Plan of Attack

Overview
We designed our project to adhere to the MVC architecture. The Display is the model, the Text and Graphics Observers are the view, and the Player is the controller.

Kenny

Kenny will aim to finish the Observers by November 23rd, which will be the view component of our architecture. Kenny will implement all features of the view component, except for the graphics observer. He will implement the Observer pattern, and move on to work on the TextObserver.

To facilitate testing, we will have a hardcoded, dummy Display as the Subject that Kenny will attempt to render with the Text and Graphics Observers while the actual model is being developed.

Afterwards, Kenny will work with Charlie to finish developing the model, including the Display, Block, Cell, and Level classes by November 26th.

Parth

Parth will aim to finish the Game and Player classes by November 25th, which will be the controller component of the architecture. The Game and Player classes will be responsible for calling all of display's methods to orchestrate a game successfully, while communicating with the user to consume commands.

To facilitate testing, since the display will not be complete, Parth will focus on checking the validity of aliases and renaming of commands, thus ensuring that the user's input is properly recognized and mapped to the correct commands.

Once the display is complete, full testing of the series of function calls will be performed, but this will be simpler since the user's input will already be mapped to the proper command logic.

The runGame() and runTurn() functions will be the main challenge of Player and Game, as this is where the game is facilitated and the model must be controlled correctly.

Charlie

Charlie will aim to finish the Display, Block, Cell, and Level classes by November 26th (with the help of Kenny and Parth when they complete the view and controller respectively). Before then, Charlie will focus on testing the state modifying methods (i.e. left(), right(), drop(), makeBlock()). To test, Charlie will run each method outside the confines of a Player as a standalone Display, and check to see if the Display's state has been modified accordingly.

Once Kenny finishes by November 23rd, he will help Charlie finish the model while integrating the Text observer. This will make it easier to see changes in the model's state.

Once Parth finishes by November 25th, he will help Charlie finish the model as well while integrating the controller. This will make it easier to execute commands, changing the model's state.

Synthesis

We will implement the basic game by November 26th, including integrating the model, Text Observer, and controller, all successfully. The 27th will be dedicated to adding the Graphics Observer while testing and debugging. The 28th will be dedicated to documentation. Finally, the 29th will be dedicated to a final review.

If some stages of development take less time than expected, we will consider possible enhancements.

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?

In our UML, we have included a cell class that can track the age of a specific cell through an age field. Precisely, each block has four cells and as soon as the block is dropped, each cell begins to count how many rounds it has existed for. At the end of each turn, indicated by the end of the function runTurn() in the Player class, each cell's age is incremented by one through the cell's incrementAge() function. Moreover, after the incrementing, each cell is checked for its age and if the cell has reached an age of 10, we would simply remove the cell from the board by running the display's killOldCells() function.

To confine the generation of such blocks to more advanced levels, cells can be given a disappear field, such that when the cell is constructed, disappear can be set to true if the block originates from the level 3 or level 4 factory method classes as an example.

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

To accommodate additional levels with minimal recompilation, our design uses a Display class as a model, containing a std::vector of pointers to an instance of each level subclass. Each level is implemented as a subclass of the base Level class, enabling polymorphism. Additional levels can be introduced by creating a new subclass of Level to define the behaviour of the new level, and adding the new level to the levels vector in the Display class. This approach localizes changes to the new level's implementation and the levels vector initialization. Updating the MAXLEVEL and MINLEVEL const fields in the Display class ensures boundary conditions remain consistent. Setting the game to new levels can be done by modifying the levelIndex, which is the index corresponding to the current level in the levels vector via the functions levelUp() and levelDown(). These functions would also update the current level pointed to by the level pointer in the Display class accordingly.

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?

In our current design, our runTurn() method in Display will return a string indicating the special effect that should be passed onto the next runTurn() ("" : no effect, "blind" : blind, "heavy"

: heavy, “force x” : force block of x). To accommodate for multiple effects, the returned effect string will be created by concatenating all of the player’s chosen effects together. For example if Player 1 chooses blind and force T, the returned effect string that will be passed to Player 2’s runTurn will be “blind force T”. In Player 2’s runTurn, the received effect string will be used to initialize a string stream. This string stream will be read. There will be one if statement to check for each kind of effect, no else statements. If blind is read, the Display’s blind field will be set to true to influence how the observer will render the display. The heavy field of the Display will be set to true, which affects the code that is run is Display.right() and Display.left(). If force is true, the current block will be replaced with the specified block.

To accommodate for additional effect types, Player will be given the option to add the effect to the list of effects passed onto the opponent. This corresponds to concatenating the effect to the effects string that is passed to the opponent’s runTurn(). Then, a new if statement will be added to check if the received effect string contains the effect, and react accordingly.

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? (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 are planning to have a map for commands, which holds the user’s alias for the command as the key, and the value to be the actual command. For example, if the user aliases the “left” command as “gauche” then the map would store <”gauche”, “left”> as a key-value pair. Then, to check if a command that the user inputs is valid, we would check the count of the user’s input in the map to check if the key exists. If it is 1, then we would run map[*user_input*] which would return the original command. This would work well as map lookup is fast, and it allows for aliasing on top of renaming commands. To support a macro language, we would once again use a map, where the macro name is the key, and the value is a vector of commands that should be run in order.

These changes would complicate the shortcuts feature, as the shortcut could now be ambiguous, and so we would create a String findUnique(string) function that would return the unique string the shortcut is a prefix to or an empty string if there are zero or more than one of such commands. To limit ambiguity, we would prevent the user from adding aliases that are prefixes of existing aliases or have existing aliases as a prefix.