Biquadris Final Report

Rithvik Kolanu
University of Waterloo

Daniel Kurien
University of Waterloo

Sami Yousef
University of Waterloo

CS 246 - Object Oriented Programming

Date: Aug 1, 2023

1 Introduction

During the past three weeks, we immersed ourselves in the development of Biquadris. Throughout development, we diligently applied various concepts discussed in class, resulting in recreation of this popular game. Our approach to building the game was multifaceted; we crafted the block, block factory, board, player, and game classes to handle different layers of logic. The result was a seamless and enjoyable gaming experience. Additionally, we incorporated an array of bonus features, such as sound effects, three distinct color schemes, intuitive keyboard and UI input, smart pointers, and visual guideline to show where blocks will drop. All these elements combined to create an enriched and dynamic gaming environment.

2 Overview/Design

2.1 Game Design

The game was implemented using multiple threads, with one thread taking input from the command line, one thread taking input SDL2 and rendering the UI, and a main thread running the game loop. The two auxiliary threads add commands to the player queues, which are made thread-safe using a standard library mutex and conditional variable.

2.1.1 Creation of Blocks

In Biquadris, the block factory design pattern efficiently generates blocks based on the abstract block class. These blocks encompass 8 inheritor classes, each representing a unique block type with defined tetromino square offsets, and the star block class that is included in level 4. Additionally, 5 level block factory classes inherit from the block factory, enabling customized block production for each level by specifying spawn probabilities and behaviors like ”heavy”. This approach ensures easy integration with

1the player.board class and simplifies random block generation due to the pattern's inherent readability and centralized block creation logic

2.1.2 Handling Block Transformations

Each block in the game is defined by an (x, y) pair representing its starting position. The individual cells within each block are represented as offsets relative to this starting position. This representation offers significant advantages, particularly in facilitating easy transformations. For instance, a simple translation can be achieved by modifying the start position while keeping the offsets unchanged. Similarly, rotations become more straightforward, as each square within the tetromino is dependent on the position of the starting block. Consequently, rotations only need to handle the new position of the main block.

The adoption of this offset-based approach for handling rotations brings several crucial benefits. One key advantage is that the responsibility for rotations lies with the block itself, rather than being computed within the board class. This not only reduces the likelihood of introducing bugs but also allows for the implementation of collision logic with improved time efficiency. By empowering the blocks to manage their own rotations, the overall design of the program can achieve greater clarity and maintainability.

2.1.3 The Players and Board logic

In the context of the two-player game, Biquadris, each instance of the Game class will be associated with two players. Each player will possess essential attributes, including a score field, a level field, and a dedicated BlockFactory corresponding to their level. Additionally, a player's applied effects will be stored as an integer. These effects are mapped to prime numbers, allowing us to determine whether a player has a particular effect applied simply by checking if their integer is divisible by the assigned number of that effect.

The board in Biquadris is constructed from multiple blocks, which are stored within a vector. The process of adding a block to the board is as straightforward as pushing the block onto the vector. To handle the deletion of specific cells, we implement the observer pattern, where the board acts as the publisher and the blocks act as subscribers. At the conclusion of each turn, we examine if any row is completely filled, and subsequently, we notify all the blocks of the specific row that has become full.

The blocks then take action by removing cells that reside on the same horizontal level as the filled row. Moreover, the cells above the filled row are shifted downwards by one cell. We also implemented a cleanup function, which removes any empty blocks and computes the new score based on the block levels. This procedure is applied to every filled row, ensuring that the stability of the board is maintained throughout the game.

2.1.4 The Game Object

At the outset of each game, the initialization process involves creating a Game object, allowing the selection of desired options, such as text-only mode, seed, script files, and start levels. Subsequently, the runMainLoop() method is invoked to handle the parsing of commands from the standard input and to execute the corresponding functions. This design exhibits notable advantages in terms of extensibility, notably enabling the seamless addition of multiple game instances, each operating on its dedicated thread. Serving as the focal point of the program, the Game object plays a central role in managing essential components, including the graphical window, player entities, and the tracking of the current turn.

2.1.5 The Window Class and OpenGL

In previous assignment experiences, it was observed that drawing to the X11 window using the Fill-Rectangle() method from the provided XWindow class resulted in suboptimal performance, which raised concerns about the overall efficiency of the system. It was evident that the sluggishness primarily stemmed from the round trip time required for each individual rectangle change, as it necessitated sending separate requests to the X11 server.

