University of Waterloo

CS 246 Design Document

Assignment 5 (Quadris)

Harris Rasheed & Amlesh Jayakumar

Fall 2011

Table of Contents

CS 246 Assignment 5 Design Document

Introduction

This document outlines the thoughts on design and architecture of the Quadris program put forth by both team members of cs246_037; Harris Rasheed (h2rashee) and Amlesh Jayakumar (a3jayaku) for CS 246 (Object-Oriented Software Development) Assignment 5 Group Project.

Project – Plan vs. Actual

There were several changes between the initial designs of the project versus the project after completion. The team's statement of intent was indeed fulfilled as all the basic features outlined in the assignment specification were successfully implemented. However, when it came to extra features for bonus marks, both members felt overambitious on some aspects.

The extra features that were not implemented were the:

- music feature; set to play the classic Tetris music on loop for the duration of the game
- hint feature; will suggest the next best possible move that the user can make in the game.

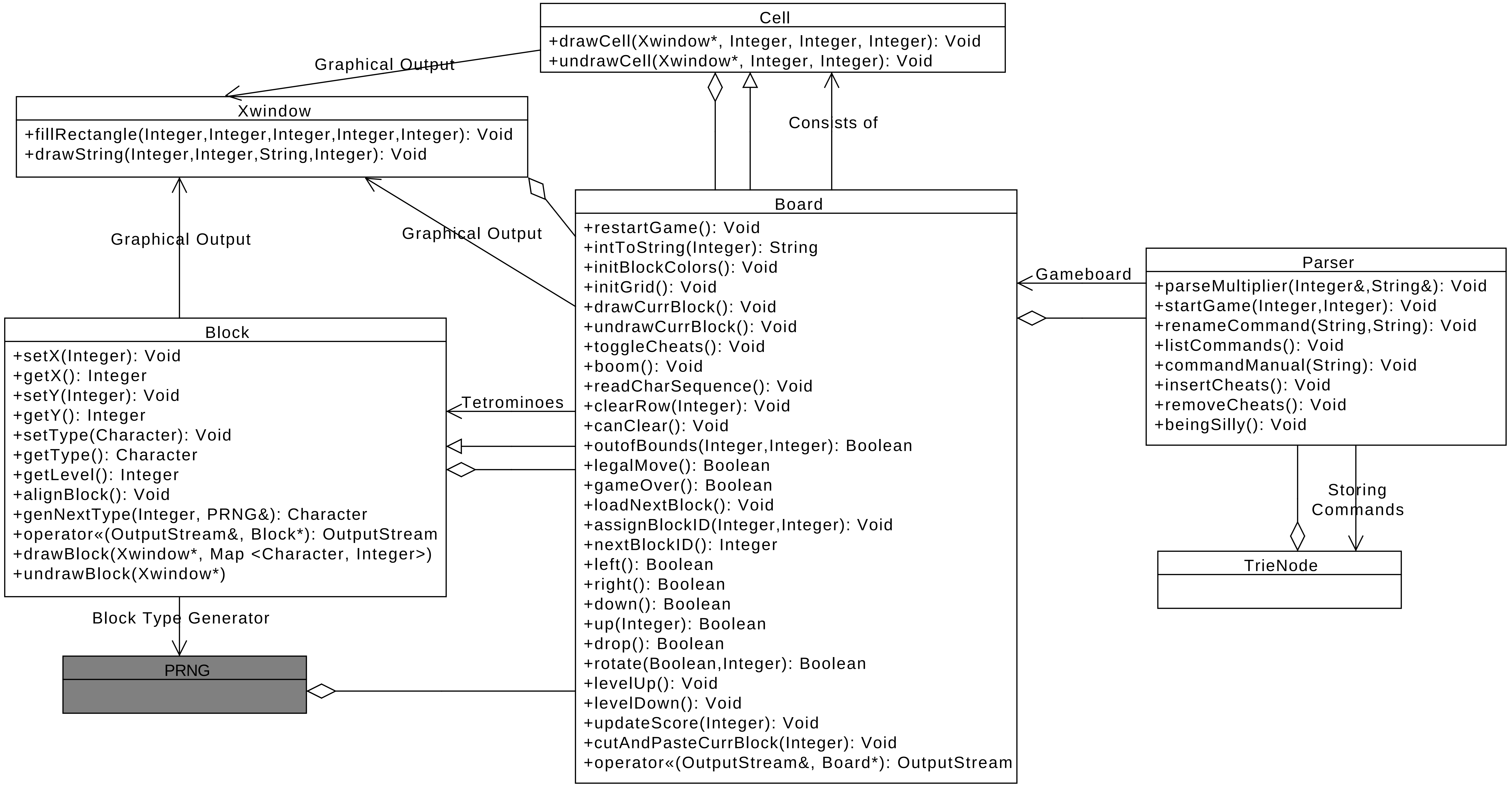
Features that were implemented but not completely include:

