Final Project Design Document

Hao Wei Huang, Ethan Li, Lucy Yu

Overview

The Front-end

Main:

Main is responsible for reading in and interpreting command line arguments. Once all command line arguments are accounted for, and after it is ensured that the command line arguments are valid, Main will construct a Game object. Main will call init on Game to initialize all components needed for the game to begin. Next, the Main will call run on Game to begin the game. Once Game::run() terminates, the program will end.

Game:

The main duty of the Game class is to manage the Player and GameDisplay classes and to read in user input. Game's public methods are init, reset, tick and run. Init initializes all components of the game. Reset simply sets all components back to their default state (Game calls GameDisplay and Player's reset, Player calls Board and BlockGenerator's reset, etc). Tick represents the execution of one player turn. Tick will call other helper methods to encapsulate the functionality for reading commands and interpreting shortcuts. When a user command is read in, Game will call that corresponding method on the current Player. During tick, Game will also update the player's score, level, and highscore on the display through GameDisplay when needed. Run simply is a loop that keeps calling tick until a player loses, a player restarts, or the game ends via eof. At the end of each iteration of the loop, the run will change the curPlayer pointer to point at the player whose turn will be next.

Player:

The Player class manages Board, Block and BlockGenerator. Board and Player both store two Block pointers (curBlock and nextBlock). Player is responsible for generating curBlock and nextBlock, and for updating Board's curBlock and nextBlock. Player is also responsible for keeping track of its level and score, however, Game is responsible for calling the Player's getter functions for score, level, and highscore in order to update GameDisplay.

In terms of tracking score as well as checking to see if a Player has lost the game, Player provides public functions for Game to access Player's score and to check if Player has lost. However, the actual functions that performs the checks to see if player lost, and calculating the how much points the Player has earned on a given turn are in Block and Board. Player simply calls those functions to update its score, and to check to see if Player has lost or not.

GameDisplay:

GameDisplay was the biggest change to the design of the front-end of the program. We realized that in order to print the boards side by side for the text display, we would have to print each board line by line in lockstep. Doing this explicitly in the Game class (like we originally intended) appeared too messy, so it was decided that functionality should be encapsulated in a class (GameDisplay). This way, with GameDisplay, once we write an overloaded output operator for it, printing the text display would be as simple as writing std::cout << display;.

GameDisplay is mainly responsible for the text display, and also for updating the score, level and highscore in the graphical display. GameDisplay has access to each of the Players' boards, and uses that to print out the text display of each board. Also, GameDisplay keeps track of each Player's score, level and highscore (which is updated via Game),

Block Generation:

Generation of blocks is handled by the classes NumberGenerator and BlockGenerator. Each Player owns their own separate instance of BlockGenerator. However, each BlockGenerator shares a NumberGenerator. The NumberGenerator is created and owned by the Game class. When Game creates player1 and player2, Game passes a pointer to its NumberGenerator to each Player, and Player uses that pointer to create their respective BlockGenerators.

The reason NumberGenerator was added is because in our original design, we did not fully understand the concept of seeding the rand() function. If each Player's BlockGenerator had its own NumberGenerator, each Player would end up getting the exact same blocks as their opponent. Instead, the NumberGenerator was added, and initialized in the Game class, to be shared by both Players. NumberGenerator is simply a wrapper class for rand() and srand(). The constructor will take too arguments, a boolean flag (willSetSeed), and an int value (seed). If willSetSeed is true, NumberGenerator will set the seed to the value of the seed parameter. The only function that NumberGenerator provides is randNum(), which takes in two int arguments representing the lower and upper bound, and returns a random number in that range.

BlockGenerator provides three public functionalities: setStream, unsetStream, and generateBlock. setStream and unsetStream are used exclusively for the norandom and random user commands. The set or unset the file which BlockGenerator will read from to generate blocks if norandom or random is called. GenerateBlock is the method that the Player class will call, it takes in an int argument representing the level, and generates a block according to the specifications of that level (reading from sequence file, or generated randomly).

Command Shortcuts:

The functionality of command shortcuts are wrapped in the class CommandInterpreter. The CommandInterpreter has two public functions, addCommand, and interpret. CommandInterpreter stores a library of valid commands (in a tree like structure: see Trie and TrieNode). The user can call addCommand to add a command to the library, and the user can

call interpret on a string to see if the string represents a shortcut for, or is, a valid command. If the string is a valid command (or a valid shortcut for one), CommandInterpreter will return the command, otherwise, it will return the empty string. The CommandInterpreter is used by the Game class to handle command shortcuts.