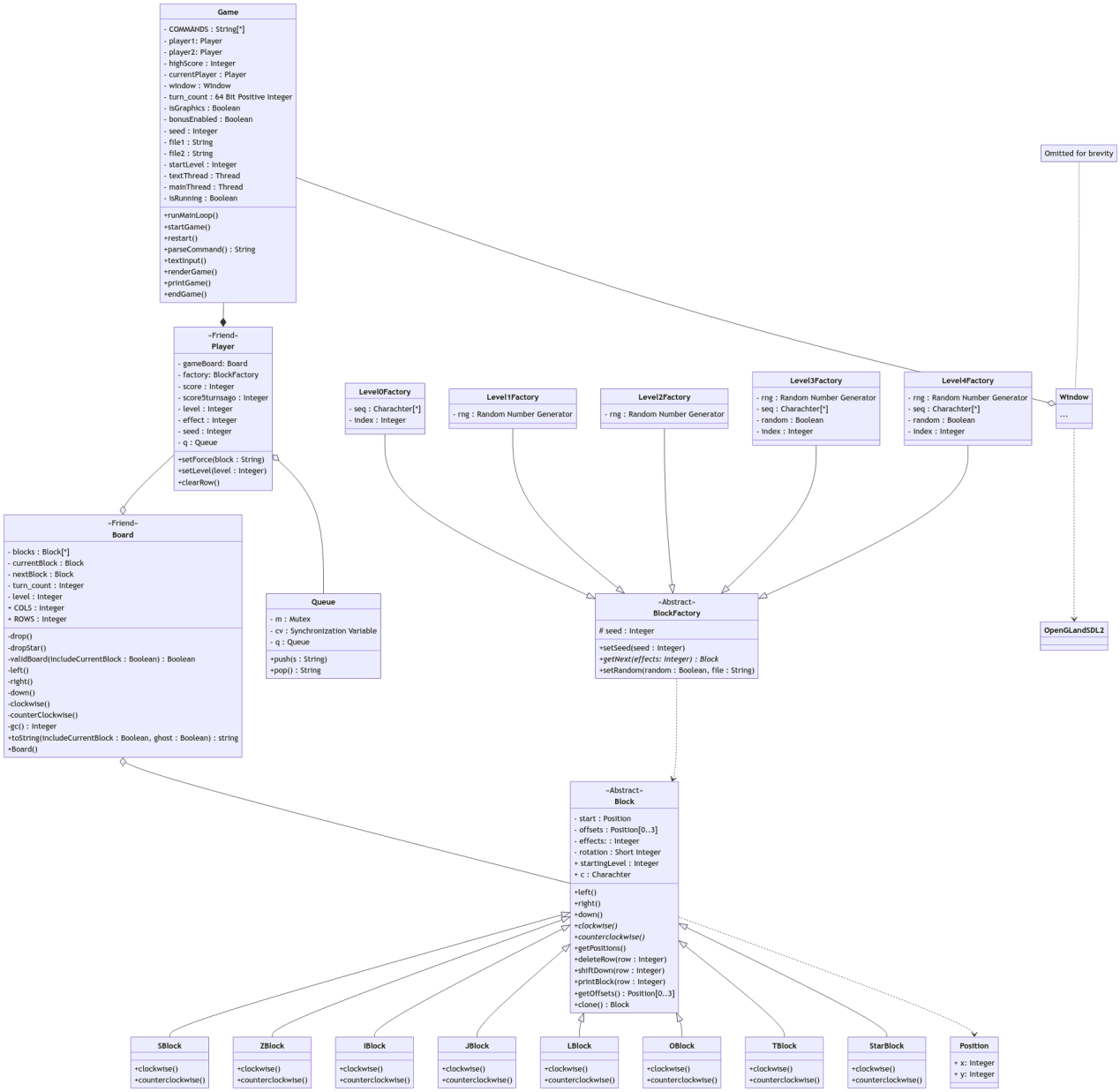
To address this performance bottleneck, a more efficient approach will be adopted. Specifically, we decided to abandon Xlib entirely and opt for a more modern approach. After some research, we decided SDL2 and OpenGL was the best choice for both cross-platform development and speed. SDL2 requires an event loop to handle input and mouse events, hence a separate thread was required. After testing, we noticed OpenGL and SDL2 were much faster than Xlib, both running locally and on the student development server using X-Forwarding.

After careful research, we realized OpenGL and SDL2 achieved this performance by utilizing STL memory allocation pool. This results in valgrind reporting un-deallocated, yet reachable memory. However, disabling this memory pool will result in suboptimal performance, and is difficult to do successfully.

