Aggregation
Weak ownership. The parent references the child but does not control its lifetime.

Plain Arrow (—>): Association
A simple reference or dependency between two classes.

Triangle Arrow (—| >): Inheritance
Indicates a subclass relationship ("is-a").

Our final UML reflects the actual structure of the implemented Biquadris system. After development, several relationships, ownership structures, and class responsibilities changed from the initial plan. The updated diagram clarifies composition, aggregation, inheritance, and association boundaries so that the architecture faithfully represents the runtime behaviour of the program.

Core Organizational Principles

High cohesion inside modules:

- Board manages grid and placement logic.
- Player owns all gameplay-specific state and effects.
- Game coordinates major components without understanding their internals.
- Display classes contain only rendering responsibilities.

Low coupling between subsystems:

- CommandInterpreter + AliasManager form an isolated parsing layer.
- Level hierarchy encapsulates block generation rules.
- Rendering is fully separate from game logic.
- Block geometry is encapsulated behind the Block/Position abstraction.

The UML communicates these design intentions explicitly through corrected diamonds, direction arrows, and inheritance relationships.

Major UML Components and Their Relationships

Game
Game is the central coordinator and owns all long-lived components.

Composition (black diamonds):

- Game *— Player (two players)
- Game *— HighScoreManager
- Game *– CommandInterpreter
- Game *— TextDisplay
- Game *— GraphicsDisplay

Associations:

- Game → Level4 (penalty logic)
- Game → Block (penalty block creation)

Player
Each Player encapsulates all gameplay state for one side.

Composition:

- Player *— Board
- Player *— Level

Associations:

- Player → Block (current, next, placed)
- Player → Score

Board / Cell / Block

Board subsystem handles spatial logic: block placement, collisions, and row clearing.

Board

- Board *— Cell
- Board → Block (association)
- Board → Position (for placement tests)

Cell

- Cell → Block (owner pointer)

Block

- Block *— Position (four for standard blocks, one for StarBlock)
- All block types inherit Block (I/J/L/O/S/Z/T/Star)

Level Hierarchy
All Level subclasses inherit from Level.

Correct UML changes:

- Level → Block (association due to generateBlock())
- Level3/4 override setRandom(), setSequenceFile()
- Level4 adds penalty methods:
  - notifyRowCleared()
  - notifyBlockPlaced()
  - shouldDropPenaltyBlock()
  - generatePenaltyBlock()

Display System

Display is an abstract base class with pure virtual render().

Relationships:

- Display → Player (reads but does not modify)
- GraphicsDisplay → Block
- GraphicsDisplay *— Xwindow (composition)

- TextDisplay and GraphicsDisplay both inherit Display

Command Interpreter System
CommandInterpreter owns AliasManager and handles all parsing.

- CommandInterpreter *— AliasManager
- No dependency from AliasManager to Game
- No ReplayManager (removed from design)

Position
Included in UML as a small supporting class providing translate(dRow, dCol).

4 Design

Class: Game
Game serves as the central controller implementing the MVC pattern, where it coordinates between the model layer (Player, Board, Block, Level) and view layer (Display classes). Game owns two independent Player instances, allowing simultaneous play at different difficulty levels. The class demonstrates the Command pattern through its CommandInterpreter, which decouples input parsing from command execution. Game also implements turn-based state management with automatic effect clearing (blind, heavy) and game-over detection with automatic restart.

Key Design Decisions:

- **MVC Architecture** - Game acts as controller, never directly manipulating display rendering or low-level game state.
- **Special Action Coordination** - handleSpecialAction() prompts for user input and applies effects to the opponent, while applyHeaviness() resolves the interaction between level-based heaviness and effect-based heaviness through cascading rules.
- **Level 4 Penalty System** - handleLevel4Penalty() implements the 5-block counter mechanism by querying Level4 object and generating StarBlocks.
- **Observer Pattern** - Both TextDisplay and GraphicsDisplay render simultaneously after each command without coupling game logic to presentation.

Class: Player
Player encapsulates all state for a single player, owning its Board, Score, Level, and blocks. This design enables complete independence between players. Player implements the Factory Method pattern through createLevelByNumber() and createBlockByType(), instantiating appropriate subclasses without exposing creation logic. The placedBlocks vector enables bonus scoring by tracking block ownership after placement.

Key Design Decisions:

- **Factory Method Pattern** - createLevelByNumber() and createBlockByType() encapsulate subclass instantiation, making it easy to add new levels or block types.
- **Collision Detection via Undo** - Movement methods (moveLeft/Right/Down, rotateCW/CCW) optimistically perform the action, then undo if Board::canPlace() returns false. This keeps collision logic centralized in Board.

- **Block Ownership Tracking** - placedBlocks vector enables checkClearedBlocks() to detect when entire blocks disappear for bonus points by querying Board::getCellsOwnedBy().

Class: Board
Board manages the 18×11 grid of Cell objects (15 visible rows plus 3 reserve rows for rotation). Board centralizes all collision detection and row clearing logic, providing a clean interface for Player to query placement validity and execute drops. The grid is implemented as a 2D vector for efficient random access.