The Back-end

Block:

Overall, the block class is intended to keep track of basic information such as the block type, the cells it occupies, number of cells in the block that is on the grid, and the level the block was created on. The information is crucial because to determine if the block is completely removed from a board when a row has been cleared. Each time a cell in a block gets removed, numCells decreases by one. If numCells reaches 0, it means the block is cleared from the board and we can calculate the score according to the level the block was created on. Once the score is recorded, we make sure to set our counted flag to true to make sure the block get recounted next time the check for if blocks are cleared is called. The main functionality of a block consist of rotating the block, and shifting the block in different directions. Whenever, the blocks moves, the cell the block points to also migrates.

We modified Block to own a list of cell pointers and the movement of the Block will depend on the pointers to the cells. The changes of these cell pointers (i.e. the movement associated with changing where the pointers point to) determine the motion of the Block.

Cell:

The Cell class is the location on the board where blocks would be set. Wherever a block was meant to be located, the cells there would be set with the correct block type or colour and the flag isFilled will be turned on. Once that block is moved the cells it used to be on would be unset and the new location would be set. This was also important for the graphics portion as each rectangle on the graphics would be a cell that would need to be drawn and undrawn to show the movement of blocks. The Cell class is directly responsible for changing where the Block should be but does not directly change other aspects of Block's class. In graphics, cell is responsible for drawing the graphics each time a cell is turned on (isFilled = True) or undrawn each time a cell is turned off (isFilled = False).

The cells will still be responsible for managing the fillType, as well as checking for isFilled or isBlind. The cell has both absolute and relative coordinates to keep track of where the cell and thus where the block should be. Cell no longer has a pointer to a block or its neighboring cells.

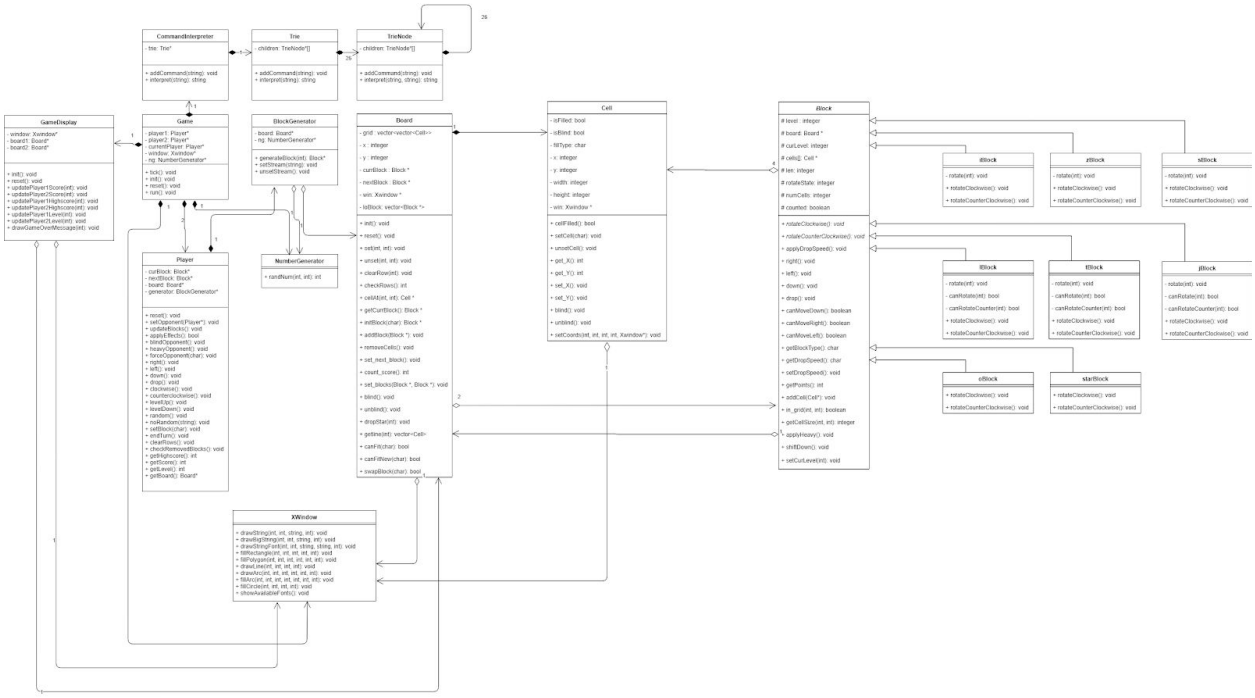
Board:

