Quadris

Project Design

Stephen Humphries (sjahumph) Mufaddal Jerwalla (mjerwall) Harsh Amin (h4amin)

Introduction:

For our CS 246 final project, we chose to implement Quadris, a game which is very similar to Tetris, but is not played in real-time. The user can take as long as he or she wants to decide on what move to make next. Quadris consists of a board 11 columns wide and 15 rows high (with 3 extra rows to allow for block rotation). The goal of the game is to score the most points possible by filling rows (which will then disappear) before the blocks hit the top of the board. Our implementation is written in C++ and outputs the game board to stdout and, optionally, to an Xwindow display. The game is controlled by text commands on stdin. This document describes our program design and answers a number of questions about our implementation.

Overview:

This section, complemented by the UML diagram which follows it, will provide an overview of the structure of our Quadris implementation.

Our program begins in the main file, which contains all code related to interacting with the user, including parsing command-line arguments and reading in and interpreting commands during runtime. After reading in the command-line arguments, the main function initializes the two displays -- the TextDisplay object and the GraphicsDisplay object -- and the Board object (which is responsible for storing and manipulating the game state). Then the main function enters the command loop, which reads in commands from stdin and passes them along to Board. When a string is read in from input, the command loop compares it against a map of stored command names, and if a match (or partial match) is found, it runs the associated code. In almost all cases, this code calls a function of Board.

Moving on to the Board class: Board is responsible for storing the state of the game, including the score, high score, current level, and all of the blocks in play. Board has an *owns-a* relationship with all of the Block objects (more on them later) that make up the game state -- a vector of smart pointers store all the Blocks that have been dropped, and a single smart pointer called currentBlock stores the Block which the user is currently interacting with. Incidentally, our program manages all memory implicitly through the use of smart pointers, which allows it to run completely leak-free. Board provides several functions which are called by main, and allow the user to modify the game-state: the *left()*, *right()*, *down()*, *rcw()*, and *rccw()* functions change either the current block's position, or its rotation. Each of those functions takes an integer as a parameter, specifying the number of times the move should be repeated (if it is possible to repeat that many times) and makes a call to the private *move()* helper function, which takes what block to call the function on as reference and does the calculations to decide if a move is valid. Thus, it is the Board's responsibility to decide when a move is allowed to be made (e.g. not being allowed to move a block through another or off of the grid). The private *move()*

function does this with the help of member functions of the Block object, which we will see next. After the Block object, we will return briefly to Board.

In order to keep track of its shape, each Block object stores a 2-dimensional vector of boolean values which represent the cells that the block occupies; that is, where characters or colours will be placed by the TextDisplay and GraphicsDisplay. Each Block also has an x-coordinate and a y-coordinate which, together, represent its position on the actual board. The Block class implements *left()*, *right()*, *down()*, *rcw()* (rotate clockwise) and *rccw()* (rotate counterclockwise) functions which Board's corresponding functions make use of. Rather than modifying a Block's own values, each of these functions return a smart pointer to a completely new Block object, in the new position or orientation (i.e. one space left/right/down or one rotation CW/CCW). These new Block objects are created without checking the locations of other Blocks in the game; that check is performed by the Board. Block also provides an *overlap()* function which returns true if the Block occupies any of the same cells as the one other blocked passed as a parameter. This function is how the Board determines if a move is legal with respect to the blocks that are already on the grid.

As well as managing the Block objects, Board also contains a BlockGenerator object. The BlockGenerator class provides a *getNext()* function which returns a new Block object, wrapped in a smart pointer. The BlockGenerator has functions to increase and decrease the level of the game, which affects the probabilities with which certain block types are created. Each call to level up or level down replaces BlockGenerator's current Level object, which is what is ultimately responsible for *creating* the next block. The Level class is an abstract base class which has concrete subclasses Level0, Level1, Level2, Level3, and Level4, each providing getNext() methods which generate the next block according to different probabilities or predefined sequences. Thus, BlockGenerator mediates communication between Board and Level.

All of the functions that control the the BlockGenerator and the levels are called by main through the Board object; so Board also must provide levelUp() and levelDown() functions, as well as functions to indicate that BlockGenerator and, subsequently, Level3/4 should read in blocks from a particular file, or generate them randomly.