Key Design Decisions:

- **Centralized Collision Detection** - canPlace() checks all block positions against grid boundaries and occupied cells, keeping validation logic in one place rather than scattered across Block subclasses.
- **Owner Tracking for Bonus Scoring** - Each Cell stores a pointer to its owning Block, enabling getCellsOwnedBy() to scan the grid and detect when entire blocks have been cleared for bonus points.
- **Row Clearing Algorithm** - clearFullRows() iterates bottom-to-top, shifting rows down in-place when full rows are found, returning the total count for scoring.

Class: Block
Block is an abstract base class representing tetris pieces. Each block stores four Position objects (except StarBlock with one), a type character, and spawn level. Block implements movement and rotation operations that work uniformly across all piece types through a bounding box rotation algorithm. The class hierarchy includes seven standard tetris blocks (I, J, L, O, S, Z, T) plus StarBlock for Level 4 penalties.

Key Design Decisions:

- **Template Method Pattern** - Base class provides common movement (moveLeft/Right/Down/Up) and rotation (rotateCW/CCW) operations. Subclasses only define initial cell positions in their constructors.
- **Bounding Box Rotation** - rotate() calculates the block's bounding box, then applies coordinate transformation to rotate cells within that box. This algorithm works for all block shapes without subclass-specific rotation logic.
- **Polymorphic Hierarchy** - Virtual destructor enables polymorphic deletion. Player and Board interact with Block* pointers, allowing uniform treatment of all block types.

Class: Position
Position is a simple data class representing a (row, col) coordinate on the board. Each Block stores a vector of Position objects to track its four cells' locations. Position provides getters and setters for row and column values, enabling Block's movement operations to update coordinates efficiently.

Key Design Decisions:

- **Encapsulation of Coordinates** - Wrapping row/col in a class (rather than using pairs or separate vectors) improves code readability and type safety.
- **Lightweight Value Type** - Position has no dynamic memory, making it safe to store in vectors and pass by value when needed.

Class: Cell
Cell represents a single position on the board, tracking occupancy status, block type character, and a pointer to the owning Block. The owner pointer is essential for implementing bonus scoring when entire blocks are cleared from the board.

Key Design Decisions:

- **Owner Pointer for Block Tracking** - Storing a Block* pointer in each Cell enables Board::getCellsOwnedBy() to scan the grid and identify all cells belonging to a specific block, supporting bonus scoring for fully cleared blocks.
- **Simple State Management** - setOccupied() and clear() provide clean state transitions, avoiding invalid states where a cell is filled but has no owner.

Class: Level
Level is an abstract base class implementing the Strategy pattern for block generation. Each difficulty level (0-4) has distinct generation probabilities and mechanics. The Level hierarchy allows runtime switching between difficulty levels while encapsulating generation logic in each subclass.

Key Design Decisions:

- **Strategy Pattern** - Each Level subclass implements generateBlock() with its own probability distribution, allowing Player to switch levels dynamically by replacing the Level pointer.
- **Level-Specific Mechanics** - Level3 and Level4 override isHeavy() to return true, enabling automatic drops. Level4 adds penalty tracking through notifyBlockPlaced() and notifyRowCleared(), demonstrating how subclasses can extend behavior.
- **Random vs. Sequence Modes** - Levels 3-4 support both random generation and sequence file reading via setRandom() and setSequenceFile(), enabling testing and scripted gameplay.

Subclasses: Level0 (sequence file), Level1 (weighted random: S/Z=1/12, others=1/6), Level2 (equal probability=1/7), Level3 (S/Z=2/9, others=1/9, heavy blocks), Level4 (extends Level3 with 5-block penalty counter and StarBlock generation).

Class: Display
Display is an abstract base class defining the rendering interface. This design follows the Observer pattern, allowing multiple display implementations (TextDisplay and GraphicsDisplay) to render the same game state simultaneously without coupling game logic to presentation details.

Key Design Decisions:

- **Observer Pattern** - Game calls render() on both TextDisplay and GraphicsDisplay after each command, enabling simultaneous text and graphical output without Game knowing implementation details.
- **Polymorphic Interface** - Pure virtual render() method ensures all Display subclasses provide consistent rendering interface, accepting high score and both player pointers.
- **Separation of Concerns** - Display classes handle all rendering logic (ASCII formatting, X11 drawing, blind effect visualization), keeping Game focused on game logic.

Subclasses: TextDisplay (ASCII output to stdout with side-by-side boards), GraphicsDisplay (X11 window with colored blocks using Xwindow class).

Class: Score
Score manages point calculation for a single player using two formulas: $(\text{currentLevel} + \text{numRows})^2$ for cleared rows and $(\text{blockSpawnLevel} + 1)^2$ for fully removed blocks. This incentivizes playing at higher levels and clearing multiple rows simultaneously.

Key Design Decisions:

- **Exponential Scoring** - Squaring the sum rewards clearing multiple rows at once exponentially more than clearing them individually, encouraging strategic play.
- **Level-Based Rewards** - Including currentLevel in the row clearing formula and blockSpawnLevel in the block bonus formula incentivizes playing at higher difficulty levels.

