Biquadris

Zhenchao Xu

Zhenglin Yu

Ying Ye

Table of Contents

Introduction

Biquadris is a two-player competitive Tetris game. Players play for the honour of the King of Biquadris. Two players take turns to drop blocks onto their boards and earn scores if the dropped blocks fill one or more rows. The number of points earned differs according to the number of eliminated rows and blocks, and the level of game and blocks. The higher the level is, the more the points gained. In each round, our game will record the highest score the players get. The player who has the highest hiScore after several rounds wins the game.

Overview

The Model View Controller architectural pattern is used when programming this game. Model contains the information of the game. View provides an interface to the players, including text display and graphic display. Controller is used to interpret the input from the players and tells the Model to change internal states. We will introduce the modules by their functionality.

- Model Classes
  - ❖ Grid

    Grid is an object that owns 11 x 18 Cell objects.
  - ❖ Cell

    Cell objects are the basic elements of game information. It has information about the state of blinded or not, the position on the grid, and the information of block "sitting" on it.
  - ❖ Block

    Block is a pure virtual class. It has 8 subclasses, including TBlock, OBlock, JBlock, ZBlock, SBlock, LBlock, IBlock, and starBlock. The Block object contains the information about the level it was generated, as well as its state and type, and provides some functions to be manipulated by Panel.
  - ❖ Level

    Level is a pure virtual class and has subclasses from Level0 to Level4. They are factories to generate Blocks, and they can be set random or non-random.
  - ❖ Score

    Score object is used to store the information of the player's score and keep recording the hiScore throughout the game.

- View Class
  - ❖ View

    Class View is defined as an observer of the Panels and provides the players an interface to play; it supports text and graphic displays.

- Controller Classes
  - ❖ Gameplay

    Class Gameplay is used to determine the current player and receive commands from the players. It passes commands to Interpreter to do the actual manipulation.

❖ Interpreter

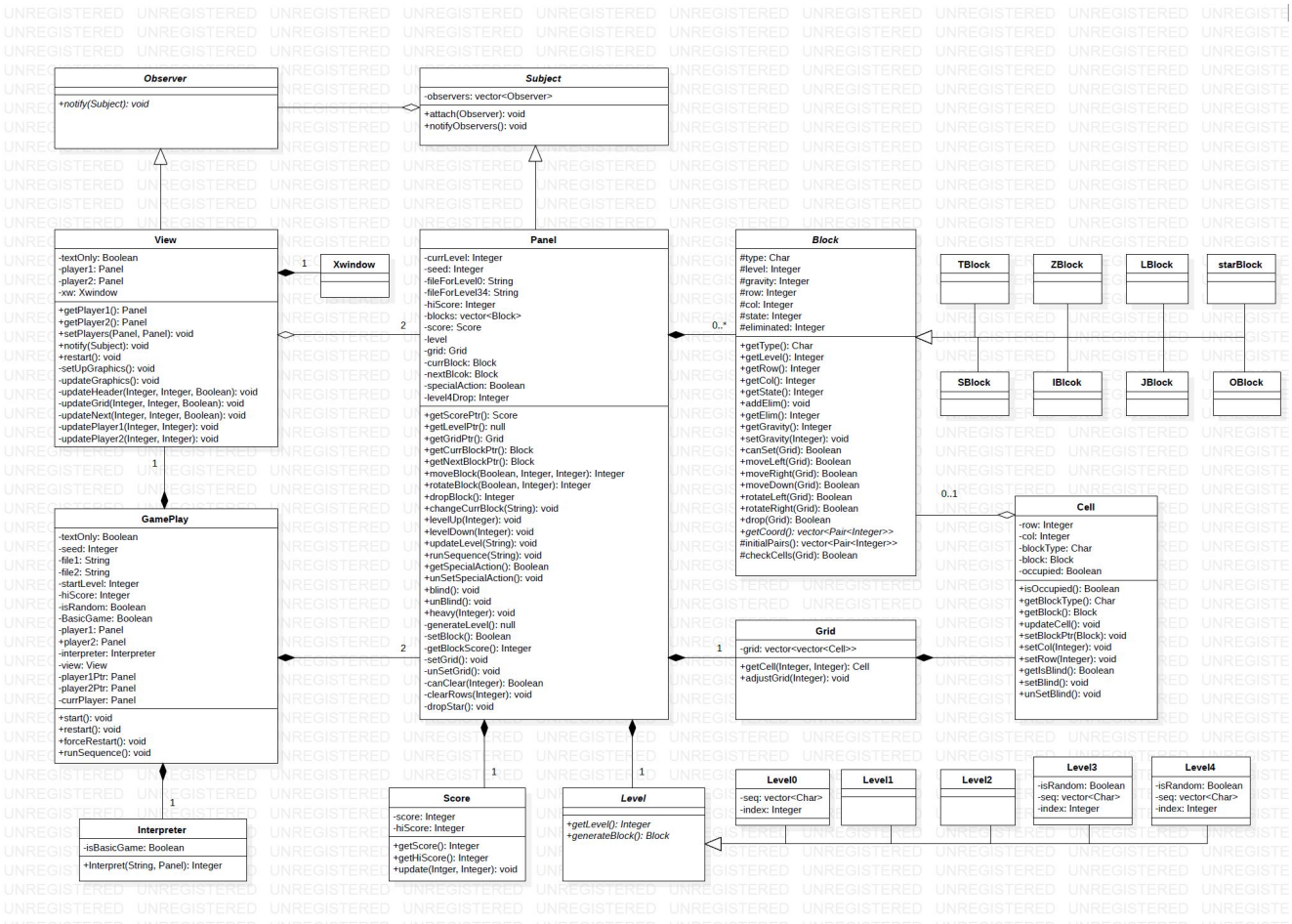
Helper class of Gameplay, the command is passed from Gameplay, and then tells Panels to change the model objects within Panel.

