CS246 Final Project Biquadris

Joon Kim, Wonjoon Shin

Index Page

Introduction

Biquadris is a Latinization of the game Tetris. It is a two-player competitive game where each player will take turns playing until one of the players loses. A player is considered lost when the next block cannot be spawned due to the pre-existing block. The rows will clear once it gets filled by the blocks and the player will gain scores according to how many rows they cleared and what level they were on. If a player clears more than two rows simultaneously, they can trigger special actions to hinder the opponent. Biquadris is not real-time like Tetris and players can take as much time as they want to make a move.

Overview

- ➢ Blocks:
  - Classes for all the Block objects for the board. Contains a vector of Cell pointers to keep track of the blocks on the board. This class manages all the block activities such as moving and rotating.
- ➢ Board:
  - Class for the Board object that players will play on. Contains 2d vector of Cells and information on score and level.
- ➢ Cell:
  - Class for each 'cell' on the Board. Contains information on it's x,y coordinates, type and whether it is fixed in position or blinded.
- ➢ Controller:
  - Class for taking inputs from the users and modifies the model classes according to the inputs. Contains pointer to two boards.
- ➢ Levels:
  - Classes for all the Levels. Contains a reference to a board. Creates the next block on the board based on the board's level.
- ➢ Observer:
  - Classes for the Observer design pattern. The Board is the subject and the TextDisplay is the observer. Changes to the board will notify the observers.
- ➢ Score:
  - Class for the score. Contains score and highscore. This class manages changes in scores and getting scores.
- ➢ SpecialAction:
  - Classes for the special actions. Contains a reference to a board. This class applies special actions to the board.
- ➢ TextDisplay:

- Class for displaying the game as text. It receives notifications from observers and displays the game as the model state changes.

Design

➢ MVC Architecture
- The overall design of this project was done using the MVC architecture. By the single responsibility principle, our project is coded so that our classes only have one reason to change using the MVC architecture. Blocks, Board, Cell, Levels, Score and Subject classes are our model part, Controller class being our controller part, and Observer and TextDisplay classes being our view part.

➢ Observer pattern
- We used the Observer design pattern for our MVC architecture so that it allows multiple views to receive notifications every time the model changes. This pattern would make TextDisplay and GraphicDisplay work together.

➢ Polymorphism
- We used polymorphism to implement Blocks and Levels classes. We have base classes that have methods and fields that are common to all the blocks and levels but specific Block and Level have their own implementation details. This allows us to decide which Block or Level to use at runtime and use those at runtime.

➢ Factory pattern
- We used a Factory pattern for our Levels class. We have an abstract base class and specific level concrete subclasses implement their own logic in creating the next block and what happens in that level.

➢ Single responsibility principle
- Our classes are designed so that each class is responsible for their part of the code. For example, Block class only has methods that impact the blocks without caring about other classes.

➢ Decorator design pattern
- We used the decorator design pattern for our SpecialActions class. We have an abstract base class and an inherited subclass for each action that overrides the virtual applyAct() method.

Resilience to Change

➢ Blocks
- If we wanted to add a new type of block, we just have to create another block subclass and pass the parameters for the base class. We just have to choose the spawning point and it's block id. Making sure the first Cell pointer in the vector points to the "center" cell that the block is rotated about.

➢ Levels

- If we wanted to add a new level, we just have to create another concrete subclass and implement the logic for that level by overriding the pure virtual methods in the abstract levels class.

➢ Command Interpreter

- If we wanted to add new commands or change the current commands, we can simply go to the controller class and change it. If we want to alter the current command, we can just replace the parameter for the if statements in the controller to the new command. If we want to add a new command, we can simply just add a new if statement and add the actions that we want the command to execute.

➢ Score

- If we want to change the way that the score is calculated on the board, we can just go to the board class and change the score to a new calculation. Currently, it sets the score when a row is cleared, but we can set the score in other functions so that the score is set in other conditions too.

➢ Command-line Interface

- If we want to add or change the command-line interface, we can go into the main and controller class and change the strings according to the new command line. We just have to pass the command line as a parameter to the controller method and we can use that command line to execute desired action in the controller class.

➢ Multiple View Class

- Since we are using the observer design pattern, and the display is notified of the changes in the model, we can just create a new View class and take the notifications and display it in the new View class.

Answer to Questions

Q: 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?

A: We could add a boolean and a counter field to the cell that counts how many blocks have fallen after and make the cell empty when 10 more blocks have fallen if the boolean field is true. Then we can make a block subclass that consists of cells that have the boolean value as true and add the counter field by 1 when a new block appears. Generation of such blocks can be easily confined to more advanced levels as we can just limit the spawn of such blocks in lower levels and only make them appear in more advanced levels by simply adding probability of the block on switch statements in more advanced level classes.

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

A: We can add a concrete subclass inherited from the abstract Levels class. The abstract Levels class has the basic Levels fields and methods and we would only need to recompile the logics for the new level and how they create the next blocks for the board.

Q: 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?

A: We can implement the Decorator design pattern that has a general abstract base class and multiple subclasses, each subclass containing the detailed implementations for an effect that we wish to be applied. The subclasses can override a method (i.e. applyAct()) for detailed implementation. Hence using this method, we would be able to apply multiple effects simultaneously, independently, without having to create an else-branch for every possible combination at the board class. To add a new effect we can simply create another subclass that overrides the base class with its implementations.

Q: 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.

A: To add a new command name, we can simply go to our controller class and put a new if branch for the new command and add a new feature under that branch. It would be fairly easy to rename the commands. We could just make the command names string variables and when the rename happens, we change the command variable to a new one that the user inputs. This all happens in the controller and it would only require recompiling the controller class.

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?

A: The main takeaway from this project was learning the ability to develop software as a team with an emphasis on Communication. Communication in particular was a key difficulty. Since we now had our partners reading our code, we had to put in extra effort into making our code readable, using variable names that had been decided beforehand, communicating with each other whenever any change was about to be made.

Overall, we designed the structure of the program together but implemented the classes separately. While this division of labour allowed efficiency when building the project, it also required us to fully trust our partners to be capable of implementing their respective classes with perfection. Testing was obviously conducted, but we also focused on discussing solutions and methods until we were fully confident in achieving our goals. This allowed us to build a solid team relationship as well as trust between each other.
Furthermore, there was also the importance of time management. Since each person had their own schedule and especially with final examinations coming ahead, the first major obstacle was coming to an agreement of a realistic goal. Then we had to assign responsibilities while carefully taking examination schedules into consideration which was a difficulty.

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

A: We would spend more time planning and scheduling our project, especially for distributing responsibilities. Some of the classes such as controller or board could not be implemented before other major classes such as cell were implemented. Hence if the person responsible for implementing the cell class had an exam schedule that was earlier than the others, it resulted in the overall project being shifted backwards by a couple of days. Keeping track of the important dates for each person and assigning responsibilities based on the remaining days would help us significantly the next time we start another project.