CS 246 Spring 2021 Final Project
Biquadris
Design Document

Amy Hwang
John Yoon
Liam Mesrefoglu

Introduction

Biquadris is a Latinization of the game Tetris, expanded for two player competition. A game of Biquadris consists of two boards, each 11 columns wide and 15 rows high. Blocks consisting of four cells (tetrominoes) 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.

Biquadris differs from Tetris in one significant way: it is not real-time. Players have as much time as they want to decide where to place a block. Players will take turns dropping blocks, one at a time. A player's turn ends when they have 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.

Overview

Our implementation of the biquadris games includes many different design techniques used throughout many classes.

There are two classes that show the actual state of the game, **TextDisplay** and **GraphicDisplay**. TextDisplay contains a vital amount of information about the game, like the width and height of the game, the board of both players. It also stores a pointer that points back to the UserInterface that it interacts with. GraphicDisplay is where players see the status of their boards. GraphicDisplay holds many integers to store the width and height of many objects that show up in the game. The class GraphicDisplay also has and uses a pointer to an **XWindow** object to actually display the game, and similar to TextDisplay, it stores a pointer back to the UserInterface.

The **UserInterface** class contains pointers to both TextDisplay and GraphicDisplay, pointers to TetrisAbstract objects that each store everything about a player's game, high score for both players and it also keeps track of which player is going to play next. Basically, UserInterface is the object that brings the TetrisAbstract objects, or in other words, the two separate games together in one class.

The UserInterface class also holds most of the methods that would be used by the players each turn. When called those methods, it uses those methods in the respective TetrisAbstract object in order to execute it. It is the most vital class in the project.

The TetrisAbstract class is an abstract class that has many virtual methods for the game to function. The class is being inherited by two child classes: Tetris and TetrisDecorator.

Tetris class is the largest class that ties many aspects of a single game together. Tetris holds the actual game information of a single player, and it keeps the game working. It has a pointer to the Grid of the game, a map that combines Block objects with pointers to BlockDispenser objects, a vector to keep track of the Block objects inside the game, and many other variables that are useful for the game to work properly. It also has many methods that it inherits from the TetrisAbstract class, which are used to make

the blocks within that tetris move, rotate and drop. Furthermore, the methods in this class are used for special actions such as blind, force or heavy. This class basically controls the whole game of one player.

TetrisDecorator class is a decorator class that is again being inherited by the child class Heavy, which has a variable “heaviness” to keep track of how heavy the blocks are in that tetris game. The reason we used decorator pattern here is because there was a need to change the functionality during runtime of the program, which is the heaviness, which called for decorator pattern use.

Tetris also holds a **Grid**, which keeps track of the grid of the game. The Grid class holds pointers to 0 to 198 Block objects that fill up the grid. The Grid class also has methods that are responsible for checking if a row has been filled up.

Tetris class also holds level objects that inherit from the abstract **Level** class, which are using a decorator pattern to achieve having different levels, and also changing the levels in runtime. There are multiple levels that inherit from the main Level class, and they mostly do the same thing other than Level4, which stores an integer named counter in order to count the number of moves without having a row filled up, in order to punish the player with a block in the middle of the grid.

Block class is used for each block in the game. 8 different block types inherit from the main Block class, including a single block, namely Block_tiny. Block class also holds a Color, a BlockShape, and other variables that define a block. Block class also has many methods that help a Block move, rotate, and change.

A **Terminal** object is for players to interact with the terminal and write code as to what they want to do in the game. It stores a pointer to the UserInterface it interacts with, and has many methods that help the players interact with the UserInterface that the Terminal object is bound to.

The **Command_interpreter** class helps convert any command the players write into the respective full command. The class also recommends the players commands if the command entered does not necessarily point to a specific command.

In some of the files that contain classes listed above, there are some exception classes that are being thrown in case there is a problem. All of them are very specific and arbitrary, so we will not go into details about them here.

These are the main classes that the game functions with. There are some other classes that help with the functioning of the game, but these are the main classes and how they work together.

Design

Polymorphism

- Throughout the program, we tried to use polymorphism for classes like Block and Level. This allows to pass on the basic features in parent class

Observer/Subject Pattern

- We used the Observer/Subject pattern for the blocks and the game to notify each other when there is a change. For example, if a finally falls down, it notifies the game and the game checks if there are any rows that were filled that the block fell down to. There are many other uses of Observer/Subject pattern in the program, it's been used a lot.

RAII Idiom

- We partially implemented RAII Idiom by using vectors and std::strings. This allows us not to deal with memory management.

Pimpl Idiom

- For BlockShape and Grid class, we used Pimpl Idiom. This facilitated us when we make changes in those classes.

