CS246 Final Project Plan of Attack

ejwang family: Eric Wang, Bob Zhang, Kevin Zeng

Introduction

The following breakdown of our plan of attack is in chronological order, as we plan on implementing different parts of our program at different times. The reason for doing this is so that program dependencies can be created properly (for example, our Cell class will be both an Observer and Subject, so those two base classes should be created before Cell). In addition, this helps keep our program modular, since different classes/functionalities are implemented in separate steps. Lastly, planning out and creating our project in different steps helps us see whether or not our project is following optimal design practices such as low coupling, high cohesion, etc.

Breakdown

1. Boilerplate Subject, Observer, main, Cell class code, commands will be added in one at a time as the features are implemented, There will unit testing for each added component with an integration tests for bigger features
2. Game board initialization
    a. Cell class: holds row, col coordinates and which block occupies it
    b. Board class: Has **vector<vector<Cell>>** to represent the board, TextDisplay and GraphicsDisplay observer **unique_ptrs**
    c. <vector<vector<Cell>> should be filled to have 18 rows and 11 columns so create a vector of cells, emplace 11 cells into it and emplace that vector into another vector 18 times
    d. Coordinate system for board: top left is (0, 0) **use (row, col) NOT (x, y)**
    e. Overloaded friend << operator that calls Text Display's <<'s operator (similar to A4Q4)
    f. Text Display has vector<vector<char>> to represent the board
    g. Text Display constructor initializes an empty board with placeholder text for Level, Score and high score on the top, Next block on the bottom
3. Create single Block class that contains the coordinates for the body of the block
    a. The block class will NOT have 7 different subclasses for each type of block
    b. Blocks only differ in shape but have the same rotate, movement and positioning systems
    c. Each block will hold 4 coordinates that make up its body, the block's initial coordinates will be determined by the block name passed into the ctor
    d. Idea: to use 4x4 array to keep track of all possible rotations of the block
    e. Block should disappear after 10 blocks have fallen
4. Spawning a block at (0, 0)
5. Create Level 0 (Should be apart of the board class)
    a. Level 0: takes blocks from file argument supplied on command line
6. Implement basic commands
    a. Left, right, down, drop, restart
7. Moving and Rotating a block
    a. Moving a block left and right
    b. Dropping a block, collision detection

c. Collision detection idea: create a “potential block” and check to see if its coordinates intersect with either the walls or other occupied cells
d. Dropping a block should trigger the next one to spawn, control moves to falling block

8. Line clearing and scoring algorithm
    a. Whenever a block is placed, check to see if rows are filled, when a row (vector) is filled, remove it, emplace an empty row at the top so the blocks shift down by one, check to see if rows are filled again
    b. Need to check if falling blocks clear another line
    c. Points += (current level + number of lines) ^ 2
    d. When a block is completely removed: Points += (level block was generated in + 1) ^ 2
    e. Current score replaces high score when it exceeds it, high score stays when restart is issued but is gone when program terminates
9. Other Levels and Next Block Algorithm
    a. Utilize **factory method design pattern** for next block decision making process, have this next block show in displays
    b. Have “Level” superclass and specific level subclasses
    c. Base virtual method would be generatePiece which generates 1 of 7 blocks depending on the probabilities in the subclass
    d. Need a random number generator based on probability (rand and srand from <cstdlib>)
    e. Level 0, 3, 4 have overloaded and overridden generatePiece functions to take input from file
    f. Levels all have an isHeavy field, 0 - 2 have it false, 3 - 4 have it as true
    g. Level 4 has another quirk where it needs to keep a counter and everytime it gets to 5 its generatePiece method would call Grid::setPiece to spawn a 1x1 block
    h. Expected use: while (!isGameOver) { spawnPiece(level->generatePiece); }
10. Implement Testing Commands
    a. Levelup, leveldown, norandom, random, I, J, L, etc. to change current block
11. Command-line Interface
    a. Text, seed, scriptfile, startlevel n
12. Multiplicative commands, auto distinguish / auto complete partial commands, renaming existing commands, macro (sequence) commands, reject invalid input
13. Hint command
    a. Find a way for game to tell the player the best spot to place a piece indicated by “?” in TextDisplay and the colour black in graphics display
    b. Hint should disappear when next piece spawns
14. Integration Test with Text Display and game so far
15. Graphics Display
    a. Should also display key elements like current score, high score, next piece in addition to the game board
16. Extra Features
    a. Common: Hold command, store high score and user settings in a file, arrow keys and keyboard instead of commands
    b. Useful Additions: ghost piece, real-time, AI

   c. Cool but not necessary: Music, More levels (alternating S and Z level, EXTRA heavy level), Modes: Hidden(when blocks are placed display does not show them, Hardrock: Play upside down, blocks are extra heavy, Expert mode: pieces spawn in the middle of the board instead of top left

17. Final Testing of extra features and graphics display

Estimated completion dates

These estimated completion dates will correspond to the various steps under the "Breakdown" heading.

