

CS246 Final Project

Biquadris

Edward Ryan, Nicholas Suprpto, Warren Elbert
e6ryan, nsuprpt, welbert

Introduction

Biquadris is a Latinization of the game Tetris, expanded for two players. The game consists of two boards where each player takes turns to manipulate their blocks. If a row is filled, the row disappears, and the blocks above move down by one unit. When the board cannot generate a new block, the player loses. Scores are calculated according to the number of lines cleared and blocks eliminated.

We implemented a version of Biquadris with features as specified in the project specification. In addition to that, we added an additional feature that allows players to rename command inputs.

Overview

The structure of our program follows the Model-View-Controller architecture pattern so that they have single responsibilities and achieve high cohesion and low coupling. There are some changes with the initial design, the classes, and the relationship between the classes. The initial design has some errors in the observer relationships.

BiquadrisModel

All of our information regarding the game is stored and encapsulated within the BiquadrisModel class. It contains a vector of Boards and a Scoreboard. BiquadrisModel observes every Board, every Cell in each Board, and the Scoreboard. It acts as the mediator between the Scoreboard, Board, and every Cell to the View.

Board

We applied the Decorator design pattern for this class to implement special actions and easy modification for such features in the future. All information about the board itself is stored within the BaseBoard class, where it contains a vector of vectors of Cell shared pointers which creates a grid-like shape of the board. The Board also has a pointer to the current block, which acts as a cursor and may manipulate it during runtime. Furthermore, the BaseBoard consists of a Level object that encapsulates the function that returns the next block. Changes in the board's state will notify its observers. The BiquadrisModel will manipulate the state of each Board according to the command invoked by the player.

Cell

A Cell object contains a pointer to a Block object, representing the type of Block it represents in the Board. Any change in the Cell's state will notify the BiquadrisModel—its observer. The Board will manipulate the state of each Cell according to the command invoked by the player.

The initial design was for the Cell to observe each Block created that will be stored in BaseBoard. After some thought, it is more intuitive to have a Block pointer in each cell.

Block

The Block class stores information about its current active cells. It consists of a Shape object where it acts as a skeleton for the Block object itself. The Block class also has a pointer to a Board to move and manipulate the Block in various ways. It is a Subject to the Scoreboard and notifies the Scoreboard when it is destroyed.

Our initial design was for the Block to be a superclass of many other types of blocks. After trying to implement, we see a pattern in certain methods and that the difference between the various blocks is their shapes. Therefore, we resolved to have a new class called Shapes that handles storing shape information for all the block types.

Shapes

Shapes store the layouts of every block type of the game with a dictionary and any other information you would need to know about the Shape of every block type.

Our initial design, as described above, does not have this class. We thought that we should create a new class for every type of block. It turns out, having a shapes class is better because doing it this way has higher cohesion.

Scoreboard

The Scoreboard class contains a vector of integers that represent the scores of each player's board. Every time a score changes, it will notify the BiquadrisModel that observes it. Furthermore, a Scoreboard object observes Blocks and Boards to know when lines are cleared in the Board and Blocks are destroyed.

Our initial plan was to let the BaseBoard have Scoreboards with the same amount as the boards. However, it seems more logical and high cohesion to create only one ScoreBoard object with a vector of scores and determine the number of boards in its constructor.

Level

A Level object stores all information required to generate what the next Block should be. There exists information about the current level number itself, random state, input files, and discrete distributions to fulfil its purpose.

Our initial plan was to implement the Level class with the Strategy design pattern. While implementing the design, we see that the only difference between the different levels is the discrete distribution variables that determine their probabilities of choosing the next block. Therefore, we have a dictionary that stores the discrete distributions.

Controller

The Controller class's primary function is to get input from the player from the command line. The Controller has a pointer to the BiquadrisModel, which it will manipulate. When the Controller receives a command, it will run the designated function on the BiquadrisModel to change the state of the game. The Controller also has a map of commands that allows renaming a command during runtime. Moreover, the Controller has a pointer to an istream. The Controller can change the command input during runtime (e.g. Read command from a file). Having a controller class allows us to have high cohesion and low coupling.

View

The View class acts as a mediator/adaptor between a BiquadrisModel and BiquadrisDisplay. It observes the BiquadrisModel for any changes about the game state and processes every information change to be compatible with the BiquadrisDisplay public interface. The View class supports multiple displays for the game since it consists of a vector of BiquadrisDisplay.