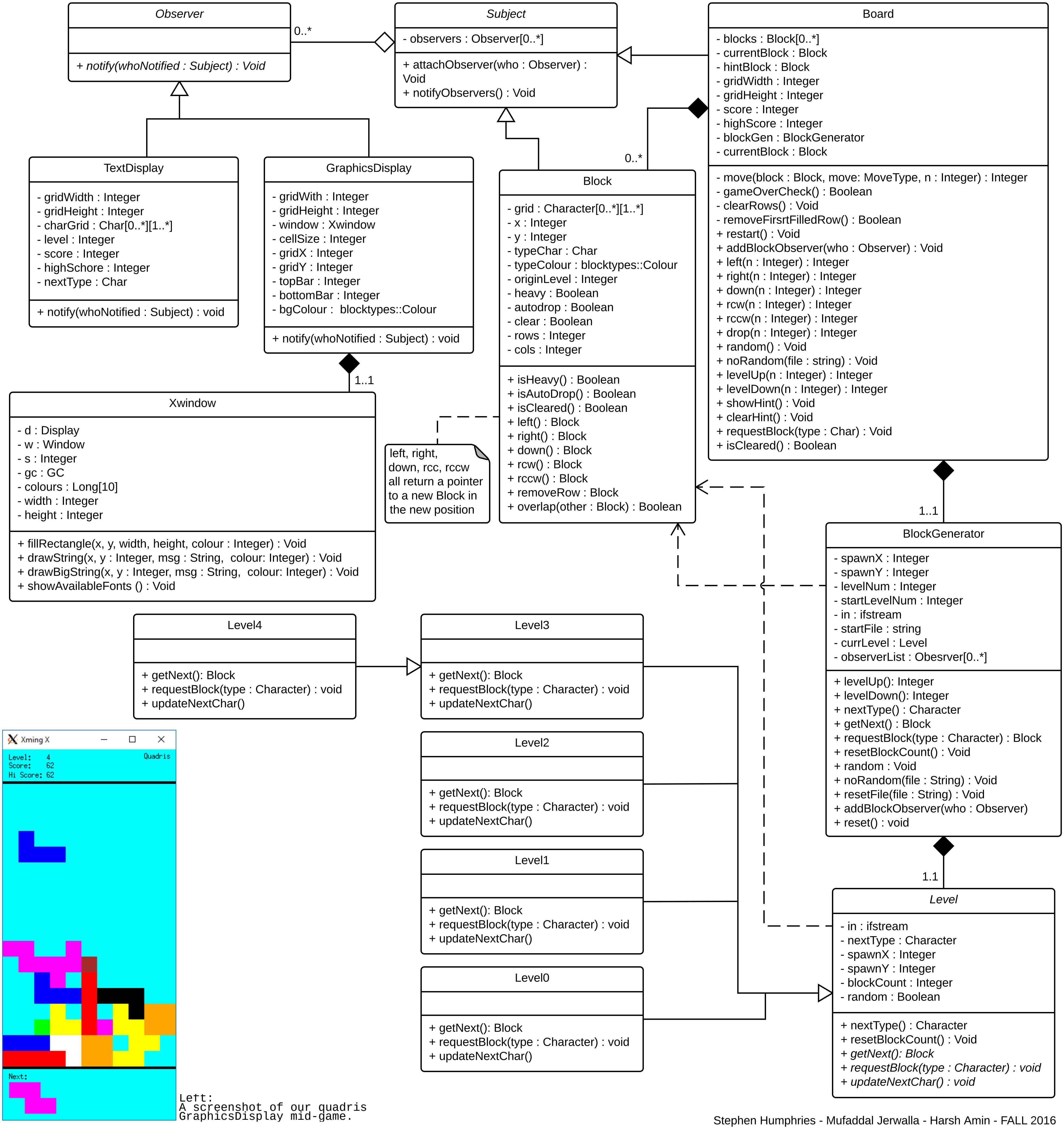
Finally, Board also provides a *showHint()* function which temporarily displays a copy of the current block in what the game considers to be the best position to drop it in. This function will be described in more depth in the design section, below.

The last important part of our Quadris implementation are the two displays which output game state information to the user. The TextDisplay and GraphicsDisplay exist, as mentioned earlier, in the main function. We make use of the Subject-Observer pattern to keep the displays up to date -- more on this in design, below.

UML:

There have been 4 major changes in our UML. They are as follows:

i) The "is-a" relationship between Level3 and Level0 no longer exists. Instead, both Level3 and Level0 inherit directly from Level. The reason for this change is to enable the noRandom



Left:
A screenshot of our quadris GraphicsDisplay mid-game.

