CS246 Spring 2021 Project - Biquadris - Design

By: j538xu, pxlin, y82hou

Introduction

In this document, we will discuss our design and implementation of the game Biquadris, which is a two-player Tetris game. In this game, two players play their own Tetris game on their own board and aim for a higher score. The game ends whenever a player's new brick cannot be placed on the board. In this report, all features and their general interpretation will be shown in "Overview"; any specific interpretation of features that are not obvious and cannot be easily interpreted will be explained in "Design"; for any features that can be added or changed from the program specifications easily will be introduced in "Resilience to Change"; all questions from project specifications will be answered in "Answers to Questions"; "Extra Credit Features" will document the extra features we have in the program; then we will answer the final questions in "Final Questions" section.

Overview

The display of the game is as follows: for each player's board, the first line displays the current player who's playing the game, then it shows the level and score under it, below that information is the actual board, and next brick is shown under the board. When the game starts, the program tells the user which player is playing the game, and asks the player to input the actions.

In the implementation of the game, we constructed 6 main classes: Game, Board, Brick, Block, Xwindow, WDisplay Invalid Inputs. From main.cc, we read in any command line argument, save those for future usage, then call Game's constructor to build and start the game. We store default filenames as "biquadris_sequence1.txt" and "biquadris_sequence2.txt" in a vector. By using command line arguments "-scripfile1 file" "-scripfile2 file", the file names will be changed. Also, the commands "-level", "-seed", "-text" will set the level of the game, seed of the random function and whether the game is running only in text. In addition, we implement a command "-enablebonus" to bring some extra features to the game. We can set the number of players, the size of the board, and even a mode that only displays the graphics with only two colors.

Constructor of Game then generates two Boards, each of them having 11 columns and 18 rows. By starting the game, Game class also generates the current brick (Board::curBrick) and next brick (Board::nextBrick) for players, and then displays the corresponding bricks on the screen. The class Game is responsible for any in-game commands listed in the "command interpreter" section in the project specification, as well as the special actions' commands. It reads in the string the user inputs, and identifies which command it is from the list of available commands, and how many times the user intends to let the program perform the command. If Game cannot recognize the command, it will throw an invalid input and inform the user.

Once the user calls a valid command, “Game” will interpret it and display a new board after the interpretation. Specifically, when calling “left”, “right”, “down”, “clockwise”, “counterclockwise”, Board would call its current brick’s function in brick: moveLeft(), moveRight(), moveDown(), clockwise(), counterclockwise(), respectively, to change each block in brick’s coordinates in Board. When the user adds a number before the command, the command will be executed that number of times. Any negative number will be considered as invalid input. When a command cannot be implemented (there are blocks in the way or it reaches the edge after movement), it will not perform the command, and the program continues. When the player “drop” the brick, or the brick is forced to be dropped by special actions or level 3 or level 4 (when the program forces them to move downward but it is impossible to do so), their round ends. For “levelup” and “leveldown”, it directly lets the board change its level (Board::level). Note that the program only allows the level to be a number between 0 and 4, inclusively. When the user attempts to let the level to be smaller than 0 or larger than 4, the program would ignore the extra level decrement or increment. (i.e., if current level is 3, and the caller calls 4leveldown, the level will be changed to 0; if current level is 3, and caller calls 100levelup, the level will be changed to 4). After the user changes the level, the level’s feature will be applied to the current player(playerTurn)’s board immediately, the bricks that already generated and displayed will not be changed, displaying the newly generated bricks in the board, where the newly generated bricks all follow the possibility specifications indicated in the project specification. When the user uses the command “norandom file”, the game will stop generating bricks with given probabilities and start to generate bricks based on the type listed in the file. When the user uses the random command, if the game is generating bricks from a file, the game stops reading the file and starts to generate blocks randomly with given probabilities corresponding to its current level. If the user enters a single letter representing any brick (any of I, J, L, Z, S, T, O), the current player’s current brick will be changed to the brick the user enters, and will be placed at the top left corner under three reserved rows. For hint, we set up a function that can loop through all the movements including left, right, clockwise, counterclockwise, while taking heavy into considerations, to give the user the best solution to place a brick in the graph so that the brick will go as low as possible in the board.

The scoring system is the same as indicated in the project specification, that is, whenever a brick is dropped and the board deletes the rows that are full, the score as well as the hi score are updated, and are displayed on the board. If score has more than 5 digits, there is no place for the numbers to display, system will print it as 99999, but still counting actual score.

When their new brick cannot be placed on the board, the player loses and cannot make moves, and the other player still plays. After the game ends, the program determines the winner according to their scores, the first person who reaches the highest score will be the winner, and asks if the user wants to play again, if they choose y (or Y), the game restarts, otherwise the program will end.