Our initial plan was to have the TextDisplay and GraphicDisplay inherit directly to the View class. After trying to implement, a certain amount of information processing is the same in both classes that can be abstracted out. Therefore, we created a superclass called BiquadrisDisplay, where displays can inherit and not repeat the processing of information required. Having a View class allows us to have high cohesion and low coupling.

BiquadrisDisplay

The BiquadrisDisplay is a superclass of TextDisplay and GraphicDisplay. It has a simple interface to update information, and it displays the state of the game to the players. TextDisplay shows the board in the command line, whereas GraphicDisplay does so in an Xwindow display.

Design

In the design of our project, we used several techniques and design patterns to solve certain design challenges.

MVC Architecture Pattern

The MVC Architecture Pattern is used to separate the program logic, presentation logic, and control logic. It helps to differentiate the different functions we have to implement in the BiquadrisModel, View, and Controller class. The Controller class focuses on getting commands from the user and passes the command to BiquadrisModel to manipulate the state of the game. After a change in the state of the game, View is notified, and it will show the changes to the player. It gives us the freedom to update each class without having to change the others.

Observer Design Pattern

The Observer Design Pattern helps our program to implement the MVC pattern and reduce coupling between the classes. The Scoreboard, BiquadrisModel, and View classes are Observers, and the Scoreboard, BiquadrisModel, Block, Board, and Cell classes are Subjects. The Observer design pattern is throughout the game logic to pass information to other classes. In doing so, the implementation of each class has nothing related to the exact behaviour of what's next in the other classes. It supports single responsibility for each class. Having interactions between classes through the Observer design patterns allows us to have low coupling.

Decorator Design Pattern

The Board uses the Decorator design pattern. There is a BaseBoard class and a BoardDecorator class. Each decorator is applied based on the game logic within BiquadrisModel. With the Decorator design pattern, we can easily apply many effects to our Board.

RAII Idiom

We entirely used unique and shared pointers to manage memory. Memory for stack-allocated variables, unique pointers, and shared pointers are easier to manage than normal memory allocation with raw pointers. They help to prevent memory leaks in our program and saves us time in debugging memory management mistakes.

Polymorphism

The BiquadrisModel has a vector of Boards. The Board inside the vector can be a BoardDecorator or just the BaseBoard. This implementation allows us to implement the same functions differently if a special action is activated. Furthermore, the View class has a vector of BiquadrisDisplay—an abstract class—to easily update TextDisplay and GraphicDisplay with the same interface even though their representation and logic are different.

Single Responsibility Principle

Each class is designed to handle their own responsibility. For example, the Controller only handles the command input from the user without having to know the implementation of the command by BiquadrisModel or the presentation by View. This helps us to maintain high cohesion and low coupling since every module focuses on its own functionality. More importantly, by assigning each class to one responsibility of functionality, we have higher cohesion.

Resilience to Change

To begin with, through MVC, we can easily add new input commands in the Controller to interact with the model. We can also easily rename command inputs in the Controller. We could even implement a touch-input controller if we want to and still use the same BiquadrisModel module. The same also goes if we're going to view our BiquadrisModel differently. Since the View class acts as an adapter to the BiquadrisDisplay, we would want to create a concrete class of BiquadrisDisplay if we implement a new display. Significant changes in controller and viewing information would not affect in any way the BiquadrisModel itself. So long as the public interface of BiquadrisModel does not change, View and Controller classes should not care about any of the implementations within BiquadrisModel.

Furthermore, going to the specific implementation of BiquadrisModel, we implemented a game where players can be more than two. In this way, our model is dynamic if we add more players. If we wanted to add more levels, we would only need to change the Level class implementation since it contains the random distributions of each level. If we wanted to add more shapes to the blocks, we would only need to change the Shapes class implementation since it contains the layout of each possible shape. If we want to change the scoring system, we would only need to change the Scoreboard class or possibly attach the Scoreboard to other classes and subscribe to a new event.

In addition to all of that, since we used the Decorator design pattern for the Board class, then it is easy if we wanted to have many special actions applied at once without having to deal with every possible combination of the special actions. Furthermore, since we used the Observer design pattern for practically every interaction between the classes, if we wanted to change specific interactions between the classes, it is easy to do so. We would only need to create a new SubscriptionType that tells us the event that had happened. An object would notify this event as a SubscriptionType, and its observers will listen to such an event and determine whether they care whether a particular event happened or not.

