CS 247 Software Engineering Principles — Spring 2021

Projects 2 and 3

Project 2 Due Date: 16. July 2021 @ 23:59
Project 3 Due Date: 3. August 2021 @ 23:59

Update history:

- 9. July 2021: updated J and L blocks representation. Every block consists of four cells.

1 Overview

In projects 2 and 3 you will develop a variant of Tetris that we call Quadris. Find the detailed game description in Section 2.

Your project will be graded based on correctness and completeness (60%), documentation (20%), and design (20%). Grading will proceed in two separate project phases, for projects 2 and 3. Further details on grading are in Section 3.

1.1 Expected Effort

This project is intended to be doable by two to three people in two weeks. Because the breadth of students' ability 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, while others won't finish at all. We will attempt to grade this two-part project 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. A perfect score will require a complete implementation. If you finish the entire program early, you can add extra features for a few bonus 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 compiles and at least does something.

2 The Game of Quadris

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

A game of Quadris consists of a board, 11 columns in width and 15 rows in height. 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 described in the following sections.

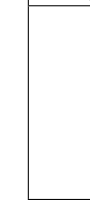
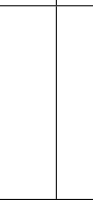
2.1 Blocks

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

I-block	J-block	L-block	O-block	S-block	Z-block	T-block
IIII	J JJJ	L LLL	OO OO	SS SS	ZZ ZZ	TTT T

Design Question: How could you design your system to allow for some generated blocks to disappear from the board if they are 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 counter-clockwise 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 of clockwise rotation are as follows:

Original J-block	Clockwise rotation		Original S-block	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.

2.2 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 cannot be rotated.

When a block shows up on the board, it should appear in the top-left corner, just below the three reserve rows. If there is no 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.

