



CS246 A5 - Biquadris

November 28, 2024

Zhenghong Chen - z253chen

Wenling Zou - w3zou

University of Waterloo

Introduction

The game of Biquadris is a competitive, two-player variation of the classic Tetris game, featuring strategic gameplay on two separate boards. Each board measures 11 columns by 15 rows, and players take turns placing tetromino blocks onto their respective boards.

The game introduces unique mechanics: when a row is completely filled, it disappears, causing the blocks above to shift down by one row. Under certain circumstances a player can pose special action towards the other player. If a player cannot properly place a new block on their board, the game ends for that player. Players can choose to start a new game or quit after one concludes. This blend of competitive elements, strategic thinking, and unique mechanics makes Biquadris an engaging and dynamic multiplayer experience.

We implemented many recommended coding practices and design patterns to make our program to minimize redundant code duplication, minimize coupling between modules, maximize code reusability, and maximize module internal cohesion.

Overview

The game Biquadris is made up of eight big components:

1. Game
2. Player
3. Board
4. Block
5. Level
6. Cell
7. Observer Pattern
8. Command Interpreter

Model-View-Controller:

The game represents the model, the observers represent the view, and the command interpreter represents the controller.

The relationship between these components is as such: A game has two players; each player owns a board and has a level; Each board owns a collection of cells, and each board has a collection of blocks; each block also has a collection of cells, but they don't own those cells. Cells are dynamically allocated. The game inherits from a subject, so that it can store a bunch of observers. The command Interpreter reads in commands from users, understands them, parses them, and executes them to manipulate the game state.

Each component was composed of one or several classes. These components each have their duties, attributes. Components collaborate together to make sure the game runs properly.

No delete keywords appear in the entire program and no memories were leaked. We leveraged RAII using smart pointers (`unique_ptr`) to manage memories instead of manually allocating and deallocating memories. Which ensures our program is safe and convenient to develop and maintain.

Following is a more in-depth and comprehensive discussion of our design of the game Biquadris.

Design

Board

At the beginning phase of the development we focused on creating an empty board to display to the terminal. We decided to use Vector as the key data structure to store our cells. The board has a 2D Vector of unique pointers that points to a series of heap-allocated Cell objects.

- `std::vector<std::vector<std::unique_ptr<Cell>>> grid`

We design the skeleton of our program surrounding these cells, the key idea is that the board owns all these heap-allocated Cell objects, and nobody else does. Anyone else who wants to store references to these cells must use raw pointers. Surprisingly, using RAII enhances our understanding of memory ownerships, which greatly facilitates later development.

Block

The block class is an abstract class, as we have different types of blocks, each with their unique initialization and rotation. However, the way we designed block movements allows the algorithms for moving the blocks left, right, and down fairly general, so that we can put them into the superclass. Blocks also have their own level variable, as we need to keep track of the level when they're created to correctly calculate the score gained when they were cleared from the board.

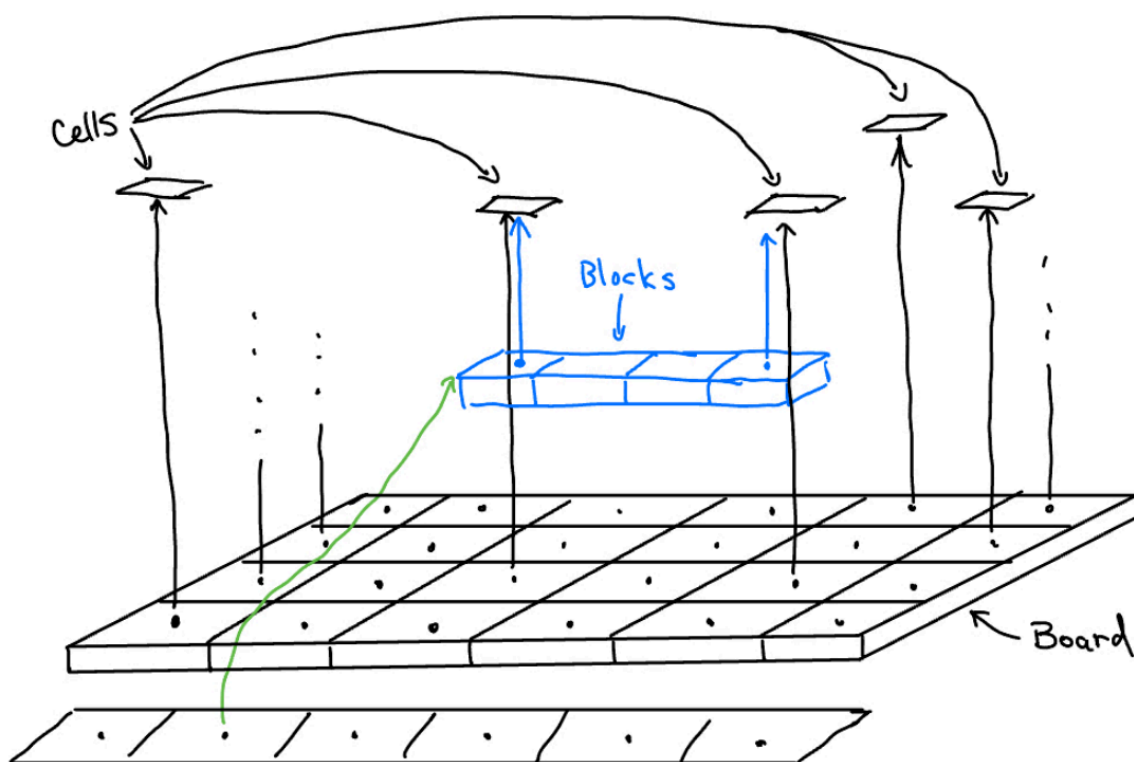
Each specific type of block (i.e. IBlock, JBlock) inherits from the superclass Block with all the getters and setters methods, movement methods, and block-update methods. All the subclass needs to implement are their initializers (initialize the block's shape and fill them with color), `rotateClockwise()`, and `rotateCounterclockwise()`, and of course the constructors and destructors. This saves us from unnecessary code duplication, also it helps improve efficiency when maintaining the code, i.e. fixing a movement bug wouldn't have us modify movement logic for each of the eight types of blocks.

