Introduction

As the final project, our group decided to implement the game of Quadris, a non-real time analogue of Tetris.

Overview

The data components of the project include

1. `Block`: currently, the game supports 7 block types: `I`, `J`, `L`, `T`, `O`, `S`, and `Z`. Additionally, an additional block type marked by the symbol '`?`' is used for hinting and in level 4, the block '`*`' is used as a penalty for not clearing a row in 5 moves, as stated in the guideline for level 4.
2. `Cell`: stores the content at each cell in the grid. Each cell stores the following fields:
    a) `numBlock`: a unique identifier that maps each cell to each block that has been dropped. Used for scoring purposes
    b) `level`: the difficulty level at which the block stored at the current cell was generated at. Used for scoring purposes
    c) `row`, `col`: the row and column of the cell. Indicates position
    d) `rotation`: information of rotation of the block at cell. Used for hinting purposes
3. `Level`: a class enumeration that is used to identify the current difficulty level
4. `Move`: each valid game move is assigned an identifier in this class enumeration. Members of this enumeration are currently: `Left`, `Right`, `Down`, `Drop`, `CounterRotate` and `Rotate`

The control components of the project include:

1. `Grid`: the most essential class that contain instances of other classes, including `TextDisplay`, `BlockHolder`, and `ScoreCounter`. It stores the current game state using a 2D vector array of `Cells`.
2. `ScoreCounter`: a class to store the high score and update current score of the player
3. `TextDisplay`: outputs the game grid onto screen as text
4. `BlockHolder`: holds the current and next block. Whenever the user enters a command, the command is interpreted and then sent to `BlockHolder` to make the appropriate modification to the current block, and the game grid.
5. `GraphicsDisplay`: creates an instance of `Xwindow` that is responsible for graphical display of the game
6. `HintGenerator`: an independent class that score the position of the current block and generates a hint regarding where to place the current block. The hint is generated by recursively testing for optimal locations to place the current block with respect to the current level of difficulty.
7. `Difficulty`: an abstract superclass whose subclasses are concrete implementations of each level of difficulty. Subclasses specify how each difficulty level acts on a block when given a certain command. Current subclasses are: `LevelZero`, `LevelOne`, `LevelTwo`, `LevelThree` and `LevelFour`.

Updated UML

[See separate submission]

Design Techniques

1. Use of the visitor pattern and polymorphism: in the essential classes, `BlockHolder` and `Difficulty`, the visitor pattern was adopted. Each level (`Difficulty` subclass) is part of the "factory" as it mutates the game board and blocks according to current level. Thus, in the `BlockHolder` class, polymorphism

eliminated to handle each level on a case-by-case basis as a dispatching function could be applied to each Difficulty object, which had its independent behaviour depending on the level of difficulty which it was supposed to model.

2. Smart pointers: to eliminate the possibilities of memory leaks caused by the use of pointers, there was extensive use of smart pointers such as `shared_ptr` and `weak_ptr`. By doing so, ownership relations and association relations between classes were clearly modelled and there was no need to implement destructors for any of the classes. Furthermore, the use of smart pointers eliminated the need to use pointers directly, and thus ensured that there would be no crashes or uncollected garbage caused by incorrect use of pointers.

Resilience to Change

1. As the visitor pattern was applied and each level is a separate subclass of `Difficulty`, the addition of a new difficulty simply involves adding a subclass of the `Difficulty` abstract superclass, and all behaviour of the new level of difficulty can be modelled solely by the new subclass. Hence, compilation is limited to a minimum as this new level of difficulty only has to be added to the vector storing difficulty levels in the `BlockHolder` class.
2. Adding a new `Block` is also quite easy as by default, the `Difficulty` class provides an algorithmic method of rotating and moving blocks that is independent on their specific shapes. Hence, defining a new block simply involves adding a new type of block to the `type.h` header file, and specifying the default initial position for the newly added type of block to the constructor of `Block`, based on the `Shape` of the new block.

Answers to Project Specification Questions

1. Each level of difficulty is a concrete instance of the `Difficulty` abstract superclass, and has its own `drop()` method. Hence, in order to allow for some generated blocks to disappear from the screen when less than 10 blocks have fallen for a particular level of difficulty, an instance of the `Grid` is contained within the `Difficulty` abstract superclass. Then, in the `drop()` method of the selected level of difficulty, the decorator could modify the provided instance of the Grid, thus allowing such a rule to be easily added and confined to certain levels of difficulty.
2. Each difficulty level has its own concrete implementation of the `Difficulty` abstract superclass and thus, adding a new level simply involves a new level. Specifically, the Difficulty object associated with each level serves as a specifies how an action such as rotation, dropping or moving affects an existing Block. With this project structure, recompilation is limited to a minimum when a new difficulty level is added into the game as one simply needs to create the aforementioned `Difficulty` subclass, and add an instance of this subclass into the `BlockHolder` class, which contains a list of Difficulty objects that represent each level.
3. Adding a new command would only involve changing the mutate dispatching function found in `Grid` and `BlockHolder` by adding an additional `switch-case` statement that handles the new command. Furthermore, adding a system that supports macro languages can also be easily done using the interpreter structure, which supports both command binding and shortcuts for these added commands. The implementation of this is explained in detail in the next section "Extra Credit Features."

Extra Credit Features

A command interpreter that supports macro definitions was added. This feature was challenging as new macro definitions must support for names of partially entered macros to be recognised, in the manner outlined in the project specification. The solution was to maintain a list of aliases for primitive commands and macro-defined commands.

Then, when a command is to be interpreted, the command interpreter first attempts to find a unique definition beginning with the same prefix as the entered command. If the process is successful, then the command is executed and based on the return value of running of the command, an appropriate output is given.

Furthermore, when the user enters an ambiguous command, a list of suggestions is presented to the user, and this is also handled by the command interpreter as the command interpreter maintains a list of potential suggestions for each partially entered command.

Finally, to support testing, a special form `!!` was added, which repeats the last successfully executed command. This feature was implemented by keeping track of the last successfully executed command and updating it after a command is successfully executed.