CS246 A5 Project Final Design
Luke McDonald, Alfred Jiao, Ashton Chan

Introduction

The game of Biquadris is a Latinization of Tetris that is not played in real-time, and consists of a two player competition. The rules are very similar to Tetris, with each player controlling blocks that appear at the top of their respective boards, dropping them with the goal of not leaving any gaps. Once an entire row has filled, it disappears and all other blocks move down to fill the gaps. The details of our group's implementation of Biquadris is outlined below.

Overview

After some refinement and discussion on how we wanted to structure the implementation of the game, we decided that the Model-View-Controller (MVC) model was the most appropriate architecture pattern for developing Biquadris.

Our Biquadris game is organized into three main class components, the game controllers, the game model, and the game viewers. Specifically:

- The game controller consists of a main Controller class, with nested BoardController and BlockController classes
- The game model consists of the following classes: ModelContainer, ModelPlayer, ModelBoard, ModelBlock, and ModelCell
- The game views consist of an abstract observer class, abstract subject class, and two concrete observer classes, namely TextDisplay and GraphicalDisplay

The details for each class will be described below.

Design

Controller

Class: Controller

The Controller class serves as the primary orchestrator of the game logic, handling the user input and game flow. Specifically, it includes the entire command interpreter which accepts command inputs from the user and converts these inputs into actions on the ModelContainer and observers accordingly. It adheres to MVC by mediating between the Model (ModelContainer) and Views (TextDisplay, GraphicalDisplay), without directly managing game

state or display logic. The class demonstrates low coupling through its delegation of specific responsibilities to BoardController and BlockController, and high cohesion by focusing solely on game flow coordination and command interpretation.

- Notable fields and methods
    - ModelContainer *Model: A pointer to the game model Model
    - BlockController: A friend nested class that handles Block operations and movement (vertical, horizontal, drop)
    - BoardController: A friend nested class that handles levels with block generation, and row clearing
    - GameplayLoop(): Manages the main game loop and player turns
    - ReadCommand(): Parses and validates user input
    - PlayerTurn(): Handles the logic for a single player's turn
    - commandMap: Maps abbreviated commands to full command names

Class: BoardController

The BoardController is a nested class in Controller that represents a more focused controller specifically handling board-level operations. It maintains high cohesion by concentrating solely on board-related functionality such as block generation and row clearing. The AddBlock method creates new blocks according to game rules, while GenerateBlock implements the level-specific block generation probabilities. This class demonstrates the Single Responsibility Principle by handling only board-related control logic.

- Notable fields & methods
    - GenerateBlock(): Creates new blocks based on current level rules
    - ClearRows(): Handles row clearing and scoring
    - AddBlock(): Instantiates new blocks on the board

Class: BlockController

A nested class that is private to the Controller, and handles individual block manipulation, showing strong cohesion by encapsulating all block-related operations. It is private to the controller to ensure that the Block information is only managed by the controller.

- Notable fields & methods
    - Controller *Parent: A pointer to the parent Controller class
    - ModelBlock *CtrlBlock: A pointer to the specific ModelBlock being manipulated
    - ModelPlayer &Player: A reference to the associated player that the block object is from

Model

Class: ModelContainer

The ModelContainer class encapsulates the state of the overall game engine, where changes to the ModelContainer are entirely driven by the Controller class. It’s a subclass of the abstract subject class from view, making it the concrete subject, thus allowing observers to pull changes from the game model. Notably, the ModelContainer only contains two ModelPlayer objects, without any knowledge of other information such as the board, or blocks.

- Notable fields & methods

    - int highScore: Field to keep track of the game's high score
    - pair<ModelPlayer, ModelPlayer> ModelPlayers: A pair of ModelPlayer objects for player 1 and player 2.

Class: ModelPlayer
The ModelPlayer class encapsulates player-specific data, which in this case, will be their game board, and their respective levels and scores. This is to ensure each player does not direct knowledge of the overall game model state, nor other players.

- Notable fields & methods
    - ModelBoard PlayerBoard: The player's respective game board
    - pair<int, int> PlayerLevelAndScore: A pair of ints containing the player's level and score
    - pair<BlockType, SpecialAction> next: A pair object that stores the player's next block in terms of its type (one of the seven) as an enum type and if it pertains to a special action (also as an enum type)

Class: ModelBoard
The ModelBoard encapsulates the state of a player's Board within the Model, which includes the contents of each cell of the board, a vector of blocks pertaining to the board, and a definition of each block with its shape and its character.

- Notable fields & methods:
    - vector<vector<ModelCell>> BoardMatrix: Defines the player's board in terms of a 2D array of ModelCells
    - vector<ModelBlock *> BoardBlocks: Stores the blocks on the board with a vector of ModelBlock object pointers
    - map<BlockType, pair<vector<ModelCell>, char>> DefBlockMap: Map each enum block type to their default configurations.

Class: ModelBlock
ModelBlock and ModelCell represent the finest granularity of the Model layer. ModelBlock manages individual tetromino blocks, tracking their cells and attributes like lifespan and creation level.

