CS 246 Final Group Project

Biquadris

Jenny Zhang, Tina Fang, Tom Pan

Introduction

The game of Biquadris is a Latinization of the game Tetris, which is an asynchronous version of Tetris. A game of Biquadris consists of two players, and each player owns a board with 11 columns and 15 rows. For each player round, a 4-cell block appears at the top-left corner of the board, and the player has the full control over that block. If the block cannot fit in that position, the player loses.

Since Biquadris is asynchronous, each player has unlimited time to place this block on their board. Players will take turn to drop the block, one at a time. One player's turn ends when he/she drops a block. Once a row has been filled, it disappears, the blocks above move down by 1 unit, and the corresponding player gains score. Player with a higher score wins.

Overview

The project's overall structure design strictly follows the object-oriented programming principals. Classes are implemented to separate responsibilities of a game in such a way that maximizes encapsulation and cohesion while minimizes coupling.

To make sure each player's board is resilient to changes in the game, an Observer design pattern is employed. In addition, a Factory method is used to make sure the game handles varying situations in different levels in a clean and efficient way.

Smart pointers and standard library (vector and map) are used to ensure the memory allocation is flawless and enhance the clarity of the codes.

Technical details and reasonings about these choices are included in "Deign", "Resilience to Change", and "Answer to Questions" sections.

Updated UML

See "uml.pdf" for an updated UML.

Design

We consider the command interpretation, game operation, game execution, and the visual display as four separate parts of the game (i.e. the displayers do not need to know the logic and rules behind the game; the game operation does not need to worry about how the game is presented to players; command interpretation doesn't need to know how the game operates and executes the commands) Therefore, the entire project is separated into four parts

- the command interpreter that receives inputs, the game that defines rules and behaviours, the board that executes the commands, and the displayers that present outputs. This design minimizes the coupling in this project.

Input and Output:

Whenever the game needs a command from the player, the game class requests a command from the command interpreter, handles that command based on the rules, passes the command to the board for execution, and notifies the displayers to update the visually presented game board.

Command-line arguments are taken by the main.cc, and main.cc directly handles them before initializing the game. Unlike the command-line arguments, input commands are processed separately in the command interpreter.

For single command, the command interpreter first separates the string into two components, the multiplier and the actual command (if there is no multiplier in the input string, assume to be 0), and then checks if the actual command is a valid command. If so, combine the multiplier and the command into a string and return it.

For multiple commands that need to be executed consecutively (sequence and macro commands), a vector of strings is used to store valid commands to be executed. The command interpreter stores all inputs to be executed in the vector. Whenever the game requires a command from the command interpreter, there are two cases: the vector is empty and the vector is not empty. If the vector is empty, the command interpreter asks the player for input and returns the valid command (if not valid, return an empty string). If the vector is not empty, the command interpreter directly returns the valid version of the command from the vector. Please find Question 4 for more detail about how the command interpreter handles inputs.

Whenever there is a change in the game, both the text-based and the graphic displayer need to be notified so they can update accordingly (if -text is not included in command-line arguments). Hence, we employed the Observer design pattern to display the game board with the Game being the concrete subject, and the displayers being the observers. Whenever there's a change to the game, the text-based and the graphic displayer will be notified. The displayers will access necessary information through accessor methods that the Game provides, and display the game board via text or graphics.

The Game:

The Game defines the rules, logic and behaviours of the game. A Game consists of 2 boards, each representing a player. Each board has a current Block that can be moved and rotated, a 2d array of cells that build the board and the blocks, and a Level that generates the next block.

Any movement of a Block would require a valid destination of the cells in the block. If the requirement is satisfied, changes are made by mutating the character and mother block of the corresponding cells.

Cells are the underlying units/elements that make up the entire board. A cell stores its coordinate, character and mother block. It provides the check method that allows the blocks to determine whether the cell is valid for block movement.