In conclusion, through the single-responsibility principle, every class is modular. We can change the implementation of a class, and other classes that depend on it would not need to care. We could easily do so through the Observer design pattern if we wanted to add interaction between classes. Lastly, it is very intuitive to determine which class deals with a specific responsibility.

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?

Our BaseBoard class contains a vector of Blocks. We can easily add an age attribute in the Block. Either we want to reset the age attribute when a line is cleared or increment it; it would be easy to do through the vector of Blocks within BaseBoard. The block will be automatically destroyed when its counter reaches 10. The Block will erase itself in the board and destruct. It will notify its observers—possibly the Scoreboard—that it is destroyed. Since we know the level the board is currently in as a private attribute in BaseBoard, we can see whether we should implement such blocks' disappearance. Thus, the generation of such blocks is easily confined to more advanced levels. The general idea of this answer is not different from what we had in the previous design document.

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

Our Level class fully encapsulates the generation of the next block. For level-specific behaviour other than on what the next block is, we can implement them in the BaseBoard since it has a level private attribute. The answer to this question changes

from the previous design document because we decided not to implement the Level class with the Strategy design pattern for the reasons mentioned above.

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?

We implemented the Decorator design pattern for the Board class. Therefore, for each effect added to the Board, we can add a BoardDecorator with the corresponding effect. Moreover, each decorator can implement the effects as specified. Implementing the Decorator pattern will remove an if-else condition since it will be a linked list of methods. With the Decorator pattern, we can have multiple effects applied simultaneously and easily add more effects in the future. The answer to this question does not differ from the previous design document.

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 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 these features would have on the available shortcuts for existing command names.

By implementing the MVC architecture, we implemented the Controller class. We implemented a map of commands in our Controller class which stores the command names as keys and values. We can add new commands to accommodate new commands for the game. Moreover, we can rename a command during runtime by changing the key of the command into the corresponding new command. To implement sequences of commands, we can add the sequence of commands into our map with the macro as its key. When the user inputs the macro, the sequence of commands will be read using stringstream. We only need to check if other commands have used the macro. If no commands have the same macro, we can add the sequence of commands into our map. The answer to this question does not differ from the previous design document.

Extra Credit Features

We implemented an additional feature in our program, which is the rename command.

Rename Command

We implemented an additional feature which is the rename command. It allows users to rename a command during runtime. To use the command, the player can enter `rename command newcommand`. The rename command will change the command into its new command in the map inside of Controller, and the old command will no longer be valid. Users must be aware that renaming an existing command into another existing command may result in undefined behaviour.

Final Questions

Question 1: What lessons did this project teach you about developing software in teams?

This project taught us the importance of planning. Making a UML design for the first deadline helped a lot in the process of developing our program. The UML design gave us a basic overview of what classes are needed in our software and how each class interacts with one another through design patterns. Without the basic outline, our work would be a mess since there would be no coordination between us. Moreover, the UML helped divide tasks so that we can work efficiently.

Furthermore, it is essential to have active and frequent communication between team members. Whenever there is a design change, every team member should be notified and understand. It is also essential to know how to use git effectively to develop programs in teams.

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

We could think of a better design pattern for the BiquadrisModel to reduce coupling with the Controller class. We can implement the Controller as an Observer of BiquadrisModel. The BiquadrisModel can notify the Controller when a special action is triggered instead of currently having a pointer to the Controller. Moreover, the View class outputting to the standard output class can be implemented only by notification by BiquadrisModel. Our current implementation of the Controller still has a pointer to the View class, which increases the coupling of our program. Reducing coupling will minimise recompilation, and adding new features will be easier.

Development wise, we would think more carefully about the first initial UML design. We had frequent cases where we decided to change the initial design, and we went back and forth on certain implementations that we had done. We could have saved precious time in having a fixed and correct design in the first place where team members can follow through quickly.

Implementation wise, we would spend more time making our GraphicDisplay more appealing. What's a game if it's not visually appealing? Furthermore, our current implementation of GraphicDisplay is slow. We could make certain procedures more efficient in updating the blocks in the window that pops up in GraphicDisplay. Lastly, we can change the hint feature to be more beneficial for the players.

Conclusion

In conclusion, we were able to implement all the functionality and features of Biquadris in our program with a design that is modular and resilient to change. Moreover, we were able to implement an additional feature to rename commands during runtime. There can still be some more improvements to reduce coupling and simplify the interactions between classes.