Final Project: Biquadris

Schedule and Responsibilities

Group Member	Content	Estimated completion date
Natalia, Kenisha, Shirley	Finish the first version of UML diagram for the program and split the tasks	07/26
Natalia, Kenisha, Shirley	Create tests to test our program	To be completed while we work on the program
Natalia	Finish the skeleton of main.cc (mainly the command interpreter), so that the program is able to run Enables basic testing when adding new features to the program	07/27
Natalia	Finish specialAction.h, specialAction.cc, level.h and level.cc (all levels)	07/27 – 07/29
Kenisha	Complete the main parts of board.h, board.cc, textdisplay.h, and textdisplay.cc	07/27
Shirley	Finish the first version of block.cc, block.h, cell.cc, cell.h and all header and implementation files for each of the blocks (eg. SBlock.cc, OBlock.cc, and so on)	08/01
Natalia, Kenisha, Shirley	Complete Level 0 (including special actions -- specialaction.cc, blind.cc, heavy.cc, force.cc and all header files) and make sure everything compiles	08/01
Natalia, Kenisha, Shirley	Implement Level 1 and 2	08/02
Natalia, Kenisha, Shirley	Implement Level 3 and 4	08/04
Natalia	main.cc revisit. Write codes to support additional command-line interface	08/04
Natalia, Kenisha, Shirley	Complete graphical display	08/5-6
Natalia, Kenisha, Shirley	Make sure everything works properly, spend some time on debugging if needed. If we have extra time, add in some extra	08/7-9

	features	
Natalia, Kenisha, Shirley	Complete design document	08/10

Introduction

Biquadris is the two-player version of the game Tetris, with various extra features including letting the players to negatively influence the opponents' game, and playing the game at different levels. The game consists of two boards, with two players dropping tetrominoes one at a time in turns. The score of a player is calculated by the number of lines and the types of blocks being cleared. If there is no room on the board for the player to place the newly generated block, the game is over for that specific player and his/her opponent automatically wins.

Overview

Cell

The Cell class is the basic component of a board, which represents a single square on the board. A board contains a vector of vectors of Cells, to simulate a board with 11 columns and 18 rows (15 normal rows plus 3 reserved rows). Each cell contains information including its coordinate on the board, whether it is occupied by a block and the level when it is created.

Blocks

The purpose of the Block class is to stimulate different types of tetrominoes, which are blocks consisting of four cells. The abstract base class Block has some accessor and mutator methods implemented that can be used by all of its derived classes, including Jblock, Sblock, and so on. Each derived class stimulates a different type of block, and the rotation details for each angle (0, 90, 180, 270 degrees) is implemented in each of the derived classes.

Board

The Board class models a board with 18 rows and 11 columns. It includes the method of moving the current block, calculating scores, changing the text display and graphics as the game proceeds. When playing the game in main, two boards are created at the same time to represent two players.

Next Block (Levels)

This part is implemented using the Strategy Design Pattern. Class board has a class Nextblock which is an abstract class. New level is attached through a public method in the class board. A new block is generated based on the pointer to NextBlock stored in each board. Class NextBlock has two protected fields which can be used directly by subclasses of NextBlock, levelThree and levelFour, and has two methods, one for generating a block and one for changing randomness. The two protected fields are used for storing the status of randomness. Overridden method of changing randomness in level 0 to 2 has no effect. Level0 produces nonrandom blocks. Other levels produce random blocks. "norandom" command only applies to level3 and 4. Level3 carries heaviness of blocks and level4 may generate star blocks under certain conditions. Level is created as a raw pointer and will be deleted in the destructor of the board.

Special Actions

This part is implemented using the Strategy Design Pattern. Class board has a class SpecialAction which is an abstract class and has a public method applyAction(). A new action will be attached to the class board if any player clears at least 2 rows in one drop of block and will be deleted once the opponent drops the current block. Note that method applyAction() really does nothing because actions except heavy only need to be applied once. Therefore, actions except heavy will be applied immediately when condition is satisfied.

