CS246 Fall 2022 Final Project

The Game of Biquadris

Final Design Document

Alex Zhang / William Zhang / Jason Yu

Table of Contents

Introduction

The Game of Biquadris is a special variation of Tetris that is not real-time, giving players as much time as they need to choose where to place their blocks. Two boards, each 11 columns wide and 15 "effective" rows high, make up a Biquadris game. The term "effective" refers to the three additional rows that have been set aside for the ease of rotating blocks but are not included as the effective rows used to arrange blocks for scoring. By entering certain commands that the command interpreter provides, players will take turns dropping blocks, one at a time, to play the game. When a player places a block on the board, his/her turn is over (except for special actions like the bonus for clearing two or more rows of blocks simultaneously, which will have an adverse effect on the opponent's game). The block that the opposing player will have to play following the player's turn is already at the top of the opposing player's board (and if it doesn't fit), then the opposing player loses.

Overview

The implementation of this project maximizes and structures around the benefits of object-oriented programming design. To separate and unite the numerous sorts of items in the game, classes have been introduced (i.e. level, categories of blocks, etc). More precisely, the observer software design pattern is created to see the boards of two players and model the changes taking place to them in order to accept and reflect changes in a clean and flexible fashion. Utilizing the factory software design pattern allows for the union of many types of building pieces, which provides subclasses with the responsibility of creating instances of object classes.

Updated UML

See “uml.pdf” for an updated UML.

Design

We have incorporated OOP design principles throughout the implementation. To clearly outline the blueprint of the program, the best approach would be to give a broad overview of our UML.

The Game class serves as a general idea of how the game of Biquadris is played. As indicated in the UML, the Game class owns two Players classes, a Command class, and a Display class. The Game is first initialized after CMDL class attempts to interpret various command-line arguments. Subsequently, the Game organizes the two Players to make movements as commands request and displays the board simultaneously to visualize the board status in both graphical and texting methods.

The Command class holds three pointers of the Play class in the private instance field (one for each competing player and one more pointing to the current player on the turn), which is for calling corresponding executing methods of the current player after Command processes the receiving demand, involving categorizing the type of commands that are in various formats and extracting the numeric multiplier.

The Player class provides the necessary information about the current player (i.e. difficulty level, score, boolean-type variables for status, etc) with the usages of mutator and accessor while controlling the movement of the Object class and corresponding result including the elimination of the rows of Cells, the score and Level class updating, and the triggered special actions.

The Cell class represents every individual cell in a playboard, which is owned by the Player as an element in the vector of Cells. It has several fields used for determining the position of the Cell in the Board, whether the Cell has been occupied by the Objects, and the sort of the Object if it is covering the current Cell. Fields in a Cell class could be extracted and changed by calling its mutator and accessor.

The Object class is an abstract class, which organizes subclasses of different categories of Objects. The superclass Object contains the basic methods that will be implemented in the same way in its subclasses, such as left, right, down, etc. The mutators in both superclass and subclasses are all returning a boolean type if necessary, clearly denoting whether the process is implemented successfully. With regard to the eight subclasses ("I", "Z", "S", "T", "O", "L", "J", and "single"), each defines its own rotating and inserting methods and accessor by overriding the pure virtual methods declared in the superclass Object. Similarly, the Level class is also an abstract class, such that the level subclasses (zero to five) will generate blocks with different patterns as desired via defining their methods specifically.

Finally, we get to main.cc, which acts as a controller of our program and handles the acquisition of inputted resources. A CMDL class is designed to manage the options on the command line, and a Command class is used for taking in the supported commands of the game.

In the design of the project, we used several techniques to solve design challenges:

Observer Pattern: The ABSPlayer class is the abstract subject class, from which the Player class inherits as a concrete subject. The ABSDisplay class is the abstract observer, and TextDisplay and GraphDisplay are the inherited concrete observer. When a subject changes state, both display classes are notified and update changes to the screen.

Factory Pattern: We defined a separate Level class which is an abstract class to create the object. There are different concrete subclasses, such as LevelZero, LevelOne… Those classes implemented their logic or randomness of creating new objects so that it is flexible if new levels need to be added.

