CS 246 W 2014: Final Project

Due: 05:05, Friday, March 28, 2014

Group Members: Sina Motevalli Bashi Naeini (20455091), and Emilio Verdugo Paredes (20452921)

Project Chosen: Quadris

For our design plan, we decided to talk about the implementation of each individual class.

- *TrieNode*: The trie was simply implemented to be able to have the autocompletion feature, based on the implementation from Assignment 3. The class is a simple node to a trie, with some data fields to hold pointers to children and a value to indicate whether the node represents a word. The functions included allow one to insert a new word into the trie, and to autocomplete a word, given that it has only one possible autocompletion. There are also two more functions that the autocompletion uses, but they are not meant to be used by the client.
- *Level*: The level class stores data for each level and generates blocks based on the data. Each instance has a pointer to a stream and a array of probabilities. If the stream pointer is not null, the level attempts to read block types from the stream it points at, and otherwise it will use the probability array to generate a type of block. This class implements the Factory Method design pattern to generate the blocks that the Board uses.
- *QuadrisDisplay*: This is an abstract class designed to be inherited by more concrete instances of a display type. It only contains a pure virtual method called `update` that takes a pointer to a Board and uses the accessor functions of said Board to update the display.
- *TextDisplay*: This is a simple text display as shown in the project description. It inherits from QuadrisDisplay, and it has no additional fields or methods other than the implementation of `update`.
- *FancyTextDisplay*: This class improves upon the simple text display. It has a more aesthetic presentation, and it also introduces a shadow feature that makes it easier to see where the current block will land. It also inherits from QuadrisDisplay.
- *XDisplay*: This is a simple graphical display, as required in the project description. It also inherits from QuadrisDisplay, but it has an additional field for a Window object, to let it interact with the screen.
- *FancyXDisplay*: This is an enhanced version of the XDisplay. It inherits from QuadrisDisplay, and much like FancyTextDisplay, it simply adds aesthetics and the shadow feature. It requires a larger screen resolution, though, and it will crash (not gracefully) on systems without the requisite resolution.
- *Window*: This is the provided window-generating class. Not much is known of its inner workings... (except that it leaks. A lot)
- *Block*: The block class is simply a data storage structure. Each instance simply holds the type of block, the level it was generated at, how many cells it currently occupies on the board, and an array indication its shape for when it is being moved. The array ended up being redundant, since we opted for a similar array contained in the Board class instead, so a future optimization could eliminate the need to store such an array for every cell.

- *Board*: This is the class that contains the game itself. It does most of the work, and it has several accessors and public methods that allow the other classes to progress the game through it. The main feature is an array containing an entry for every cell in the board, each entry containing a pointer to the block that occupies it or `NULL` if there is no block. It also has a pointer to the block being currently moved, and a pointer to the block that will drop next. It also contains several arrays that describe the position and orientation of the current block (and also its shadow). The Board also contains several Level objects to denote the several levels available, and a field to determine the current level. These arrays are what made the Block's arrays redundant. It also has two fields to keep track of score and high score, and a few internal methods that merely simplify the coding. In terms of public functions, it has accessors for the current level, for the type of the block at each cell, the position of the shadow cells, the orientation of the next block, the score, and the high score. It also has methods that allow the client to start a new game, change the level, move and rotate the current block, and finally a command that updates the Board, including all score calculation and generation of a new block.

- *BufferToggle*: This class simply enables accepting input without pressing enter. It uses the `termios` package to modify terminal behavior. Credit due to Luc Lieber for writing this utility class.

- `main.cc`: While our Board class manages most aspects of the game, the main program file manages all the input and all options for our program, which gets fairly complex. The first unusual feature is a structure called `RealTimeData`. This is used to pass values into the function that will be running in our display update thread. The function `realTimeUpdate` is the function used by said thread to update the display at regular intervals. To get the display to work nicely with the multithreaded nature of the program, we needed to call the `XInitThread` function, which notifies the X server that we are using multiple threads. The main function then behaves exactly as a simpler implementation would: we create a dictionary trie of commands, and insert all valid commands in it. We create some flag values, and we process the command line arguments to determine which flags should be set. Based on these flags, we then create the appropriate display objects that will display our game. If the real time option is chosen, this is also the point where the display update thread is initialized. If the character mode was chosen, the terminal is now set to accept commands without pressing enter. Now we enter our main game loop, where we receive a command from the terminal, and interpret the command. The loop also has a section of code that allows the repeated commands and autocompletion to work, and it updates the display(s) after each command. The loop exits when it reaches an end-of-file or when the game has been lost. The boolean flag that determines whether the game is running is then set to false so the display update thread will terminate, and all generated objects are deleted.

The following are the questions included in the Quadris project description:

- How could you design your system to make sure that only these seven kinds of blocks can be generated? In particular, that non-standard configurations (where, for example, the four pieces are not adjacent) cannot be produced?

  We solved this problem by having the constructor specifically make blocks of the given types. A character is generated that dictates the type of block we will create, and then the constructor simply generates the appropriate values for the block positions. This prevents any other type of block from being generated.

- 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?

This could be very simply done by adding a timer field to the blocks. The timer would decrease every turn, and a method could be implemented in the Board to check for and delete any blocks whose timer has run out using our already built `eraseCell` method. This is also easy to confine to higher levels, since the method would have access to the currentLevel variable.

- How could you design your program to accommodate the possibility of introducing additional levels into the system with minimal recompilation?

  Our program uses a discrete class for the levels, so creating a new instance of these objects would merely require slight changes to the constructor of the Board, and then unlimited (up to how many integers the platform can hold) levels can be generated.

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

  For the first question, we could simply insert a new command into our dictionary trie, and we could add one more condition to the main game loop. In theory, creating a Command class could be better as it has less coupling, but we found the implementation of such a class to be too complex to be worthwhile. As for renaming commands, this is again possible to achieve with a Command class, but we could also achieve it in our current main file by using an array of "key words" instead of string literals, and we would need to add a remove method to our trie (since we removed from the Assignment 3 version to simplify the class).

The following are the two questions given in the project guidelines:

1. What lessons did this project tech you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

   When a program is well-thought out and it has independent modules, it is very nice to work in a team since different people can work on different modules at the same time with little or no interference. This helps us appreciate object-oriented programming philosophy. However, we did find that whenever we wanted to work on the same files, some problems could arise. This could be eased by using better software for managing our code (we used Google Drive, when we could have used GitHub). We also agree that there needs to be a lot of trust between team members, since everyone needs to independently make sure their code runs perfectly. Even in this program, which is relatively small, it would be hard for either of us to completely read and modify the code that the other wrote, so even larger programs would be a nightmare if we couldn't trust other people to implement their parts properly.

2. What would you have done differently if you had the chance to start over?

   One of our biggest regrets is the extra memory being used up by the Block class. It was initially meant to be used by the program, but we found a better way to store the data we needed later on, making this data redundant. However, it was too deeply coupled with our code by that point, and removing it would have cost us too much time. We also would have preferred to use GitHub, or some other software more oriented to code-sharing and version management, even though Google Drive worked relatively well.