Decorator Pattern

- Other than that, we used the decorator pattern to decorate the game itself, namely using it on TetrisAbstract class for the Heavy class, to make blocks fall with each move as required. More mods to the game can be added using this way, but we were only required to add heavy blocks, so we stopped there.

Factory Pattern

- Lastly, we used the factory method in order to generate new blocks to fall down. Since there was a need to create blocks as the game kept going, we found the factory method most suitable for this task. In the BlockDispenser class, each of the dispensers that inherit from BlockDispenser is responsible for creating a new block of each type when called for.

Resilience to Change

As we discussed in the design and overview of biquadris, we have tried to create minimal recompilation when we need to modify features.

Abstract Level classes

The Level class is an abstract class, and all 5 different levels have been implemented by inheriting the Level class and adding the specific feature required (i.e. Level_x). Thus, this abstract class allows us to add new levels easily with different features and challenges.

Block Class

The blocks in the game inherit from the Block class. This means, within the limitations of the blocks being 4 by 4, a new block X can easily be added to the game by adding a Block_X class and a BlockDispenser_X class. Different and more complicated blocks can be added in the future levels.

Controlling Terminal class

Terminal class, a.k.a Controller class is responsible for doing initial set-ups for the game, and it also handles the input from users. By implementing this class, we are allowed to easily add the commands and require minimal changes to do so.

Inheritance in Tetris

Our Tetris class is designed as an abstract virtual class. This implies that by inheriting from the TetrisDecorator class, we can also easily create other properties for the game.

Answers to Questions

Blocks

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?

- To allow blocks to disappear, the most intuitive and easiest method we could think of is to implement the integer counter field in block classes. This counter will then increase as we create blocks and it will be resetted to 0 when it reaches 10 to clear the blocks or the blocks have been cleared (whichever comes first). To see whether the blocks have been cleared or not, we would have to implement a method or boolean in our Tetris class to have this counter be "notified" towards the change. Moreover, to have such blocks be confined to different levels, we must implement another field, such as integer level field so that we could keep the blocks consistent with a certain desired level.

Next Blocks

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

- For minimum recompilation, it is best to create the abstract class Level and create subclasses such as Level_x (i.e. Level_4) which inherits the class Level. With this inheritance method, we would only have to create additional class Level_x and override the common Level methods and add any additional methods we need for specific levels.

Special Actions
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?

- To apply several effects at the same time, we could implement them using decorator patterns. For Conway's life game, one of the questions from our previous assignment, we used decorator patterns to accommodate multiple rules for certain cells. It wraps up the cell with the rules. Similarly, we can wrap up our Tetris of other players with the actions and we can simply add the subclass under the class Decorator if we invent more kinds of effects. The use of decorator will naturally prevent from having else-branch.

Command Interpreter
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.

- To accommodate new command names, we could create a class Terminal which will control all the commands coming from the user. By having a class Terminal, we only have to compile the least file since Terminal would only add one method that corresponds to the new command. It will not be too difficult to adapt renaming since we could just replace the "old" command name to desired one. We have used the controller class in previous assignments, and our Terminal class functions the same way. With this separate class, we create low coupling. As we actually implement the command interpreter, we found out rather than using the Terminal class as interpreter, it is more efficient to create the Command Interpreter class and create the algorithm to analyze the command by each character and find the similar/ or exact one out of the pool.

<u>Extra Credit Features</u>

Command Recommandations
As we implemented the command interpreter, we also added a feature that suggests similar or possible commands meant by the command entered by the players if there exists multiple options (i.e. for 'le', levelup, leveldown, and left will pop up in recommendation).

High Score / Player's turn
We also store the high score from both the players and put it onto both of our displays throughout the program. We also implemented the information message box where it tells the user whose turn it is to play.

Final Questions / Conclusions

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

- Sometimes people may not agree on the same solution for a problem. It's good to discuss and try to understand each other's points. Also, learning to use a remote repository efficiently will greatly boost the performance of the group, since that way, more people can work on different features of the game easier.
- Communication is very important when working with teams on projects like this, because a class you implement might collide with a class someone else implements. Group members should definitely make their intention clear as to what they are working on and how they are planning to go about achieving that.
- Writing stable code is very important. Rather than writing code that works only for that stage of the development, writing code that minimizes coupling and maximizes cohesion can help a lot in the long run.

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

- We would have started earlier and got more work done earlier. Starting early would definitely benefit us, as we had to work really hard during the exam week in order to complete the project. If we started when the project was released, we could have been more relaxed.
- Also, having a reasonable UML diagram. We changed our UML along the way from the one we made at the beginning of this project. Although, of course, some changes can be necessary since it is basically impossible to predict a whole project at this scale, we realized we should have planned better for the challenges we would face. In the end, our finalized UML was very different from the one we designed at the start.