Design of Final Project: Biquadris

Kevin Jin, Yibo Ma, Yiran Sun
CS246 Spring 2021
University of Waterloo

August 13, 2021

Directory

Introduction

The video game Biquadris is a Latinization of the game Tetris, expanded for two player competition. It 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. You have as much time as you want to decide where to place a block. Players will take turns dropping blocks, one at a time. A player's turn ends when he/she has dropped a block onto the board (unless this triggers a special action; see below). 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

- **Cell**

  Definition: Single cell (square) with different types contained in a board

  Description: Every cell is only constructed when a board is constructed, and it remains in its original position until the program terminates. If a cell's type is modified and the cell is not set to "blind", it will draw a new square in the corresponding colour in the Xwindow to update the graphical display. If a cell is set to "blind", it will become a black square in the graphical display. If a cell is not "blind", it appears in the colour corresponding to its type.

- **Block**

  Definition: Tetromino that consists of 4 cells (vector of cell pointers) with its shape determined by type.

  Description: When the board generates a block, a new block is constructed at the top left of the board. However, if there isn't enough space, the block will be set to "doesn't exist" and the board will observe this to determine whether the game should be terminated. The movement and rotation of a block is accomplished by changing the cells that it points to and setting the cells' types accordingly. When performing a horizontal or vertical move, the function returns a boolean to indicate whether the move is valid or not. This information is useful for the board when a "heavy" block is moved.

- **Board**

  Definition: Game board of a single player that contains 11x15 cells (vector of vectors of cell pointers).

  Description: Board contains a vector of vectors that contains totally 11x15 cell pointers, which constitutes the game board. Also, it contains a vector of block pointers. Therefore, the board knows the movement and status of all blocks. Board directly or indirectly handles most of the player's operations. It updates the game board (vector of vectors of cell pointers) in it by traversing, clearing out the full rows and calculating the new score. The board implements the methods of generating new blocks; moving, dropping and rotating blocks by calling the methods of the block. Also, there are three boolean variables in Board to help track the special effects (Blind, Heavy and Force) on the current board, the methods in Board will react differently based on these booleans.

- **Level**

  Definition: Block generator for a certain level of difficulty

Description: A block generator can either generate blocks according to a sequence contained in a file or randomly according to the specified rules. The board uses the block generator stored in it to retrieve what type of block should be generated next.

- **CmdInt & Command**

  Definition: Command Interpreter for text-based commands

  Description: The interpreter uses the given string (command) to determine what the command is, whether the command is valid, whether it has a multiplier prefix, if so, what the number of times that the command should be executed. A structure (Command) that contains all the information stated above that will be useful for executing commands is returned by the command interpreter.

- **TextDisplay**

  The text-based display of the game boards is displayed every time a player makes a change to the board (i.e. making a move, changing the game level). It takes the necessary information from the two boards and displays them side by side along with some spaces accordingly. It also determines who wins the game.

- **GraphicDisplay**

  The graphical display, consisting of an Xwindow, of the game boards is automatically updated when a cell in the boards is modified. When generating the next blocks, changing the game levels and scores, the corresponding methods must be called in order for the graphical display to be updated.

- **Main Function**

  The main function is the controller of the entire program. It supports various command line arguments, and reads in commands, and connects all classes with different functions together to make the program behave as specified..

Design

- **Object Oriented Programming:**

  Encapsulation: Our program is built using various features of Object Oriented Programming. When we implement modules for the program, we use the idea of Encapsulation throughout our design. Using the level class as a specific example: we hope to hide key information, for example, the order of the blocks from the players. Hence, when blocks are generated with a file, sequences in the file are hidden from players, and there are no ways to allow players to manipulate the sequence, except for changing the file. When blocks are generated from seed, players can only set the seed, instead of knowing which seed is actually being used. Other classes of our program also follow this principle. In the board module, players could not mutate the position of the block once it is dropped or modify the cells contained in the board.

