# CS246—Assignment 5, Group Project (Fall 2016)

J. Avery C. Kierstead B. Lushman

Due Date 1: Friday, November 25, 4:55pm
Due Date 2: Monday, December 5, 4:55pm

**This project is intended to be doable by two people in two weeks. Because the breadth of students' ablilities in this course is quite wide, exactly what constitutes two weeks' worth of work for two students is difficult to nail down. Some groups will finish quickly; others won't finish at all. We will attempt to grade this assignment in a way that addresses both ends of the spectrum. You should be able to pass the assignment with only a modest portion of your program working. Then, if you want a higher mark, it will take more than a proportionally higher effort to achieve, the higher you go. A perfect score will require a complete implementation. If you finish the entire program early, you can add extra features for a few extra marks.**

**Above all, MAKE SURE YOUR SUBMITTED PROGRAM RUNS. The markers do not have time to examine source code and give partial correctness marks for non-working programs. So, no matter what, even if your program doesn't work properly, make sure it at least does something.**

**Note:** If you have not already done so for Assignment 4, make sure you are able to use graphical applications from your Unix session. If you are using Linux you should be fine (if making an ssh connection to a campus machine, be sure to pass the `-Y` option). If you are using Windows and putty, you should download and run an X server such as XMing, and be sure that putty is configured to forward X connections. Alert course staff immediately if you are unable to set up your X connection (e.g. if you can't run `xeyes`).

Also (if working on your own machine) make sure you have the necessary libraries to compile graphics. Try executing the following:

```
g++ -L/usr/X11R6/lib window.cc graphicsdemo.cc -lX11 -o graphicsdemo
```

## The Game of Quadris

In this project, your group will work together to produce the video game Quadris, which is a Latinization of the game Tetris.

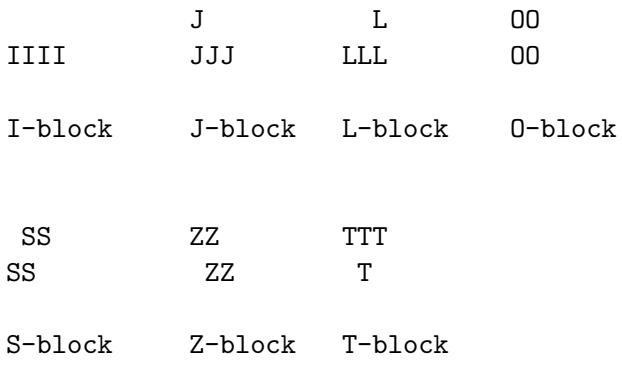
A game of Quadris consists of a board, 11 columns wide and 15 rows high. Blocks consisting of four cells (tetrominoes) appear at the top of the screen, and you must drop them onto the board so as not to leave any gaps. Once an entire row has been filled, it disappears, and the blocks above move down by one unit.

Quadris differs from Tetris in one significant way: it is not real-time. You have as much time as you want to decide where to place a block.

The major components of the system are as follows:

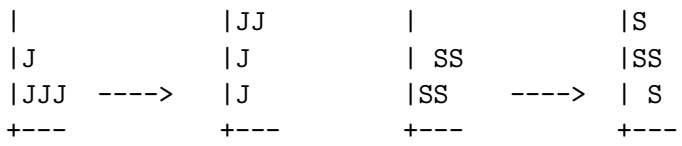
## Blocks

There are seven types of blocks, shown below with their names and initial configurations:




**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?

Blocks can be moved and rotated. When a block is rotated, it should be done such that the position of the lower left corner of the smallest rectangle containing the block is preserved. For a clockwise rotation, this means that the lower-right corner of the block should take the place of the lower-left corner of the original block. For a counterclockwise rotation, this means that the top-left corner of the block should take the place of the lower-left corner of the original block. A few examples follow (clockwise rotation):




The coordinate axes shown above should be understood as being fixed in space, and are there to show the position of the rotated block relative to that of the original.

## Board

The board should be 11 columns and 15 rows. Reserve three extra rows (total 18) at the top of the board to give room for blocks to rotate, without falling off the board. If a block is at the extreme right-hand side of the board, to the extent that there is no horizontal room to rotate it, it can't be rotated.

When a block shows up on the board, it appears in the top-left corner, just below the three reserve rows. If there is not room for the block in this position, the game is over.

When a block is dropped onto the board, check to see whether any rows have been completely filled as a result of the block having dropped. If so, remove those rows from the board, and the remaining blocks above these rows move down to fill the gap.

