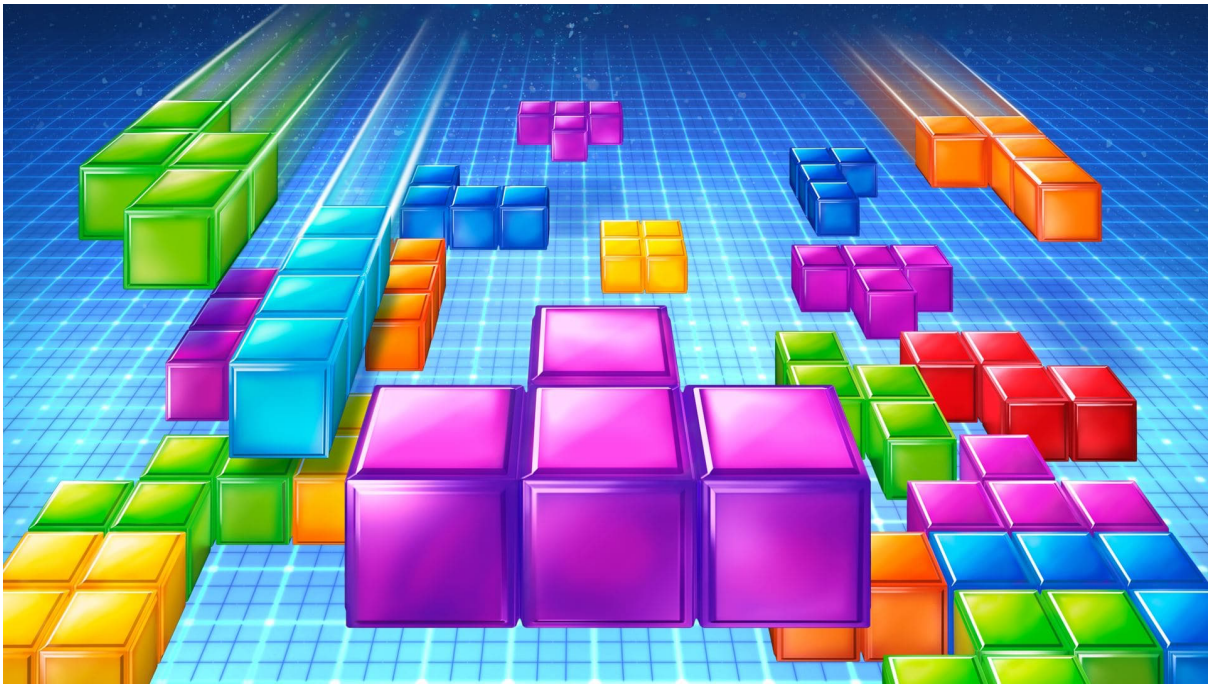


Biquadris

CS246 A5

Ruiqi Hao (r24hao)
Rickie Duan (r25duan)
Jackie Cao (yxcao)

Introduction

The game of Biquadris is a modified version of the classic game of Tetris. The main difference is that Biquadris is not real-time. It consists of two players and each player has their own board and takes turns dropping blocks, one at a time. They can move the blocks around, either left to right and/or rotate them. A player scores when a row is cleared or when a block is completely removed. The game is over if a new block is unable to be initialized, the corresponding player's game is over.