❖ Panel

Panel object is used to manipulate the movement of the blocks, eliminate the filled rows, and changes the state of the Model accordingly. Panel simulates the block moves around the grid, and tells the cells on the grid if the block can be set on the grid.

UML

This is the UML for our final project, we changed some of the features and some relationship between classes. These changes will be discussed below, under design section.



Design

To solve different challenges in the implementation, various design strategies are applied to our programming design. In addition, we changed some of pattern and ideas in our final project.

- Iterator Design Pattern

In the Panel module, the generated blocks are pushed onto a stack; we use the iterator design pattern to find the block we want.

- Observer Design Pattern

The observer pattern is integrated with the MVC pattern. In this design pattern, View object is an observer of the Panel, and the Panel is defined as a Subject. Every time Panel changes its block or the state of the Cells, it will notify observers. Then the View object will provide the players with the updated display.

- Factory Design Pattern

The generation of the Blocks follows the factory design pattern. Every subclass of the Level will generate a unique pointer of Block according to the current level. The level contains a file to determine the order of the following blocks. Every time the Level changes, the order of the blocks starts over. Moreover, the factory can be controlled from the command line such that the sequence of next Blocks can be changed.

- Interpreter Design Pattern

The gameplay object uses an interpreter; if we want to add new commands, it is easy to implement it in the interpreter. And we only need to recompile the interpreter class.

- MVC Architectural Pattern

The Gameplay, Interpreter, and Panel are the Controllers; they are used to change the state of the Model. View class is the View. The Cell, Block, Grid, Score, and Level objects are the Models.

- Polymorphism

Polymorphism is used in our program for both Level and Block. Dynamic dispatch is widely used in the function related to Level and Block. In Panel object, the functions are always called by passing a superclass Level or Block, but eventually, the compiler determines which actual function to be used. Because Level and Block are abstract, the methods will be overridden.

- Non-Virtual Interface Idiom

All public methods are non-virtual, and all virtual methods are private; for example, in the Level class and Block class.

- Single Responsibility Principle

We tried our best to follow the single responsibility principle. Gameplay controls starting game, restarting game, reading inputs, and changing player. Interpreter interprets the command from gameplay, and passes it to panel. The core class Panel contains unique pointers to Grid, Score, Blocks, and Level. However, it only works on simulating the movement and elimination of the blocks. A Grid contains Cells, while a Cell contains its own information. Level generates new

Blocks. Score stores the score and highest score of the Panel. Block contains its own information. View is responsible for the display. Each class has single responsibility, and all of them are designed towards high cohesion.

- RAII Idiom

There is no delete nor shared pointer in our code. Only unique pointers and vectors have been used. Every object is defined as a unique pointer object, and the destructor will clean up all of its heap memory. Therefore, there will be no memory leak.

