CS246 Final Project
Biquadris

Junlin Yu
Sylvia Ding
Henry Tian

Introduction:

A game of Biquadris consists of two boards, each 11 columns wide and 15 rows high. Blocks consisting of four cells appear at the top of each board, and you must drop them onto their respective boards so as not to leave any gaps. Once an entire row has been filled, it disappears, and the blocks above move down by one unit. Notice that Biquadris is not real-time. You have as much time as you want to decide where to place a block. Players will take turns dropping blocks, one at a time. A player's turn ends when he/she has dropped a block onto the board (unless this triggers a special action). During a player's turn, the block that the opponent will have to play next is already at the top of the opponent's board (and if it doesn't fit, the opponent has lost). Scores will be calculated base on the number of lines players cleared and their levels. The player with the higher score wins the game.

Overview:

We use Model View Controller Architectural Pattern for this game.

Model:

- **struct Posn:**
    - This struct has fields x and y in order to store the position of cells in 2D board.
- **class Block:**
    - The class Block has 8 derived classes, and each Block owns a vector of Posn to store 4 positions of cells in order to represent a block. Also, we have public methods to operate those blocks, such as left, right down, and so on. Since rotate is different for each derived class, so we choose pure virtual classes to define method rotate.
- **class Level:**
    - The level class has 5 derived class: level0 to level4. We have methods to allow us read file input, and then created blocks. Method createBlock is different to each derived class, so we choose to use pure virtual methods to define this function. Moreover, we have a field to determine if our creation of block is random or not.
- **class Score:**
    - We use this class to store the current score and the highest score. And we have a level class in so that we can add score depends on games' level. Moreover, we have public methods to allow us to access or change those fields directly.
- **class Board:**

    - Class Board has fields current block and next block to allow us to know the block we operate now, and next. And we use a vector of Block to store the block already in the board. Same as above classes, we have methods to allow us directly change or access the fields in the class Board. Also, we have special actions in board and we can control those actions by methods heavy, force, and blind.

View:

- **class View:**
    - The class View is used to display the boards to clients, by either only text or text and graphics. There is a composition relationship between the View class and the BoardControl class. The BoardControl class has a field of View to control the display of the boards in the BoardControl class.
    - The class View includes 2 fields: a Boolean called isGraphic, and a smart pointer of an Xwindow object called xw. The Boolean field isGraphic is used to indicate whether the client runs the program in text-only mode. The default behavior is to show both text and graphics, and the text-only mode only displays text. The smart pointer field xw points to an Xwindow object which is used to do a graphical display using X11 built-in functions.

Controller:

- **class BoardControl:**
    - This class contains two Board. One for player 1, and another for player 2. We use this class to do the commands we received from input such as left, right, down, and so on. We use public methods to finish those commands. Also, we may have a field called round to determine which round is this game in. (player 1 or player 2)

Design:

MVC Architectural Pattern:

In this project, we use MVC Architectural Pattern. In MVC, the program state, presentation logic and control logic are all separated so that we can modify any one of them without modifying the other two. We have BoardControl class for users to input commands and translates it to the actions that stored in the model classes. We also create several model classes such as Block, Level, etc. to manage the game's data and its modification. Those model class doesn't know anything about how these data is presented to the user. We also have a View class that manages the interface to present data. It manages how models' data should be presented to the user, in our case, it manages the text-based interface and the graphical interface. The benefits of this approach are increasing cohesion and decreasing coupling.

SRP (Single Responsibility Principle):
We implement our game by separating the whole project into many classes, each of the classes will have its own unique functionalities and be responsible for a single part of the project. For example, the View class will only handle the TextDisplay and the GraphicDisplay, it will not need to know how the Block class transforms the blocks and create the blocks.

Factory Pattern:

We use the Factory Pattern to implement Level class. We first create an abstract Level class, then we will implement concrete subclasses for each specific level. If we want to introduce additional levels into the system, we only need to add one more concrete subclass. This ensures us that without needing to know anything about how the choice of which block to create is made, we can also change what the Level pointer is pointing to in order to change the level for the game.