The Board class is representative of a single board per player. It shows all its Cells, the current block that is being controlled, and the next block that will be added on. Board's intent was to provide an area for the cells to stored in and provide a wrapper like effect in managing

those cells. The Board has an aggregate relationship with the Cell class so when the board is updated, the Cell are as well. It also has a composition relationship with the Block class to keep track of the Block and manage it in list of placed blocks. The key functionality of Board is to check for filled rows at the end of each turn. If a row is confirmed to be clear, board is in charge of reinitializing the locations of the board once the row is cleared. In our graphics, board is responsible for drawing the grid lines and “NEXT:” string on the bottom of the grid. Moreover, it draws the next block about to be placed on the board and initializes the current block at the top left corner of the board.

A new feature added to the board was a list of blocks (formerly in the Player class). The reason we moved this field from the Player class to the Board class is that the Player class should only need to keep track of its score, not calculate it. Now, Board provides public methods for Player to call in order to clear rows from the Board, as well as methods for returning the amount of points earned by clearing rows and removing blocks. Player and Board both have a curBlock and nextBlock field. This is because block generation is handled in the Player class. When Player generates a block, Board will be instructed to point to the newly generated block. This allows Board to access the Block without having to worry about block generation (because if Board also handled generation, it would need to keep track of its current level, which should be done in Player). Another change we made to Board was that Board is now responsible for printing the next block as well as the current block being dropped. Printing the next block was originally intended for the Game class, but we realized this would result in high coupling.

Updated UML (See Submitted PDF)



Design Techniques

Sequence File Feature

The main problem with enabling the sequence file command was that when reading a sequence of commands from a file, one of those commands being read could be another sequence file command. Due to this, implementation of the sequence file command became more complicated than just having an istream variable that we switch between cin and and ifstream. To mitigate this problem, **a stack** (represented by a vector) was used to store all the commands from a sequence file. Each time a sequence file command is called, the commands stored in the file are read in, and added in backwards order to the the stack (or to the back of the vector). Additionally, each time a sequence file command is issued, a boolean flag (readingFromFile) is set to true. A private helper method is used to get the next input, and this method will either pop the top command off of the top of the stack (if readingFromFile is true), or read in a command using cin (if readingFromFile is false). Each time a command is popped off from the file input stack, a check will run to check if the stack is now empty. If it is, the readingFromFile boolean flag will be set to false.

Command Shortcuts

A tree like structure (Trie) is used to enable shortcuts. A Trie contains an array of 26 TrieNodes (one for each letter of the alphabet). A TrieNode has an int field (usedIn) that keeps tracks of how many words in the Trie that the letter that that TrieNode contains is used in. Also, each TrieNode an array of 26 TrieNodes (the array is called “children”). Each level of the Trie represents a letter of a word. For example, a Trie containing the single word “drop” will look like:

d → r → o → p

When interpret is called on the Trie, Trie will simply take the given command fragment, and travel down the tree letter by letter until either a leaf is reached, or the command fragment runs out of letters. If a leaf is reached before the command runs out of letters, then that means the command does not exist, and interpret returns the empty string. If the command runs out of letters and we have not reached a leaf (or if we are at a leaf), we will check if usedIn of the node we are currently in is more than 1. If so, that means the command fragment given is not sufficient to distinguish an actual command, and interpret will again return the empty string. Otherwise, interpret will continue down the trie until a leaf is reached, and return the correct command that the fragment represents (this is using **accumulative recursion**).

Observer Pattern (Cell and Block)

We implemented the observer pattern between Block and Cell. The Block has a composition relationship with the Cells which it notifies to set and unset and keep track of the location of each block. This allows the Cell to notify Block when it has been unset, during the situation when rows are being cleared. Whenever the block moves, it updates what the cell pointers in cell to accurately locate the position of the cells.The Cell acts as the observer while getting information from Block (subject) to display onto board. Through this pattern, we were

able to efficiently pass information to the block to successfully manage and draw both accurately.

