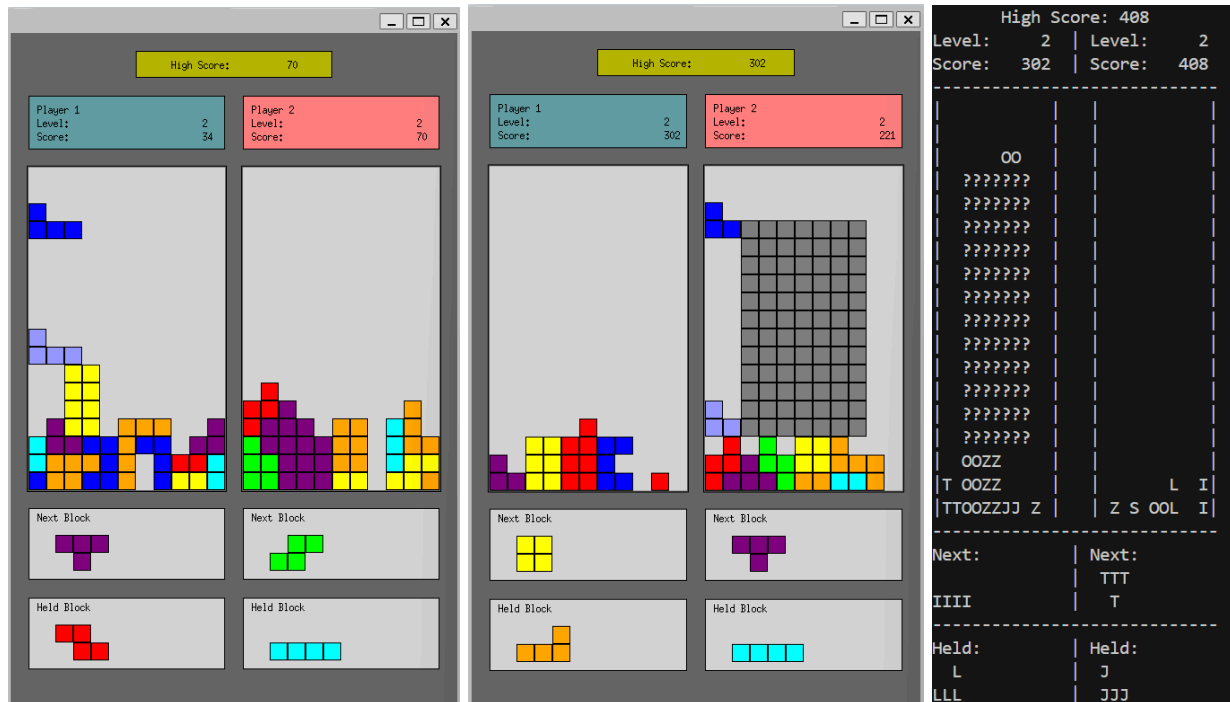


The Game of Biquadris

Filip & Haris



Introduction

Biquadris is a Tetris-inspired video game that differs from the real-time classic by introducing turn-based mechanics and a unique strategic element through special actions. Unlike traditional Tetris, where players must respond rapidly to falling blocks, Biquadris allows players to plan their moves carefully in a non-real-time environment. This shift transforms the fast-paced arcade experience into a more thoughtful and deliberate game.

The program is implemented using **C++** and employs object-oriented principles for modularity and clarity. Key components include the **Block**, **Player**, and **Board** classes, which work in unison to manage gameplay elements. The program leverages **vector** data structures to dynamically update the game board and blocks, ensuring flexibility and efficiency. Additionally, input handling is designed to allow players to interact with the game intuitively, whether through raw keyboard inputs or command sequences.

This report outlines the design, implementation, and testing of the program, highlighting its core functionalities, challenges faced during development, and areas for

potential improvement. Throughout this project, we gained valuable insights into object-oriented design, algorithm development, and effective problem-solving in software engineering.

Overview

Main

The main file handles all user interactions with the game to maintain encapsulation and avoid user-class interactions. When running the game executable, users have access to 5 possible command-line arguments, which can be combined in any order. The following arguments are:

- `"-text "` runs the game in text-only mode, disabling graphics
- `"-seed " xxx` sets the seed value for the random number generator to xxx, ensuring a randomized block order
- `"-scriptfile1 " xxx` and `"-scriptfile2" xxx`, where xxx represents an input file consisting of block letters, allows users to define the order of blocks
- `"-startlevel" xxx`, where $0 \leq xxx \leq 4$, initializes the starting level of the player to xxx. Invalid arguments will result in the closest possible level.
- `"-enablebonus"` activates the raw terminal for keystroke patterns and enables the "hold" feature, which allows players to reserve a block for later use.