Stephen Humphries - Mufaddal Jerwalla - Harsh Amin - FALL 2016 functionality such that only Level3 and Level4 would read contents from the new file without changing the file from which Level0 would read [Level0 will continue to read from the either the default sequence.txt file or FILE_NAME provide via the command line argument “-scriptfile”.

ii) A dependency had been added from Level to Block. Level is dependent on Block since according to its specification it must create and return a smart pointer to a Block, but at no point does the Level *own* a Block.

iii) The relationship between Block and BlockGenerator no longer has an “owns-a” relationship between. Similar to *ii* above, BlockGenerator is dependent on Block because it must return a smart pointer to a newly-created Block but never *owns* it.

iv) A number of member functions were added to Level, BlockGenerator, Block and Board in order to provide a more efficient implementation as well as to deal with intricacies that occurred when implementing Level3 and Level4.

Design:

The most important part of our program design is the way that we handle blocks and their interactions with each other and with the displays. Each Block object contains a 2-dimensional vector of boolean values, representing which cells make up the block’s shape, and two integers, representing the block’s x-position and y-position relative to the bottom-left corner of the board. The Board object contains a vector of smart-pointers to Blocks, which stores the “static” blocks – the ones that have already been dropped and “committed” to the grid. The Board also has a single smart-pointer to a Block, called currentBlock, which is the block that the user may move around, rotate, and drop.

A specific challenge that we faced when considering the movements of the blocks was how to determine whether a move is legal, and how to not allow it if it isn’t, while keeping the displays posted on any changes made. Our design solves this problem by having each Block object represent only a single immutable state of a block. Each time the current block needs to be moved or rotated, we generate an entirely new Block object in the new position, and compare it to all the other blocks on the grid to check if the move to the new position is legal. This is done using a Block member function called overlap which returns true if the Block overlaps with the other Block that is passed to it as a parameter. If the move is legal (i.e. overlap returns false for all the Blocks that have already been dropped), we replace the old Block object with the new one; if it is not a legal move, we discard the new Block. This strategy also makes it easy to update the displays without having to draw or modify more than is necessary: before the old Block is deleted (which happens automatically, thanks to our use of smart pointers), it calls the displays’ notify methods and the displays clear only the cells that the Block occupies. After the new Block object is created (in a different position), it calls the displays’ notify methods and they fill in the cells that it occupies. This avoids having to, say, redraw every block each time a single one changes.

As mentioned above, we used the Subject-Observer pattern to facilitate communication to the TextDisplay and GraphicsDisplay. Our Block class inherits from Subject and our display classes inherit from Observer. Each Block has a vector of pointers to its Observers, which it uses to

Adding new Block types is very straightforward; to initialize a new Block, we read initial grid details for a certain type of block from a map stored in blocktypes.h using a function blocktypes::grid(char), passed the character describing the type of the block. Thus, to add a new block we would just need to add an entry to blocktypes.cc and then modify an existing Level or add a new Level such that the new block type would be generated. No changes to the Block class are necessary, unless the new Block requires special functionality. In that case, Block could be inherited from or modified itself.

Adding new Levels is straightforward thanks to the design outlined in question 2, below.

Adding a new kind of display would not be too difficult: since both GraphicsDisplay and TextDisplay are inherited from the Observer abstract base class, we would just need to create a new Observer subclass and use the addBlockObserver() function in Board to indicate that all new blocks should be created with the new display Observer added to their observer lists.

Answers to Questions

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?

Original Answer

Each block object would include a counter which would be initialized to 10 and would be decremented by one every time a new block is dropped. Once the counter reaches 0, the block would be removed from the array of blocks and the displays would be updated. The generation of such blocks could easily be confined to more advanced levels because each Level provides its own getNext function, which could easily be modified to set a auto-disappear property to true. Any advanced levels added would simply necessitate an adjustment to the block class as described above, adding a new Level subclass, and implementing a function clearBlock() in the Board class which removes a block once its counter reaches 0.

Depending on the level of difficulty, we might also include an adjustBoard() function which, in an easy level, would adjust the board by calling the drop function on each of the blocks until they could not go down any further. It would start from the bottom row and go up the rows. It would then look for any filled rows and clear them. In a medium level, the same would be done except the blocks would only be moved down by one. In a hard level, the block would be removed but the rest of the board would not be adjusted. Note that this last option would be the easiest to implement and is the most consistent with normal Tetris.