Scoring

Scores are calculated using methods in the board class to check for filled rows and cleared blocks, and calculate the obtained score based on the level when the blocks are created. Scoring does not use any design pattern. The hi-score of the players will be kept. Each time the current score exceeds the hi-score, the hi-score is updated and the current score goes to 0 if the game is restarted. Messages about the highest score and who wins will be printed when any player loses. The methods for calculating scores and all fields related to scores are private fields in class board. However, note that instead of adding scores when a single block is cleared, additional scores will be added if all blocks with the same type on the board are cleared. This new score is calculated similar to the method a single block is cleared, except that the level of block is determined based on the left bottom remaining block having the type of block that is going to be cleared. This is different from the requirement in score because this requires the field vector<vector<Cell>> in class board to become vector<vector<shared_ptr<Cell>>>. This change will cause too many adjustments in other sections of code, especially board.cc and view.cc. It is possible to implement the additional score calculation correctly, but we don't have enough time to make this become true at this point. You can check the method detectRow() in board.cc if you want to take a closer look at the code.

View

The purpose of the View class is to be able to implement the graphical display. This is implemented using the Observer pattern, since each board has a pointer to the same View object (which is instantiated in main) that is notified whenever a block is dropped or moved. This way, the board does not have to be redrawn each time a block a move is made, which improves efficiency as well as user experience, since there is less wait time for each block to be drawn. Updating the score or level is also a very quick process, as the pointer to the View object will be notified whenever there is a change, and this also does not require any parts of the rows or columns to be redrawn.

TextDisplay

The TextDisplay class is used for printing out the text version of the game. It includes a pointer to the opponent's Board so that we can access the opponent's data (their score, level, the current setup of their blocks, and their next block) to be printed.

Design

Polymorphism

In this program, the block class and the level class are abstract base classes, in which different types of blocks and different levels are inherited from these two base classes, respectively. The virtual

member functions in the abstract class allows us to override these methods in derived classes. Polymorphism and inheritance gives the program the ability to accommodate multiple types under one abstraction. For instance, a call to a member function will cause a different function being called depending on the type of object that invoked the function. Functions such as clockwise(block) will execute different functions based on the type of block (Sblock, Zblock, etc.) that invokes the clockwise(block) function.

The Strategy Pattern

Special Actions and Next Block parts both use this pattern because levels and actions change as the game progresses. Fields like bool set_seed may vary depending on the command randomness. Therefore, the algorithm used to return blocks of level 3 and 4 may vary. Also, each level (except level 3 and 4) and each action have no common behaviors. Therefore, Strategy Pattern is a good choice to implement these two parts. Since the superclasses NextBlock and SpecialAction are pure abstract classes and their methods are overridden by their subclasses actions and levels, the specific applyAction method or generateBlock method of each action/level can be called by simply using the pointer to NextBlock/SpecialAction. Class board also has a private field int level_n which stores the current level. Thus, whether or not to apply heaviness (move block 1 row downward) is determined by checking if current level is 3 or 4 in move methods. When a player moves block, especially in left and right, the existence of class heavy will be determined using dynamic cast. If the return value is not nullptr, apply heaviness. When a block is dropped, pointers to action will always be deleted and be assigned with nullptr. Pointers to NextBlock, current block, next block, and SpecialAction will be deleted in the board's destructor so that there is no memory leak.

By using a strategy pattern, these two parts are high degree of cohesion and low degree of coupling. They have a high degree of cohesion because classes next blocks and special actions are only responsible for implementing methods of generating new blocks or changing randomness, and applying actions respectively.

Class NextBlock has a low degree of coupling because it only uses class block to return a specific type of new block. It does not require any fields from other classes. Class SpecialActions really doesn't do anything. It has high cohesion and low coupling because it mainly supports the functionality of the program as an identification of an action and only needs to make adjustments in the method of applying actions on board.

High Cohesion and Low Coupling

In order to lower the coupling among different modules, we considered the following features in our implementation. First, except for output operators, none of the classes in the program has friend classes or friend functions. This lowers the coupling among different classes and protects the encapsulation of the program. Second,

Resilience to Change

The NextBlock part is highly resistant to changes. For each adjustment of the probability of a type of block, only one or two fields of class level need to be changed. If there is a new rule for a specific level, changes are needed only in the generateBlock method in that level class. If there is a new level

added to the game, only a new class needs to be implemented in the level module. Nothing needs to change if class block or any other classes has any new feature.

SpecialActions part is resistant to changes. If a new action which disappears after a block gets dropped is added to the game, changes are needed only in a method addAction. Changes might be needed if the new action changes the way of movements. If current actions need to last for several rounds or even last until game is over, adjustment can be added to the method applyAction in the subclasses of SpecialAction and call this action->applyAction (action is a pointer to SpecialAction) in the corresponding method of movements. The program knows which applyAction of which action needs to be called because SpecialAction is a pure abstract class and all its methods are overridden.

