Biquadris Design Blueprint

Rithvik Kolanu
University of Waterloo

Daniel Kurien
University of Waterloo

Sami Yousef
University of Waterloo

CS 246 - Object Oriented Programming

Date: July 19, 2023

Project Management

To approach this project successfully, we had a detailed discussion to establish a well-structured workflow that would effectively accommodate our team's diverse schedules and workloads. Recognizing the importance of flexibility and remote collaboration, we prioritized the need for an asynchronous work environment, given that each team member was simultaneously handling various course-related responsibilities.

After careful consideration, we unanimously concluded that implementing a git-based workflow would serve as the cornerstone of our project management strategy. This decision was driven by the realization that utilizing a version control system like Git would offer us unparalleled efficiency in terms of file sharing and change monitoring. GitHub was particularly appealing as it was well integrated with Visual Studio Code and offered native Mermaid support; a JavaScript based diagramming tool.

By adopting Git, we have the advantage of seamless collaboration, enabling each team member to work on their assigned tasks independently, without being constrained by time zones or conflicting schedules. With the ability to work asynchronously, we can ensure a steady progression of the project, as each member could contribute at their own pace and convenience.

The version history feature of Git also works as a crucial asset for our project. It provides us with a comprehensive record of all modifications made to the codebase, allowing us to track and trace any changes introduced throughout the development process. With a well-maintained version history, we can easily revert to a stable and functional iteration of the project, ensuring that we always have a reliable copy to submit in case we encounter memory leaks or unfixable bugs.

1 Game Design

Similar to many turn-style games, we plan to have a main event loop that will handle the logic for commands and rerenders.

1.1 Creation of Blocks

In Biquadris, the block factory design pattern efficiently generates blocks based on the abstract block class. These blocks encompass 7 inheritor classes, each representing a unique block type with defined tetromino square offsets. 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 the Board class and simplifies random block generation due to the pattern's inherent readability and centralized block creation logic.

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.

1.3 The Players and Board Logic

In the context of a two-player game like 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. This procedure is applied to

every filled row, ensuring that the integrity of the board is maintained throughout the game.

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.

1.5 The XWindow Class

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 will handle the drawing process manually by directly rendering each element onto an XImage. By doing so, we can significantly reduce the number of requests made to the X11 server. Instead of sending individual requests for each rectangle alteration, we will consolidate the changes into a single XImage and utilize the Xlib function XPutImage() to transmit the entire image to the server in one go.

This optimization strategy aims to streamline the rendering process and minimize the overhead associated with numerous requests, thereby enhancing the overall performance and responsiveness of the graphical operations within the X11 window.

2 Responses to Questions

Question 1: Lifespanned Blocks

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 2: Additional Levels

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 3: Additional Effects

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. For example, if blind = 2, heavy = 3 and force = 5, player one could send both those effects by sending $2 \times 3 = 6$ 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 4: Additional Commands and Macros

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.

3 Division of Roles

July 20-23: During this period, our team will focus on building the fundamental framework of the project, encompassing block, board, player, and game classes. Additionally, we will create a functional demo to thoroughly test game limits and interactions. To ensure efficiency and specialization, Rithvik will be in charge of block creation, while Daniel and Sami will work on player-game logic. Moreover, we will integrate a basic graphical window to enhance user experience.

July 24-27: Advancing the project further, we will concentrate on developing sophisticated game logic, diverse levels, and captivating visual effects, as well as implementing a robust scorekeeping system. Our dedicated efforts will be devoted to identifying and addressing any existing bugs and memory leaks. Furthermore, to elevate the project's functionality, we will introduce an advanced command interpreter featuring level support. For this stage, Dani will take the lead in creating level factories, while Sami and Rithvik will collaborate on the command line interpreter, essential for spawning level-specific blocks.

July 27-30: In this phase, we will prioritize the refinement of the project by systematically eliminating bugs and memory leaks that may persist. We will incorporate additional features to enhance the overall experience and commence working on comprehensive documentation. Emphasis will be placed on improving code readability and implementing appropriate design patterns to ensure the project's maintainability and scalability. Feedback from stakeholders and team members will be taken into account, and we will collaborate to implement valuable enhancements.

July 31-Aug 1: As we approach project completion, we will focus on finalizing the documentation and preparing the project for submission. Concurrently, rigorous testing and debugging will be conducted collectively to ensure a robust and polished final product. Our aim is to meet the project completion and submission deadline, demonstrating a successful and well-executed endeavor.