Project: Quadris
Group Members: h2cha, j565lee

Overview

1. Design of the game
    a. Explore features of the game and the specifications
    b. Consider the design pattern
    c. Possible bonus features (if decide to add)
    d. UML
2. Breakdown of time management and each group members' roles
3. Project development
4. Testing
5. Debugging
6. Final markup

Plan of Attack

Complete by November 25th 2016 (Due Date 1):

- UML for the program
- Plan of Attack
- Workload separation

The first week will be focused on the design of the UML, and then splitting the workload depending on the difficulty and by significance of completion order. UML is expected to be finished by Thursday alongside with the plan of attack document.

Complete by December 5th 2016 (Due Date 2):

- Development: .h files and corresponding .cc files
- Merging code
- Testing
- Debugging
- Update UML
- Include bonus features if time is allowed
- Final design document

All group members would be involved with the development:

November 25th -26th, 2016

- All: Create .h files with .cc files that have incomplete methods so that issuing the command **make** builds **quadris** without error.

November 27th - 28th, 2016

- H2cha: Implement Cell, TextDisplay, and Board by employing the **Observer Pattern.**
- J565lee: Develop Controller according to the assignment specification.

November 28th - 29th, 2016

- H2cha: Complete TextDisplay and start working on GraphicsDisplay.
- J565lee: Develop Level and its subclasses using the **Factory Pattern.**

November 30th, 2016
- H2cha: Finish GraphicsDisplay and develop Board
- J565lee: Implement Block and its subclasses and help h2cha to develop Board

December 1st - 3rd, 2016
- All: Put all classes together to build **quadris**. test and debug it.

December 4th, 2016
- All: Finalize the required documents with the final UML

December 5th, 2016
- All: Plan and simulate for the demo.

Q & A

Q1a: 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?

A1a: Add a private field called 'age' to Block and its accessor method. Increment it whenever blocks fall. Delete all blocks in the board who are older than 10 for each turn.

Q1b: Could the generation of such blocks be easily confined to more advanced levels?

A1b: Yes. Let the class Level decide what blocks to be deleted.

Q2: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

A2: Implement an abstract class Level and add subclasses whenever we want an advanced level. And let Level have the operations(move, rotate, etc.) of block such that the operations are done based on the level.

Q3a: 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?

A3a: Implement the public methods of Model as general as we can. Thinking of them as different types of lego blocks, a change or an addition of command simply can be built by combination of those methods at Controller only.

Q3b: How difficult would it be to adapt your system to support a command whereby a user could rename existing commands?

A3b: Not so much. Let Controller have the string fields for each existing commands and work with those fields instead of fixed string.For example, if (command == right) {...}. Whenever we want to rename, assign that field with different string. However, the new name has to be checked if it does not conflict with other names.

Q3c: How might you support a “macro” language, which would allow you to give a name to a sequence of commands?

A3c: Store commands in a istringstream. And have them as a map with sequence name and its corresponding istringstream. Whenever the command is one of the sequences, get input from its istringstream.

Q3d: Keep in mind the effect that all of these features would have on the available shortcuts for existing command names.

A3d: We will be going to implement the shortcuts by following: 1.store all commands in a vector. 2. Count the number of commands that has the given command as a substring. 3. If the number is 1, identify the given command as a valid shortcut. If not, the given command is an incomplete command.