One challenge we ran into with OpenGL and SDL2 was rendering text onto the window. We discovered we could either create the bezier curves manually, or use a library to convert fonts to OpenGL textures. We decided to use the latter approach since it is much easier, still necessitated complex use of OpenGL documentation.

For more information, see the window.h class.

3 UML


Game
- COMMANDS : String[*]
- player1: Player
- player2: Player
- highScore : Integer
- currentPlayer : Player
- window : Window
- turn_count : 64 Bit Positive Integer
- isGraphics : Boolean
- bonusEnabled : Boolean
- seed : Integer
- file1 : String
- file2 : String
- startLevel : Integer
- textThread : Thread
- mainThread : Thread
- isRunning : Boolean
+runMainLoop()
+startGame()
+restart()
+parseCommand() : String
+textInput()
+renderGame()
+printGame()
+endGame()
Omitted for brevity
«Friend»
Player
- gameBoard: Board
- factory: BlockFactory
- score : Integer
- scoreSturnsago : Integer
- level : Integer
- effect : Integer
- seed : Integer
- q : Queue
+setForce(block : String)
+setLevel(level : Integer)
+clearRow()
Level0Factory
- seq : Charachter[*]
- index : Integer
Level1Factory
- rng : Random Number Generator
Level2Factory
- rng : Random Number Generator
Level3Factory
- rng : Random Number Generator
- seq : Charachter[*]
- random : Boolean
- index : Integer
Level4Factory
- rng : Random Number Generator
- seq : Charachter[*]
- random : Boolean
- index : Integer
Window
...
«Friend»
Board
- blocks : Block[*]
- currentBlock : Block
- nextBlock : Block
- turn_count : Integer
- level : Integer
+ COLS : Integer
+ ROWS : Integer
-drop()
-dropStar()
-validBoard(includeCurrentBlock : Boolean) : Boolean
-left()
-right()
-down()
-clockwise()
-counterClockwise()
-gc() : Integer
+toString(IncludeCurrentBlock : Boolean, ghost : Boolean) : string
+Board()
Queue
- m : Mutex
- cv : Synchronization Variable
- q : Queue
+push(s : String)
+pop() : String
«Abstract»
BlockFactory
seed : Integer
+setSeed(seed : Integer)
+getNext(effects: Integer) : Block
+setRandom(random : Boolean, file : String)
OpenGLandSDL2
«Abstract»
Block
- start : Position
- offsets : Position[0..3]
- effects: : Integer
- rotation : Short Integer
+ startingLevel : Integer
+ c : Charachter
+left()
+right()
+down()
+clockwise()
+counterclockwise()
+getPositions()
+deleteRow(row : Integer)
+shiftDown(row : Integer)
+printBlock(row : Integer)
+getOffsets() : Position[0..3]
+clone() : Block
SBlock
+clockwise()
+counterclockwise()
ZBlock
+clockwise()
+counterclockwise()
IBlock
+clockwise()
+counterclockwise()
JBlock
+clockwise()
+counterclockwise()
LBlock
+clockwise()
+counterclockwise()
OBlock
+clockwise()
+counterclockwise()
TBlock
+clockwise()
+counterclockwise()
StarBlock
+clockwise()
+counterclockwise()
Position
+ x: Integer
+ y: Integer

4 Resilience to Change

When approaching this project, we planned on dividing the logic into as many modular pieces as possible, which resulted in a very flexible and adaptable model. The main components, such as blocks and factories, are abstract classes that hold pure virtual functions, which ensures that new derived classes can be implemented quickly with little friction. This actually came in handy during the project, when the star (*) block was implemented much farther down the development cycle than the rest of the blocks, and was quickly implemented by inheriting from the same abstract class. By focusing on modularity throughout the project, features such as effects and bonus additions were also easily implemented by adding new methods at any time, which could be called in other classes. For example, when implementing an effect, a specific method could be added to the board class and called in the game class, ensuring that the board's private elements were affected at the right instances. Having methods to interact with private fields also enforced the structural integrity of our project, making certain that the addition of new features didn't create devastating unintended consequences. Lastly, the inclusion of new levels is also supported, as each level has its own block factory designed to accommodate unique spawn probabilities. In this case, a new level would just require a new inheriting class, as well as additional methods to handle additional game rules.