The level object is responsible for creating the next block. Since different levels have different strategies for constructing new blocks, we decided to employ the Factory method. The abstract Level class provides an interface to the Game for block creation, and the subclasses decide which block to create. Please find question 2 for more details on Levels.

Resilience to Change

The program is built using Object-Oriented Programming. It is extremely resilient to changes, and it's resilience can be proved in the following three criterias: coupling and cohesion, resilience to syntax and rules changes and minimal structural modification and recompilation.

Coupling and cohesion:

We designed our program in a way that maximized cohesion and minimize coupling. When it comes to analyzing the level of coupling in our program, each class has it's own .cc and .h files with it's .h files being included where necessary. This strengthens the resilience of the program as the change is only need to be made to a specific class/module rather than the whole file. For example, a game class owns two boards, and a board class has two blocks pointers as private fields. If a change is to be made on the left function of block which moves the block to the left,then only the function left in file block.cc needs to be changed while the .cc and .h files of board and game remain unchanged. Unrelated parts of the program are separated where necessary.

To maximize cohesion, various private and public functions are constructed with each focusing on one goal. The functions within one implementation file are designed to work toward a common purpose. Instead of putting all code in function, the tasks are broken down into small tasks with each task being completed by one member function. For instance, in block.cc the public function left is responsible for moving the function to the left while the public function right is responsible for moving the function to the right. Although they have different functionality, they work towards a common goal which is moving the block to the desired position. With OOP being utilized, the public and private functions also work together to implement different actions and behaviours of the class In this way, not only is the cohesion maximized, it also facilitates encapsulation (instead of working with the fields we can assign the public methods to complete the missions for us) which also makes the program easily manageable.

Resilience to syntax and rules changes:

When it comes to the addition and subtraction of rules or features, abstract superclass comes into play. We use abstract class for various purposes, and one of the most important reasons is that it can be easily changed. We use virtual functions when applicable these functions can be overriden in subclasses, and if overridden is not necessary, it avoids repetitive code implementation subclasses as they can just use the virtual function in the parent class. If, for instance, level 5 is required, then only a subclass called level5 needs to be added to the parent class of level, and it can define of its own way of creating nextBlock by overriding the parent function's pure virtual method creatNextBlock which provides flexibility to changes.

Global constants are also used properly. In our case, number of rows and columns of the board are set to be constant as seen in many files including board.cc and graphicViewer.cc. This is really convenient when it comes to changing the size of the board as we only need to change the value of the constants without going through everywhere in the code that uses number of rows and columns.

Minimal structural modification and recompilation:

With high cohesion and low coupling accomplished and abstract class utilized, the program is extremely clear, organized and well-structured. If a change is needed, then changing the implementation of one class is sufficient which avoids structural modification. The relationships between classes are extremely organized in our case: game owns boards which owns blocks which owns cells.

Another way of reducing recompilation is that, forward declaration are used wherever possible. In the case where a forward declaration is adequate, the usage of include statements is shunned to minimize recompilation.

Answers to Questions

Question 1:

We can implement this by constructing an abstract class having Block as an abstract superclass with fields, methods and virtual functions included as necessary. Level would be another abstract superclass having level 0 - 4 as child classes using Factory Method. Level also has pure virtual functions, and each child class will overwrite the pure virtual functions according to their level of difficulty. One of the protected fields of the superclass Block would be a pointer to the class Level. Block would have eight subclasses with each representing one type of block and a field indicating whether it's a disappearing block or not. Since each subclass of Block has access to the protected field the Level pointer within Block, each method in the subclass can vary based on the Level it points to and the indicator.

Besides, each block can also have an indicator indicating if it's cleared and a counter that counts the number of blocks fallen after the placement of this block. Every time a block has fallen, the counter would increment by one, and if it exceeds 10 it will be destroyed and cleared from the grid, and the rest of the grid would be moved downward as necessary.