- Notable fields & methods:
    - vector<ModelCell> OwnedCells: A vector of cells belonging to the block
    - int Level: Stores the level at which the block was created for scoring purposes

Class: ModelCell
ModelCell represents individual board positions, maintaining their state and ownership information. Both classes exhibit strong cohesion and single responsibility, focusing on their specific data management tasks.

- Notable fields & methods:
    - pair <int, int> CellIndex: Coordinate pair to maintain position data on the board
    - ModelBlock *OwnerBlock: Reference to containing block

View

The View layer implements the Observer pattern to maintain loose coupling between the game state and its visual representation. The Observer abstract class defines the interface for state change notification, while the Subject abstract class provides the infrastructure for managing observers and triggering updates.

Class: subject

This abstract class implements the Subject role in the Observer pattern, managing observer registration and notification.

- Notable fields & methods:
    - vector<Observer *> observers: Vector of observer pointers to support observer pattern implementation
    - attach(): Registers observers
    - notifyObservers(): Updates all registered observers when game states are changed

Class: observer

This abstract class implements the Observer pattern, defining the interface for concrete observers.

- Notable fields & methods:
    - notify(): Pure virtual method for updating observers

Class: TextDisplay

The TextDisplay class provides the text-based representation of the game state, implemented as a concrete observer. This observer will be updating automatically to model changes while remaining completely decoupled from the model's implementation details.

- Notable fields & methods
    - ModelContainer *subject: Maintains a pointer to the concrete subject ModelContainer to observe game state changes and manage the displays accordingly
    - notify(): Updates the text display based on the board changes. Uses an overloaded operator for displaying player boards and next blocks.

Class: GraphicalDisplay

Provides graphical representation of the game state, implementing the Observer pattern through concrete implementation.

- Notable fields & methods
    - draw(): Renders the game state
    - getColor(): Maps block types to Xwindow colors
    - notify(): Updates display on state changes, will only update cells that were affected

Resilience to Change

When constructing our design, by focusing on maintaining clear responsibilities (high cohesion), and minimizing coupling (loose coupling) for classes and functions, we were able to maintain flexibility.

High Cohesion:
Our classes and functions are designed in accordance with MVC principles. One of our key goals was to really emphasize the fact that the model *(which is really an aggregation of several classes that comprise the state of our program)* is a structure responsible solely for storing and providing data. Additionally, within the model aggregate class, sub-model classes *(e.g., a Cell is subordinate to a Board)* do not directly modify or access information from their super-model classes *(e.g., a Board or a Block are both superior to a Cell)*.
Example:
Suppose we decided to make the board taller by 10 rows. Since there is minimal upward reliance from sub-model classes, the change will be as simple as initializing a board with a larger 'height' specification – and then any implementation for classes subordinate to the board class will not need to be modified whatsoever.
Key Details:
Where possible, we tried to use dynamic containers to store data and information. For example, we used enum classes for both BlockType and Special Actions, allowing easy addition of new types of blocks and actions. Additionally, we used a map to store pairs of Block Type and initial board position.
We also used int return values for most controller-type functions so that we can address special situations as necessary in higher levels of control flow. For instance, after a given command is read, we can pass an int all the way up to the gameplay loop control flow if necessary.

Minimal Coupling:
As is generally resultant of high cohesion, we sought to minimize the reliance of classes within the aggregate model. This was particularly challenging, due to the fact that the player, board, cells, and blocks are all quite related. Our way to mitigate this was by creating specialized controllers within the controller class that could perform various model-related operations without relying on adding methods and additional fields to the actual model. This meant that individual model classes could reduce their reliance on other model classes, and we could still create a convenient interface for modifying the board on various levels of abstraction (i.e., through BoardController or BlockController). Hence, we could modify particular mechanics or details pertinent to one part of the model.
Example:
Suppose we wanted to make a default block be three cells, not four. We could easily do this by modifying the initial map for the block types.
Key Details:
As mentioned above, high cohesion often yields loose coupling. This is evident in our design – since we tried to make model classes and functions very granular and cohesive, they do not rely significantly on the state of other model classes. Most functions that require iteration also do not use a default range (e.g., iterating through a block's cells does not stop at three – it

continues iterating until it reaches the end of the vector containing cells). Hence, most implementation details are not “constrained” by other classes – allowing a flexible and easily modified design.

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?

Given our current planned implementation, each “block” is an object. We could have a derived class for “vanishing” blocks from our regular block class. In our derived class we would add a field that keeps track of the block’s lifespan which will allow us to easily remove them from the board if not cleared after the lifespan reaches a certain number. With the creation of each block, we can set its lifespan to 10 and in our gameplay controller, with the addition of each block we would modify the existing lifespan of any vanishing blocks. When the lifespan reaches zero (i.e. 10 more blocks have fallen), we can then call the respective methods to delete and remove the vanishing block properly. In order to confine the generation of such blocks to more advanced levels, inheritance works well as then we can generate the vanishing blocks when needed, directly in our gameplay controller when the level is set to a certain number, and normal blocks otherwise in lower levels.