- renaming feature; renaming commands only allows strings comprised of lowercase alphabets.

A feature that was implemented but not included in the designs was:

- for cheat mode, a command to allow the user to move the current block up which could potentially allow the user to select an alternate area to drop the tetromino on
- a shadow of the current tetromino which is being moved around is generated on the bottom of the board so that the user knows where the block will fall when they drop the block.

The UML diagram outlining the system's architecture can be found on the next page.


Cell
+drawCell(Xwindow*, Integer, Integer, Integer): Void
+undrawCell(Xwindow*, Integer, Integer): Void
Graphical Output
Xwindow
+fillRectangle(Integer,Integer,Integer,Integer,Integer): Void
+drawString(Integer,Integer,String,Integer): Void
Consists of
Graphical Output
Graphical Output
Board
+restartGame(): Void
+intToString(Integer): String
+initBlockColors(): Void
+initGrid(): Void
+drawCurrBlock(): Void
+undrawCurrBlock(): Void
+toggleCheats(): Void
+boom(): Void
+readCharSequence(): Void
+clearRow(Integer): Void
+canClear(): Void
+outofBounds(Integer,Integer): Boolean
+legalMove(): Boolean
+gameOver(): Boolean
+loadNextBlock(): Void
+assignBlockID(Integer,Integer): Void
+nextBlockID(): Integer
+left(): Boolean
+right(): Boolean
+down(): Boolean
+up(Integer): Boolean
+drop(): Boolean
+rotate(Boolean,Integer): Boolean
+levelUp(): Void
+levelDown(): Void
+updateScore(Integer): Void
+cutAndPasteCurrBlock(Integer): Void
+operator«(OutputStream&, Board*): OutputStream
Gameboard
Parser
+parseMultiplier(Integer&,String&): Void
+startGame(Integer,Integer): Void
+renameCommand(String,String): Void
+listCommands(): Void
+commandManual(String): Void
+insertCheats(): Void
+removeCheats(): Void
+beingSilly(): Void
Storing
Commands
TrieNode
Block
+setX(Integer): Void
+getX(): Integer
+setY(Integer): Void
+getY(): Integer
+setType(Character): Void
+getType(): Character
+getLevel(): Integer
+alignBlock(): Void
+genNextType(Integer, PRNG&): Character
+operator«(OutputStream&, Block*): OutputStream
+drawBlock(Xwindow*, Map <Character, Integer>)
+undrawBlock(Xwindow*)
Tetrominoes
Block Type Generator
PRNG

The specific differences in the UML in the previous two pages are in:

- Parser
  - Additions
    - +renameCommand(String,String): Void
    - +listCommands(): Void
    - +commandManual(String): Void
    - +insertCheats(): Void
    - +removeCheats(): Void
    - +beingSilly(): Void
- Board
  - Additions
    - +toggleCheats(): Void
    - +boom(): Void
    - +up(Integer): Boolean
    - +drawShadowBlock(): Void
    - +unDrawShadowBlock(Boolean): Void
  - Amendments
    - +left(Integer): Boolean
    - +right(Integer): Boolean
    - +down(Integer): Boolean
    - +clearRow(Integer,Integer): Void

The majority of changes come from implementation of bonus features which was not included in the initial designs because the team was unsure of the amount of bonus features that would be implemented in the final program.

The remainder of changes were made to the block movement functions (left, right, up, down) in order to facilitate minimal redrawing of the blocks by recognising the iteration that the block is currently moving on. It undraws the block from the graphical window on the first iteration.

Assignment Specification Approach

We will outline the approach that we took to the assignment specification in the following four areas.