## Display

You need to provide both a text-based display and a graphical display of your game board. A sample text display follows:

```
Level:      1
Score:      0
Hi Score: 100
-----------


TTT
 T









         S
        ZSS
       ZZIS
       ZJI
     OO JI
IIII OOJJI
-----------
Next:
  L
LLL
```

Your graphical display should be set up in a similar way, showing the current board, the current block, the next block to come, and the scoreboard in a single window. The block types should be colour-coded, each type of block being rendered in a different colour. Do your best to make it visually pleasing.

## Next Block

Part of your system will encapsulate the decision-making process regarding which block is selected next. The level of difficulty of the game depends on the policy for selecting the next block. You are to support the following difficulty levels:

- Level 0: **(Make sure you at least get this one working!)** Takes its blocks in sequence from the file `sequence.txt` (a sample is provided), or from another file, whose name is supplied on the command line. This level is non-random, and can be used to test with a predetermined set of blocks. **Make sure that `sequence.txt`, and any other sequence files you intend to use with your project, are submitted to Marmoset along with your code.**

- Level 1: The block selector will randomly choose a block with probabilities skewed such that S and Z blocks are selected with probability $\frac{1}{12}$ each, and the other blocks are selected with probability $\frac{1}{6}$ each.
- Level 2: All blocks are selected with equal probability.
- Level 3: The block selector will randomly choose a block with probabilities skewed such that S and Z blocks are selected with probability $\frac{2}{9}$ each, and the other blocks are selected with probability $\frac{1}{9}$ each. Moreover, blocks generated in level 3 are "heavy": every command to move or rotate the block will be followed immediately and automatically by a downward move of one row (if possible).
- Level 4: In addition to the rules of Level 3, in Level 4 there is an external constructive force: every time you place 5 (and also 10, 15, etc.) blocks without clearing at least one row, a 1x1 block (indicated by `*` in text, and by the colour brown in graphics) is dropped onto your game board in the centre column. Once dropped, it acts like any other block: if it completes a row, the row disappears. So if you do not act quickly, these blocks will work to eventually split your screen in two, making the game difficult to play.

For random numbers, you can use the `rand` and `srand` functions from `<cstdlib>`.

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

## Command Interpreter

You interact with the system by issuing text-based commands. The following commands are to be supported:

- `left` moves the current block one cell to the left. If this is not possible (left edge of the board, or block in the way), the command has no effect.
- `right` as above, but to the right.
- `down` as above, but one cell downward.
- `clockwise` rotates the block 90 degrees clockwise, as described earlier. If the rotation cannot be accomplished without coming into contact with existing blocks, the command has no effect.
- `counterclockwise` as above, but counterclockwise.
- `drop` drops the current block. It is (in one step) moved downward as far as possible until it comes into contact with either the bottom of the board or a block. This command also triggers the next block to appear. Even if a block is already as far down as it can go (as a result of executing the `down` command), it still needs to be dropped in order to get the next block.
- `levelup` Increases the difficulty level of the game by one. The block showing as next still comes next, but subsequent blocks are generated using the new level. If there is no higher level, this command has no effect.

- `leveldown` Decreases the difficulty level of the game by one. The block showing as next still comes next, but subsequent blocks are generated using the new level. If there is no lower level, this command has no effect.
- `norandom file` Relevant only during levels 3 and 4, this command makes these levels non-random, instead taking input from the sequence `file`, starting from the beginning. This is to facilitate testing.
- `random` Relevant only during levels 3 and 4, this command restores randomness in these levels.
- `sequence file` Executes the sequence of commands found in `file`. This is to facilitate the construction of test cases.
- `I`, `J`, `L`, etc. Useful during testing, these commands replace the current undropped block with the stated block. Heaviness is detemined by the level number. Note that, for heavy blocks, these commands do not cause a downward move.
- `restart` Clears the board and starts a new game.
- `hint` Suggests a landing place for the current block. The game should suggest the best place to put the current block (spend some time thinking about what "best" might mean), but it must not suggest a position that is not legally reachable (for example, any postion that cannot be reached without moving the block upwards). The hint should be indicated on the text display by a block made of `?`'s in the suggest position, and in the graphics display using the colour black. On the very next command, no matter what the command is, the hint must disappear from the displays.

End-of-file (EOF) terminates the game.

Only as much of a command as is necessary to distinguish it from other commands needs to be entered. For example, `lef` is enough to distinguish the `left` command from the `levelup` command, so the system should understand either `lef` or `left` as meaning `left`.