Model-View-Controller Architecture Pattern: The users will use the Command class to input commands that manipulate the Model classes, including the Game, Player, Object, Cell, and Level. The Model classes handle the data and update the Display classes, which present the Models' data to the user.

Code Flexibility

We have designed the program in a way that makes adding and removing game functionality easier. A few examples include the addition of new Object and Level. Particularly, the overall architecture of our application adheres to the principles of minimum coupling and high cohesion and object-oriented software design.

Under the Object base class, corresponding subclasses are created for each type of block. For Level, similar implementations have been introduced. By this methodology, creating a subclass under the appropriate base class is all that is needed to add a new block or level. Since the majority of the methods may be directly inherited or overridden, it would be beneficial from an implementation and efficiency standpoint. Furthermore, the factory method allows each level to generate different types of blocks, which further simplifies this process.

To highly realize encapsulation, which is undeniably an essential part of software programs taught in our class, we have made an effort to employ classes rather than structs and provide less public methods. Each class's characteristics are all private (the most paramount advantage of class compared with structure), increasing software security because only its subclasses may access them if necessary, which avoids external modification and even access.

Global variables are avoided as well, preventing any modules from sharing global data that would potentially change the value or state by mistake. By effective practicing the four types of

relationships of classes, our program minimizes unnecessary shared information which better encapsulates the individual classes as well as making the classes easier to manage.

Thus, we have demonstrated why our program is adaptable to change, such as addition and subtraction.

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?

Answer:

Have the abstract class of Objects be the parent class of two more subclasses: RegularObjects and DisappearingObjects. This allows DisappearingObjects to have additional properties. For example, having a counter for the number of objects placed after the current block was placed allows the block to delete itself once the counter reaches zero.

Further extension of these two subclasses with the seven possible shapes of the tetris blocks. This allows the implementation of the factory method as well as allows the ratio of RegularObjects and DisappearingObjects to fluctuate based on the difficulty, or can generate specific blocks due to special actions. Using this method allows the RegularObjects and DisappearingObjects to have their own unique library of possible shapes, and it can be relatively easily implemented by adding more subclasses to RegularObjects and/or DisappearingObjects. This method also allows the properties of DisappearingObjects to be easily changed (blocks fallen before disappearance, unique letter/color representation, unique score calculations, etc.)

Question 2:

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

Answer:

By implementing the level, such as LevelOne and LevelTwo, as subclasses of an abstract Level superclass, we can easily add different levels by building subclasses and can alter corresponding aspects such as block generation, special properties, etc. This way, only the new subclasses, and the main function need to be compiled, and the other parts of the program can remain the same. Moreover, we introduced the factory method, by which each level can generate

its objects as instruction requires, with the pre-set probability of distinctive types of block occurring randomly for instance.

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?

Answer:

Although the Decorator design pattern could be used, since the special actions affect different aspects of the gameplay (display, game mechanics, etc.), it can be difficult to implement. Another possible solution to this is to add boolean fields to the Player class for each type of special effect (isHeavy, isBlind).

Simultaneously, we could make the commands of the moving option return a boolean type indicating whether the efforts have been implemented successfully, which provides a straightforward indicator for other methods to confirm the status of moving actions.

Additionally, when the commands that are affected by these special actions are triggered, a single if-statement can check for each type of effect, and it eliminates the need for an if-else statement for every possible combination. If more special actions need to be implemented, we could simply add more boolean fields into the Player class's private fields.

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

Answer:

Firstly, to implement the features required (rig -> right), a vector of strings is needed as a reference for all possible commands. By looping through the vector, we can see if any substrings of commands of length n (where n is the length of the user input) match the user input. Using this method, adding new commands can be easy as we could easily add more commands into the vector of strings. To execute these commands, however, can be more difficult. Since we don't

know the classes/functions needed by the new class, it can be troublesome. A possible solution is to have the command class implement an aggregation relationship with all the classes, and set up accessors and mutator commands to allow for the commands to be more easily implemented.

To allow users to use a macro language, a map could be implemented, where the key would be the macro for the command, and the value would be a vector of strings, with each string being a valid command. Since the keys must be unique, it prevents the users from using the same macro for two or more sequences of commands. This way, the command interpreter could simply check the map to see if the input is a macro and execute commands accordingly.