- Block
- Board
- Command Interpreter
- Bonus

Block

The blocks in the game consist of all the one-sided tetrominoes (i.e. tetrominoes that can be translated and rotated but not reflected). Each tetromino is represented by a 4-by-4 grid of Boolean values that stores its shape. For example, the S tetromino would be depicted as:

F	F	F	F
F	F	F	F
F	T	T	F
T	T	F	F

Rotation

To facilitate rotation, the following bijective functions between 4-by-4 grids, that rotate a given 4-by-4 grid clockwise and counter-clockwise respectively, were used:

$$f_{CW}\big((i,j)\big) = (j, 3-i) \qquad \text{(rotates the tetromino clockwise)}$$

$$f_{CCW}\big((i,j)\big) = (3-j, i) \qquad \text{(rotates the tetromino counter-clockwise)}$$

The above functions come from the fact that, when rotating a grid clockwise, the j^{th} column gets transformed to the j^{th} row, and the i^{th} row gets transformed to the $(3 - i)^{th}$ column (f_{CCW} is then achieved by merely taking the inverse of f_{CW}).

The generation of the next block is based on the current level of the game. By using the PRNG (Pseudo-Random Number Generator), we generate each block type using the given probabilities for each (non-zero) level. For example, in level 1, the probabilities we were given for generating each block were:

Block Type	Probability
S	$^{1}/_{12}$
T	$^{1}/_{12}$
L	$^{1}/_{6}$
J	$^{1}/_{6}$
I	$^{1}/_{6}$
O	$^{1}/_{6}$
Z	$^{1}/_{6}$

The PRNG is then used to randomly generate a number x in the range [0, 11], which is in turn used to determine the next block's type (since each number in [0, 11] is chosen with a probability of 1/12).

Board

Representation: Cells

The game board is represented by a 18-by-10 grid of Cells (where each cell stores the required information of one particular 1-by-1 square of the grid). This was done because each cell of the grid, if non-empty, needs to store information about the tetromino it belonged to (for example the level it was generated in, its type etc.).

If the cell contains part of some tetromino T, then it stores:

- T's block ID
- the level T was generated in
- the character representing T's block type (e.g. 'S', 'Z' etc.)

The 'block ID' of a cell is merely a unique number in [0, MAXBLOCKS-1] (where 'MAXBLOCKS' is a rough upper bound on the maximum possible number of different blocks on the board at once, i.e. 180). This member is used to determine when a block has been completely erased from the board after a number of rows have been cleared. After a tetromino has been dropped and pasted permanently onto the board, we assign the cells, where it is placed, with an unused block ID (i.e. a number in [1, MAXBLOCKS] that isn't currently the block ID of any other cell on the board). This facilitates the checking of whether a block has been completely removed after rows are cleared, as we merely look for assigned block IDs that no longer exist on the board.

Take Figure 1a for an example. It shows the (block ID, type, level) assigned to all the dropped blocks on the board respectively (where 'level' is the level that the particular block was generated in). Figure 1b then depicts the board after the current block has been dropped, and the filled rows have been removed.

