Design

Overview

The Project consists of three main modules which are the input and output, block and board modules. The Parser (input) and Display (output) modules are implemented in the Game class that communicates with the Board class and reflects those changes on the output. The Board class implements the two Boards, as a grid of Cells. The class Cell contains information about the type of Block it belongs to and its position in the Board. The Board contains a vector of Blocks which have fallen onto the Board. The Block is also a vector of cells at different positions that make up the seven kinds of blocks in the game. The Board and Block communicate with each other. The Board communicates with the Game after every round of the Game and the changes made to the Board are reflected by the Game class in the Display. The Game class takes in a string that is then parsed into passable input to Board.

Design

Part of the challenge of designing the Game class is handling all the different sources of information appropriately. You have the istream to interpret, the turn to keep track of, the two different boards and their various gamestates to constantly check, as well as the two different types of output to maintain. Simply, it was clear early in the design that writing this all into one large class, let alone function, would be too much to handle. Thus, we decided that it would be better to divide Game into three components: Interpreting input, Calling/Checking Boards in a game loop, and Displaying output. Then, we created two sub-classes to handle the entirety of interpreting input (Parser) and displaying output (Display) respectively.

This greatly simplified things, and allowed Sean to focus on each part one at a time. This was especially helpful as the Parser was easy to tackle early in the program while the Board and Blocks were being built, as Parser could be tested on its own. By the time Parser was fully tested and complete, Sean was able to build the game loop based on the revised versions of Board and Block. This involved several functions being pushed on the Game to handle. This was because while in principle certain aspects of the game should be handled by Board, in practice, it was much simpler and modular to push the responsibility to Game as opposed to the already stuffed Board class.

The ASCII and UI displays were easy to implement even without knowing how the game was run. This is because of how we designed Board and Block's display function respectively to output (in essence) a screen capture of the game as a 2D vector of chars. If this 2D vector was accurate to the true game occurring in the Board and Block class, Display didn't have to know how the game was running. All it had to do was call on Board to give it just the info it needed through getter functions. Implementing Xwindow was a challenge, and was the last component of our program fully implemented because the coordinates had to be hard-coded.

The Board class is decorated with three kinds of Boards, HeavyBoard, BlindBoard and ForceBoard. The special moves in the game are implemented through these classes.
The Board class has its own display variable which is a vector of characters and contains information on what kind of cell is present at each position in the Board. The display is updated through each round of the game and the information is passed on to the display class. There is also a Level class which produces different kinds of Blocks based on the level. There are five levels here so there are five level classes which inherit from the base Level class.

The Block class used inheritance to define different types of Blocks that exist in the game. The Block was also decorated with a class called Heavy Block that will be used in a special case. There is usage of Decorator Pattern here. The Move function to move left, right and down was defined in the Base class. The Rotate function for each type of Block was defined in their respective classes. The Drop function was defined in the Base class as well. Each type of Block had a different implementation of the constructor to construct different kinds of shapes in the game. The Block contains a Board pointer as a member variable in the class which is used to add changes to the Board. It contains a vector of cells located at appropriate positions in the board to draw the Shape.

Resilience to Change

- If we want to add a new kind of Block to the game we can do so by adding a Class of that kind of Block which inherits from the Base class Block. DisplayUI would then be updated to include a new type of color for the new block type.
- We can add a new specification to the Board by adding a Class to the decorator pattern hierarchy. Then, we need to simply add a command in the ‘scomm’ map in parser to accept a new type of command, as well as add another if-statement in Game to accommodate for the proper decoration.
- If we want to add a new type of command, we simply add it to the ‘comm’ map in parser. If the command is meant to map functions together, Game class would have to be edited to include an if-statement to run those functions in sequence, likely with a Command array.
- A new Level can also be added by adding a new subclass that inherits from class Level. Then, we just need to edit the level subclasses below the current level to point to the new level in the ‘levelUp’ function.
- New command-line flags could be edited into the main class, calling the respective Game class function that would also have to be created.
- Any internal changes to the game logic would result in little (if any at all) to the Game, Parser, Command, or Display class. Because the internal workings are handled by Board and Block, Game class has low coupling with Board in terms of game logic.

Answers to Questions

Q: 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?