1. 3 hours (coding) - Finish by November 27
2. 2 hours - Finish by November 28
3. 4 hours - Finish by November 29
4. < 1 hour - Finish by November 30
5. 1 hour - Finish by November 30
6. 2 hours - Finish by November 30
7. 3 hours (rotations + collision) - Finish by December 1
8. 2 hour - Finish by December 1
9. 3 hours - Finish by December 1
10. 2 hours - Finish by December 2
11. 2 hours - Finish by December 2
12. 2 hours - Finish by December 2
13. 2 hours - Finish by December 3
14. 1 hour - Finish by December 3
15. 2 hours - Finish by December 3
16. If there is time at the end of our project
17. 2 hours - Finish by December 3

Responsibilities

Bob: Block class (moving, rotating, dropping blocks, collision detection), multiplicative commands, command autocompletion, GraphicsDisplay class, main file
Eric: Game board initialization, Game board state, TextDisplay class, command-line features
Kevin: Scoring, Line clearing, Levels, next block generation, testing commands, block rotate, collision detection

Questions for Quadris

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?

Answer: Each Cell should have a integer counter variable that keeps track of how many turns it has been since the Cell has been placed on the screen. Upon the placement of a block, each of its Cells' counter variables will start off at 0, and will be incremented by 1 each time a new block is placed. Once this variable reaches 10, remove the Cell. This could be easily confined to more advanced levels by including a conditional statement, that checks to make sure this clearing of blocks only occurs if the level is advanced enough (level greater than 3, for example). In the class that handles levels, we would vary the way we create new blocks/Cells; if the user is at a level in which this removal of blocks should be included, an additional boolean flag could be passed to indicate that the counter variable needs to be incremented each turn. Otherwise, simply create new Blocks/Cells without worrying about the counter variable (if the current level is not advanced enough). This process of removing blocks could also be placed inside a function, so that the conditional could call this function only during more advanced levels.

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

Answer: We could create a separate class that controls/contains all the code regarding the levels (handles things such as levelling up, probability of blocks in each level, etc.) This way, when a new level is added, only that class (and possibly a very small amount of other classes/files which are impacted by this Level class) need to be changed and recompiled. However, the bulk of the code will remain the same, and will not require any change or recompilation. This is because things such as the dimensions/functionality of the Grid, algorithms to rotate the blocks, etc. do not vary depending on the level, and are instead constant throughout the game. Designing our program this way will accommodate the possibility of introducing additional levels into the system, while also minimizing recompilation and maintaining good design practices such as MVC and low coupling.

Question 3: 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.

Answer:

Autocompletion Accommodating for New and Renamed Command Names

How our command autocompletion will work will allow autocomplete to be easily adapted to work with newly added or renamed commands. Our autocompletion will determine if the given input is a substring of only a single available command name. If so, the command that contains the input will be executed. The available commands will be stored in a vector of strings. Command names will be stored and read from a text file. Adding a newly added or renamed command to the file will allow that command to be executed when the user provides a substring of that command (which is only a substring of that command and that command only) as input.

In order to add a new command to our program, an extra ‘else if’ conditional block needs to be added to our main file and code for the command must be incorporated in the Grid class, as the Grid class will process all commands.

Command to Rename Existing Commands
A command which renames existing commands would only need to automate a variation of the process detailed above. Input could come in the form of “rename <oldname> <newname>” where oldname is the current name of the command and newname is the string which will replace the old command name. The rename function will first make sure that newname is not the same string as another command. If newname matches another command, an error message will be shown and oldname remains unchanged. The rename function will then find oldname in the vector of strings used in the autocomplete code and replace it with newname. Then, in main, the conditional statement comparing user input to oldname will need to compare user input to newname. This could easily be done if oldname was stored in a string vector or variable in main. By setting the value of the variable to newname, user input equalling newname will perform the operation that used to be specified by oldname.

Command Macros
Command macros could be implemented as follows: When the name of a macro is inputted, the string contained in cin will be manipulated to include the list of commands to be executed by the macro. This could be done via a combination of cin and stringstream. This will execute the commands in the desired order.
For small macros, it may be feasible to copy over the code of each desired command into the macro’s conditional statement.