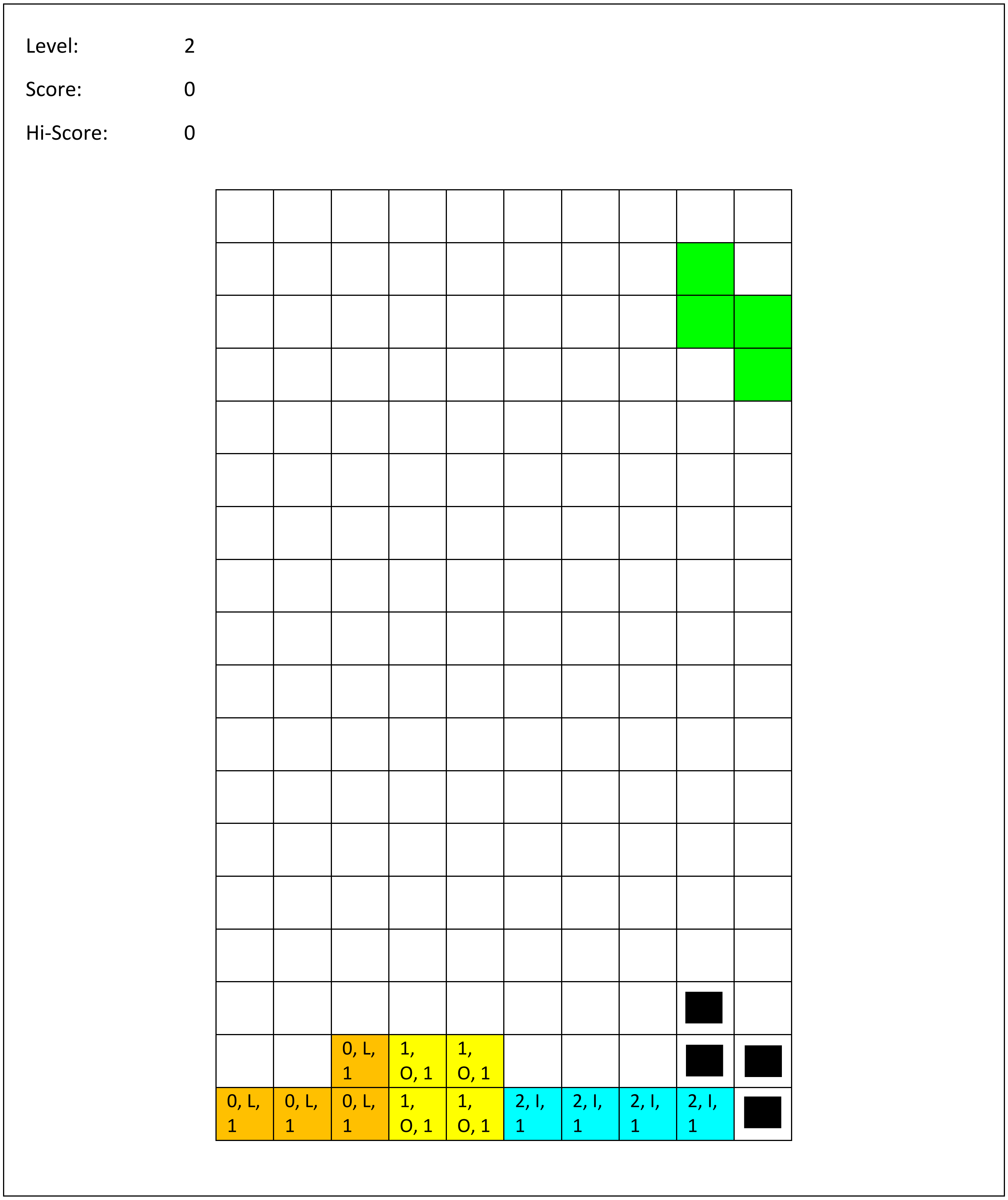
At this point, the used block IDs before and after the clearing of the board is compared to check if any block has been completely erased. In this case, the block with ID # 2 (i.e. the I tetromino) was erased, and thus the new score takes the level, in which block of ID #2 was generated in, into account during scoring.

$$Score = (Level\ 1 + 1)^2 + (Level\ 2 + 1)^2$$

Scoring for clearing the I block from Level 1 + Scoring for clearing the row from Level 2

$$Score = 4 + 9 = 13$$

The dropped block then gets assigned to the block ID #2, as that is the next possible ID that is not being used. The character representing the cell's block type is used during output (the letter for text mode, and the corresponding class Tetris colour in graphics mode).