A: This is simple in our current implementation. When a block is created, it is appended to the end of a vector. When the current block drops into the pile, it notifies the board it belongs to. The board can then check the length of the vector of blocks, and remove any blocks older than 10. Since it is the board's responsibility to remove its blocks, this is well within its capabilities.

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

A: To add a new level, we need to:
1. create the level subclass, which overrides the getBlock() method, and move() method
2. add a pointer from the previous level to the new level, and from the new level to the previous level, to support the double linked list structure.

Q: How could you design your program to allow for multiple effects to be applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination?

A: Multiple effects are already possible in the existing design, since effects are implemented with a decorator pattern. To add a new effect, the following steps must be done 1. create the effect subclass, which overrides the display, and move methods 2. add the effects command name to the Parser class as a member of ‘scomm’ map, and associate it with the class just created in the Game class. In total, there are only two classes that need to change.

Q: 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.

A: To add new command names, we add the command name and syntax to the Parser, Game class adds another if-statement to its ‘play’ command to accommodate for non-board behavior, else gets passed into Board class to handle.
To rename commands, Parser can (on recognizing input as ‘rename’) rename the designated key in mapping ‘comm’ to the new name.
To support the ‘macro’ language, upon command ‘macro n’, Game parses in the next n strings as commands designated to the macro as Command objects, saved to a vector of commands within a map of key strings to value vector of commands. Then, add the string command via the ‘addCommand’ function to Parser to the ‘comm’ map. Now, as part of the ‘play’ loop, iterate over the map of key strings to value vector of commands for every parsed input. If it matches with a key value, run the vector of commands.

Extra Credit Features

- All memory management in the program is done using vectors and smart pointers and very few raw pointers. The program has no delete statements.
- Xwindow display is aesthetically appealing and has minimal hard-coding for coordinates.

Final Questions

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

A: Let’s start with what we found difficult. Compared to the previous assignments, completing A5 required substantially more planning. When coding a project on your own, it's easy to have a mental map of how your program should be designed and work on it from there. It was clear after our first meeting, that each of our mental maps of Biquadris differed from the other. Certain ideas had to be rejected or revised and we were able to amalgamate our ideas into a single functional program. This wasn’t without a lot of work however. We needed to properly communicate our ideas and have them hold against different edge cases our group members could come up with.

This refinement of our ideas was the unexpected positive side of working in a group. When you work alone, it’s easy to develop a blind spot in your code and not see obvious errors in your implementation. Having others around to say ‘what about this?’ is frustrating in the moment, but ultimately leads to a more complete program. This also saves time down the road when trying to debug your code.

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

A: We would have liked to manage our time a little better. There would have been a well defined plan to finish tasks on time. This would have given us more opportunity to explore additional features in our code as well as address bugs that we encounter during testing.

Honestly, we should have allotted about the same amount of time to debug our code as much as writing it, as there was a lot of crunch time down the stretch to get the code completely working. We probably also should have come up with a better testing suite from the get-go to test as we code, so as to not have to handle all the errors at the very end. It was hard to determine where errors occurred when there are 20+ files to keep track of, and it would have been better to have an augmented test-harness to try out each class/batch of files individually.