Forward Declaration:
We add the forward declaration of the View class to the class of BoardControl so that we can avoid the possible problem of an including cycle. By using the forward declaration, we can also reduce the C++ build times.

RAII Idiom:

RAII Idiom is an important part of our project. By using RAII, resources that must be explicitly cleaned up, such as heap-allocated objects, are bound to resources which are cleaned up automatically. There is no delete in our code, instead, we use unique_pointer and shared_pointer so that our program will not have a memory leak even if an exception is thrown. This greatly reduces the difficulty and frequency of debugging when implementing this game.

Low coupling and high cohesion:
We have low module dependency in our project. Our modules interact with another module through a stable interface and they don't have to be concerned with the other module's internal implementation. We didn't use friend classes or public fields. At the same time, we maintain high cohesion in our project as we follow the Single Responsibility Principle.

Resilience to Change:

In order to achieve high cohesion, we use abstract classes to implement our base classes. We let the concrete classes inherit the abstract classes so that when we want to add some new functionalities, we will only need to add a new derived class. For example, we design our Level class as an abstract class and we create 5 subclasses representing 5 levels.
We also use the factory design pattern for the Level class. The advantage of the factory design pattern is its modular expandability of the application. It provides an interface for object creation but lets the subclasses decide which object to create. For example, our Level class is incorporated with the factory Method, we include a pure virtual createBlock method that returns a Block pointer to one of the eight blocks. The actual Level subclasses will handle the specifics of constructing those blocks. This guarantees that we may alter what the Level pointer is pointing to in order to change the level for the game without having to understand how the decision of which block to generate is made. This will let us achieve the goal of resilience to change.
What's more, we use the Inheritance hierarchy in our project. Our block class is implemented as an abstract class. we define 7 subclasses representing 7 types of different blocks. by using the inheritance hierarchy, if we want to add more blocks, we can easily create new block subclasses.
MVC Architectural pattern will also make our project resilient to change. Our BoardControl class is responsible for the commands users input such as left, right, and down and passes these commands to Model classes. Since we have a separate Controller class, we can easily add a command if we want to have some additional functionalities. Our new command will not affect other commands since we have low module dependency in our project.
We follow the Single Responsibility Principle and implement our game by breaking up the entire project into numerous classes, each of which will be in charge of a specific aspect of the project and have its own distinct functionalities. At the same time, there are few module dependencies when we are implementing this game. Through high cohesion and low coupling, we reached the goal of resilience to change.

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?

We will set up a private field in the board class called counter_uncleared to count the generated blocks. We will also set up a Boolean field in the block class to decide if the blocks are visible or not, if the Boolean value is true, then the block is visible. When the counter is greater than 10, we set the generated blocks to be invisible by changing their Boolean values. When a row is completely filled and needs to be cleared, cleanrole() will be called. In this function, we will reset the counter to 0. We will also have a field indicating its current level in the block class as we generate blocks in different levels with different rules. In order to let the generation of such blocks be easily confined to more advanced levels, we will only increase the counter when the level indicator in blocks is equal to a certain advanced level.

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

We can use the factory design pattern to accommodate the possibility of introducing additional levels into the system with minimum recompilation. The advantage of the factory design pattern is its modular expandability of the application. It provides an interface for object creation but lets the subclasses decide which object to create. We first create an abstract Level class, then we will implement concrete subclasses for each specific level. If we want to introduce additional levels into the system, we only need to add one more concrete subclass.
For example, we could incorporate the Factory Method into our Level class, by adding a pure virtual createBlock method that returns a Block pointer to one of the eight blocks, the rules of creating those blocks will be encapsulated by the concrete Level subclasses. This ensures us that without needing to know anything about how the choice of which block to create is made, we can also change what the Level pointer is pointing to in order to change the level for the game.
When we are introducing a new level called level 5, we just need to implement that new level subclass. The added subclass will have all the necessary information for the new level. We will only need to compile the level 5 subclass.
Since we only need to compile the subclass we just added, this will let us minimize recompilation.

