November 21, 2018

Walker Hildebrand, Amirali Sharifzad

Project Plan

Deadlines:

Where RED is Amirali's work, BLUE is Walker's work and PURPLE is group work.

Description	Date (2018)
Interpret command line parameters • Error check and parse the parameters passed in when executing the program	Monday, November 19
Interpret action commands (rotate, move, restart, etc.) • Error check and parse the commands that a user enters into stdin	Monday, November 19
Due Date 1	Wednesday, November 21
Create a basic version of the important classes: Block (and derived), Cell	Wednesday, November 21
Create a basic version of the important classes: Game, Player, Board, Level	Wednesday, November 21
Add functionality for moving blocks • Allow the user to move falling blocks on the board, with error checking	Thursday, November 22
Add functionality for rotating blocks • Allow the user to rotate falling blocks on the board, with error checking	Thursday, November 22
Print display (text)	Friday, November 22
Add functionality for dropping blocks • Move falling block down until it can go no further • Convert falling block to a placed block on the board	Friday, November 22
Implement "restart" functionality • Clear the boards and points, etc.	Monday,, November 26
Clear filled lines from the board • Implement functionality to check for filled rows and then clear them	Monday, November 26
Trigger special actions	Monday, November 26
Display boards (window)	Monday, November 26
Implement block generation (from sequences in files) • Parse input files in order to generate blocks • Loop when the file is exhausted	Monday, November 26
Implement block forcing (L, O, Z, T, etc. commands) • Implement functionality for choosing your block via command	Monday, November 26
Calculate points • Calculate the points a player receives after each turn based on the number of lines cleared, blocks cleared, and level	Monday, November 26
Random block generation • Allow blocks to be generated randomly in certain levels dependent on the	Monday, November 26

seed at the beginning	
Implement Win/Lose functionality	Tuesday, November 27
Implement “Heavy” effect ● Both on levels 3 & 4 and as a player effect ● Stack multiple heavy effects ● allow 3left before falling once	Tuesday, November 27
Implement “Blind” effect	Tuesday, November 27
Implement “Force” effect	Tuesday, November 27
Implement 1x1 block for level 4	Tuesday, November 27
Improve on the design of the project (dependencies) & fix bad code	Wednesday, November 28
Finish coding project (no enhancements)	Thursday, November 29
Finish the first draft of the document	Thursday, November 29
Finish coding project (with enhancements)	Friday, November 30
Finish the final document	Sunday, December 2
Due date 2	Monday, December 3

November 21, 2018

Walker Hildebrand, Amirali Sharifzad

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?

A Boolean field (`disappearing`) could be added to the block class to indicate whether the block should disappear after a certain number of turns. Two Integer fields (`life` and `startingLife`) could also be added to the `Block` class which would represent the number of turns until the block would disappear. After each turn, if no rows were cleared, we would iterate through all of the blocks that are `disappearing` and then reduce their `life` by one. If after a turn a row is cleared, all of the `disappearing` block's lives would be reset to their starting life. Once a block reaches a `life` of 0, the block and all of its cells will be removed from the board.

To restrict the generation of `disappearing` blocks to more advanced levels, a public method could be added to the `Block` class which sets the fields of the `Block` object. By default, blocks would not be disappearing, but then in more advanced levels, blocks would have the method called on them to make them `disappearing`. Additionally, this would allow us to dictate how long a block would take to disappear depending on the level.

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

Our design will feature an abstract `Level` class with a pure virtual `getBlock()` method that returns the next block in the concrete subclasses in which `getBlock()` is overridden. `Level` will also have other fields that dictate what special effects (such as being "heavy") a `Level` has. This means that in order to add a level, a single subclass that inherits `Level` would need to be added, without any additional change to any code that uses `Level`s.

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?

Our design would include an enumerator for each type of effect. The `Player` and `Level` class will contain a vector of `Effect`s that are currently affecting the player/level. When an effect may be triggered, the `Effect` is looked for in the player and level, and then the respective code for the effect will be executed for each occurrence of the specified effect. There will not be a need for one else-branch since we would be using a vector of `Effect`s.

Walker Hildebrand, Amirali Sharifzad

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.

Our design includes an enumerator for each command that the user can input, paired with a string that represents the text for executing the command. The program would take in command input and then return all of `Command` enumerators that match. If exactly one was to match, then the code for the command would be executed. This means that in order to add a new command, all one would need to do is to add a Command enumerator and a string pair. A command for renaming would be as easy as adding the command pair and then updating the string that matches the given command.

In order to support a "macro" language, we could abandon the enumerator-string pair and instead use a `Command` class. This would allow new macros to be added at runtime. The `Command` class would have a field for command arguments and a field (`subcommands`) which is a vector of other `Command` pointers. For base commands, this vector of commands (`subcommands`) would be empty, but for macros, this would contain all of the other commands (with their arguments) that are to be run when the macro is executed.