Game
+ play(istream &) : void
+ setTextOnly(bool) : void
+ setSFT(string) : void
+ setStartLevel(string) : void
+ setSeed(string) : void
1
1
Command
+ getType() : string
+ setType(string) : void
+ getTimes() : int
+ setTimes(int) : void
Parser
+ parse(string) : Command
+ sparse(string) : Command
+ getInt(string) : int
1
2
Level
+ updateBlockCount(bool): void
+ createBlock(char): AbstractBlock*
+ getPrevLevel(): Level*
+ getLevel(): int
+ setRandom(bool): void
+ getNextLevel(): Level*
Level0
+ setRandom(bool): void
+ getNextLevel(): Level*
+ getPrevLevel(): Level*
+ getLevel(); int
Level1
+ getPrevLevel(): Level*
+ getLevel(); int
+ setRandom(bool): void
+ getNextLevel(): Level*
Level2
+ setRandom(bool): void
+ getNextLevel(): Level*
+ getPrevLevel(): Level*
+ getLevel(): int
Level3
+ setRandom(bool): void
+ getNextLevel(): Level*
+ getPrevLevel(): Level*
+ getLevel(): int
Level4
+ setRandom(bool): void
+ getNextLevel(): Level*
+ getPrevLevel(): Level*
+ getLevel(): int
ForceBoard
BlindBoard
+ display() : vector<vector<char>>
HeavyBoard
+ getNextBlock() const : unique_ptr<AbstractBlock>
DecoratorBoard
+ display() const : vector<vector<char>>
+ displayNextBlock() const : vector<vector<
+ move(string) : bool
+ getNextBlock() const : unique_ptr<Abstra
+ levelUp() : void
+ levelUp() : void
+ levelDown() : void
+ getLevel() const : int
+ getScore() const : int
+ getHighScore() const : int
+ restoreRandom() : void
+ setNoRandom(string) : void
+ setFile(string) : void
+ changeCurrentBlock() : void
+ forceCurrentBlock(char) : void
+ forceNextBlock(char) : void
+ gameOver() const : bool
+ setSeed(int) : void
+ setScriptFile(string) : void
+ getCell(int,int) : shared_ptr<Cell>
Board
+ display() const : vector<vector<char>>
+ displayNextBlock() const : vector<vector<
+ move(string) : bool
+ getNextBlock() const : unique_ptr<Abstra
+ levelUp() : void
+ levelUp() : void
+ levelDown() : void
+ getLevel() const : int
+ getScore() const : int
+ getHighScore() const : int
+ restoreRandom() : void
+ setNoRandom(string) : void
+ setFile(string) : void
+ changeCurrentBlock() : void
+ forceCurrentBlock(char) : void
+ forceNextBlock(char) : void
+ gameOver() const : bool
+ setSeed(int) : void
+ setScriptFile(string) : void
+ getCell(int,int) : shared_ptr<Cell>
AbstractBoard
+ display() const : vector<vector<char>>
+ displayNextBlock() const : vector<vector<
+ move(string) : bool
+ getNextBlock() const : unique_ptr<Abstra
+ levelUp() : void
+ levelUp() : void
+ levelDown() : void
+ getLevel() const : int
+ getScore() const : int
+ getHighScore() const : int
+ restoreRandom() : void
+ setNoRandom(string) : void
+ setFile(string) : void
+ changeCurrentBlock() : void
+ forceCurrentBlock(char) : void
+ forceNextBlock(char) : void
+ gameOver() const : bool
+ setSeed(int) : void
+ setScriptFile(string) : void
DisplayT
+ draw(unique_ptr<AbstractBoard> &,unique_ptr<AbstractBoard> &) : void
1
DisplayUI
+ draw(unique_ptr<AbstractBoard> &,std::unique_ptr<AbstractBoard> &) : void
1
DisplayAscii
+ draw(unique_ptr<AbstractBoard> &,std::unique_ptr<AbstractBoard> &) : void
Cell
+ getX() const : int
+ getY() const : int
+ getType() const : char
+ setType(char) : void
*
1
*
AbstractBlock
+ move(string) : void
+ drop() : void
+ Down() : bool
+ rotate(int) : void
+ removeRows(vector<int>) : void
+ getCells() : vector<shared_ptr<Cell>>
+ setBoard(Board *) : void
+ isEmpty() : bool
+ setX() : void
+ getLevel() : int
+ setY() : void
+ getX() : int
+ getY() : int
1
DecoratorBlock
+ move(string) : void
+ drop() : void
+ rotate(int) : void
+ setBoard(Board *) : voi
+ removeRows(vector<in
+ isEmpty() : bool
+ setX() : void
+ setY() : void
+ getX() : int
+ getY() : int
HeavyBlock
+ move(int) : void
JBlock
+ rotate(int): void
+ setBoard(Board*): void
MiddleBlock
+ rotate(int): void
+ setBoard(Board*): void
TBlock
+ rotate(int): void
+ setBoard(Board *): void
ZBlock
+ rotate(int): void
+ setBoard(Board *) : void
OBlock
+ rotate(int): void
+ setBoard(Board*): void
IBlock
+ rotate(int): void
+ setBoard(Board*): void
SBlock
+ rotate(int): void
+ setBoard(Board*): void
LBlock
+ rotate(int) : void
+ setBoard(Board*): void