- Reduced compilation dependencies

Forward declarations are used in the header file, if possible. All compilation cycles are avoided, for example, Observer and Subject.

- No Throw Guarantee

Operations are guaranteed to succeed and satisfy all requirements even in exceptional situation. If any exception occurs, our program will handle it properly. There is no throw in our program; therefore there will not be any throw exception.

- Variance between Due Date 1 & 2
  1. In our first design, we did not use the interpreter design pattern. And we set several observers and subjects, but later on, we found that some of the objects are not subject, for example, the level is not a subject in our program. The view observer will only be notified when Panel changes its field's state. Then we made the Panel be the only subject. The original problem became much easier.
  2. We were using shared pointers, but our instructor said we better use a unique pointer. So, we eventually changed to use unique pointers.
  3. Levels were altered. Initially, we put all of the functions and fields in the super class. However, we found that some of the levels do not need all of the fields. So, we chose to put these specialized fields into the subclasses.

Resilience to Change

- Level Class: Abstract Class and Polymorphism

If a new level, for example, level 5, is added, we can add a subclass Level5 to the abstract class Level and override the generateBlock() method to implement the logic of generating blocks in the new level. Given that the group of level classes only works on generating new blocks, Level classes have high cohesion. Level classes use no other module, except Block, so it the coupling is minimized.

- Block Class: Inheritance Hierarchy

If a new block type is introduced, we can create a new block subclass that inherits from the abstract Block class and overrides the getCoord() method. The movements of blocks are implemented in the Block class for all kinds of blocks, so minimal effort is needed to add a new type of block. Similar to Level, the group of Blocks works only for one aim, that is, to store the information of the block to facilitate the manipulation of Panel. Thus, cohesion is high. It has to pass a grid pointer to indicate where it is. No other modules are used; therefore, the Block modules have

low coupling.

- Score Class: Score Recorder

Score does not need to use any other modules, and it only records the current score and the historically highest score. Therefore, the Score class has high cohesion and low coupling.
If we want to add any other features toward score, only score class has to be modified.

- Panel Class: The Main Controller

If additional operation is added, the Panel can be modified and be implemented with the new feature. The operation is usable after adding it into the command interpreter. Panel needs to use Score, View, Block classes, Grid, Cell and Level, because they are the components of Panel, it has high cohesion and high coupling.

- Interpreter Class: Command Interpreter

If a new command is added or if an existing command is changed, we can modify the Interpreter class to adapt to these changes. We can add whatever new command to the interpreter class, as we want. When we recompile our program, only the Interpreter class will be compiled. The task of the Interpreter class is to interpret commands. Since our program is designed to follow the single responsibility principle, the interpreter is highly cohesive. The Panel module is coupled in the interpreter module. However, we must include Panel in our program because Interpreter has to make Panel change its state; therefore, it is low coupling.

- View class: User Interface

If we want to add or change something in the text-based or graphical display, the only module that needs to be modified is the View class. If we want to add a new window or change any character in the text display. We can simply change very little fields within View class. Methods in the View class are only used to support display; therefore, View has high cohesion. Since we used an observer design pattern, View objects have to include the Panel module and Xwindow module; we cannot get rid of these two modules. Thus, our View class has low coupling.

- Grid class: Resizable Grid

Our implementation allows change to the size of the board since we store gridWidth and gridHeight as constants. When the size of the grid is changed, we can simply replace the value of those two constants with the new ones. Grid is low coupled with Cells because it is made by Cells, and Cells only contains its own information about color, position, isblinded, etc. Therefore, Cells and grid have low coupling.

- Gameplay Class: Expandable for Multiplayer

Since this game is the expanded version of Tetris for two-player competition, the way we handle players is efficient. If the game is to be expanded to facilitate more players, because we have the Panel class for a single player, we need to add a new Panel pointer in the Gameplay class and update the display by modifying View class accordingly. The Gameplay class only works in the processes of the game, including starting the game, restarting the game, changing players, and taking inputs. It has high cohesion. Furthermore, it only needs Panel and interpreter modules; therefore, it has low coupling.

- Summary of Resilience & Low Coupling & High Cohesion

Every module has high cohesion because of our Single Responsibility Principle design, it is very easy to change and add in our modules. We can always find the responsible module and add new features in it. Our module dependencies are low. Only one friend relation is used for text display using operation<<, and the classes have low coupling except Panel class. Though the Panel class is relatively high coupling, it is minimized.