  Abstract Classes and Inheritance: We also use the idea of abstract classes and inheritance when designing the level module and the block module. For the Level class, we have the function for generating blocks declared as pure virtual, and each Level subclass has its own version of override function which follows the specified rules. Although different blocks have different possibilities to be generated in different levels, those levels share various similar characteristics, and hence, it is essential to use inheritance to avoid unnecessary code repetition. Similarly for the Block class, which implements the function rotate as purely virtual in its abstract class. While in the subclasses of Block such as IBlock and SBlock, the rotate function is implemented according to the shape of the specific type of the block, while each subclass of Block represents a different type of block. The abstract Block class not only contains accessors and mutators, but also functions which could be applied to every type of block. For example, the function drop only needs to make sure that the block is dropped to the end, and does not really need to care about the type of the block. Hence, it is implemented in the abstract class.

RAII Idiom: The design of the level module also follows the RAII: Resource Acquisition Is Initialization principle. All resources and heap allocated objects related to the level are cleaned up automatically, which means that we do not need to manage memory on our own. All classes related to the level are allocated as share pointers in the main function, and inside level classes, as I mentioned before, we store items in vectors.

- **Coupling and Cohesion:**

Our Program follows the high cohesion and the low coupling principle. For cohesion, all Block classes inherit from the abstract block class and they work together for the block movement as effects of read in commands (drop, left, counterclockwise…). All Block classes are linked together to achieve the same goal. On the other hand, the command class and the CmdInt class have an association relationship. They work together for the goal of identifying the input command, and modify and suggest modification on the command names. Similar to the Block classes, our Level classes are also subclasses of the abstract Level class (inheritance relationship) and they work together to generate new blocks with probabilities corresponding to each level.

Our program also follows a low coupling principle. The Command class only has an association relationship with CmdInt class without having other relationships with other classes. Display classes (textdisplay and graphdisplay) both only have an aggregation relationship with the board class and do not have any other relationship with other classes. The only relationship which level class (except for inheritance relationship with its subclass) has with other classes is the aggregation relationship(board has level). The block class has a cell class and a board class where the board class owns the block class. This is because blocks need to know information of cells and properly act on the read in commands. All couplings in our program are necessary, and we have minimized our coupling.

- **Differences between the final design and the design on Due Date 1:**

Removal of Decorator Pattern: In our UML design on Due Date 1, we planned to use the decorator pattern to implement the special actions. However, during coding, we realized that it was much easier to use some boolean values to implement the special actions since they only exist temporarily. Therefore, we decided not to use the decorator pattern because it was less efficient.

Addition of New Fields and Methods: In order to implement every feature of the game, we had to add many new variables and functions. For the Block class, we added a getLevel function which was required when calculating scores, a drop function which increases the efficiency, a clearRow function which was useful when a row is cleared, etc. For the Level class, we added various functions that allowed the block generator to generate blocks according to game rules easily. For the Board class, we basically created a completely new class. We realized that implementing the game board was far more complex than we expected and we decided that the board was going to be connected tightly with the main function. Therefore, lots of new functions were added in order for us to successfully implement the main function.

Association between board and blocks: In order to implement the move and rotate functions in Block, a block must have access to the board and the cells in the board. Therefore, we added a Board pointer in Block so that the blocks could be easily modified according to the positions of other blocks in the board.

Resilience to Change

Our Programs are suitable to make any changes which may occur without large changes on the whole program. In this part, we will outline some possible modifications or changes on the existing features, or any new features which may be added to the program.

- **A new level is introduced (i.e. level 5):**

  Since we implement the level module using abstract class and inheritance relationship, it is relatively easy to introduce a new level to the game. If the new introduced level does not have any special effect, such as heavy in level 3 and the dropped block in level 4, then the only files we need to modify are the main.cc (main function), level.h and level.cc, which contains all interface and implementation of the abstract class' functions. We will just need to add a new class in the file, which also uses the overridden generateBlock function to produce a character. If the new level contains data which the abstract class does not have, depending on the level of Encapsulation, we could use mutator or accessor methods to get or change the data. While in the main function, since our program does not need to create a new level class every time a leveldown or levelup command is read in, we just need to create one or two share pointers, depending on what we need. We also need to modify some parts of the main function, for example, Seed xxx command will also have effect on the new level.

  On the other hand, if the new level includes some special effects similar to level 3 and level 4, depending on the new effect, we might want to add some fields to the board to record the effect. We should also modify setLevel, move, rotate and update the board accordingly to the new features. Nevertheless, we still only need to modify the board module, main and level module to adapt the change, which is relatively easy to do.