Block Inheritance Hierarchy

The Block class has an inheritance hierarchy where there is a base class and each of the different types of blocks are subclasses to the original Block. The base block class has implemented functions for basic move features like move down, right and left and drop. Those move features also has its own helper function such as canMoveDown, canMoveRight, and canMoveLeft that checks for constraints. The rotate functions (clockwise, counter clockwise) are pure virtual because each block has its own ways of rotating. Thus in each of the subclasses, we override the rotate functions. The base class also contains 2 major parameters that are utilized in the subclasses; numCells and rotateState. The numCells represent the number of cells that the block has to keep track of and starts at 4 for all blocks except the star-block (which starts at 1). The rotateStates keeps track of which state it is currently in and that is changed every time a block is rotate per block. The initial rotateState is 1 for all blocks. These features are updated by the subclass block types but are declared in the base class.

Resilience to change

Adding New Levels

Since all the logic for generating blocks are encapsulated in BlockGenerator, the addition of a new level would not affect the functionality of Game, Player, Board or Block. Instead, it will only require an extra if statement block in the generateBlock function, as well as the addition of a function for handling the probabilities required for generating a block on the new level.

Adding New Effects

Since the application of effects on Players relies on a boolean flag based system, the addition of a new effect would only require the addition of a new boolean flag, and a corresponding setter function for that flag (which Game would call on Player). Not much other change would be needed, as the logic for carrying out the effects of these special effects are handled in applyEffects (in Player). Also, the nature of the flag system already supports the option of having multiple effects applied at once.

Adding New Commands

Since the interpretation of commands is encapsulated in the CommandInterpreter object, adding a new command would be simple, as we would not have to manually recalculate which shortcuts are valid for which commands.

Adding More Players

Adding new players would simply mean that Game now manages a new number of Player objects, and that GameDisplay now will need to store the score, level, highscore and board of any additional players. Adding new players would not have any discernible effects on the rest of the code.

Adding New Types of Blocks

The movement of the blocks is very resilient to changes as we do not check according to the type of block for movement (excluding the rotation). If a new type of block would be added, it can be easy incorporated with only the rotation states needing to be added as it would still be able to move left, right, down, and drop (with collision checks), without needing to rewrite any of the code. In terms of graphics, it is also very easy to change colors of the block for presentation. Since everything is all implemented in the cell class, we can easily assign colors for each block (we know the filltype).

If rules regarding the number of lines needed to be cleared to trigger special effects, it is easily dealt with through the design of our program. Since our checkrows function in Board return the number of rows cleared at the end of a turn, we can easily relay the information to Player class which we can change the number of rows needed to be cleared to trigger an effect.

Expanding the Board

Since the board is managed by a vector of cell vectors, it is very easy to expand the board. Since the board class takes care of a single board we would only have to change the dimensions of the vectors to expand the cell. The only component that needs to be changed is the constants the help checks constraint when a block moves right, left, and rotate.

New Graphic layout

It is extremely easy to enhance our graphic design because we let the board draws the grid, and the cells fill in the corresponding status of the cell (whether it is unfilled: white, or filled with the block's color). We have a gamedisplay class that deals with the remainder of the graphics such as player score, level, and highschore.

Answer to questions

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?

The only change we made was having the list of Blocks stored in the Board class instead.

Each block will have a list of pointers to the Cells that are representing the Block on the Board. If the Block is to be removed from the board, we could access those cells through those pointers, and reset the cells to their default state, then update the board so that the other blocks move down. Then we will remove the Block from the list of Blocks stored in the Board class. It can easily be utilized in more advanced levels as we are only going through lists of cells and blocks to reset the values. Generation of blocks can be easily confined to more advanced levels, as it would only require nesting the generation of that block in an if statement.

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

Since Player is responsible for keeping track of its score, introducing a new level would require an increment to the variable in player that stores the max level a player could obtain (to reflect the new level). Also, any effects that the new level would have on the generation of blocks would be accounted for in the private implementation details of the BlockGenerator class. Since player only will ever call generateBlock in Block to get a new block, the way Player interacts with BlockGenerator would remain the same. This does not differ very much from the original answer. However, with the new design, not all of the logic of the player's turn (for example, reading in commands) are handled in the Player class (they are now in the Game class). However, since Game does not deal with any logic revolving around levels, adding new levels should not change the Game class.

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?

As indicated in the original answer, we used boolean flags in the Player class to keep track of the effects being applied on Player. Although our game only allows users to apply one effect at a time on the opponent, our design actually does allow multiple effects to be applied at once. This is because the effects are applied during different stages of the player turn, and as such are incorporated into various functions of the Player class. The logic for applying those effects are wrapped in an if statement, which simply checks if the corresponding boolean flag for that effect is set to true, if so, that effect will be applied. Having multiple effects apply at once will be the same as setting those boolean flags set to true at the same time.