Answers to Questions

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?**

   To allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen, we could use the observer pattern. Our Board class could have a vector of Blocks, where we push back every Block that is dropped. Board could also have a counter field that is incremented for every block dropped (i.e. whenever Board::drop is called, we increment the counter). Whenever the counter is divisible by 10, we can notify our TextDisplay. Our TextDisplay can 'get' the Block (via an accessor method) at the front of the vector of Blocks so that it can erase that Block in the display. Once we return to the method Board::drop, we can remove the Block that is at the front of the vector of Blocks that was just erased from the board.

   If we use this design, then the generation of such blocks can easily be confined to more advanced levels, since it would not be the block's responsibility to know when they should be erased, it would be the board's responsibility.

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

   When introducing an additional level, we only need to add one more concrete class under the NextBlock abstract base class (ie. the class that represents the new added level is inherited from the NextBlock abstract base class), and override the function generateBlock() so that we can generate an instance of the new added class whenever we need to. This only requires us to recompile the header and the implementation file related to the new added class. For instance, if we added a level named Level5, then only Level5.cc and Level5.h require recompilation, while files for other levels stay the same.

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?**

   In order to let multiple effects occur at the same time, we can use the decorator design pattern where each effect is a unique class inherited from the decorator, and the decorator is inherited from the abstract base class AbstractBoard. Thus, effects can be wrapped around the created Board object at runtime to achieve the goal that multiple rules are applied simultaneously on a board. At the same time, the

decorator design pattern avoids the case that the program is required to have one else branch for every possible combination, since all rules can be added at runtime and there is no need to use control flow to determine the combination of methods that should be applied to the board. If more kinds of effects are invented, we only need to add more corresponding classes inherited from the Decorator, implement the corresponding rules for each effect, and only recompile the header and implementation files related to the new effect, while files for other effects stay the same.

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.**

If there are additional command names or changes to existing command names, I will add a new else if statement or adjust an old one in the loop of command-line argument. If there are two commands that will interact with each other, I will loop in the command-line until I find these two commands and apply them first. Therefore, I only need to change the command-line argument section of main.cc and only need to recompile main.cc. If there is a rename command in arguments, I will check the next two arguments and store the third string in string s. While checking whether the input is "left", "counterclockwise", and command names, I make the checking statement into "or" and will let the program also check whether the input is equal to string s1, s2 …. s11. The initial value of s1...s11 are the original command names. They may be changed if there is a rename. I don't need to change features for "I", "J", etc because these are already in the macro languages. Therefore, users can use macro names in further input. They can also use old command names.

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?**

In this project, we discussed the overall structure of the program together, including drawing the UML and deciding what design patterns we will be using. Then we split the tasks such that each person is responsible for implementing several modules, before we put all of the code together for compilation and testing. Overall, we worked efficiently as a team and kept frequent communications among group members. However, there are two major lessons that we learned by working as a team to develop software.

First, most of our group members are not particularly familiar with Git or Github,

which is a setback to our software development process. Although we created a Git repository to manage all project files, some of the advanced features of Git such as creating a Git pull and merging branches are almost never being used. Hence, we did not use Git to its full extent due to lack of experience. However, Git makes our file management much easier since all of our changes are updated immediately, and this greatly helps us with our communication. We learned that skills related to Git are vital both in school and in workplace, since it keeps

What's more, we learned that frequent communication is vital for the software development process. Although we determined the overall structure and design patterns we will be implementing in the program, there are still many details that might significantly impact the overall implementation that are not discussed in a timely manner. For instance, we did not discuss whether we should use vector<vector<cell>> or vector<shared_ptr<cell>> to stimulate the board, and this has caused a lot of discrepancies in our later implementation. As a result, we spend hours fixing the implementation for other modules in order to accommodate this change (from vector<shared_ptr<cell>> to vector<vector<cell>>). We learned that it is vital to determine the implementation details as early as possible, and frequent communication is necessary in the software development process.

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

First, we can compile our code as we are writing it instead of compiling all of our code after we have almost finished all the modules. This will make our process more efficient since modules are easier to debug than the whole program. Also, we should test our existing code before proceeding to write the next module. Second, we can spend more time learning more Git features so that we will be able to use advanced features including merge branches, creating pull requests, and so on. These features of Git can greatly help the communication among group members and help us to manage our project files easier.

Another thing we would like to do if we had more time is figure out a way to make the graphical display take less time to finish being drawn. For example, whenever a block is moved, it redraws each cell separately. If there were a next time, we would try to figure out a way (for example, maybe by experimenting with different graphical libraries other than Xlib) to get the entire block to be redrawn all at once.