In addition, commands can take a multiplier prefix, indicating that that command should be executed some number of times. For example `3ri` means move to the right by three cells. If, for example, it is only possible to move to the right by two cells, then the block should move to the right by two cells. Similarly, `2levelu` means increase the level by two. Prefixes of 0 or 1 are permitted, and mean that the comand should be run 0 times and 1 time, respectively. It is valid to apply a multiplier to the `drop` command. The command `3dr` (or `3dro` or `3drop`)) means drop the current block and then drop the following two blocks from their default position. If blocks are heavy, a multiplied command is still followed by only own downword move (example: the command `3left` moves the block three positions left, and then one position down). It is not valid to apply a multiplier to the `restart`, `hint`, `norandom`, or `random` commands (if a multiplier is supplied, it would have no effect).

**Although it is not required that your program respond to invalid input, we highly recommend that you arrange that invalid input (e.g., misspelled commands) not cause your program to crash. It will cost you precious time during your demo otherwise.**

**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.

The board should be redrawn, both in text and graphically, each time a command is issued. For the graphic display, redraw as little of the screen (within reason) as is necessary to make the needed changes. For multiplied commands, do not redraw after each repetition; only redraw after the end.

## Scoring

The game is scored as follows: when a line (or multiple lines) is cleared, you score points equal to (your current level, plus number of lines) squared. (For example, clearing a line in level 2 is worth 9 points.) In addition, when a block is completely removed from the screen (i.e., when all of its cells have disappeared) you score points equal to the level you were in when the block was generated, plus one, squared. (For example if you got an O-block while on level 0, and cleared the O-block in level 3, you get 1 point.)

You are to track the current score and the hi score. When the current score exceeds the hi score, the hi score is updated so that it matches the current score. When the game is restarted, the current score reverts to 0, but the hi score persists until the program terminates.

## Command-line Interface

Your program should support the following options on the command line:

- `-text` runs the program in text-only mode. No graphics are displayed. The default behaviour (no `-text`) is to show **both** text **and** graphics.
- `-seed xxx` sets the random number generator's seed to `xxx`. If you don't set the seed, you always get the same random sequence every time you run the program. It's good for testing, but not much fun.
- `-scriptfile xxx` Uses `xxx` instead of `sequence.txt` as a source of blocks for level 0.
- `-startlevel n` Starts the game in level `n`. The game starts in level 0 if this option is not supplied.

## Grading

Your project will be graded as described in the project guidelines document.

Even if your program doesn't work at all, you can still earn a lot of marks through good documentation and design, (in the latter case, there needs to be enough code present to make a reasonable assessment).

**Above all, make sure your submitted program runs, and does something!** You don't get correctness marks for something you can't show, but if your program at least does something that looks like the beginnings of the game, there may be some marks available for that.

## If Things Go Wrong

If you run into trouble and find yourself unable to complete the entire assignment, please do your best to submit something that works, even if it doesn't solve the entire assignment. For example:

- can't do block rotations
- program only produces text output; no graphics
- only one level of difficulty implemented
- does not recognize the full command syntax
- score not calculated correctly

You will get a higher mark for fully implementing some of the requirements than for a program that attempts all of the requirements, but doesn't run.

**Every time you add a new feature to your program, make sure it runs, before adding anything else. That way, you always have a working program to submit. The worst thing you could do would be to write the whole program, and then compile and test it at the end, hoping it works.**

A well-documented, well-designed program that has all of the deficiencies listed above, but still runs, can still potentially earn a passing grade.

## Plan for the Worst

Even the best programmers have bad days, and the simplest pointer error can take hours to track down. So be sure to have a plan of attack in place that maximizes your chances of always at least having a working program to submit. Prioritize your goals to maximize what you can demonstrate at any given time. We suggest: save the graphics for last, and first do the game in pure text. One of the first things you should probably do is write a routine to draw the game board (probably a `Board` class with an overloaded friend `operator<<`). It can start out blank, and become more sophisticated as you add features. You should also do the command interpreter early, so that you can interact with your program. You can then add commands one-by-one, and separately work on supporting the full command syntax. Take the time to work on a test suite at the same time as you are writing your project. Although we are not asking you to submit a test suite, having one on hand will speed up the process of verifying your implementation.

You will be asked to submit a plan, with projected completion dates and divided responsibilities, as part of your documentation for Due Date 1.

## If Things Go Well

If you complete the entire project, you can earn up to 10% extra credit for implementing extra features. These should be outlined in your design document, and markers will judge the value of your extra features.

## Submission Instructions

See **project_guidelines.pdf** for instructions about what should be included in your plan of attack and final design document.