In `main.cc`, the game board is initialized by creating two players, each with their respective levels and game state according to user set-up. The program then handles three input modes: standard input, command sequence from a file and raw keystrokes to process

- Movement commands (left, right, down, clock, counter-clock, drop)
- Special action commands (blind, heavy, force)
- Game commands (levelup, leveldown, restart, hold)
- Block Manipulation (I, J, L, O, S, T, Z)

Between commands, the `Board` class itself is alternating player turns.

Board

Next, we move down the UML hierarchy to the `Board` class. The `board.cc` file implements a two-player game board that tracks turns, high score, and bonus features. It defines the board layout using global constants for width and height, and overloads the `operator<<` to print the board to standard output. In text-mode, `board.cc` displays both player boards side by side, showing each player's score, level, next block, and an optional hold block. To format the output, we utilized the `<iomanip>` library for customized formatting.

Beyond printing, the `Board` class manages player turns via `switchTurn()` and `whosTurn()` methods. The `board.cc` file provides an interface for clients, in this case `main.cc`, to move blocks with methods like `shift()`, `clockwise()`, and `counterclockwise()`, without requiring access to the `Board` or `Player` classes. This improves encapsulation by preventing direct interaction with lower-level classes. Block manipulations such as `drop()`, `setBlock()` and `holdblock()` are also streamlined through the `Board` interface, which internally calls these methods on the current player.

The `Board` class also facilitates special action commands and maintains the game state. It supports starting a new game via `start()` and resetting the board (excluding high score) with `restart()`. Level management is handled by the `levelup()` and `leveldown()` functions, which interact with the `Level` class. Additionally, we incorporated the Observer design pattern to notify display components and other observers of any state changes that may occur, such as block drops, special actions, game restarts, or board updates.

```

High Score: 408
Level: 2 | Level: 2
Score: 302 | Score: 408
-----
      OO
  ??????
  ??????
  ??????
  ??????
  ??????
  ??????
  ??????
  ??????
  ??????
  ??????
  ??????
  OZZ
T OZZ      L I
TT00ZZJJ Z | Z S 00L I|
-----
Next:      Next:
IIII       TTT
          T
-----
Held:      Held:
L          J
LLL       JJJ

```

Player

The `Player` class manages the individual player's state and gameplay mechanics. It handles the game area, next block area, and hold area, which were all implemented as 2D `vectors` of characters (`vector<vector<char>>`). The class also tracks the player's score, level, special effects and a list of played blocks.

The `Player` class facilitates block operations such as `shift()` and `rotate()`, which are invoked by the `Board` class. Depending on the player's level, certain movements can trigger a downward shift (in levels 3 and 4, blocks are considered "heavy"). The `drop()` method also manages block drop mechanics and updates the score based on two scenarios

(1) When rows are cleared, the additional points equal

$$(\text{numRowsCleared} + \text{curLevel})^2$$

(2) When an entire block previously placed is cleared, additional points equal

$$(\text{levelAtCreation} + 1)^2$$

`Player` handles the special effects

- `blind()/undoblind()`: Toggles board visibility
- `heavy()`: Influences `shift()` behavior

-
- `force()`: Overrides and forces next block selection

Throughout the `Player` class, the helper function `change()` is frequently used to update the game area and other areas by toggling between block characters and empty spaces.

Block

The `Block` class is responsible for initializing blocks based on a character parameter that defines their type. It stores both the x and y coordinates in a vector of integers, updating these coordinates whenever the block is moved or rotated. The current rotation state is tracked via `positionID`. The block's creation level is stored and can be accessed by clients using `level()`, which is important for determining movement behavior based on the player's level. Key operations in the `block.cc` include:

- `shift()`: Moves blocks left/right/down with collision detection
- `rotate()`: Changes orientation using `rotatedCoords()`, a lookup table for all block rotation states
- `clear()`: Handles block clearing and gravity effects
- `hasMoved()`: tracks whether block has moved from its initial position

The `Block` class interacts with the `Board/Player` class through the `vector<vector<char>>` area parameter, which is passed to many of its methods.

Level

The `Level` class manages block generation across 4 difficulty levels (0-4):

- Level 0: Block sequence is loaded from a user file (`-scriptfile1/-scriptfile2`)
- Level 1: Random block generation, with S and Z blocks having a $\frac{1}{12}$ chance and the other blocks have a $\frac{1}{6}$ chance
- Level 2: Random block generation with uniform distribution
- Level 3: Random generation where S and Z blocks have a $\frac{2}{9}$ chance, and the other blocks have a $\frac{1}{9}$ chance. In this level, the `Player` class handles "heavy" block movements by retrieving the level via `getLevel()`
- Level 4: Identical to level 3, but every fifth block placed without clearing at least one row drops a star block in the centre of the board. This behavior is managed through interactions between the `Player` and `Block` classes, not directly by the `level.cc`

The `Level` class also overloads the `operator+=` and `operator-=` functions to allow clients to increment or decrement the level. Additionally, if a seed value is provided by the user via `main.cc`, it is passed to the `Level` constructor to initialize the random number generator.