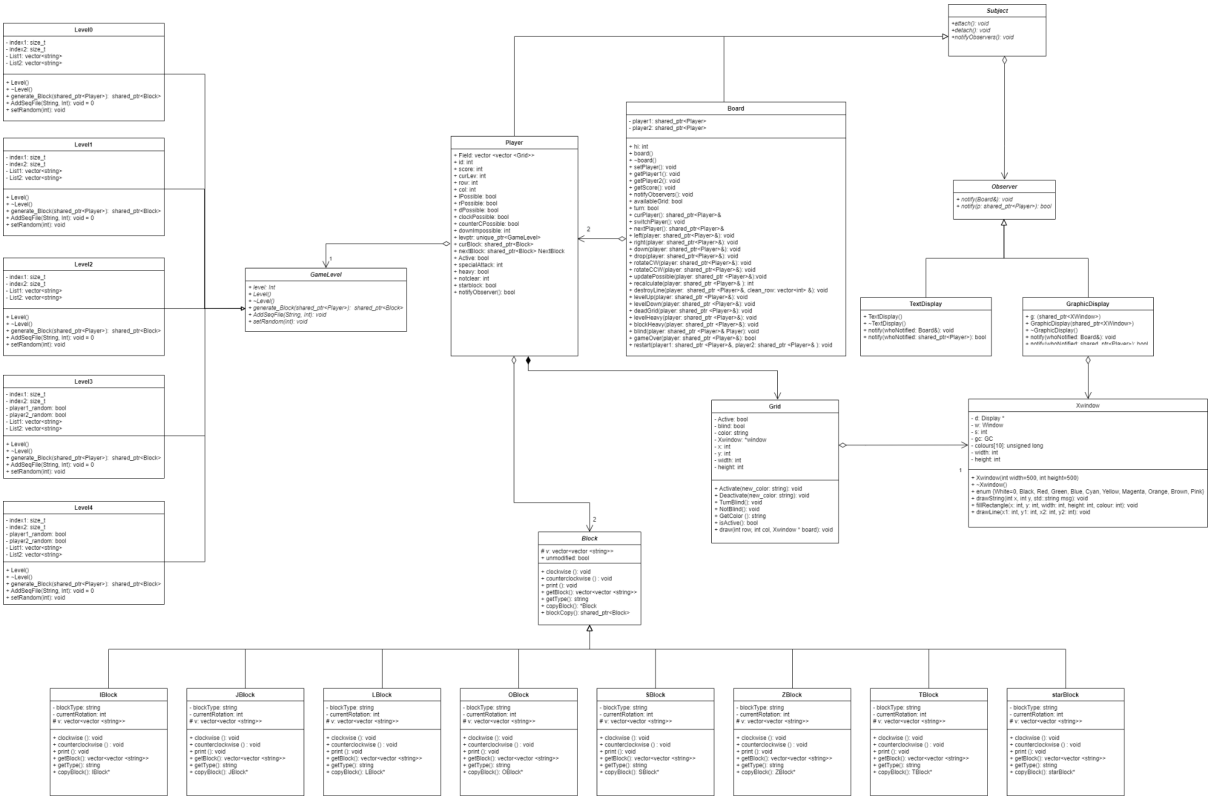
Overview:

This whole program utilizes 6 different classes:

1. Player
2. Board
3. Grid
4. GameLevel
5. Block
6. Observer

We use smart pointers and vectors for memory management between these classes. A detailed UML is listed below for Biquadris game structure.

Updated UML



Design

Player:

Player is struct instead of a class because we want the board to be able to access all of its fields. Thus it is better we just make it a struct. The purpose of the player class is to hold all necessary information for the current player. It owns a Field, a 2-D vector of grid (the actual game board), current and next block pointers and current level. In each player, there are fields showing the bool value There are also booleans to check if any movements of the block will be possible. Also, the special action switches are in player.

Board:

Board is the class that contains the main game logic. It controls the whole game process, and is responsible for managing player turns, interpreting commands (left, right, down, drop,

clockwise and counterclockwise) and updating players' status (keep track of each player's score and the higher one).
This is essentially done by two shared pointers to each player, allowing each function to access the player field and information. For each given command, we have a corresponding method to check whether the move is valid or not, and then update the xPossible fields in each player. We will recalculate the player's score when a block is being placed and causing lines to be cleared. The feature for each level will also show up here.

Grid:

Grid is a single basic square on the game board. The Grid class is used to form a Field variable for Player. A basic cell helps to determine where a Block is on the Field, and the space availablity to move left, right, or down. The purpose of this class is to help us update changes in the game board to text and graphic display. Cells can support various kinds of data types and store enough information for use. In Grid class we have fields that store the position, the color, and the state (Active or Blind).

GameLevel:

GameLevel is the superclass for all levels(Level0, Level1, Level2, Level3, Level4). The purpose of level is to generate blocks regarding the level. It either is randomly generated (all levels) or read from a script file (Level 0, 3, 4) provided to the command interpreter. It is part of the factory method pattern. Also, we are able to set seeds for getting the random sequence of blocks, so we don't get the same random sequence.