The graphic display is similar to the board’s display, except that it does not ask the player for their input, that is, all actions and inputs are done on text display, while the boards are shown in the graphics display. Another difference is that we allow different colours for each type of block on the graphics display and we are also able to implement a colour blind mode to make all the blocks the same colour for the user if desired.

When a hint is asked, our program finds the combination of commands that allows the user to drop the current brick at its current position, onto the board with the lowest possible height, and the most number of blocks being at the lowest height. For example, if the player currently has a block I, the program will ask the user to drop it horizontally because that way, four blocks are at the lowest height instead of only one block.

Design

The way we find the corresponding inputs from the users' abbreviated input is implemented in the Game::getInput() function. In that method, we checked the input string with the substring of every command up to the length of input, to see if we found a match. If we find no or more than 1 match, then we consider it invalid input. Otherwise, we use the match as the command. Note that for each character in all commands, they can be either capitalized or lowercase, and do not need to be consistent. That is, Levelup, LEVELUP, lEvELuP, levelup, LeVELU are all valid inputs for "levelup". For any command that has a number at the front, instead of repeating each action manually, we used a for loop to loop it that number of times, and meanwhile check if it's at bottom if the player is level 3 or 4 or the other user applied special actions each time in the loop, increasing the efficiency. For any value in front of the commands that is a negative number, we treat it as invalid input.

Board has a 11*18 grid which is a vector of vector of Block*, so that each grid[r][c] is either nullptr, representing an empty block, or a pointer to block, which contains the block's information which are its row, column, type and the brick it is in. This is convenient when displaying the board, when a position is nullptr, it prints ' ', and it prints the type otherwise. Each Block is recorded in both the board containing it and the brick containing it. Each Brick is recorded in the board containing it, and it records the blocks that form it. Each Board records the blocks and bricks it contains. We use unique_ptr to save Block in order to avoid "new" and "delete", so that we do not need to worry about memory leaks issues.

When we add a brick to the board, the board's vector Board::bricks will save it as part of the board, then the Brick will save all of its blocks in Brick::component, and all blocks in the component will save this brick in Block::brick. Then, whenever we need to update the board to clear the rows that are full, we call the Board::update() function to implement it. It first checks which row has no nullptr, which is rows that are full, then it clears the line and makes sure every brick cannot be dropped further after the removal of the blocks, then it checks the rows again and repeats the above operations until there are no full rows in the board. Note that if a brick's middle row is cleared, the blocks on top will drop. We decided that all the blocks of a brick need to be adjacent to each other, so no block will drop lower than the lowest row of the brick. update() also records the number of lines that are cleared, and calls all blocks in the row to remove themselves, which would fully remove it from both the brick and board containing it.

When initializing a brick, we first make four unique pointers of blocks, then put them at initialized positions at the top of the board according to their types, and put Blocks in Brick::component to record them. We use a structure to set up bricks's movement. The structure "point" contains two integers representing row and column. For each movement, we save the original position and the position after the desired movement as two vectors of point, and

then use a loop to loop through each point in the new position's vector. If it does not go out of board and there is no brick other than "old" brick in the way, it will delete the blocks in the old brick and generate the blocks in the new brick and put them in Brick::component and grid in board.

When creating a brick for its current level, we use the function std::rand(). For level 1, since the first five types in the vector have a probability of 1/6 = 2/12, and the last two types have a probability of 1/12, if we use (rand() % 12)%7, the chance of getting 0-4 is when rand()%12 gives 0-4 and 7-11, which doubles the chances of getting 5 and 6. For level 2, since the probability is the same, we can simply use rand()%7. For level 3 and 4, since we want the last two types in the vector have a probability of 2/9, and the first five types have a probability of 1/9, if we use (rand()%9)%7, the chance of getting 0-1 is when rand()%9 gives 0-1 and 7-8, which is 2/9 each and the rest has a probability of 1/9 each. Now, if we use 6-(rand()%9)%7, the 0-1 becomes 5-6, which are the two types we want to get the probability of 2/9 and the rest has a probability of 1/9 each.

In the hint section, we implemented this feature in the game class. When a hint is requested, the program uses the copy operator to create identical boards to the player who asks for the hint. For each identical board, the program will check each combination of clockwise, right, and left to find the most matched combination for the current brick to be dropped. Counterclockwise needs not to be considered because rotating clockwise accounts for it. The program creates local variables to store the lowest height and the number of blocks that can reach the lowest height during the simulations and update them when the record is beaten. The program also uses two integer variables to store the combination of clockwise rotations, and horizontal movements that leads to the best match to drop the brick.