Command

The `Command` class is responsible for handling command processing and validation in `main.cc`. It operates independently of the game itself, without connections to other classes. The class has a lookup table of valid commands, called `commands`, which is stored as a `vector<string>`. It processes both text input and keyboard input, mapping keyboard shortcuts to commands such as the spacebar for “drop” and arrow keys for movement. Additionally, the `Command` class tracks user provided multipliers which may precede a command using the `countCommand()` method.

Design

In developing Biquadris, we encountered several design challenges, ranging from efficiently managing game state updates to implementing real-time rendering with optimized performance.

1. **State Management and Object Representation:** To efficiently handle the dynamic nature of the game state, the program utilizes the vector class. The `vector` class is perfect for dynamically manipulating data because it allows for flexible resizing, enabling the program to easily add or remove elements as the game state changes. This includes maintaining the current game area, next block area, and hold area, which are all stored in `vector<vector<char>>` types. These structures allow easy manipulation and comparison of the game states at different points in the game.
2. **Observer Pattern for Real-Time Updates:** The Observer design pattern was implemented to handle the communication between the game state and the rendering systems. The game state, represented by the `Player` class, acts as the subject, and the rendering system, represented by the `GraphicalObserver` class acts as the observer. Whenever there is an update to the game state (the movement of a block), the observer is notified. This ensures that the game’s UI always reflects the most recent state without unnecessary checks or redundant updates.
3. **Efficient Rendering Using Difference Calculation:** A significant design challenge was ensuring efficient rendering, especially when dealing with updates to large game areas. As seen during Assignment 4, render graphics can be painfully slow. Instead of redrawing the entire game area on every update, a difference calculation method was implemented. The `differences()` function compares the current and previous game areas, returning only the coordinates that have changed as a `Coords` struct which is essentially a list of X and Y coordinates. These differences are then sent to the rendering function, which updates only the modified sections of the screen. This optimization minimizes the computational cost and enhances the game’s performance.

-
4. **Error Handling and Robustness:** Since the game involves real-time interactions, robust error handling was critical. The `Command` class was implemented to handle all the conversions from input to valid commands ensuring that only valid commands were accepted. This also allowed for more complicated processing of commands such as multiplied commands and incomplete commands being handled gracefully. Additionally the renaming of commands was implemented using a `Map` data structure. This ensures that each new name can only be used once to limit ambiguity.

Resilience to Change

To ensure resilience to change, the design of the program was built with flexibility and extensibility in mind.

Firstly, the Observer design pattern is implemented which ensures that the game state and rendering are handled completely separately. This means that changes to the layout or visual representation of the game can be made independently of the game state. Similarly, small changes to game logic such as the addition of a new block or special effect would have very little impact on the existing system. Since the game state and rendering logic are decoupled, modifying or extending the game's functionality can be done without altering the observer classes. This makes it easier to introduce new game elements or make adjustments to the existing ones with minimal risk of breaking the program.

Additionally, the use of flexible data structures and global constants ensures that the game can be resized to any reasonable dimensions. Adjusting the dimension of the board is as simple as changing a single number. The graphics also render correctly with different sized boards. Moreover, constants are used to define the sizes of the various elements in the GUI, which means adjusting their sizes would also be trivial.

Another note to point out is that most of the functions implemented are extremely reusable. Very few functions have side effects that prevent other functions from using them. The functions don't have to rely on the internal implementation details of each other which maximizes cohesion and minimizes coupling.

Finally, the game's internal structure does not depend on blocks being made up of four squares, which means that larger or smaller blocks made up of a different number of squares can be introduced seamlessly, maintaining consistent gameplay mechanics. The separation of block properties from the game logic further ensures that we are able to make changes in block behaviour if we desire without disrupting the flow of the game, making it adaptable to various game modes or additional features.

Overall, the combination of modularity, dynamic data structures, and object-oriented principles ensures that the program can easily accommodate future changes or new requirements with minimal disruption.

Answers to Questions

Question 1: How could you design your system (or modify your existing design) to allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen? Could the generation of such blocks be easily confined to more advanced levels?

Having blocks disappear after 10 more blocks have appeared would be extremely simple. As we track the blocks that have been played so far, and the number of turns it has been since a row has been cleared, we could combine these pieces of information to clear blocks that have not been cleared in the last 10 turns.