We'll discuss how blocks move, rotate in later sections.

Cell

Cell class stores two integers: xCoordinate and yCoordinate, and a character: type. The methods inside are setters and getters for each variable, and an isFilled() method which returns if the cell is occupied by some blocks.

Block Movements



Above picture is a schematic diagram of how our board, blocks, and cells work together to achieve the game's movement and collision detection functionalities.

As discussed in the overview, the board owns a 2D vector, or a plane of Cells. Each element in the vector points towards a heap-allocated Cell instance. Meanwhile, the blocks that are currently on the board also point to some cells that the board owns. The Block in our program is really just a "shape" of an actual block that humans understand in real life, they keep track of what cells that make it a block that they're supposed to be at on the board. This design is the basis for conducting later implementation of block movements, line clearings, and collision detections. It allows for maximal code reuse, and minimal code duplication.

Simply speaking, the block move by doing a the following:

1. Check if the block is allowed to proceed with this movement.
2. If yes, then proceed the movement.
3. If no, do nothing.

The block checks if it's allowed to move left, down, or right, first by emptying all its cells; then they grab a vector of raw pointers that points to cells that'll represent where the block will be at after the movement, and check if any cell within is already filled. If yes, that means after the movement, the block will overlap with another block, which is not valid, which makes the movement do nothing. Otherwise, the movement will be valid, and we let the block accomplish that move.

Similar for rotate, except that we sort of "hardcoded" out the algorithm for block rotation. Simply saying, block of type 'J', 'L', 'T', have four possible shapes, type 'S', 'Z', 'I', have two possible shapes, and type '*', 'O', have one possible shape. When initializing each block, each cell's location on the board is setted specifically. When rotating the block, the block superclass has a variable rotationIndex which represents the current shape of the block, and the block rotates according to their rotationIndex. When checking if the rotation is valid, the block calls the same method as when moving blocks, we designed it to be that they can be achieved this way. The function shared here is called `Block::isValidMove(std::vector<Cell *> newCells)`, `block.cc`, line 231.

Moreover, it is totally unnecessary to implement both the `rotateClockwise()` and `rotateCounterClockwise()`, since rotating counterclockwise is essentially the same as rotating clockwise three times, all blocks only have clockwise implemented. Rotation for 'O' and '*' is trivial, they didn't do anything, so no implementation.

Clear Line After Drop

Clearing line is the part that we spend most time designing and implementing. In order to make sure the block can tell what parts of it were cleared from the board after a line or multiple lines are cleared, and when it is fully eliminated and should return a score, we have to update all blocks' position after each line is cleared, removing any parts that are cleared, copying the rows downward, calculate scores gained from the clearing, etc.

Our logic to clear lines or multiple lines are as follow:

1. Drop the block
2. Count how many rows are filled
3. Starting from bottom most row to the uppermost:
 - a. if the row is filled
 - i. empty a row (set all cells at that row empty)
 - ii. iterate through all blocks to check if any parts of the block are empty, if so remove those pointers from the block.
 - iii. iterate through all blocks to check if any parts of the block are above the row being cleared, if so, shift those cells to cells that are down by one on the board.
 - iv. starting from the row being cleared, copy all rows above it down by one.
 - b. if the row isn't filled
 - i. move on to the next above row
4. calculate the score for clearing any blocks and clearing any lines, based on their levels.