Block:

Block is an abstract class. There are eight subclasses under the Block it: IBlock, JBlock, LBlock, OBlock, TBlock, SBlock, ZBlock and StarBlock for level 4 only. In each subblock, we use 2-D vector with size of 5 * 4 to store the block since we need to reserve space for the block and three extra rows at the top of the board. There is a private int field called currentRotation in each subclass to update the rotation state. Different numbers correspond to different status. All subclasses override pure virtual functions in Block. The getCopy method will return a deep copy of the current block used to test whether the clockwise or counterclockwise operation is possible.

Observer Design Pattern

The Observer pattern plays a significant role in the Model-View-Controller architectural pattern. In this pattern, a target subject manages all the observer objects that are dependent on it, and actively notifies them of any stage changes. This is achieved by calling the methods provided by each observer, TextDisplay and GraphicDisplay. In our design structure, the Player and Board classes are Subjects. When a Board subject changes state, TextDisplay and GraphicDIsplay classes are notified and the screen is updated accordingly. This makes it so the level and score are displayed at the top above each of the player's game areas. Then we add the space where the blocks will drop and then display the next block that will spawn for both players after a dash line. When a Player subject changes, it generates the block at the top left of the player's game Field and checks if a down is possible.

Factory Method Pattern

The factory method is a creation design pattern that solves the problem of creating product objects without specifying a specific class. This factory method is used within the level class to spawn different blocks. Each level class has their own logic of creating new Blocks. Thus, we have the flexibility to add new levels. GameLevel defines an interface for creating a Block, but let subclasses decide which type of Block to instantiate.

Polymorphism

The block and level class uses a single interface to perform in different ways, depending on how the method is used.

Resilience to Change

Our code has the ability to adapt to change and overcome challenges. We can easily add or remove functionalities to the game, such as additional level and deletion of a type of block. The general structure of our program demonstrates resilience in object-oriented software development from the below aspects.

Modularity

We have six main classes to represent the game and we believe that this has low coupling and high cohesion. High cohesion involves grouping functionality based on separation of

concerns for flexibility and reuse. We follow the Single Responsibility Principle to make sure each module has one specific goal. So, when we recompile, as few files as possible are compiled again. Low coupling is the approach of connecting modules in a way to be as independent as possible. Modules should know as little as possible about their related modules. This means when we change a module, and this doesn't require changing a related module. For example, if we change an implementation for a method in Grid class, it doesn't affect the implementation in Board class as it just calls the method via Grid. In terms of Blocks, block is an abstract class so more blocks can be created as needed. For example, we created the starBlock easily for level 4 in addition to the initial seven main blocks with minimal changes.

Redundancy

If modularity allows you to change tires without breaking the car, then redundancy is the foresight to keep a spare tire in the trunk. For example, one type of GameLevel doesn't work, we still have other levels to test with. This is achieved from the use of inheritance relationships. The GameLevel is an abstract class of different levels. So, we can simply add a new level without interacting with Level 0-4. Redundancy extends so we don't rely on a specific Object.

Encapsulation

In object-oriented programming, encapsulation is defined as binding data together with the functions that manipulate them. Hence, we endeavored to use classes as opposed to structs and private or protected fields only. That means we can't directly access any function in a class. Instead, we need an object to access the function, which uses the member variables of the class. This enhances the safety of the program as data abstraction or hiding occurs between classes. It also helps us group related data and functions together, which makes our code more concise and easy to read. Moreover, the decoupled components can be developed, tested, and debugged independently and concurrently. Any changes in one object does not affect the other components.

In conclusion, we demonstrated resilience to change in our program from modularity, redundancy, and encapsulation.

Answer to Questions

Question1:

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?

Answer1:

We could add a counter which is a private field in each block for counting how many rounds a block has been on the screen(not cleared). For example, this should generally act like a queue since it will be first in first out. The block who first stays within the board for 10 rounds will disappear. If we were able to clear one of the blocks, then we reset the counter to be 0. Else, we should find some generated blocks and remove them from the board.

The generation of such blocks could be easily confined to more advanced levels. In advanced levels, the class specifies a certain method of removing a block regarding rounds, which can be modified according to different rules about removing blocks. We could have an indicator of level in each block, which is an integer field in the Block, then the counter only gets operated on certain levels.