Thoughts after completion

To expand on the first paragraph of the above answer, we would just need to add a function to Board that runs through the vector of Blocks that have been dropped and decreases all of their counters, likely using a countDecrease() function addeed to Block. If any of the counters reach 0, the Board would set the Block to clear, notify the displays, and delete the Block. Paragraph 2, notify them when it is drawn or cleared. The Board class is also a Subject; it notifies the displays when the level, score, high score, or next block type change.

Another big challenge was presented by the *hint* command, which requires that a copy of the current block be temporarily displayed in what the game considers to be the best position in which to drop it. This challenge was two-fold: first, we had to figure out how to display a hint only until the next command is entered, and second, we had to figure out how to choose the "best" position and rotation. Board includes a second Block smart pointer, called hintBlock, which stores the Block object representing the hint. This Block is generated when main calls Board's *showHint()* function, then it notifies its observers (i.e. the TextDisplay and GraphicsDisplay). The hintBlock continues to exist until main reads in the next command; then main calls Board's *clearHint()* function, which notifies the displays that the hintBlock should be cleared, and resets the smart pointer (deleting the Block object). To actually figure out where the hintBlock should be positioned, *showHint()* makes use of the same internal *move()* function that is used to move the currentBlock. *showHint()* generates a vector of temporary Block objects by trying to move each of the 4 possible rotations for the block to each possible x-position and then down as far as possible. The "best" of these Blocks is selected on the basis of having the lowest top (so the lowest y-position + height), and any ties in lowest top are broken by choosing the Block that is the farthest to the right. This choice was made because, since blocks spawn at the top-left, it is usually best to fill up the right side of the board first.

Coupling and Cohesion

In our program design, we have strived to achieve a low level of coupling and a high level of cohesion through attempting to stick, as much as possible, to the single-responsibility principle. The Block class is responsible for storing the state of an individual block and is itself quite uncoupled; it does not depend on any of our other classes. The displays (i.e. TextDisplay and GraphicsDisplay) are highly uncoupled because they only depend on Block and Board to get the relevant information for displaying changes, and they are highly cohesive as their sole functionality is to display the information and do not make any changes. The Board class is, admittedly, highly coupled with Block and BlockGenerator, but this is permissible, since its purpose is to store and control the game state. It is highly cohesive, since all of its interface methods are for the purpose of controlling the game state. There is a moderate amount of coupling between Levels, but in this case, it is a good thing, because it allows for a more efficient implementation of the higher levels, which would otherwise include a lot of duplicate code (Level3/4).

Resilience To Change

Our design is highly resilient to change; some examples of this are described in the question answers from Due Date 1 which can be found below.

It is easy to rename commands or to add new commands, simply by adding the command name and a unique enumerator to our command map in main (see details in question 3).

above, describes how the rest of the Board could be adjusted in response to the block being removed.

Question 2

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

Original answer

In order to be able to efficiently add new levels, we have a container class called BlockGenerator [it contains an object of class Level] which generates new blocks depending on the current level of the game and levels up or down depending on the input command. To level up or down, BlockGenerator has levelUp() and levelDown() functions which, when called, both create new Level objects according to the new difficulty and replace the existing Level object. Separating the Level object form the Board with BlockGenerator decouples all functionalities to do with block generation and switching levels from Board.cc. When creating a new level, we will create a subclass to Level (which is an abstract class) and include the relevant implementations for the subclass. On compilation, BlockGenerator.cc will be re-compiled in order to inform itself of the new level and the new Level subclass will be compiled as well. This will minimize recompilation when adding new levels to the game, since Board will not need to be recompiled.

Thoughts after completion

This is exactly how we implemented the Levels in our Quadris program. It worked very well and made it quite simple to add each of the new Levels.

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.

Original answer

We have chosen to design our system in a way such that, with minimal re-compilation, names of commands can easily be changed and new commands can easily be added. To support this functionality, we will create an enumeration with unique enumerators representing each function that can actually be called on a board (e.g. Board.left(), Board.right(), Board.down(), etc). We will also create a map which links the command names (strings) that the user can enter to the enumerator for the associated function. When a command name is read in, a switch statement on the enum (which will have been acquired from the map) will handle actually calling the function, and, optionally, reading in additional parameters such as a file name. Adding or renaming a command will only necessitate recompilation of the file containing the main function and the file containing the command-enum map.

With this design, it would be easy to support a command to rename or duplicate an existing command as it would need only to modify the key string in the command-enum map.