- **The effect/rules/features of the existing level changes:**

  This situation is similar to the situation above, where a new level is introduced with special effects. The difference is that in this case, we do not need to add a level subclass to the level module. We still need to modify the generateBlock function, along with the move, rotate, update function in the board module. Depending on the number of levels' effects are changed, we might modify fields of abstract level class or the subclass of level depending on what we need Some board fields should also be modified/deleted or added to adapt the changes.

- **Command names are changed:**

  We have an extra feature with the command name, rename, which is described more detaily in the extra feature section. Everytime players hope to change the command name, they can just enter this command. Note that it is also possible to change the name of the rename command.

- **Effects of Command change:**

  This really depends on which command's effect is changed and how it is changed. For some very tiny changes, for example, leveldown decreases the level automatically by two instead of by one, the only section which we should modify is the main function. However, for most changes on the commands' effects, we not only need to modify the main function, but also modules which relate to the specific command. For the existing commands, left, right, down, clockwise, counterclockwise, and drop are related with the block module (block.h and block.cc), levelup, leveldown, nonrandom and random are related to the level module (levels.h and levels.cc) and I, J, L, etc. and restart are related with the board module (board.h, board.cc). Each time a command's feature changes, with no special feature added to the command (i.e. heavy effect), then we only need to modify the related module to the command. For example, if the rotation method changes, we only need to modify the block module, and more specifically the rotate function. If a special effect is added to the command, then beside the module relates to the command, there is a large chance that we also need to modify the board module to add effects. Sequence command is a special case. It is implemented directly in the main function by changing the input stream which a pointer points to. Hence, if special effects are added, then the board module is modified through functions and fields, otherwise, only the main function needs to be modified.

- **A new command is added:**

  If a new command is introduced to the program, our program can also handle it in a relatively easy method. The command name is first added to the class CmdInt, which contains a vector of commands' names. Now, our program will be able to identify the new added command even if the input is in abbreviation. Depending on which module the command is related with, we may only need to build/modify an extra function in the board module which calls the related function in the block, level or display module, then modify the module which relates to the new command by implementing new field and functions, or use the combination of the existing functions. It really depends on how the newly added command actually works. For example, if the new command "rightdown" moves the block one cell to the right and one cell downwards, we only need to modify the command module to add the new command name, then modify the main function to identify the new command then call the move function in the board. We do not need any modifications on board module, block module, or level module, even though the new command is related to the block module. Our existing function can deal with it. On the other hand, if a new command "extend" is added to extend the size of the block by two (i.e. I-Block is changed from an 1x4 block to a 2x8 block), then since board module has composition relationship with block module, new new function should be created in board to call the corresponding new function in block module, which is also newly implemented (and probably pure virtual and overridden in subclasses).

- **Shape of Block changes:**

  Again, I will use I block as an example, suppose I block now has a H shape, due to our program following the modulation principle with high cohesion and low coupling, we only need to modify a very small part of our program. We have two choices: First, create a new Hblock class and leave the Iblock class. This could be useful if sometime later, we want Iblock back. We will first implement Hblock which is also a subclass of abstract block class, with its unique constructor and overridden rotate function. However, if we choose to do so, we need to change the command section(I to the E), the level module (I to the E), the main function and the display module. Although all we need to do with these modifications is tiny (one line per function), they could be annoying. The benefit for players is that they could type in E to obtain a H block and if they hope to generate blocks from file, the file could obtain E block by containing character E.

  On the other hand, we could just modify the existing IBlock class instead of creating a new subclass, then display a message to the player that now I shape Block becomes E block. By doing so, we only need to modify the constructor and the rotate function. If a player hopes to generate blocks using files, they need to be more careful, since they should still use I instead of E. A good thing is that players do not need to modify the content of files which they created before (do not need to change I to the E).

  In general, the first choice is more suitable if the program has many modifications, and the second choice is more suitable if the program only needs to be modified once and does not have many users.

- **New type of block is added:**

  This is similar to the choice one in the condition where the shape of blocks change (condition 6). A new subclass is added which also inherits from the abstract block class. The type of the new level actually does not matter, except for we might need to draw a larger rectangle at the bottom to display the next block. (Currently we have a 2x3 rectangle, but if a larger block, for example, H shape block, we may need a larger rectangle). On the other hand, when introducing a new type of block to the game, if players choose to generate blocks using seed, with an extra type of block, there is a large chance that the possibility of the next block corresponding to different levels also changes. Hence, we might also hope to modify the generateBlock function in the level module and board module to generate the new block. These modifications usually only need to add or modify one or two lines, and are easy to do.