5 Responses to Project Specification Questions

Question: 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? **Answer:** To address the requirement of allowing some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen, we have devised a system based on a turn-based counter. This counter will increment after both players have placed their blocks. The counter is represented as a global unsigned long, providing ample capacity to store up to 4 billion turns ($2^{32} - 1$). In our modified design, each block will possess a lifespan variable. When a block is generated, its lifespan will be set to the current turn number plus 10. Permanent blocks have their high lifespan set to a large value ($2^{32} - 1$) to indicate their permanence. As the game progresses, if the global turn counter surpasses or equals a block's lifespan, the respective block will be removed from the block stack within the board class and subsequently deleted from the heap. The remaining blocks above it on the board will be shifted downwards to fill the vacant spaces.

It is worth noting that if the game is designed to be played in real-time, the tick counter would function as a timer. In such scenarios, the use of an unsigned long might not suffice, and alternate timekeeping methods would need to be explored and integrated into the system.

Question: How could you design your program to accommodate the possibility of introducing additional

levels into the system, with minimum recompilation?

Answer: To introduce additional levels into the system with minimal recompilation, we could create an abstract board class, with level classes inheriting from the board class. For the current specifications of the game, there would be 5 level classes, with each level class specifying a distinct level factory, probabilities any additional game logic. To add an additional level, we could just create a new level class inheriting from the abstract board class, which would only require one extra file and a recompilation of one file, game.cc.

Question: How could you design your program to allow for multiple effects to be applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination?

Answer: This problem can be solved by assigning each effect to a unique prime number value. In order to assign effects to the other player, the prime numbers would be multiplied together to create a key value that can be sent through the game class. Each forced letter would have its own unique prime number. For example, if blind = 3, heavy = 2 and force J = 5, player one could send both those effects by sending 2*3 = 6 to player two. Then, player two would check what effects were applied by calculating the modulus of the key value by each unique prime number value. For example, 6 % 2 == 0 and 6 % 3 == 0, so blind and heavy were sent over. 6 % 5 != 0, so force was not sent over. This eliminates the need to have an if conditional to check every single possible combination of effects, reducing the number of conditionals to the number of effects. Multiple effects can also be called simultaneously.

Question: 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.

Answer: Our command interpreter is designed with careful consideration for efficiency and extensibility. In the Game class, we have implemented the parseCommand function, which efficiently handles text commands from the input stream and returns the corresponding command string. The mainLoop member function in the Game class utilizes an if-conditional chain to determine the appropriate action to take based on the returned command string. To accommodate new commands, we simply need to add the new command name to the vector of strings storing existing commands and include a corresponding conditional in the command interpreter. This process allows for easy modifications and minimizes the need for recompilation,

as we can rely on the existing game class. In the event of renaming an existing command, we can efficiently handle it by searching the vector of strings for the matching command and making the necessary adjustments. While string comparison is generally considered to have O(n) time complexity, the fixed and relatively short nature of each command ensures constant time efficiency.

Moreover, we have the opportunity to enhance the interpreter by introducing support for macros using an unordered map, or Hash Map. This implementation enables constant time (on average) retrievals, optimizing performance. To handle the macros effectively, we utilize a stringstream as a buffer to queue the expanded commands. The parseCommand function reads from this buffer unless it's empty, in which case, it falls back to reading from the standard input. When a macro is inputted, we store it along with its expansion as a key-value pair in the unordered map. Subsequently, when parseCommand() is called, it scans the unordered map to check if the received command is a macro. If so, it fetches the corresponding expansion from the hashmap and appends it to the stringstream buffer, acting as a queue. As a result, the buffer efficiently manages the expanded commands for future processing.

6 Extra Credit Features

Our group also implemented extra credit features to enhance the gameplay experience, with a primary focus on the graphical display. One of the key features added is the integration of keyboard functions, allowing players to interact with the game without using text commands. By using the arrow keys, players can now easily move the blocks, while the 'd' and 'a' keys enable clockwise and counterclockwise rotation respectively. The space key has been assigned to drop the block, which is designed to also end the current player's turn. A noteworthy implementation is the synchronization of text and graphical displays, ensuring a seamless transition between moves. This was done by adding command queues for each player, making sure that commands are called in the right order with no lag.

Another significant enhancement lies in our sound effects that adds a new layer to how players interact with the game. When players drop a block or successfully clear a row, the game produces appropriate sounds, enhancing the overall engagement. Also, our group introduced interactive buttons that allow players to level up, level down, or restart the game. Each of these buttons has been complemented with unique sound effects, further elevating the gameplay experience.