Answers to Questions from project

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?

In our Grid object, we can add a new private method used to check if the generated BlockID is ten times larger than every Cell contained in it. If it is, we set the Cells as plain. Otherwise, Cells do not change. Every time we update our grid, the Cells are checked and updated. To confine to more advanced levels, we need to double-check with the Level in the Panel.

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

We can have an abstract class Level and several subclasses corresponding to different levels. When additional levels are introduced, we can simply add a new subclass that contains the new features. The new class is the only file that is changed, so it is the only one that needs to be recompiled; this would require minimum recompilation. In addition, we could implement the related classes within one file if they are working toward the same function.

3. How could you design your program to allow for multiple effects to 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 design pattern can allow the application of multiple effects simultaneously. We can create an abstract decorator class and then create subclasses for every special action. Decorator pattern allows effects to be applied to the object without affecting the behavior of others. We can add more subclasses if more kinds of effects are invented. Using decorator avoids having one else-branch for every combination because it will wrap up all effects when they are applied.

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.

We can create a class to manage all the commands. We can assign each command a method, and

the addition of new command names just needs addition of methods corresponding to that command. This requires a little change to the code, and only the class of commands needs to be recompiled. To support the command of renaming existing commands, we can call the method of the original command if we get the new command. For example, for “rename counterclockwise cc,” when we get the command cc, we can call the method of counterclockwise. We have the sequence file command that executes the sequence of commands found in the file. To support macro language, we can use the sequence function and give a name to the call of sequence. For example, we can use the command “1lef1ri” to call “sequence 1left1right.txt”, so the sequence of commands “left right” is given the name “1lef1ri”.

Extra Credit Features

We used runtime command to change if we are playing the bonus game or playing the standard game. To implement runtime changing, we have to depend on our Gameplay class; it has a field called isEnabled. Once the user input “enablebonus”, the field turns true, and the bonus game is invoked.

1. We added an extra command called “withdraw”, it will undo the player’s last drop. It is challenging because we have to record every block generated. We have to find the last block dropped by the current player, then find its position on the grid, then tell the cells to change their state. The most difficult part is without forced block. To solve this problem, we popped the current block, then push a new one. Eventually, we got a new block, then notify the observer.

2. We added an extra special action called “clearrow”. The action is triggered when the current player cleared at least two rows at once. Then the opponent’s last row will be destroyed without any score added. Furthermore, the remaining parts of the destroyed block will not generate points for the opponent after being cleared.

3. We added a keyboard listener. Players can choose if they want use the keyboard to play or not. The keyboard mapping is stated in the table. We use this feature to facilitate testing, and improve user experience. The challenge we met is to find out how keyboard send value to the system, and how to make it compile properly. Before implementing this feature, every command has to be input after press Enter, but Now players just need to press a single key. The new library ncurses is not covered in this course, we spent a lot of time to learn new stuff, did a lot of trouble shooting, eventually implement it into our program. The commands are shown below.

↑	↓	←	→	+	-	B	F	H
clockwise rotation	down	left	right	levelup	leveldown	blind	force	heavy
I J L O S Z T		E	X	W	C			
change current block		enable bonus	disable bonus	withdraw	clearrow			

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?**

   The key to the success of developing software in teams is communication. Before starting the development, we have to communicate with each other to share our strengths and weaknesses in order to assign the tasks effectively. A detailed schedule helps us complete the project efficiently since all of us have clear expectations and goals. The first step of development is designing. The program consists of many classes, and different team members can have distinct thoughts on the design. Communicating is crucial in designing, and brainstorming together boosts the process of designing the program, which allows us to learn from each other. During the coding process, we communicate with other members through documenting the code. With clearly documented code, we spend less time on understanding others' code; hence we speed up the process. It is also important to give teammates on-time feedback. We need to make sure the program works well with every newly introduced feature, so timely feedback of testing is necessary. We also need to share our progress frequently so that we can ensure the schedule is followed and offer help to teammates.

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

   The most challenging work is collaborating with teammates and merging our code together. We should probably learn to use Git first and then develop our program. Or, we could provide the header files first and provide the interface for each other, then implement different modules separately.