- **New special effect/or effect is added:**

This condition is similar to the 4th and 5th condition, and it really depends on which command is added. Let's use the effect mentioned in question 1 as an example. Our board actually has a field which is a vector of pointers to the block to identify every block which has dropped in the game (including the current dropping one). To add this new effect, we would add a field in the cell class to identify which block owns this cell. We will also modify the updateBoard function in the board class so that when a row is complete and the upper blocks drop down to fill the gap, the number which identifies which block has that cell will also be updated. When the vector of a pointer to block has more than 10 items, each time it will check the 10th block before it. In our design, a block contains a vector of pointers to cells. We could find out if that block is completely eliminated or not (if that vector of pointers points to cells which do not have the same index with the block). This will be done either in the updateBlock function, or build a static local function and be called in updateBlock. This is a relatively complicated effect, and although it is complicated, we only need to modify the cell module and board module by adding one field (with corresponding accessor and mutator, adding one helper function and modify the existing updateBlock function to achieve this effect, and modify the main function correspondingly. Most of the new effects will only relate to two modules of the program and they are easy to achieve.

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?**

   We could include a count down field in each block, starting with 10. Every time a new block is generated, the count is decreased by 1. If the count reaches 0 and the block is still not fully cleared, we would clear the entire block by clearing each cell contained in the block from the board. At the same time, since the block contains vectors of cells, after each block is dropped, the thing we need to do is to loop through all blocks, and find blocks which count reach 0. Then what we need to do is to set any cells which are still contained by the block to contain space to indicate that it does not belong to any blocks, and lastly, delete the block. If we only want to invoke this feature in more advanced levels, for example, only invoke when level is 3 or 4, our design also allows us to do so. We could add a field which uses an integer to indicate which level the board is currently at. At the same time, the block is actually owned by the base-board, and in the base-board, there is also an indicator which reminds the board the current level that the game is at. Hence, we can just copy the level indicator from the board whenever a levelup or leveldown command is read in, and to determine whether this feature will be added or not (if not turned on, the count field will still exist, just that we do not call functions related to it.

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

   We could make the Level class an abstract class so that every time an additional level is introduced, we could simply add a new Level subclass in separate .h and .cc files, which will provide the information and rules required for the new level of difficulty. Therefore, whenever a new level is introduced, only the new .cc file is required to be compiled. At the same time, we will introduce a controller static function which will be used to run the seed, and generate appropriate blocks. All the levels, for example, level 0, level 1 and the new introduced level will inherit from the abstract class of level to avoid any unnecessary duplications.

3. **How could you design your program to allow for multiple effects to 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 could use the decorator pattern for special actions to allow multiple effects to apply to a player's board at the same time. Since the current three effects do not conflitted with each other, using a decorator pattern would be the most appropriate method. If a new effect is invented, we could add a new concrete decorator and there is no need to introduce if-else statements for applying the effects. However, if there is a conflict between two or more effects, we would have to consider something similar to the precedence of rules for the life game we did in Assignment 4.

An alternate approach is that we could set up an indicator in the board, to identify which effect has been added on. If the program reads in commands that suggest that one special method is added to the player's board, we will first check the indicators to see if this effect has already been added. If the effect has already been added, then we won't bother to make the effect work again. Note that we will only have indicators for heavy, blind, but not force. For force, no matter how many force commands have been called, the new force command will overwrite the last force command.

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.**

We will have a command class, which contains all commands that the game currently has. So every time a command is added, we would simply need to recompile the command.h and command.cc file. At the same time, in the command class private field, we will have a vector of strings which represent the name of commands. Every time a command is renamed, the corresponding string in the vector will be mutated. We will also have a match-command function to help the program understand any recognizable command names. So basically, the procedure of reading commands is that:

1. A command is read-in.

2. We will call the command identifier, and then the correct command will be called and it will loop through the vector of strings and find out which string is matched with the command, then call the correct command function.