Interpreting the commands

We wrote an interpreter class to parse the command into two strings, a multiplier and a command string. The interpreter can tell what the command represents as long as it can distinguish the command from others. This is achieved by looping through all possible commands and checking if the command entered is the sole substring of any command stored in the interpreter class.

The interpreter is also responsible for executing the command after interpreting. Interpreter class has a pointer to the game object, and they can run the command for \$(multiplier) times, if applicable.

Game

The game class is the center of the program. It solely contains all states of the entire game. It has two players, it owns an interpreter, and some methods to run the game, restart the game, switch turns, and notify observers.

Design Patterns

- Iterator Pattern
- Observer Pattern
- Factory Method Pattern

We use iterator pattern in many circumstances when accessing “Cell *” and “Block *”, range-based for loops were used upon accessing elements within the Vector container structures.

We used an observer pattern to allow for easy updating of game’s views when game states are changed thanks to the subscription mechanism. Adding new observers can be easily achieved, and the abstraction completely separates the game and the observers. Observers define their unique way of presenting data to the users, nothing about the view game class would have to be responsible.

Factory Method Patterns were introduced when we designed the levels. Each level was basically responsible for only one duty: generate a new block that follows its own rules such as random versus nonrandom, probability of getting S and Z blocks, star block in level 4, etc. When the player object gets the order of creating new blocks, all it does is to inform the level object that it owns to return a new block, and the player can then easily store the block for later use. There’s no detail that player class would need to know about any rules at any levels of generating a new block, the responsibilities were delegated to the different level classes.

Resilience to Change

Classes like Block and Level are abstract superclasses, they have a bunch of concrete subclasses. Adding a new Level or a new Block will be easily achieved. The new class will only need to implement the part that is special and unique, and the rest of the general parts will be inherited from the superclass. This also minimizes recompilation, other existing classes wouldn't need to be recompiled except for the new classes that need to be compiled first time.

Low Coupling: Components such as Game, Player, Board, Block, Level, Cell, Observers interact with each other through well-defined interfaces and abstractions. The use of the Observer pattern decouples the game logic from the presentation layers, allowing each module to function independently without tight dependencies on others. Modularization of Blocks and Levels minimizes dependencies between components.

High Cohesion: Each class and module in the game has a focused responsibility:

- Game manages the overall game state and flow.
- Player encapsulates player-specific data and behaviors.
- Board handles the game board's state and operations like clear-line.
- Blocks can be moved, rotated, and the logic is encapsulated within the class.
- Level were responsible of generating blocks
- TextObserver and GraphicsObserver handle text-based and graphical representations, respectively.
- Interpreter decodes the command and executes them to update game state
- Methods within classes are closely related, supporting the primary purpose of the class.

As modules are loosely coupled, changes in one module (e.g., updating the graphics in GraphicsObserver) do not necessitate changes in others (e.g., Game or Player). This isolation allows us to modify, add, or remove features with minimal risk of unintended side effects. High cohesion within modules ensures that related functionalities are grouped together. When a change is required, we can focus on a specific module, improving efficiency and reducing the complexity of maintenance tasks.

We can work on different modules simultaneously without interfering with each other, thanks to the minimal interdependencies enforced by low coupling.

High cohesion ensures that the core logic remains consistent and reliable, even as peripheral components change, keeping our game relevant over time. Faulty modules can be replaced or updated without impacting the entire system, enhancing the game's overall robustness.

Moreover, the overall designing of how board, block, cell work together minimizes code duplication, and maximizes code reusability. If we decided to implement each block's `moveLeft()`, `moveRight()`, `moveDown()`, `drop()`, `isValidMove()`, `updateBlock()` exclusive to each block, the workload

required for any potential future changes of how the program works can be extremely heavy, and it'll also be more error-prone if any part of the component were missed (i.e. forgot to update `moveLeft()` logic for Z block, it'll took hours to pinpoint the error).