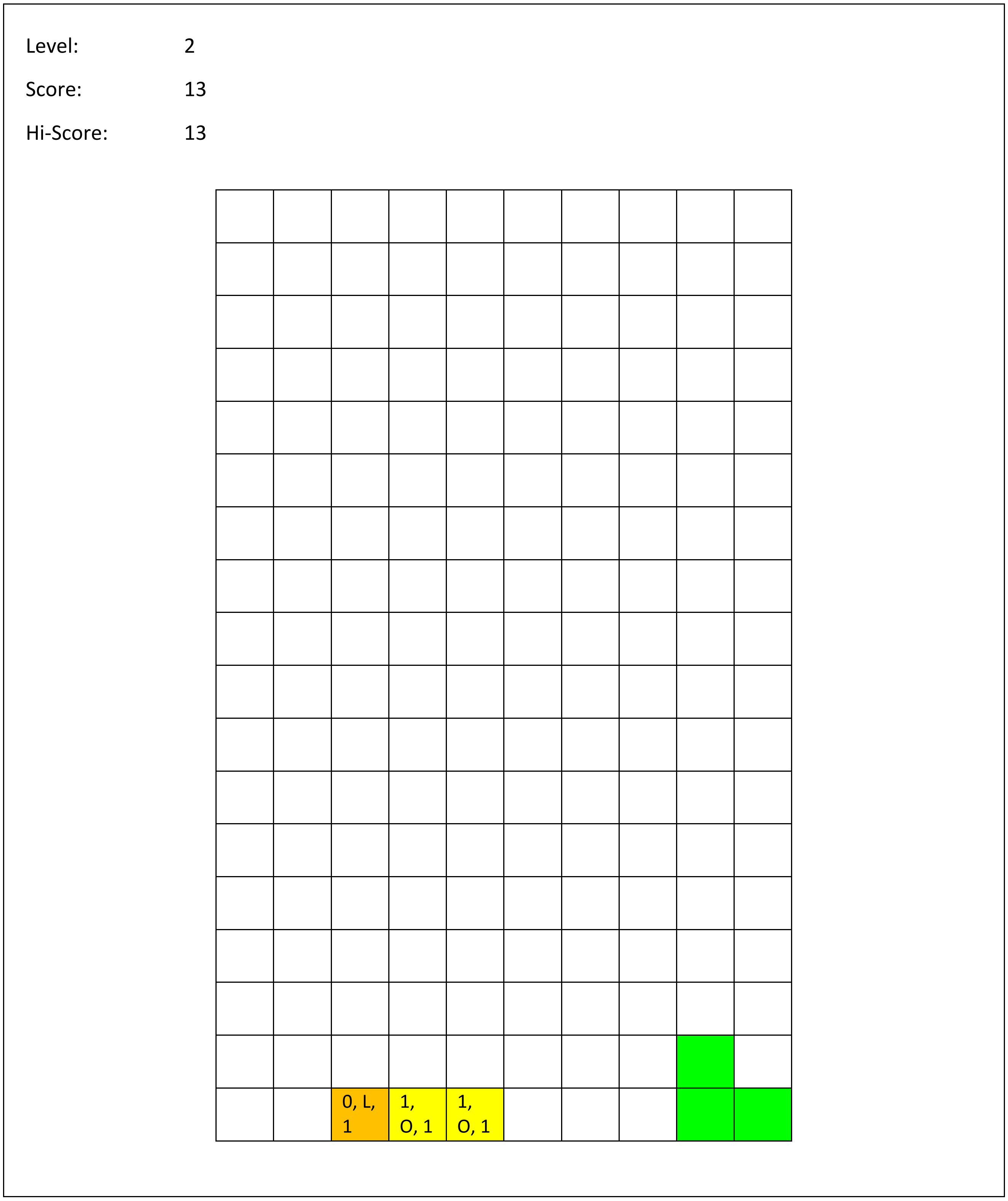


Level: 2

Score: 0

Hi-Score: 0

Figure 1a



Level: 2

Score: 13

Hi-Score: 13

Figure 1b

Current Block

The current block, i.e. the block being manipulated by the user, isn't actually part of the board, but instead merely placed on the board when it is output (one could think of the current block as a sheet of tracing paper hovering over the game board that is temporarily drawn onto the board for the output process). This makes it easier to detect collisions between our current block and the existing blocks on the board as well as to check whether our current block is out of bounds. Figures 2a and 2b illustrate this 'tracing paper' idea of storing our current block.