Question 2:

Since the level of difficulty highly depends on how we select the next block, the solution we came up with is to make use of the virtual constructor, that is, the Factory method. This allows us to produce the next block according to different levels while minimizing recompilation and encapsulating our decision-making process. Specifically, we created an abstract Level class with a public pure virtual createNextBlock method that returns a pointer to a block object. The subclasses are concrete levels(i.e. Level 1, Level 2, etc.), each providing a createNextBlock method that overrides the virtual method. The createNextBlock method will return a pointer to a block object that complies with its own next-block policy and perform any extra effects such as the "heavy" blocks for Level 3 and the additional 1*1 blocks for Level 4. Accordingly, a pointer to a level object will be added to our player class as a private field. This allows us to call the createNextBlock method on the level object and construct the next block for the player.

When introducing new levels, it suffices to add a new concrete Level subclass. The new Level subclass will provide a createNextBlock method that performs the same functionality as the createNextBlock method in other Level subclasses. When the make command is executed, the newly created Level subclass will complete its initial compilation and the ultimate biquadris program will relink, but no recompilation will occur.

Question 3:

We chose to add boolean values that identify the presence of effects and their corresponding functions in the *Board* class. By adding boolean values in the Board class, we will be able to keep multiple effects active simultaneously by using only if branches. For each effect, we will have only one if-branch that calls the function that applies its effect to the program. We also added accumulators for round-specific effects. By using this way, we are able to allow for multiple effects to be active simultaneously while making the addition of effects simple and clear.

Question 4:

To accommodate the addition of new command names/changes to existing command names with minimized recompilation, we used “the pImpl idiom” and created *CommandInterperter* classes. The *CommandInterpreter* file will only have a forward declaration of the *CommandInterpreterImpl* and the *CommandInterperter* class will only have a pointer to the *CommandInterpreterImpl* class defined in its .cc file to reduce the compilation dependencies. All inputs from the player are passed to the *CommandInterpreter* class, and it will pass the valid command to the *Game* class for execution.

Inside the *CommandInterpreterImpl* class, we utilize the std::vector from STL to create a vector object called *commands* as a private field. This vector object stores all the command names in string format. Whenever a change to existing command names is needed, we only need to change the name in the vector. Then, if a new command is added to the system, we will push the new name to the vector and add necessary function codes in the *Game* class.

There is another std::vector<std::string> object *remainCommands* in the private field of CommandInterpthat stores any remaining commands to be executed from the sequence and macro commands. If *remainCommands* is not empty, whenever game requests a command from *CommandInterpreter*, the valid version of the first command in *remainCommands* will be directly returned to the game. Then, the first command will be erased from *remainCommands*.

To support the rename command, we would add std::map object called *renameMap* to the private field of the *CommandInterperterImpl* class, which is relatively simple. Whenever the user executes the rename command, the command interpreter will first ask for the original function name, then it wil ask for the renamed function name. If the renamed function name is already present in *renameMap*, we would decline the player’s request, otherwise, we would insert the new name of the command as the key and the original name of the command as the value into the map. Now, when a command is received from the player, we would first check the *renameMap* to identify any matched value before passing the valid command to the *Game* class.

To support a “macro” language, we created another map object called *macroMap* as a private field of the *CommandInterperterImpl* class to store the name of the macro (std::string) and its corresponding sequence of actions (vector<std::string>). When a player types macro command, we first asks for the name of macro, then, we asks for the sequence of commands that the player wants to include in the macro, ending with a keyword “done”. Then we stored the name as key and its corresponding commands as value into *macroMap*. When we want to execute the macro, we would first match its corresponding vector object in *macroMap* and use the vector's iterator to loop through the sequence of commands. For the first command, we first try to find any matched value for the command in the *renameMap* object, replace the original value with the matched value. Then, we stores the remaining commands into a std::vector<std::string> *remainCommands*. If there are commands in *remainCommands*, all commands will be pushed backwards and let macro's commands execute first. If there is a multiplier for the macro command, all commands in the macro will be stored after inserting the first macro's remaining commands. Lastly, we return the valid first command.