Additionally our game offers three visually appealing themes: light, dark, and dracula. These themes cater to different preferences and enhance the overall visual appeal of the game. The combination of keyboard controls, immersive sound effects, and visually appealing themes elevates the overall gaming experience to new heights and makes it more fun for the players.

Lastly, each block in play has a "ghost", showing where the block would land if the drop command was called. This allows users to play the game easier, and visualizing the board better. This was done by cloning the current block and calling drop on the clone, and updating the "ghost" block everytime movement

or rotation was called.

To implement the graphics for this project, our group made a choice to switch from X11, which our team had used in a previous assignment to OpenGL. This decision yielded significant improvements in graphics speed. Nonetheless, the transition to OpenGL presented challenges due to its steep learning curve, specifically when adapting it to suit our game requirements. Overcoming these issues involved referring to OpenGL documentation and looking through examples on forums such as Stack Overflow to address the encountered errors. Through these efforts, our group achieved success in completing the implementation and were delighted with the final results.

7 Final Questions

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?

Creating a well-defined plan of attack and a structured design is of utmost importance in working on developing software as a team. Our utilization of Unified Modeling Language (UML) proved to be invaluable during the design phase (DD1). By investing significant thought into our design, we were able to establish the primary structure of our program, deviating only slightly during implementation (as mentioned earlier). This approach facilitated the development process and allowed us to focus on specific components individually. With the aid of UML, we gained a holistic view of how each element fit into the overall design and interacted with other components within our project.

During the course of our project, the integration of Git and GitHub also proved to be instrumental in our development. Our well-structured workflow involved two branches: 'main' and 'dev'. We emphasized maintaining the 'main' branch as a stable and production-ready version of our game, free from any known bugs or pending fixes.

The 'dev' branch, on the other hand, served as an active development area where our team members collaborated and implemented new features, improvements, and bug fixes. This approach allowed us to work on new functionalities without jeopardizing the stability of our 'main' version.

To ensure code quality and a seamless integration of changes into the 'main' branch, we adopted a code review process. One of our team members was responsible for reviewing the changes made in the 'dev' branch and ensuring that they met the required standards. This review process ensured that only well-tested and refined code was in our 'main' branch.

The utilization of Git's version control capabilities enabled us to easily track and visualize the changes made to the codebase over time. Each group member could monitor modifications made by others, detect potential issues, and offer valuable feedback. This collaborative approach not only fostered effective communication within the team but also led to a more robust and stable final product.

By leveraging the power of Git and GitHub, we streamlined our development process, while maintaining a clear separation between stable and experimental code. This version control system became an invaluable tool throughout the project, allowing us to work in a coordinated and efficient manner, resulting in the successful completion of our game with a high level of code quality.

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

One critical improvement we could have made from the outset of our project was adopting smart pointers instead of initially relying on raw pointers and subsequently having to transition to using smart pointers later in the development process.

By leveraging smart pointers right from the beginning, we could have significantly enhanced the overall robustness and safety of our codebase. Smart pointers automatically handle memory management, reducing the risk of memory leaks and dangling pointers that often plague projects employing raw pointers.

Additionally, smart pointers provide us with a more intuitive and convenient interface for working with dynamic memory, mitigating the need for manual memory allocation and deallocation. Switching from raw pointers to unique pointers proved to be challenging due to unexpected debugging issues that surfaced during the transition. Although, in the end we were able to complete the transition, starting with smart pointers would have streamlined our development process and allowed us to focus more on the core functionality of our project.

8 Conclusion

In conclusion, this project has provided our group with a valuable opportunity to apply the core concepts we learned in class, fostering a deeper understanding of software development. Throughout the project we gained insights into the intricacies of the software development cycle, the significance of planning, and the art of effectively resolving various challenges that arose. This hands-on experience has been instrumental in enhancing our abilities as a cohesive team and has reinforced the importance of collaborative efforts in achieving success especially in coding.

During the project's progression, we encountered and conquered numerous obstacles, which further sharpened our problem-solving skills and resourcefulness. These experiences served as invaluable lessons, teaching us to embrace adversity as an opportunity for growth and learning. We also learned to leverage various aids and tools to streamline our development process, resulting in improved efficiency and productivity.

As we reflect on this experience, we look forward to utilizing the skills we have acquired and building upon the knowledge we have gained in our future endeavors within the field of software engineering. The lessons learned from this project will serve as a solid foundation for tackling complex challenges and devising innovative solutions in our professional journeys. Together, we aspire to make meaningful contributions

to the world of software engineering, armed with the confidence and expertise derived from this project's successful completion.