- **Separation of Concerns** - Score encapsulates all scoring logic, keeping Player focused on game mechanics rather than point calculations.

Class: CommandInterpreter
CommandInterpreter parses user input, handling multiplier prefixes (e.g., "3ri"), command abbreviations (e.g., "lef" → "LEFT"), and user-defined aliases through its internal AliasManager. This design decouples input parsing from command execution, allowing Game to focus on game logic.

Key Design Decisions:

- **Command Pattern** - Separates command parsing from execution. CommandInterpreter resolves input to a full command string and multiplier, which Game then executes, enabling easy extension of command sets.
- **Prefix Matching** - findCommandByPrefix() allows users to type minimum distinguishing prefixes (e.g., "lef" for "LEFT"), improving usability while maintaining unambiguous command resolution.
- **Alias System** - AliasManager maintains a map of user-defined shortcuts, enabling customization. ADDALIAS and REMOVEALIAS commands are handled internally by CommandInterpreter.

Helper Class: AliasManager
AliasManager maintains a map of user-defined aliases to full commands. Provides addAlias(), removeAlias(), resolve(), and clear() methods for managing custom shortcuts.

Class: HighScoreManager
HighScoreManager handles persistence of the high score across game sessions by reading from and writing to a file (default: highscores.txt). This ensures the high score persists even after the program exits.

Key Design Decisions:

- **File-Based Persistence** - Constructor reads the high score from file on startup, updateIfHigher() writes new high scores immediately, ensuring data persists across sessions.
- **Graceful Degradation** - If the file cannot be written, updateIfHigher() keeps the in-memory value and issues a warning rather than crashing, maintaining game functionality.
- **Single Responsibility** - HighScoreManager focuses solely on high score persistence, while Score handles per-player scoring calculations.

5 Resilience to Change

1. Extending the Block System
The block system was designed to allow new piece types to be added with minimal impact on the rest of the program. Each block subclass specifies its geometry using a small set of Position objects, while shared movement and rotation logic lives in the base Block class. This separation keeps block construction cohesive and prevents duplication across subclasses. The decision to introduce a dedicated Position class made this structure especially flexible. By encapsulating coordinate manipulation, Position allows block subclasses to define shapes declaratively rather than handling arithmetic directly. As a result, neither the Board nor the Game needs to know anything about the internal layout of a block; they rely only on the Block interface. Because of this modularity, adding a new block involves creating a subclass and defining its cell layout, with no changes required elsewhere. The late addition of our StarBlock demonstrated this: it integrated cleanly without touching Board, Player, or Game logic. This shows that the system can accommodate further block variants or rule extensions while keeping coupling low.

2. Adding New Levels
The level system was built around a single abstraction: every level inherits from the Level base class and

overrides only the behaviour it needs. Game interacts with levels solely through this interface, so it does not depend on the details of how blocks are generated or how randomness is handled. This keeps coupling low and ensures that level rules remain self-contained. Because each level's logic lives entirely within its own subclass, adding or modifying a level requires no changes to unrelated modules. Features such as Level 4's penalty block or sequence-file behaviour were integrated by updating only the corresponding subclass. Introducing a new level follows the same pattern: create a new subclass, override the relevant functions, and the rest of the system continues to work unchanged. This design supports scalable gameplay evolution without architectural disruption.

3. Display and Rendering Flexibility
Rendering was deliberately isolated from the core game logic. Both TextDisplay and GraphicsDisplay implement a shared display interface, which allows the Game class to draw the state without knowing how it is being rendered. This created a clean boundary between gameplay rules and presentation. Because text and graphical output are handled by separate modules, extending or replacing rendering behaviour is straightforward. The coloured text display was added late in the project without modifying any gameplay files, showing that presentation changes can occur independently. Similarly, the graphics module relies on the Xwindow component, so future changes to layout, colours, or rendering strategy only require updates within the display modules. This separation keeps coupling low and ensures the rendering layer can evolve without affecting gameplay mechanics.

4. Player State and Special Effects
The Player module focuses solely on per-player state: their board, current block, level, score, and temporary effects. Game interacts with this state through a small number of well-defined methods. This design allowed us to implement features like blind, heavy, and forced blocks in a cohesive and localized manner. Special effects map directly to internal flags or small state variables inside Player. Game triggers these effects through explicit function calls rather than directly modifying Player fields. Because of this encapsulation, adding new effects would only require updating Player and the command logic. No changes would be needed in Board, Block, or Level. This keeps player-related behaviour flexible and avoids complex branching logic scattered across multiple modules.

5. Command Processing and Alias Support
Command parsing was intentionally decoupled from gameplay through the CommandInterpreter and AliasManager modules. All parsing, shortcut handling, multipliers, and alias resolution occur before commands ever reach the Game class. Game receives normalized instructions, which lets command behaviour evolve independently from gameplay logic. The alias system demonstrates the flexibility of this approach. Users can define custom command names or remap existing ones entirely through AliasManager, with no source code changes. Support for partial commands and numeric prefixes was added late in the project without modifying any gameplay files. Because the interpreter produces standardized outputs, future extensions such as macros or new command types can be integrated with minimal disruption to the rest of the system.