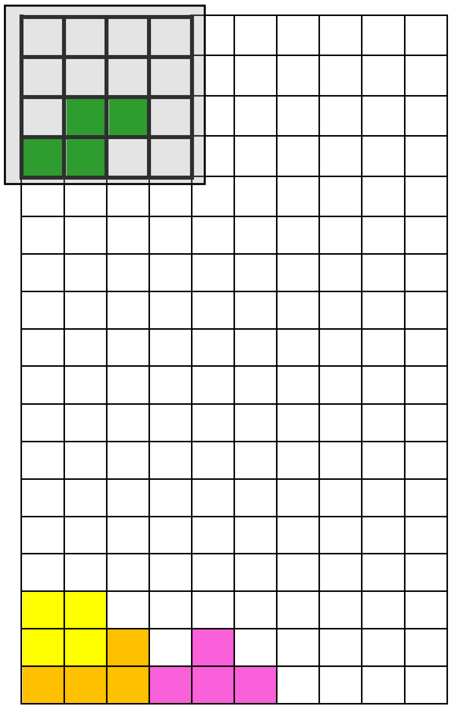


Figure 2a: Current block stored on the tracing paper

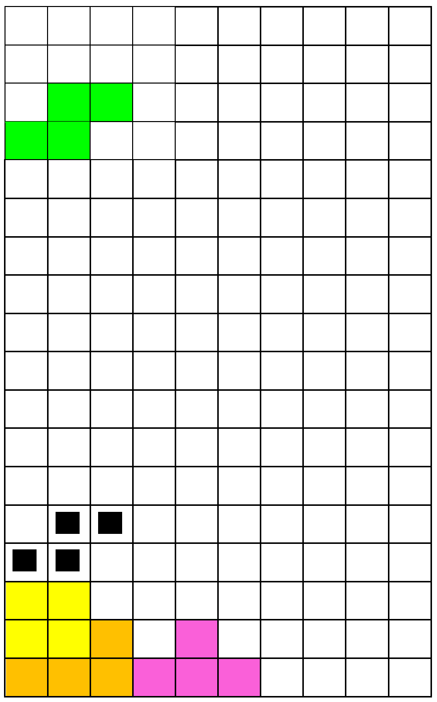


Figure 3b: Current block pasted onto the board

Graphics

To make the game board more visually pleasing, an encompassing grid was drawn (to define the boundaries of the board). These boundaries are drawn as thin rectangles, with a spacing of 25 units (thus each cell of the grid is a square of length 25), where the bottommost and the leftmost boundary lines are drawn 125 units from the edge of the window. This yields a simple coordinate transformation that helps draw the current block onto the board:

$$\theta_{curr}((x, y)) = (25y + 125 + 1, 25x + 125 + 1)$$

This transforms the top left coordinates of a cell from the board onto the window. The 25 * x, and 25 * y, accounts for the fact that the point (x, y) gets mapped to a point on the window that already has x squares of length 25 above and y squares of length 25 to the left of the point (x, y). The addition of 125 is to account for the position of the initial gridlines (which are 125 units from the edge of the window). When drawing/undrawing the current block, to guarantee that none of the drawn gridlines are disturbed, we draw each cell of the current block as squares with length 23 so that it fits perfectly inside the cells of length 25 on the board (this is what the '+ 1' in both the coordinates account for).

A similar transformation was constructed to display the next block in the smaller side window:

$$\theta_{next}\big((x,y)\big) = (40y + 70 + 1, 40x + 70 + 1)$$

The dimensions were chosen so that the window that displays the next block is visually pleasing.

Minimal Redrawing

The board is redrawn only when a change has occurred (i.e. only when the state of the board/current block has been altered by the last move). There are a few cases where the board or current block could be unnecessarily drawn/undrawn:

- when clearing a row, or restarting the game (i.e. when the move requires rows to be undrawn and possibly redrawn)
- when having entered an illegal command (e.g. trying to move a block out of bounds or into an existing block on the board)
- when a redundantly large multiplier has been appended to the beginning of a command (e.g. 1000left when the block can only move 2 cells left)

To avoid undrawing cells unnecessarily, the vector 'usedCells' is created to keep track of the cells on the board that aren't empty (i.e. that contain a portion of some previously dropped block). Thus, when a row is cleared, instead of undrawing and redrawing all the cells on the board, only the used cells on or above the cleared row is undrawn and redrawn. This vector also makes it easier to manipulate the used cells as a whole by implementing methods that universally affect all the used cells (e.g. drawing/undrawing of the board, clearing a row, restarting a game, the 'boom' or 'clear' cheat etc.). This isolation of the used cells from the board also helps facilitate the debugging phase, as it induces a sort of modularity in the program (so if there is a bug, it is easier to pinpoint its location). For example, upon running 'restart', if the board isn't completely cleared, then the error is associated with the clearing of the used cells as opposed to the actual board itself, making it simpler to locate the bug and fix it.

