Design Document

Jiazhan Song, Robin Pan, and Pranjal Raihan

Plan of Attack:

Est. Completion Date	Activity	Responsible Members
Nov. 17	● Review project specifications, guidelines and time constraints ● Prepare design document	All members
Nov. 18	● Brainstorm project design ● Rough sketch of UML (base classes)	All members
Nov. 20	● Setup git repository ● Setup build system (CMake/Make)	Pranjal Raihan
Nov. 21	● Complete UML diagram ● Set up header files according to UML	All members
Nov. 22	● Finalize and submit design document	All members
Nov. 24	● Implement simple and data-only classes	Robin Pan
Nov. 24	● Implement Event ● Implement ObserverSlot	Pranjal Raihan
Nov. 27	● Implement Level	Robin Pan
Nov. 27	● Implement Block	Pranjal Raihan
Nov. 27	● Implement TextDisplay	Jia Zhan Song
Nov. 30	● Implement CommandLineArguments	Robin Pan
Nov. 30	● Implement CommandInterpreter	Pranjal Raihan
Nov. 30	● Implement Board	All Members
Nov. 30	● Implement GraphicDisplay	Jia Zhan Song
Dec. 1	● Bonus features	All members
Dec 4	● Testing ● Final revision of project ● Writeup	All members

Important Classes

Event
The Event class semantically represents an event in the progression of the game. It maintains a list of listeners/observers that can subscribe to the Event. The owner of an Event can notify all observers. Additionally, the template parameters to the Event class define what kind of data is passed from the subject to the observers when notified. This class is movable-only.

ObserverSlot
The ObserverSlot class is a friend of the Event class. These two classes are tightly coupled together to form the basis of the subject/observer pattern. An ObserverSlot represents one observer of one Event at a time (or nothing in its empty state). It employs RAII to tie the lifetime of the observer callback to this object. On destruction, the observer that this ObserverSlot represents is removed from the associated Event. This approach ensures that observer lifetimes are very easy to track - much like how std::unique_ptr makes it easy to track the lifetime of resources.

Level
The Level class represents the different difficulty levels available to the program. Each sub-class represents a single level and inherits the operations from the base class. A sub-class is responsible for generating the next block in a game of Quadris. Block generation relies on a specified sequence that is stored in the class. As well, all Level subclasses have a unique score-generating function.

Block
The Block class is a base class whose derived classes represent different block types that can appear in game. Each block has a public position which represents the topmost and leftmost part of the block. The protected field, occupiedPositions, contains the position of every cell position occupied by the block relative to its position.

Display
The Display class is a base class for the different ways that the game is shown to the user, namely, via text in standard output (TextDisplay) and graphically with X11 (GraphicDisplay). This class has no public interface. Its only responsibility is to set up ObserverSlot's for Event's in the game so that TextDisplay and GraphicDisplay may easily implement their functionalities.

TextDisplay
The TextDisplay class handles the standard output of the program. This would print the state of the board after each command.

GraphicDisplay
The GraphicDisplay class creates an X11 window that represents the state of the board. This window is updated after a game's Event's.

CommandLineArguments
The CommandLineArguments class parses argv to get command line argument values

CommandInterpreter
Parses in-game commands from standard input and creates a corresponding Command object for the Board class to execute.

Score
The Score class stores the score of the current Quadris game, as well as the high score of the current run of the program. The class raises its two events whenever score or high score are updated.

Board
The board class represents the grid which the Quadris game is being played on. A 15 by 11 2D array of cells is used to store different block states. Board is an observer class which uses events to notify its observers (TextDisplay and GraphicDisplay) of changes in a game.

Design Decisions
Most of the abstractions in our program are achieved through object oriented principles, namely inheritance-based polymorphism. This is evident in our Block, Display and Level classes, where commonalities are expressed in the base classes only. Because of this design, multiple implementations can exist completely independent of each other.

The different functionalities of the program are broken up into classes (separation of concerns) so control flow is easy to follow. Composition is frequently used so that object lifetimes are easy to predict. RAII is heavily used: even dynamic allocations are RAII-enabled with the help of smart pointers.

The subject-observer pattern is used for the board to communicate to the displays. This decouples the display from the board itself, forcing the display to act only on the data provided. All non-trivial communication is made explicitly through events.

Questions
Question:
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:
To allow for generated blocks to disappear if not cleared before 10 more blocks have fallen, an age field can be added to the Cell class described above. A new active Cell has its age field set to 0, and this field is updated every time a new Block object is generated. This gives Board the ability to keep track of the age of each Cell in its two dimensional Cell array. After the current block has fallen and the new active Cells have their age set to 0, the Board will iterate through two dimensional Cell array and clear any Cell that has reached an age of 10 (or more) while ignoring inactive Cells. Before a new Block is generated, all the active Cells will have their age incremented by 1 accordingly.

This feature can be limited to certain levels by adding a virtual function to the Level class which returns an int representing the lifespan of a block. The base level class can return the sentinel value of 0 to represent the block has infinite lifespan. Board would be referring to this returned value to determine which block to delete.

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

Answer:
To accommodate for introducing additional levels, have an abstract Level class. The different levels will be accounted for in the derived classes, and adding a new level will simply be a matter of adding a new Level subclass. The Board also needs to aware of the new level when executing levelup and leveldown commands.

Question:
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:
To create the most general approach to interpreting commands, use a CommandInterpreter class which parses input and creates a command object for the board to execute.

To allow for renaming, there needs to be a rename command. The CommandInterpreter class will contain a map of strings to commandType enumerations instead of constants. To rename a command, the rename command should change the map. An exception could be the rename command, depending on whether it is desirable for the rename command to be renamed.