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?

Decorator is a suitable design pattern for this situation. It can let us add functionality or features to an object at run-time instead of adding to the class as a whole. We can add features incrementally by adding new subclasses.
We create an abstract class called Effect and then we implement concrete subclasses for different effects. For example, we can build an abstract Effect class.

We implement a ConcreteEffect subclass and a Decorator subclass containing a pointer to an Effect inherited from the Effect class and a pure virtual method Operation. We can then build a linked list of decoration objects by having a pointer to an Effect. The linked list terminates with a ConcreteEffect object. By using the Decorator pattern, each effect is wrapped through a linked list. This ensures that our program can apply multiple effects simultaneously and each effect will not be affected by the others.
We can simply add more subclasses for more kinds of effects under the abstract Decorator class if we want to invent more kinds of effects instead of modifying existing code.
Before using Decorator, we might need a class for every possible combination of effects. However, with Decorator, when we try to call Operation, it will automatically choose its implementation by class type. Thus, we will prevent our program from having one else-branch for every possible combination.

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 the source and minimal recompilation? 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?

We create a new class to contain all the command methods for the game. It is a controller to manage interaction to modify data. If we want to have new commands, we could simply add new methods to this class. This will let us have minimal changes to the source and minimal recompilations.
For example, we create a BoardControl class to contain all the command methods to control the board such as restart, drop, etc. If we want to add a new command unique, we simply create a new method called unique in BoardControl class. BoardControl will then translate the user interface event using its methods to a game model event. Before using this way, there might be different command methods in different classes, causing multiple recompilations. With our BoardControl being invited, all command methods are in one class and minimal recompilation is achieved.
If we want to add new command names or change to existing command names, we could use Switch to do this, by adding an extra case, we can add a new command name. If we want to change a command name, we can change the name of the case. It is also very easy to edit Switch cases since they are recognized easily. In contrast, If we use if statements, it is difficult to edit since nested if-else statements are used.
In order to support a macro language, we could add an extra method that wraps a sequence of commands and give it a new command name. For example, if we want

to set a sequence of commands as a single command, we could simply add a method called macro to BoardControl class.

Extra Credit:

Smart Pointer:

There is no delete in our code, instead, we use unique_pointer and shared_pointer so that our program will not have a memory leak even if an exception is thrown. This greatly reduces the difficulty and frequency of debugging when implementing this game.

Player’s Manual:

We implement the player’s manual so that players can get more familiar with the commands and rules of different levels. In order to use the manual, the players can type “help” whenever they feel confused during playing this game.

ScoreBoard:

After the game is ended, both displays will display which player wins and their high scores respectively.

Status Light:

In the graphic display, we establish two status lights on the top of the board to indicate the role of the players. The light will turn green if it’s one specific player’s turn.

Final Question:

Question 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?

It’s really important to implement a UML diagram. Before we began to implement code for this project, we first discussed together the main structure of this game.
In order to cooperate as a team for this project, we then distributed our tasks based on our UML diagram. Each of us was responsible for one part of the Model View Controller classes. This is very close to the real-world programming situation in which everyone is responsible for their own class and fulfills the encapsulation at the

same time.

A good style of code is very important. At first, it was difficult to read and edit teammates' codes since we don't have much documentation. We then made clear documentation for every block of our code. This greatly increased our efficiency.

Good communication is the key for a team to reach its goal. In the past two weeks, we manage to deal with some common difficulties together and built a trustworthy team.

One major difficulty for this project is updating our codes to the teammates so that we can debug and implement additional features more efficiently. However, since all teammates in our group haven't used GitHub before, we had to transfer our files through google drive at first. Although we are only a 3-people group and we have perfect communication, we still encountered problems of overwriting past codes. Thus, we studied how to use Git and Git did a perfect job in version control. By using Github, we can easily see the difference between the newly uploaded version's code and the past version's code.

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

If we have the chance to start over, we will learn how to use GitHub in the first place since version control is so important for developing software in teams. Overall, we as a whole team did a perfect job on this project.