In the case where an illegal move is made, or when a large multiplier is added to the beginning of a command, the current block could potentially be undrawn and redrawn unnecessarily. To ensure that the board is redrawn minimally, in graphics mode, we keep track of whether the previous move altered the current state of the game board. Thus, the board is redrawn when absolutely necessary, and this also accounts for redundantly long multipliers appended to the beginning of commands (e.g. when the user attempts to move the current block 1000 times to the right when it can only legally move three spaces [1000right]. The program performs the first three movements and then terminates when the right command becomes illegal).

Level 0

When in level 0, the block types are input from sequence.txt and stored in an array of characters. These characters are then subsequently loaded as the game proceeds. In the event that a non-standard block type is input from sequence.txt, we mark this type in our array as '*' to indicate that it is non-standard. Then, when loading the next block from our array, if the type is a '*' (i.e. if it was a non-standard block type), an error message is output, and the character is then skipped (this is to let the user know that a non-standard block type was entered). When the block types from sequence.txt have been exhausted, the game is restarted.

In the event that the level was changed when the current block is the last block in sequence.txt, the next block would be empty (as being on the last block in sequence.txt, there wouldn't be a next block). To avoid this we load a new next block based on the level we moved to.

Command Interpreter

The command interpreter is based in the 'parser' module of the project (i.e. parser.h, parser.cc). The command interpreter reads the relevant commands from the standard input stream and provides the relevant functionality by executing the appropriate functions.

Auto-completion

The functionality to accept commands in full form and any unique prefix was dealt with by using a trie. Commands were fed into the trie on start-up and if a command prefix was passed into the program, the command interpreter searched the trie for the possible commands.

- If nothing is found, the user is informed that the command is invalid and does not exist
- If a unique command is found, the appropriate function is called and executed
- If multiple corresponding commands are found, the user is informed that their input is ambiguous and that they must input a unique substring.

The functionality for standard commands is in the 'board' module, as they potentially modify the game board in some way as opposed to some of the bonus features implemented such as 'help' and 'man'. These bonus features are implemented within the 'parser' module.

Multipliers

The command interpreter also handles multipliers. The assignment specification outlined that commands should be able to execute the specified amount of times if a number was attached as a prefix to the command. For this option, the program reads in each command and parses the prefix of the string that contains the number, if present. This number is used in a loop to perform the specified procedure multiple times. The rest of the command is recognised by the command interpreter if it is valid. Certain commands are unaffected by multipliers such as restart, rename, help and man.

Cheats

Cheat commands only exist in the trie if cheat mode is activated. During activation, the index for the array storing all active commands is extended to include the cheat commands. When cheat mode is deactivated, the cheat commands are removed from the trie to prevent recognition and functionality of the commands and the index of the active commands array is reduced so that the cheat commands are considered 'out-of-bounds'.

Bonus

There are several bonus features that need to be addressed.

Renaming Commands

The renaming command feature uses various structures to aid in its functionality.

- All command names are stored in an array an each command is associated with its index.
- Commands available for use during the game are stored in a trie which allows the auto-completion feature to work.

When a command is being renamed, the old command name is replaced in its array location by the new command name, the old command name is removed from the trie and the new command name is inserted into the trie. The renaming feature, however, prevents the user from renaming a command to one that already exists or one that is a prefix of a command that exists so there are no collisions when a command is input. The use of the trie structure has greatly aided in the ease of implementing this feature.

Since the trie is limited to storing lowercase alphabets, the renaming feature can only change command names to strings consisting solely of lowercase alphabets.

Tetromino Shadow

The tetromino shadow feature aids the user visually by displaying the 'shadow' of the current block (i.e. the spot the current block will end up at if dropped). The shadow is drawn, and undrawn, whenever the current block is drawn, or undrawn. Thus, the shadow also follows the minimal drawing specification.

A problem that arose when redrawing the shadow was when the shadow partially overlapped with the current block. In this case, when the shadow is undrawn and redrawn, part of the current block is affected as well. To ensure that this doesn't happen, the current block is redrawn whenever the shadow is undrawn (to potentially fill up any missing parts).

To graphically simulate a shadow, the cells that make up the shadow were drawn in black. Also, the cells were drawn as squares with length 15, to contrast the squares of length 23 that represent the tetrominoes.

Cheat Commands

For bonus, a cheat mode was created to turn on/off the availability of the following three commands.

Up

The up command allows the user to shift the current block one cell upwards. This is the only move that allows the user to renter the 3 rows that are reserved for the current block upon its generation. This command is very similar to 'down' in its functionality.