Question2:

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

Answer2:

We can use inheritance to create our level classes. The inheritance lets us use the information and methods defined in the GameLevel class from which we inherit. Then all level subclasses, such as level 0, level 1, etc are derived from the GameLevel class. We also plan to make the superclass GameLevel abstract, which provides the ability to accommodate multiple level types. If we wish to add an additional level into the game, all we need to do is to add a new concrete class under this abstract class. This achieves minimum recompilation since only the newlevel.cc will be recompiled.

Question3:

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?

Answer3:

In this case, the decorator design pattern will be the ideal solution. The decorator design pattern allows users to add new behaviors to an existing object without alternating its structure. We can then combine several behaviors simultaneously by wrapping a target object (in this case the Board class) with multiple decorators.

We make the Decorator Class's reference field accept any object that follows the interface. This allows us to invent more effects to the Board, by creating more subclasses with different functionalities under the Decorator abstract class. Hence, if several effects are applied simultaneously, we can add the combined behaviors of all the wrappers to the opponent's board. Notice that the other instances of the same class are not affected by the object with the modified behavior. Each subclass of the Decorator class follows the single responsibility principle. This prevents our program from having one else-branch for every possible combination as each subclass only has one responsibility and we just need to combine them when needed.

Question4:

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.

Answer4:

We could design our system to accommodate the addition of new command names, or changes to existing command names, with minimal changes to source and minimal recompilation using pimpl idiom. By using a Command Interpreter (a new class which can call different commands with corresponding methods needed. In this case, if we want to add a

new command, we can simply just add a new method. For renaming an existing command, we add a rename function for the Command Interpreter class where we use the oldname and newname as the parameter. Thus, only the Command Interpreterroller class and the implementation of the corresponding method will be recompiled without affecting other classes.
We could give a name to a sequence of commands by creating a vector of shortcuts to commands. We could read in the sequence of commands first, then store the commands into the vector starting from index 1 and the name in index 0.

Extra Credit Features

Game Cover:
We created a cover for Biquadris at the beginning to display when graphic mode is enabled. It gives a nice preparation stage for the players. They get to press any key on the keyboard to start the game whenever they are ready.

Uppercase Command Interpreter:
To accommodate different command input preference, we also added uppercase letters as consideration when reading in commands. This gives the flexibility and tolerance for typos.

Final Questions

Question1:
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?
Answer1:
We all learned a lot about developing software in teams after this project and here are the 4 key ones:

- The breakdown of the whole project should be accurate and clear. It is the basis of team success. Efficiency can be greatly enhanced if everyone is clear about the feature their part of code should do.
- The connection between each module should be in low coupling. This will give a big hand to debugging.

- The code style and documentation is indispensable. Each person has their own unique style of writing code but while working on a common project, uniformity in specifications are required. Also, documentation is needed so that everyone else can understand the code to be able to work together.
- Remote collaboration with VS Code and GitHub is the base of team code. A version control system, specifically GitHub, was used to enable remote collaboration. We all had little to no experience working remotely prior to the group assignments, so we took the time to research and acquaint ourselves with these platforms. The learning curves proved to be rather steep, but a lot of time has been saved as a result.

Also, continuous change is always part of writing code. We should always plan for the worst. Improving efficiency and accomplishing team goals as quickly as possible is what any team is pursuing.

Question2:

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

Answer2:

Overall, we would spend some more time on brainstorming. Instead of paying attention to details, we should focus more on the overall design. It is also important to have a command interpreter to test along the implementation.

Creating Pseudo Code Prior to UML Design

We did not explicitly put out any pseudo code prior to constructing the UML for the submission due on the first deadline. To organize the general hierarchy of the programme, we simply brainstormed as a group and created mind maps; however, if we ever had the chance to redo the entire project, we would all prefer to write some pseudo code before drafting the UML because the overall structure would not be fully sorted out until the coding portion was finished. This successfully avoids process loss.

Learn More About Git Beforehand

We were using gitlab to share and manage our project files. Although we've gone through the quick git intro posted on piazza, there were still sometimes when complications come up. For example, nothing is updated after git pull. This arises when we remove some files and try to

git pull the clean version of that file. Instead of solving git issues during meetings, our work could be more efficient by learning Git in the preparation stage of the project.