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? How difficult would it be to adapt your system to support a command whereby a user could rename existing commands? How might you support a \macro" language, which would allow you to give a name to a sequence of commands?

Adding Commands:

The theory of the design for solving this problem remains the same as the one described in the original answer. However, the functionality of interpreting commands are now dealt with in the Game class. Adding a new command would require adding that command (using addCommand) to the CommandInterpreter in Game. “readCommands()” is also now in the Game class, and the addition of a new command would require an extra if block in that function to check for that command, and an arbitrary amount of code to the Player class to implement that command.

Renaming commands:

To support renaming commands, we would need a string variable that holds the string that maps to each command (we could also use a Map for this purpose). Changing the name of the command would simply require setting the string variable for that command to a different string value. The new string value would have to be added into CommandInterpreter, and the previous string value for that command would have to be deleted (note that this requires that a deleteCommand method would have to be implemented for CommandInterpreter). This answer is the same as the original answer (CommandInterpreter represents our Trie structure here).

Macro Language:

In the new design, the file input stack vector is a “macro language” representation of a sequence of commands. This differs from the old answer, which uses a Map to store aliases to commands, and can then be used to interpret a list of aliases and execute a list of commands. The new approach is less efficient in terms of space, but more resilient to change, as adding or renaming commands would not affect the file input stack at all.

Final Questions

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

Creating software in teams can be very difficult especially when roles are not well defined and not everyone’s strengths and weaknesses are well known. Furthermore, with our lack of experience in both C++ and developing large software, it made it a very hard but riveting experience. Regarding technical skills, our entire group was able to successfully learn how to use git. Not only did we learn the basic commands of pushing and pulling, we learned how to set it up and link it to our respective editors. Moreover, we learned to handle merge conflicts, preventing merge conflicts, and reverting to older versions.

Relating to designing the program and dividing work, we were able to understand why it is so important to be able to work in close vicinity of each other. It is more difficult to communicate our ideas online and to work on similar parts of the program if we are not all working on it together and beside each other. With that said, it is extremely inefficient to work on every part of the program together. Thus, explicit roles must be declared when splitting tasks.

To ensure everyone understands their roles, all members of the group should first plan together then split up their work. It is very differently from working alone when you can change ideas and processes whenever you want. We learned the importance of planning together beforehand to ensure ease in splitting up roles.

Additionally we learned the importance of planning out every single detail of the program. If one part of the design is left too vague, it increases the chance of having deviations from the original plan. Even the smallest deviation from the plan can compromise the entire program, as other parts of the program will have to be changed to reflect the deviation.

Lastly, we learned how crucial regression testing is. Often, fixing certain bugs, or adding new features would cause other parts of the program that were previously working fine, to fail. Writing test cases to check that the program does not regress to old, wrong behaviour is crucial in ensuring that a program as big as the final project runs with minimal bugs.

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

We decided to split the program into a back-end (Board, Cell, Block) and a front-end (Main, Game, Player, GameDisplay, block generation, command shortcuts). So when the work was split among the group one person was assigned to implement the front-end while the other two group members were assigned to the back-end. However, the front-end turned out to take much less time to implement than the back-end. As a result of this, the linking of the front-end and the back-end was delayed due to issues with the back-end. This pushed the project behind schedule, and left less time for testing than we would've liked. Additionally, since we were planning on implementing graphics at the very end, this also limited the amount of time we had to implement the graphical interface. If we started over, something we would change would be to first work on the back-end collectively as a group, make sure the back-end was working, and then move on to the front-end.

Another thing we would change is that we would've put more careful planning into the graphical interface. When we initially wrote out the implementation details for the project, we did not put too much consideration into exactly how adding the graphical interface would affect the rest of the code. What we implemented was everytime a cell was changed, we would draw or undraw that cell. Therefore, with the movement of blocks, we unset and reset cells one by one. With each set and unset, the cell gets drawn and redrawn. Therefore, the graphics show how each cell is set and unset during any movement of the blocks. This detriments the aesthetics of the graphics. What we should have planned out was drawing the cells after the cells have been moved into place to show one swift movement for each movement rather than all the intermediate steps of showing each cell being set and then unset. This may have made the graphics appear as if it was faster.