CS246 Final Project: Biquadris

Overview:

- Block:

  An object that contains a 2D vector of ints storing its coordinates. The object also contains all of the rotation values for each of the various Block types (i.e. Z, S, T, etc.). The object is able to change its coordinates, as well as rotate them, depending on the Block types rotation values.

- Board:

  Board houses a 2D vector of Blocks called grid, along with the functions that control the movement of Blocks within grid. Additionally, Board stores boolean values for the implemented special effects, which change how the board is displayed in the Controller and how blocks are moved in grid.

- Controller:

  An object that stores the two boards used to play the game. After the playGame method has been called, Controller will interpret the user's keyboard inputs and manipulate the current player's board accordingly. After the drop method has been called, it will switch to the other player's turn, and change the current board pointer to that player's board. If the requirements for a special event have been triggered, the method applySpecial will be called to prompt the player for a special event. If there has been a winner, or a tie, displayWinner will be called to display the winner.

- Generation:

  An abstract class that implements readFile(), which reads from a file and assigns those values to a vector and swapRandom(), which sets the random boolean and reads from the passed file name depending on random.

- LevelX:

  Where X is the level, this object is used to generate a character.

- RAIILevel:

  An object containing a vector of shared_ptr<generation>. This way, all the game's levels can be stored in one place. It will pass the shared_ptr to a level using the getLevel method.

- Test:

Interprets command line arguments. Also responsible for starting the game, restarting the game, and prompting the user to play again.

- Window:

  Written similarly to the given window files in A4Q4. Has the constructor and destructor to display and destroy the window. Sets up the function fillRectangle that displays a filled rectangle given its coordinates, width, height and color. Also has drawString, which displays a string given its coordinates and the string message. Additionally, the array color_vals stores colors that are used in the display.

- Tetris_graphics:

  Creates the game grids for both players. Has the function display_block which displays a piece given a reference to the window, a vector of a vector of integers representing the x and y coordinates of a piece, the player type, and the block type. When a player is positioning a piece before deciding where to drop it, the function erase_block is used to erase a piece before displaying its new position.

Changes from DD1:

- Removed all the various Block classes, condensing it into a singular Block class.
  - This was to simplify our files, as the existence of a separate Block class did not add anything of value to the code. Thus it was condensed into one file.

- Added the generation and level classes.
    - Originally, the plan was to have the level and block generation occur in the board. This proved to be difficult to work with, as well as clutter up our code. Thus, the task of generating blocks was given to the factory method.

- Added the RAIIlevel class.
    - Similarly to adding the generation and level classes, all of the levels were going to be stored in the board. This proved to be difficult with managing memory. Additionally, storing the current level of each player on the board class was a bit difficult. Thus, the RAIIlevel class was implemented to help with these issues.

- Adjusted the main.cc to test.cc.
    - Main.cc was originally going to just interpret command line arguments. This was changed, test.cc now interprets command line arguments as well as creating both board classes on the stack, and restarting or quitting the game.

- Split the Display class into window and tetris_graphics.

  Initially, we weren't entirely sure how to implement the display class as none of us had used it before. We decided to split the class into window, which contains X11 functions, and tetris_graphics, which contains functions created by us. We used the instructor provided code from A4Q4 to write window.cc and window.h.

Design:

Techniques and patterns we used to help solve design challenges:

- Factory:

  The abstract generation class with concrete level classes. These classes implement their own methods to generate new Block types (passed as characters). Thus, the genBlock command will vary depending on the level type.

- RAII:

  Shared_ptr are used in the RAIILevel class to manage the memory of all the various levels in a single class. Additionally, Board uses shared_ptr to store all the Blocks in the 2D vector. There are no deletes in our code.

- Polymorphism:

  RAIILevel uses a vector of generation pointers. The object holds the pointers for all levels, which are stored in the vector, making it a polymorphic vector. The virtual and override tags are used specifically in the generation of blocks to dynamically generate them according to the level rules.

- Exceptions:

  Specifically used when adjusting the game's level. RAIILevel throws an error when attempting to access a non-existent level. In the controller, when attempting to use a command like 5levelup, the level will be swapped to level 4, and throw an error on attempting to swap to level 5. The controller ignores this and continues with level 4.

Resilience to Change:

- Level:

  Levels are stored in RAII level as generation pointers. If a new level were to be implemented, one would only need to create a class that inherited the generation class and implement the block generation logic, then add it to the initialization of RAIILevel. Since this is mostly self-contained in the created level, aside from the one line of code added to RAIILevel, it is easy to create new levels.

- Blocks:

  Each block type is stored in the block class itself. That is, each character block has its own specific values stored in the block class. To add a new block, all you would have to do is add the initial coordinates, the block type, and it’s rotation values.

- Commands:

The controller class handles all of the user's inputs. It then uses the methods in the board class to apply those changes. If we want to add a command, we would simply add a condition in the playGame method, and work out the logistics of the command using the board's methods.

Answers to Questions:

- Question 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 add a counter variable to the Block class. When a block is dropped, we go through each Block in the grid. If a block has a counter of 10 or more, the block is removed. Otherwise, it's counter is incremented by 1. Since it is a variable confined to each individual block, the generation of the blocks won't change with the advanced levels.

- Question 2: How could you design your program to accommodate the probability of introducing additional levels into the system, with minimum recompilation?
    - As mentioned in the Resistance to Change section, additional levels can be added with the creation of a new class. There is minimum recompilation as the new level class and RAII would need to be recompiled.

- Question 3: 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 decorator design pattern would be the most appropriate solution in order to implement multiple effects simultaneously. The abstract effect class would call the normal board methods respective to their purpose, while the decorated special effect classes implement their own logic on top. If more effects are create, we can implement more subclasses under the abstract effect class.

- Question 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? 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? Keep in mind the effect that all of these features have on the available shortcuts for existing command names.

  - We can implement this by storing all commands in a vector of strings. This means that by default the array of strings will contain commands like “left”, “right”, “up”, “down”, etc. and then we can give instruction to the user that at the start or during any time of the game, they can press the reserved letter ‘s’ to open settings. Once the user opens settings, a settings function is called where until the letter ‘q’ is received, it:

    - Reads in a) command to change b) new name for command

    - It then finds ‘command to change’ the vector of strings (all the commands), and changes the vector at that index to contain ‘new name’ for command.

    The way this vector of command works is that every time user input is read in (whether from text or graphics), instead of using constant strings to compare and see what the next command is, it uses the vector of commands and each position is associated to a specific function regardless of the string literal at that position.

Final Questions:

- Question 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?
  - This project has taught us the importance of collaboration and clear communication. As the implementation to our code was not commented, lots of changes to the main code from each member would need to be understood by all the members before the code could be changed or adjusted, which resulted in a less efficient workflow. Additionally, Github and Git are quite difficult to learn right off the bat. However, after a week of working with Git and Github, we realized the importance of it when working on projects in groups. Changes were implemented seamlessly, and if there was a change that was not well received, it

could be removed easily. As many of us have worked on group projects without Git and Github, the change in convenience really shows why Git is an industry standard.

- Question 2: What would you have done differently if you had the chance to start over?
    - Overall, our project ran quite smoothly. We made extremely good pace in the first week. However, we began to slow down when attempting to implement the various features that we had not planned out well, or required a design pattern that was thought to be unnecessary. Thus, we could have created a plan that was more concrete and well thought-out by implementing design patterns.