We also made some changes to the original design submitted. We deleted class Level since level changes all the time, and adding the level class and letting it create the board every time the level changes are unnecessary and make the program really slow. Then we realized that having only the functions we write was not enough to implement all features the project desired, so we added a few more functions and fields in every class to implement all features. For example, in our new design, we realized for the convenience of updating each brick and board, it was necessary to add a brick pointer to each block and a board pointer to each brick. This way, every time a new brick or block is created in the board, it updates itself onto the board instead of having the board to search every time to update what is added. Then, we also realized we cannot just use std::exceptions to determine an invalid input. As a result, we created an invalidInput class that will be thrown each time the user inputs something invalid, then the invalid message will be displayed and a new input will be requested.

Resilience to Change

For the fields of Game which are the setup of the game, we did not use constants, we used variables instead. So the entire program can run safely even if a different height and/or width are entered. We also kept the player number as a variable so the game can also run with not only 2 people, but 1 player, or more than 2 as well. When implementing the display, we realized that having too many players or a board size that is too big or too small would either make the game have no sense, or require a very large display that would not look visually appealing on the display. So we

decided to put a minimum and maximum on the variables height, width, and player number. When writing the graphics part using Xwindow, we also realized that the display would be too big for the maximum number of players we allowed, so we created 2 different block sizes to be used depending on the number of players that are playing. In this way, we can change the number of players, which is also the number of boards in the game, as well as the width and height of the game. Thus, to implement those, if we wish to change the width and height of each board or number of players in the game, we only need to require the user to enter it after -enablbonus, and store those numbers into different variables, then use those variables to construct a game.

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?

Answer: We will answer this question in two parts: determination of screen not cleared before 10 more blocks have fallen; elimination for the blocks to disappear. For the first part, similarly to what we have done for level 4, we add an int field in class Board; we can call it notCleared for now for clearer further explanation. Every time we access the Board::update() function which is the function we call when the "drop" command is called, the rows are cleared in the screen. If no screen is cleared, we record by adding 1 in the notCleared integer. For the second part, whenever we are deciding that we want some blocks to disappear, we only need to check notCleared to see if it is 10 or more, if it is ten or more, then we call Board::removeBrick(Brick *brick, bool change_score) to remove the brick. (we assume the block indicated here is not single blocks but bricks in our program. If it means block in our program, we can also firstly save those blocks in Board::bricks as single blocks and also call the same function removeBrick to remove them). It can be confined to more advanced levels since the only thing we need to add is the notCleared integer, and as long as we treat them as normal bricks in board, it does not require any other functions to make it more complicated.

Difference with answers before starting the project: When we answered the question, we neither consider counting the number of times it is not cleared, nor consider saving the blocks before making changes to them.

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

Answer: We implemented the action of each level mostly in the board class. If we are to introduce a level, we simply have to add the feature of the level into the board, and the board will automatically follow the rules of the level in the game. This way, when the game is being played, we do not have to add many if statements in playGame() in different scenarios for each player to determine if a rule of a level is to be applied. This also reduces the amount of time to debug for each scenario because all we needed to do was to set breakpoints within the board class and we can see what is going on in the board that is creating an error, making fixes accordingly. This way, breakpoints throughout each scenario in playGame() is not necessary and we save a lot of time.

Difference with answers before starting the project: since we deleted the class Level which is unnecessary in our program, the entire procedure of applying the levels to the functions has changed. We did not consider the complexity of the program before, but realized it after we started implementing the program.

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?

Answer: Since effects are applied every time a brick's position changes, we are doing it in game.cc when we read in inputs. In the while loop we take commands and call board to execute the commands, we can add if statements after the executions to apply the effects simultaneously. However, if there are more kinds of effects, adding a helper function in game would be necessary. Note that the reason we should not add it in the class Board is that the effects usually involve actions of both players, for example, the special actions are generated to one whenever the other player has achieved some specific effects. The way we design it will not be using else-branches. Since the effects will be applied simultaneously, we only need if statements. Whenever one condition is not met, we will not apply the effect, so it will be unnecessary to add else-branches.

Difference with answers before starting the project: we planned to use decorators but turns out it would be unnecessary since simple if statements in the loop would be sufficient.