Another aspect that indicates how our design promotes resilience to change is revealed when we debug our code. We spent roughly 60% of time implementing all the modules, and 40% of time debugging and optimizing. When we were designing the program and composing our Plan-of-attack, we already had a standard in our mind that our program must appear elegant, in a way that modules are well encapsulated and manifest maximum cohesion, different modules are related with minimum coupling. This is not for the purpose of scoring higher marks, but to save us time and workload to deal with any potential demand in urgent bug-fixes, or expansion in the game function. One example of how designing a high cohesion and low coupling program helps us improve efficiency is when our algorithm for updating the block's position after clearing a line is buggy. This bug leads to severe undefined behaviour and segmentation faults, any blocks attempting to perform any operations below the lines previously cleared will either break to random pieces or lead to a SEGV and crash the entire program. The process of pinpointing the bug is painful, but once we notice what's going wrong, all we do is simply fixing one method (to be more specific, one line of code) to fix the entire program (If you want to check it out it's at board.cc, `Board::moveCellsDownByOne`, line 103). Another example would be the SEGV during the exit of our program. This bug wasn't discovered until the end of the implementation phase. We notice this bug when we're using Valgrind to check any memory leaks, and we notice the program crashes every time after the Game object gets destructed. We later realize the problem goes with the observers. We previously constructed the observers in the main function, which is stack allocated memories, so the memories should be automatically destroyed when the program goes out of scope, but our destructors for the observers assumes they're heap allocated and attempt to deallocate the memories, which contradicts. The bug occurs not in the way we implement them, but in the way we use them. This is really handy because fixing bugs no longer requires refactoring of code, we fix them by correctly using them, which is not to stack allocate the observers, but construct the observers during construction of game class, and the RAII will handle the memories. MVC, observer patterns, and well designed encapsulation of modules really helps us a lot here. We made sure modules interact with each other by interfaces and not by accessing and manipulating the fields directly, so that fixing bugs at lower level wouldn't require code refactoring at higher levels, and vice versa. There are also a lot more debugging sessions like the above two examples, and it is hard to imagine how difficult debugging would be if we wrote a high coupling, low cohesion program without making use of standard OOP design practices.

Answers to 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?

We can add a timestamp to each Block, and the timestamp will keep track of the lifespan of each Block. After 10 blocks have fallen and some blocks that still haven't been cleaned, they'll disappear from the board. By using the factory method pattern, we can easily allow different levels to have different rules and algorithms for generating the next block, hence we can easily set such "self-disappearing" blocks to only be generated in higher levels.

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


We would design our program to utilize strategy patterns and factory method patterns. Let's say we want to add a level 5 to our program, by having our program structured in a way that there is an abstract Level class, and concrete subclasses 0 to 4, to add level 5 we only need to write a new subclass called level5, and when recompiling the program, only the level5.cc needs to be recompiled, Level class, other Level0 to Level4 subclasses, and all clients using levels through the abstract Level interface don't need recompilation.

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?

There are different approaches, the most fundamental one being having a bunch of variables to keep track of each of the current effect's state, the downside of that is whenever we want to add a new effect, we'll have to modify a series of files to make sure new effect's state are kept track as well as displaying them. Even if the handling of the states would be a pain as many branchings are needed. Decorator pattern would then be a desired approach. We can have each effect layered upon another, all upon a basic game board, hence the effects became a "linked list" and it would cost much less work and recompilation when adding new effects.

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.

When designing the data structure to store the commands, we determined that a map would be an ideal choice. We can have the keys to the map to be string values which represent the user's inputs, and the values to be the command objects. Similar to the answers to the above questions, this will allow for minimal changes to source codes and minimal recompilation, as the users of the command system would only need to communicate to the modules through passing parameters, which minimizes coupling and delegates the responsibilities of handling different commands to the command interpreting module. This also allows for the renaming of the existing commands, which can be easily achieved by adding a new



pair into the existing command map whose key is the new name of the command, and the value pointing to the same command object. Then, persevering the original command pair or not can be determined by the program's requirement, and minimal code would need to be changed.

In order to support a “macro” language, we can add a new map whose keys represent the macro's names of type strings, and values to be vectors of command objects. Therefore, executing a macro would be equivalent to executing a “list” of commands, which is easy to maintain and easy to understand.

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?

Developing the Biquadris emphasized the importance of clear communication and collaboration. We learned that breaking down tasks and ensuring everyone was aligned was key to maintaining progress and avoiding overlapping work. We used tools like git for version control and github for storing the code to facilitate collaboration. Along with the encapsulation design of our modules, these tools allow for parallel development, so each member can work on their modules and merge the code later on. Version control also ensures changes that are lethal and should be reverted to previous versions is achievable. There were instances where we had different ideas about implementing features. These experiences taught the importance of balancing individual opinions in order to make progress while respecting everyone's ideas.

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

We tested some parts of our program and left some parts not tested as they should have been at the beginning stage, which leads to bugs being caught late in development. Although we incorporated unit tests, integration tests, and regression tests, we would use a test-driven development (TDD) approach to ensure that each component was robust before moving on.