Supporting a macro language would also not be too difficult using a variation of this system. One simple way to support macros would be to store the string representing the command sequence, and then interpret it only when the macro is called. This approach raises a few issues, though; if one of the commands included in the macro is renamed, then the macro would need to be redefined. We could get around this by translating the command sequence into its associated enumerators and storing that, instead of storing the original command names. To store macros including commands which require additional input, like filenames, we could do the same, but also store the additional input separately in a string or sstream. For example, a macro for the command sequence “`left down left sequence seq.txt norandom nor.txt right`” could be stored as a list of enumerators `{LEFT, DOWN, LEFT, SEQUENCE, NORANDOM, RIGHT}` and as a string or sstream “`seq.txt nor.txt`” which would be read from as appropriate.

Thoughts after completion

This is pretty much exactly how our Quadris program is implemented, with the single difference being that the command-enum map mentioned above is not in its own file, but rather is a part of main.cc. This choice was made in order to simplify the interactions between main and the command loop, since the command loop needs to be able to access the Board object that is owned by main. As an extra feature, we added a command to rename an existing command, in the way described above. Unfortunately, we did not have time to implement the other command interpreter extensions, but the methods described above would still be valid.

Extra Credit Features

The most important extra credit feature that we implemented was not explicitly managing any of our memory. As suggested, we exclusively used smart pointers (specifically, instances of the *std::shared_ptr* class) to hold dynamically allocated memory. We chose to do this right from the start, so that we would not have to switch from raw pointers once the implementation was finished. Examples of objects managed by shared_ptr objects are the Blocks owned by Board, the Level owned by BlockGenerator, and the GraphicsDisplay in the main. GraphicsDisplay is in a smart pointer rather than being on the stack like TextDisplay because we needed to be able to selectively disable it with the “-text” flag. The most difficult part of using smart pointers was just adapting to the syntax of declaring them; other than that, they greatly simplified things, and made debugging much less of a struggle. Our program also runs completely leak-free, with next to no additional effort exerted.

Another extra feature we implemented was the ability to specify the seed based on the time; instead of using command-line argument “-seed ####”, the user can pass the argument “-seed time” and then the seed will be based on the current time. This makes it much less of a hassle to not have the same default block order each run.

A third extra feature we implemented was adding a “rename c1 c2” command to the command interpreter. As described in the question answer above, this just re-maps the given command c1 to call command c2, instead. This was very easy to implement, due to the fact that our program was designed specifically to accommodate such changes.

Another feature we implemented was offering the user a choice of whether to play again or to quit after losing the game. When the game over condition is met (i.e. when the next block cannot spawn without overlapping an already-existing block), Board raises an exception, which is caught by main. Main asks for input whether to play again or not: if yes, it calls Board.restart() and continues reading command input, and if no, quits, deleting all dynamic memory as the stackframes are popped.

The final extra feature we implemented is reading and writing the high score to/from a file. When the Board constructor runs, it reads highScore from “.quadrisHighScore”, and when highScore is updated, it writes out to the same file.

Final Questions

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

This project taught us a few things about developing software in teams. First of all, we all greatly furthered our knowledge of using git as a central version control system; two of us had never used it before and one of us was only somewhat familiar. This will be a valuable skill in any career in a software-related field. An important lesson that we learned that was associated with collaborating on code was the importance of communication and making sure that everybody is kept on the same page when changes are made to the UML layout, method functionality, or simply function names. We had a few cases where two of us were using different names for getter and setter functions, and the problems resulting from that could have been avoided had we exercised better communication.

Another thing we learned about developing software teams was that it is quite valuable to have a second or third pair of eyes when writing or debugging code. Many sloppy mistakes when we had one person debugging near the end of the project were caught by an observer.

It was also quite valuable to be able to exercise knowledge transfer between the individuals with a better understanding of how the program was supposed to work or with a better understanding of the C++ language and the individuals with gaps in their knowledge.

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

If we had the chance to start over, we would have started earlier so that we would have had more time to implement extra features. As it stands, we were successful in implementing the entire core functionality, but there are a number of extra features that we would have liked to include but did not get a chance to.

One change that we would make to the structure of our program if we had a chance to do it again would be to implement the BlockGenerator/Level inheritance structure as a visitor pattern in order to reduce the coupling between those classes. Using a visitor pattern would have made it easier to add new Levels with greater feature sets without needing to modify the BlockGenerator class that we used as a mediator between Level and Board.

(EOF)