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

Implementing a factory method pattern for the different levels of the game would allow us to switch the policies surrounding the generation of blocks and game rules dynamically at runtime. In this case, levels 0-4 would be a subclass of an abstract Level” class, with different characteristics in each block creation method. Introducing a new level with different policies would be as simple as creating a new subclass, defining its policies on which classes to instantiate, and a recompilation of the one superclass.

Question 3: 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?

The decorator pattern would be an elegant solution for handling multiple simultaneous “special actions” in Biquadris. For instance, by creating a base Block interface/abstract class that defines core behavior, we can then implement decorator classes for each effect that involves blocks (heavy) that wrap around a Block and modify/add behavior. The key advantage is that

effects can be stacked by wrapping decorators around decorators (e.g. Block -> HeavyDecorator -> AnotherEffectDecorator…), making it easy to add new effects without modifying existing code or creating complex conditional branches for every possible combination. This decorator pattern can also be used for effects on the Board object such as the special action blind, and other Board effects we may want to add. This keeps the system extensible while avoiding the need to modify existing implementation when new effects are added.

Question 4: 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.

In order to allow our system to accommodate the addition of new command names and make changes to existing command names with minimal changes to source code, the key would be separating the command names and actual implementations of the commands with a command registry system. We can implement a command registry system that maps commands to their respective implementations and also store our command binds in a separate configuration away from our code. Within our registry we can also implement partial recognition to match partial commands to their respective implementations. This way adding a new command only requires creation of the command class itself and registering it with the system, and changing command names would not change any implementation of the commands themselves.

With a proper implementation, adapting our system to support command renaming should be straightforward. Similarly with the above implementation, we would need to have an alias map that stores user-defined command names and recognizes them to their respective commands. For any potential conflicts regarding aliases and built-in command names we would also require some sort of priority system to deal with the conflicts. We would also need to consider the naming itself, ensuring that new command names don't conflict with existing shortcuts, etc., and a method to save all user created shortcuts.

A macro system could be implemented by treating macros themselves as first-class commands that contain sequences of other commands. We would need to implement a way to record and store input of commands, as well as a method to execute macros and run each

command in the sequence. We could also consider a way to edit existing macros (name, contents), however more importantly we would have to consider the impact of the command shortcut system for macros as well.

Final Questions

Question 1: What lessons did this project teach you about developing software in teams?

One key lesson was the importance of coordinating responsibilities effectively. Early on, our team struggled with overlapping work and encountering merge conflicts, particularly when two or more of us were making changes to the same sections of the code. This highlighted the need for an initial planning phase, where we collaboratively defined the program's overall structure, delineated responsibilities, and set clear boundaries for individual tasks. With a well-documented understanding of the game's architecture and our individual roles, we found it easier to work on the code base in parallel without many merge conflicts.

In addition, throughout this project, we realized how crucial in-person collaboration is when working in a team, especially with an interconnected project like Biquadris. While working remotely has its benefits, we found that meeting in person fostered a level of real-time communication and problem-solving that significantly improved our efficiency. For instance, when planning the project's overall structure, meeting in person allowed us to use tools like whiteboards to sketch out our UML diagrams and visually break down tasks. This was far more effective than discussing through Discord, as it helped everyone get on the same page quickly and resolve ambiguities in the design. In-person collaboration also proved invaluable during debugging sessions. When one of us encountered a bug, having the team physically present allowed us to brainstorm solutions, test fixes immediately, and rapidly iterate on the code. This immediate feedback loop often resolved issues faster than asynchronous communication could

Another challenge was communicating and prioritizing bugs. When issues arose, we initially struggled to convey the details effectively, leading to delays in resolving problems. As we progressed, we adopted a more systematic approach, categorizing bugs based on their impact on the game's functionality and prioritizing those that blocked the core features like block manipulations. This practice emphasized the value of clear and timely communication, which kept the project on track and reduced misunderstandings.

Lastly, we also learned that mitigating merge conflicts requires not only careful planning but also disciplined use of version control. Regularly merging smaller changes and maintaining an updated codebase helped us avoid major conflicts later in the development cycle.

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

One thing we would do differently is restructure the command-parsing system. The current implementation relies on a very large command map in Controller.h, (over 70 entries) and a complex string-parsing implementation in ReadCommand in order to actually process user inputs. This approach makes adding new commands or modifying existing commands more complex, as we would have to directly modify our command map and implement the corresponding parse/handling logic in our controller. Rather we could consider implementing a proper command pattern with respective dedicated command classes for each action type. This would now encapsulate command logic and allow for command validation to be more straightforward, as well as a less complex way to parse and handle command multiples. A command pattern would also allow us to expand deeper and implement features such as command history, or even an undo functionality. With this we would also implement error handling to provide clear feedback about invalid command inputs, rather than the existing "silently continuing" execution in our current implementation.

Another thing we could improve on and implement differently if starting over is refactoring the BlockController class purely to improve the design and readability component of the class. Currently the implementation contains very complex movement functions, particularly rotate, as well as nested lambda functions. For example, our rotate method contains nearly identical code for our clockwise and counterclockwise rotations, which leads to less reusability as well as increased maintenance overhead.

These changes would significantly improve code readability, reusability, and maintainability, while making it easier to implement new game functionality and provide a better structured foundation for future game enhancements.