To allow users to rename commands, we could implement another map where the key would be a valid command, and the values would be the names given by the user. When the user enters a command, the command interpreter would first check to see if the input is a macro and if so, it would execute those commands. If it is not a macro, we could use a for loop to compare the input with valid commands as well as renamed commands. If the input does match, then that command would be executed.

Extra Credit Features

A resurrection mechanism:

We introduced a chance to resurrect a player who has failed the current game. The current player is considered to have failed the game if one does not have a legal move for the new block. In exchange for losing half their points, we give him/her the opportunity to continue the game by eliminating ten rows of blocks (counting from the 4th row from the top, which is the first "effective" row, to the 13th row).

Testing mode:

To distinguish the testing mode and formal playing model, we added one extra command-line argument, -enabletest, which enables actions of changing block type, restarting the game, sequence, random, and no_random, which would be otherwise forbidden.

Command renaming:

For command input convenience, we support the renaming functionality, by which a player could customize a preferred name for the particular command. For instance, one would like to rename counterclockwise as “ccw”.

Added special actions:

The following options are available when a player successfully eliminates two or more rows at once:

- Double his/her next score earned to the total: addition of 8 if originally getting 4
- Invalidate the opponent's next score: no matter how many points the opponent should have scored next, he/she will end up with zero points added to the total

Final Questions

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?

This project is structured in a thorough and educational way so that we are enabled to leverage what we have learned in class and develop our interpersonal and soft skills for future career paths while collaborating within teams. The following are what we learnt from completing this object-oriented project in teams after thorough introspection:

Conduct Effective and Comprehensive Unit and Union Tests on Key Functionalities:

Unit and integration testing of the project's major features would yield a significant return on investment because it is rather large-scale ever since we have got exposed to computer science assignments. Multiple unit tests are run throughout the testing, debugging, and polishing stages to verify the accuracy of the current components and maximize the highest efficiency of proceeding stably without debugging frustration when the implementation of methods is getting large.

More precisely, the performance testing procedure has been improved with the usage of the grey-box testing methodology. To evaluate the program's robustness, massive test cases, including edge cases and error handlings, have been constructed.

Collaborate Effectively in Both Communication and Code Writing

All members of the group have gained a deep understanding of the value of effective communication and task distribution within software teams. Through this valuable opportunity, we have obtained the beneficial experience of utilizing GitHub, which helped us save a lot of time collaborating remotely and synchronously.

Additionally, we experienced the flexibility of working in small groups, increasing our initiative in learning and working, and enhancing motivation and team spirit. In terms of running software

design, we felt the efficiency of understanding other people's code through documentation, and we further realized the great security and stability that encapsulation and cohesion brought to projects with large amounts of code.

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

The following are the things we would have changed if we could have started the project over again:

Pre-writing Pseudocode Before Designing UML:

We did not explicitly put out any pseudo code prior to constructing the UML for the submission due on the first deadline. To organize the general hierarchy of the program, we simply brainstormed as a group and created mind maps; however, if we ever had the chance to redo the entire project, we would all prefer to write some pseudo code before drafting the UML because the overall structure would not be fully sorted out until the coding portion was finished. This successfully avoids process loss.

Synchronize Coding Styles :

Group members have different preferences in terms of coding and documentation styles, thus in some cases, there is a need for secondary confirmation or discussion of ideas with members, resulting in unnecessary time wasted in understanding other members' code. If we have the opportunity to recompile the code, to ensure code maintainability and documentation consistency, we would negotiate and agree upon the style to employ amongst all three. After the entire program is coded, each group member is responsible to recheck one's code and style.

Conclusion

In summary, this project does an excellent job of synthesizing the knowledge acquired in CS246. Throughout the course of the project, we had the opportunity to employ a number of object-oriented programming design patterns and collaborate on a software project. The project's design meets the standards set out by the NVI idiom for ideas like polymorphism, low coupling, high cohesion, and others. We could not only improve our understanding of the core concepts taught in this course, but also develop a solid project experience that would be beneficial to our future career development.