Clear

The clear command allows the user to clear a specific row on the board. However, the user won't get any points for their score for a row cleared in this manner. To make sure that this happens, the method that deals with clearing a row, 'clearRow()', takes in a variable that specifies whether the command was called via the cheat 'clear' or by another method. (Note: Multipliers have no effect on this command)

Boom

The boom command clears the entire board. The implementation of this command illustrates one of the advantages of storing all used cells in a vector. This way, we merely have to undraw all the cells in the used cells vector as opposed to unnecessarily undrawing all the cells on the board. (Note: Multipliers have no effect on this command)

Help Command

The help command aids and guides the user when in doubt of the available commands or their functionality. The two commands 'help' and 'man' both aid the user by providing hard-coded documentation from the 'parser' module of the code. The decision to hard-code the documentation into the code was made in order to remove dependencies on outside files. The 'help' command displays a listing of available commands. This varies if cheat mode is activated or not. The 'man' command describes the functionality of a given command.

Note: The Plan of Attack mistakenly outlined a feature for minimally redrawing blocks in graphics mode and a feature informing the user if an ambiguous prefix is input for bonus marks. However, they were specified in the assignment and are required components so they are no longer addressed in the bonus section or stated as a change in the design. This was because it was planned for and implemented although for different purposes.

Assignment Design Specification Question

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

The system incorporates the seven standard block configurations built into block.cc. Noting, also, that these seven tetromino configurations include all possible combinations where the four pieces are adjacent to each other. This ensures that the program only allows these seven block configurations to be selected.

The seven blocks are selected depending on the current level with the given probabilities. If the program is in level zero, it reads in the characters appropriately and generates the corresponding block. If a character that does not correspond to any of the seven standard blocks is included in sequence.txt and is encountered by level zero, the letter is skipped and an error message is output to inform the user that an incorrect block type was specified.

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

In order to add further levels to our system, another conditional clause would have to be included in block.cc's genNextType method with the appropriate level number. The appropriate specific functionality or relevant probabilities would have to be set-up in the clause to select blocks accordingly for the given level. Block.cc depends on PRNG.h and window.h, which would only differ for a major graphics module change or if the random number generator class needed to be processed differently. As a result, only block.cc, and main.cc would have to be recompiled if an additional level is to be added to the system.

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)?

We stored all the standard commands in a global dynamic array. Thus to add a new command, all one would have to do, apart from adding the command's functionality, is to include the command name in the array and a conditional clause to recognise if the added command is input by the user. In addition, the array size would have to be increased by one.

Our implementation includes the ability to rename commands. The program makes use of a trie that inserts all of the standard commands on start-up. For a command to be renamed, the old command simply needs to be removed from the trie and the new command inserted in its place. The program would rename the command in the global dynamic array to its new name and operate with the new command name doing the old functionality. During the renaming process, the program checks whether the user is attempting to rename a command to an existing command name. The program also prevents the user from renaming commands to a string with numbers, symbols and alphabets in uppercase. This is because the trie has been built with the functionality to support only lowercase alphabets.

Lessons Learnt

As a group, both members related their feedback while working on this assignment and we found many benefits and challenges to team software development.

Issues

There was a difference of opinion when setting a code documentation standard. Both members had different styles in commenting code and setting the one method that we would both use was often disputed several times during the development phase. Good coding and documenting practices should have been set and been clarified earlier in the project.

Benefits

Both partners found the project far easier when there were two people to manage the workload and it was discovered that both partners had their strengths in the various areas which made the project's code, functionality and documentation stronger. This shows that projects that require teamwork can be far stronger than one built by an individual if co-ordinated well.

The fact that both members of the project were roommates helped with its speedy completion. This also allowed us to keep strong communication channels and meet all of the deadlines put forth within the Plan of Attack document surprisingly.

Co-ordinating the various code modules between each other allowed both members to recognise the documentation required in their modules by the level of understanding of their partner. This better understanding makes both members better programmers as a result.

Conclusion

Overall, both team members felt that the project was a fantastic experience and gave a small taste of what team software development in a professional working environment is like. This project has helped further the understanding of good software engineering practices and the importance of time management for both team members. The team hopes to be able to indulge in a similar endeavour in the near future.