3. If the command name is changed, we will simply change the string which will be used in the command identifier function to identify the command.

For the question related to the "macro" language, we will allow the player to define the "super command" when they are playing the game. However, players must follow the given format. The key word that indicates the player is trying to define a new command is "Name". We will add two new vectors in the class of command. The first one is a vector of strings which would contain the player defined commands. The second vector is a vector of vectors of strings, such that each vector of strings will correspond to a unique element vector which I mentioned before. The defined function name will be right after the command "Name", then after that is the list of commands which make up the new defined command. For example, if I type in:

Name supermove clockwise lef ri right

And this the third command that the player ever defined. Then the vector1[2] == "supermove" and vector2[2][0] == clockwise vector2[2][1] == left vector2[2][2] == right vector2[2][3] == right. (we will also identify all the strings except the newly designed one.

Whenever a new command is read in, the function will not only try to identify with the original vector, but also the first vector which we just created. However, this feature actually makes the program less efficient and is suggested to not do so.

Extra Credit Features

- **Smart block selector**

  When a block selector takes its blocks in sequence from a given file, it accepts block types in both uppercase and lowercase, and it ignores any invalid block types by skipping to the next one. For examples, if “T” and “t” are read, they are treated in the same way and a T-block is generated, but if a character other than “I”, “i”, “J”, “j”, “L”, “l”, “O”, “o”, “S”, “s”, “Z”, “z”, “T”, “t” is read, the selector will skip over it and read the next character. This functionality avoids any unexpected termination of the program when reading sequence files.

- **New command: “rename”**

  The “rename” command renames an existing command by giving it a new name. For example, if the command “counterclockwise” is renamed to “cc”, the next time a player calls “counterclockwise”, it will become an invalid command. But if “cc” is called, the current block will be rotated counterclockwise. If the renaming is successful, a message indicating that the old command has been replaced by the new command will be displayed. However, if the player attempts to rename a command that does not exist, a message indicating that the command is invalid will be displayed and the player will have to re-enter the commands.

- **New command: “hint”**

  The “hint” command provides a hint for the player by displaying a randomly generated hint message and some links for tutorials on playing the game Tetris.

- **New command: “help”**

  The “help” command displays the list of available commands, and for each of the commands, a brief description of the command and its syntax are provided. This command is intended to help users familiarize with playing the game by providing a user interface guideline.

- **New command: “quit”**

  The “quit” command allows players to quit the game halfway through with a confirmation message. After the game is terminated, there will be a message indicating which player has won or the game is tied based on the scores of the two players.

- **Command suggestion for invalid commands**

  Every time an invalid command (a command that cannot be recognized by the command interpreter) is called, the program will search for the command that is closest to the given command. For example, if “le” is entered, the command interpreter will not be able to know whether it was intended to be “left” or “levelup”. Therefore, the program will display “Invalid command. Did you mean left?”. This function is also used when an invalid command name is given for renaming, when it will display “Could not find an existing command to rename. Did you mean [command name]?”.

- **Winning message**

  When the game is won (one player loses), a winning message will be displayed in text. The message shows which player has won the game, the current scores of the two players, and the highest score of the two players.

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?**

   Over the period working on this project in a group, we learned that communication is crucial in collaboration. Our team has three members, each living in a different timezone, so it was quite challenging for us to work together at the same time. We had to divide the project into different parts and had each of us working on specific parts. However, when actually working on the project, we found that it was difficult to stick strictly to the plan. We had to make changes when implementing the functions and modify our design over and over again. In order to make sure that everyone is on track, we had regular voice meetings, which turned out to be really helpful over time. Besides, we also realized that it is extremely important to have a plan in the beginning when working on a group project. Even though we did make lots of detailed changes, the overall plan remained the same and it was beneficial for us to have a general idea of what we were working towards building. In conclusion, this project tought us that it is important to plan ahead and communicate well when developing software in teams.

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

   If we had the chance to start over, we would definitely start working on the project earlier because all of us had been busy during the final few weeks of the term, especially during the week of finals. Although we were able to finish the project on time, we had some really good ideas for extra credit features that could improve our project, but we were unable to implement them because we were running out of time. We felt disappointed that we could improve our project but didn't have the time to do so. However, we would like to improve it by introducing some new features after the project is due just for our own interests.