Question 2: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

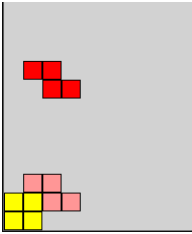
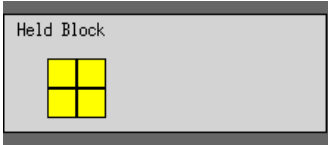
Introducing additional levels is as simple as writing another else-if condition and updating the `MAX_LEVEL` constant. We opted for this approach to keep the program as simple as possible. The `Level` class would need to be recompiled along with the `Player` class and `Board` class as they depend on each other.

Question 3: How could you design your system to accommodate the addition of new command names, or changes to existing command names, with minimal changes to source and minimal recompilation? (We acknowledge, of course, that adding a new command probably means adding a new feature, which can mean adding a non-trivial amount of code.) How difficult would it be to adapt your system to support a command whereby a user could rename existing commands (e.g. something like rename counterclockwise cc)? How might you support a “macro” language, which would allow you to give a name to a sequence of commands? Keep in mind the effect that all of these features would have on the available shortcuts for existing command names

Adding additional command names is as simple as adding the name to the vector of existing commands and updating the main class to handle the new command. Changing existing command names is similar in that the vector of possible commands would need to be updated as well as the main class to handle the new name. The program incorporates a `rename` function that allows the player to rename commands for easier input. To incorporate a “macro” language, a new function would need to be implemented in the command class but the reusability of the other functions means that they would likely not need to be changed or be changed very little.

Extra Credit Features

Five extra credit features were implemented.

1. **Raw Key Inputs:** This allows for the arrow keys to move blocks and all the functions to be activated with a single key press. Special actions must still be typed in as well as file names but the processing of raw key inputs makes the game feel much smoother and intuitive for the player.
2. **Phantom Blocks:** This feature shows the player where their block will fall on the GUI. It can sometimes be hard to align the blocks in the correct position but the phantom blocks allow the player to see where their block will fall. These phantom blocks appear in a lighter colour and do not interact with the other blocks as they are purely visual.
3. **Holding Blocks:** This feature allows the player to hold a block if they do not want to use it yet. Blocks can only be held if they are in the top row though as to prevent players from swapping their blocks in levels 3 and 4 to avoid the heaviness of their blocks.
4. **Renaming Commands:** This feature allows the player to rename commands to allow for easier inputs. This is implemented in the command class which is independent of the main function which means it is extremely versatile.
5. **No Delete Statements:** The program never explicitly handles its own memory via delete statements. There are just two occasions for which we use shared and unique pointers. One for declaring the Observer object and another to track the blocks that have been played for scoring purposes.

Final Questions

Question 1: What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

This project taught us several valuable lessons about developing software in teams. First, we learned the importance of clear communication and coordination between team members. Regular check-ins and discussions helped ensure that we were aligned on the project's goals, and it made resolving issues much faster. We also realized the significance of dividing tasks into manageable portions. By breaking down complex problems and assigning responsibilities based on each team member's strengths, we were able to make steady progress and maintain high productivity.

Additionally, we gained insights into version control practices, especially the need to regularly commit changes and resolve conflicts promptly to avoid disruptions in the

development process. Working on this project also emphasized the importance of testing and debugging collaboratively, as team members could spot potential issues from different perspectives.

Question 2: What would you have done differently if you had the chance to start over?

If we could start over, we would implement the blind functionality a bit differently. The way it is right now, is that it covers the actual board with question marks. This makes the graphics renderer print each question mark individually which can be slow. Instead, if the blind function left the player's board alone and sent a different version of the board with question marks to be printed, and some information to the observer to cover part of the screen, the graphics would render faster. This was not done, because originally, the observer just printed the board that the player had. So if the player's board did not have question marks on it, it would render the board as normal but the text display would be correct with the question marks. As a result we decided that the players' boards must actually be covered with question marks for the graphics to render. After optimizing the graphics to only display what changed, a more efficient blind function could be implemented but this would require some changes to the player class which could cause other issues so it was avoided.

Conclusion

In conclusion, this project provided us with invaluable experience in both collaborative and individual software development. We gained a deeper understanding of the importance of effective communication, task division, and maintaining a modular, flexible approach to code design. Our hands-on work with version control, debugging, and testing highlighted the need for ongoing attention to detail and teamwork. The project reinforced the necessity of thorough planning, clear documentation, and iterative improvements. The lessons learned will undoubtedly be applicable to future projects, allowing us to approach software development with greater efficiency, collaboration, and adaptability.