Extra Credit Features

Macro

Macro allows the clients to enter a sequence of commands and provide a name to the sequence which would then be stored. The next time the client calls this sequence of commands by its name, the program would automatically executes the commands that were stored under the name in order.

Smart pointers and vectors used without any memory leak

Smart pointer vectors are used whenever possible. It is advantageous because we don't need to manage memory by ourselves and there is no delete statement needed. No memory leak is present. On difficulty encountered was that segmentation fault will occur after changing pointers to smart pointers, but we were able to find the root cause of the errors and fixed them.

Rename commands

This functionality allows commands to be renamed. The commands can be renamed based on clients' preference. For example, the command "clockwise" can be renamed as "wiseclock", and the nextime the client wants to rotate a block clockwisely, the new command "wiseclock" can be entered to trigger the rotation. One restriction would be a new command cannot match to any existing commands.

Enter names

The game allows player1 and player2 to name themselves and when the boards are printed, their names would show on the graphic and text viewer. When a game is finished, a game summary is present clearly showing which player has won and their scores respectively.

Bonus system

Aside from normal scoring system described in Biquadris.pdf, we added a bonus system ourselves that brings more surprises to the players! There is a 5-point bonus whenever a player clears 5 blocks of the same type, but there is noly 1 bonus for each type! This system was implemented using std::vector object to store the number of blocks that got cleared for each type, and we used getter and setter to retrieve and update the bonus.

Score Comparsion

If a play just loses because its next block cannot be initialized on the board, that would be too boring! To make things more interesting, we added the score comparsion at the end of each game to determine the winner. To add this system, we added getter and setter for scores in the project and modified the summary scene in both graphics and text displayers.

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?

Collaboration and Communication:

Collaboration saves time on larger-scale program. Each individual is responsible for their own components which shortens the overall time costs to code the program. Additionally, communication is really important when it comes to teamwork. Always give your team updates.

Using version control software (i.e. GitHub) when collaborating is extremely beneficial because commits and changes can be easily tracked and reverted.

Design and plan ahead:

UML is extremely helpful when designing larger programs. With UML being utilized and program being designed beforehand, the overall process and structure of the program and the order of operations became so much clearer. Because we had everything planned out, all we needed to do is using the UML as a reference to code with a clear and fresh mind. If we didn't complete UML, it's not hard to imagine how messy the code could be and how struggling the completion could be.

Each individual is responsible for their own code pieces. Although each individual did not participate in all parts of the code, each had an idea of what each function does which by just referencing the UML which was extremely helpful when debugging.

Unit Test:

Conduct unit tests first before integration. It is essential to conduct unit test harness for each component and ensure each part works before combining them together, otherwise it would be overwhelming when debugging. It's not possible to have a working version without thoroughly testing each component.

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

Double check before merge:

Double checking your branch before merging ensures master always has the correct version. Constantly pulling from master and updating the current branch are necessary. If the wrong version is accidentally merged into master, it would be problematic to revert the changes and inform all group members to rerun git pull which may cause some code loses by accident.

Read instructions carefully

Reading the project description and understanding all its requirements are important. Rather than rereading the description everytime we code and misunderstanding the requirement, we can, instead, jot down important points the first time reading it while highlighting the details that could be missed. By doing this, we can avoid changing the code back and forth to meet the requirements caused by misinterpretation of the requirements.

Conduct unit testing and debug separately

Although coding together can be time-saving, debugging to gether using integration tests at the first place is unwise. Each individual could have debugged their own code pieces first using unit testing first and integrated the code piece after making sure each part works as desired.

Conclusion

In conclusion, Biquadris is a fun game to play. Our group learned a lot of lessons, both coding knowledge and interpersonal skills in a collaborative environment. We were able to strengthen our coding skills and further digest the core Object-Oriented Programming concepts throughout this experience.

Thank you.