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: In class Game, we have a helper function getInput() that takes a list of commands, and applies the inputting rules from the project specifications to the strings the user provides, then it returns the correct command from the list of commands we provide. Since it applies the rules to any string to find the closest and unique one, if we need to add new commands, we only need to add the new command as a string in the vector of strings that is passed to our function, and the function will check the command and return the correct command from input. We did not use "or"s in if and else if statements to determine the correct commands. This is because there are too many combinations and strings we need to consider; adding a function to evaluate the strings will be much more efficient and reduce many unnecessary codes. In this way, it would be really easy to adapt our system to support new commands. The same method would work for any macro language. For any new features, we need to add new functions to the board, and since Brick and Block have almost every action we can apply to the program, we do not need to add functions in class Brick and class Block in most cases. We can add the macro language as a string to the

vector of strings and pass it to the getInput function, and then if it returns a macro language command, we apply the actions to the program like what we did for any other commands.

Difference with answers before starting the project: before starting the project, we only considered adding functions for the newly added features, but did not consider reading in the input, which is a large proportion of the implementing process.

Extra Credit Features

Full extra credit features are to be shown if user calls “-enablebonus” and without “-text” in command line.The features are:

- Number of users: there can be 1 to 7 players playing the game, since the computer screen is only large enough to show 7 boards side by side clearly, we only allow as many as 7 players to play. There can be as many players as possible if we want, but here we only allow 7 players to make the display nice and clear for users to watch.
- Width and height: users can choose to set up a specific width and height for their game, both have a minimum of 10 and maximum of 20 to make sure that the game can be set up in a proper way (not having small width or height that the game can’t even work in the first place). In order to ensure fairness in the game, we make all boards having the same width and height. When a player is level 4 with even length, the bricks that drop every five uncleared rounds are located at the middle right column (e.g., column 6 when width is 10).
- Colourblind mode: when the user selects colourblind mode, their graphic display will have all brown blocks instead of bricks with different colours.

Final Questions

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

Answer: After doing this project, we have a better understanding on developing software in teams. We first learned that we should always make a thorough and reasonable plan before doing a group project. If we had not planned well, we would have been confused about what we should do on certain dates since everyone has their own daily plans and different exam periods, it is almost impossible to keep in touch at any time we need to. Having this plan can make everyone complete their tasks without too much communication. In the plan, we first built the structure of the project and then started with some basic classes that consist of the whole project. Then we start to divide the program into different parts, where one part will only be coded by one of us. This way, each part can only be coded by one of us so others won’t change its meaning which may cause some errors. We thought that this was a good idea since programming allows us to build different parts of a program in different classes, and by using this advantage, we built classes without worrying if the changes of one's part of the program would influence others’. However, when we combined the classes and started to make changes to the whole project which would influence all classes,

we faced some problems. Since we were using gitlab to upload and download our codes, when two people modify codes at the same time, it is really hard to combine two newer versions into one (gitlab does not have the “live” editing feature). Thus, others will not code while someone is modifying the code. Instead, we decided that we should use that time to think about what the next step is and how to implement it. One advantage we had is that one of our teammates is in a completely different time zone. Thus, we can basically code 24hrs, while we can have 6hrs per day to discuss new outcomes. Also, we make sure the code runs well and the new feature implemented correctly before we update the whole project to gitlab. During the process, we also had some different opinions when we tried to solve some problems. We learned that since we were developing in a team, we should always have more communication with each other and consider others' ideas before insisting on our own opinions. From others’ opinions, we learned about new knowledge and different methods when implementing specific features. From this experience, we learned both academic knowledge and collaboration development through doing this project in a team.

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

Answer: If we can start over, we would spend more time on planning the whole structure. If we compare the uml diagram from dd1 and dd2, we have added a lot of more fields and functions in the latest diagram. Even though we still have almost the same classes, when we were writing it, we found that all classes lack different kinds of functions and fields, including functions that return values of fields for other classes to use, functions that implement a different kind of object (for example, the single block in level 4, and fields that record certain values which we did not consider about when planning it). For example, we did not plan the functions for “hint” in our program, which would require us to build a copy constructor in Board. Since it was not part of our plan, every time we add a new field to Board, we forget to initialize it in the copy constructor, which causes some valgrind errors. If we have a chance to start over, we will make better simulations of how our program’s classes should be like, and make sure to include as many fields we need as possible to reduce debugging time. Also, in some of our versions, we found some functions had too many lines of code, which may be solved by separating them into different functions or adding helper functions. This is also something we could prevent if we planned the classes more thoroughly. To fix those problems, we may have a decorator for the board to record what level it is currently at, what special action is taking place. The brick class should also have a decorator level that denotes which level it is when the brick was generated. This may make our code cleaner and much easier to follow and understand. What is more, we should have started our testing process earlier, meaning that we should finish coding earlier. This is because many errors were generated during the testing period and we needed to debug, change codes, and test all previous test cases again. This whole process took longer than we thought, so finishing earlier would definitely have reduced our testing time.