2.3 Display

You need to provide both a text-based display and a graphical display of your game board. A sample text display follows (the row numbers on the left side are optional):

```
   Level:      1
   Score:      0
   Hi-Score: 100
   -------------
 1
 2
 3
 4  TTT
 5   T
 6
 7
 8
 9
10
11
12
13           S
14          ZSS
15         ZZIS
16         ZJI
17       OO JI
18  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, with each type of block being rendered in a different colour. Do your best to make it visually pleasing.

2.4 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 a probability of 1/12 each, and the other blocks are selected with a probability of 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 of 2/9 each, and the other blocks are selected with a probability of 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 Level3, 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 into two, making the game difficult to play.

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

Design Question: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation? (Hint: a design pattern!!)

2.5 Command Interpreter

Your program should allow user interaction with the system through text-based commands. Commands to be supported are:

`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` Drop 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 the 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 the 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, S, Z, O, T Useful during testing. These commands replace the current undropped block with the designated block. Heaviness is determined 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" means), but it must not suggest a position that is not legally reachable (for example, any position 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 suggested 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 display.

Some additional features to be supported:

- 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`.
- Commands can take a multiplier prefix, indicating that the entered 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 command 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`, `3drop`) means drop the current block, as well as the subsequent two blocks, from their default position.
  - If blocks are heavy, a multiplied command is still followed by only own downward move. For 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.
- **Your program MUST handle invalid input by repeatedly querying for correct input like typical command-line applications.**

Design 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 the 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.

2.6 Scoring

The game is scored as following:

- When a line (or multiple lines) is cleared, you score points using the following formula:

$$\text{scored points} = (\text{current level} + \text{number of lines})^2$$

  For example, clearing 1 line in level 2 is worth $(2+1)^2 = 9$ points.
- In addition, when a block is completely removed from the screen (i.e., when all of its cells have disappeared), you score bonus points using the following formula:

$$\text{bonus points} = (\text{the level at which the block was generated} + 1)^2$$

  For example, if you got an O-block while on level 0, and cleared it in level 3, you get $(0+1)^2 = 1$ point bonus.

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.

2.7 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 do not set the seed, you always get the same random sequence every time you run the program. It is good for testing, but not much fun.

`-scriptfile xxx` Uses `xxx` instead of `sequence.txt` as 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.

2.8 Graphical Interface

Your program needs to work on the University systems. Therefore, this project will require you to set up X11 connection forwarding to experiment with the GUI. Some guidelines follow.

- Linux (and Linux Student Server): It should be sufficient to just `ssh` with the flag `-X`, that is instead of typing
  `ssh id@linux.student.cs.uwaterloo.ca`
  you instead type
  `ssh -X id@linux.student.cs.uwaterloo.ca`
  You may have to install `xlib` if you don't already have it.
- Windows: If you are in Windows then you will also need to install software, in this case probably `xming`. If you are using putty then you should make sure to enable X11 forwarding under the X11 tab under ssh in putty.
- Mac: Similar to Linux, you would need to ssh with the `-X` flag as above, but you must install xQuartz and have it running.

Also, if you are working on your own machine, make sure you have the necessary libraries to compile the graphics. Try executing the following: `g++ -L/use/X11R6/lib window.cc graphicsdemo.cc -lX11 -o graphicsdemo`

3 Grading

Your project will be graded as described below.

Even if your program does not 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 meaningful! You do not get correctness marks for something you cannot 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.

3.1 Correctness and Completeness

The correctness and completeness component of your project will be assessed by the teaching team using the list of required features as outlined above. You must develop your game with close adherence to the list of features. Bonus marks will be assigned by the TAs based on the bonus features you have introduced in your design.

WARNING: If your project submission on Marmoset does not compile, you will receive a grade of 0 on the correctness and completeness portion of the project. The TAs will NOT, under any circumstances, change your code or allow you to re-submit the code after the deadline.

3.2 Documentation and Design

The design of your project is an important part of your overall assessment. For top grades, it is not enough that your program simply works; it must demonstrate a solid object-oriented design. As a result, you will need to spend considerable time planning out your classes and their interactions, aiming to minimize coupling while maximizing cohesion. You must plan for change in your program. Imagine that the specification will change on the date before the final due date (it won't, but imagine it will). How easily can you accommodate these changes?

As part of your design, you should anticipate various ways your project could change. Perhaps a change in rules, or input syntax, or a whole new feature — who knows? Plan all of your project features to accommodate changes, with minimal modification to your original program, and minimal recompilation. In your final design document, outline the ways in which your design accommodates

new features and changes to existing features. Be specific. You may wish to implement some of these for extra credits. **This discussion itself is the deliverable for Project 2; therefore, if you spend less than a page on it, you probably aren't saying enough!**

Your documentation and design will be assessed via written documents that you will submit to Marmoset as pdfs.

3.3 If Things Go Wrong

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

- Cannot do block rotation.
- 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 does not run.

INCREMENTAL ENGINEERING! 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 can do would be to write the whole program, and then compile and test it at the end. There is no successful leap-of-faith in software engineering.

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

3.4 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. Here is our recommended plan of attack:

- Develop the game in pure text first.
- First, implement a routine to draw the game board (probably a Board class with an overloaded friend `operator<<`). Start off with a rather blank and bland game board, and add incremental features to make it more sophisticated.
- Next, implement a command interpreter as early as possible, so that you can interact with your program. Incrementally add more commands, but work on one command at a time. Work on supporting the full command syntax only after all the required functionalities for the commands are implemented.
- Spend time to work on a test suite at the same time as you develop your code. Although we are not asking you to submit a test suite, having one on hand will speed up the process of verifying your implementation.
- Save the graphics for last. Do not do both in parallel.

You will be asked to submit a plan, with projected completion dates and divided responsibilities as part of your documentation by the Project 2 due date.

3.5 If Things Go Well — Extra Credit Features

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.

Both project phases give you the opportunity to earn up to 10% extra credit for enhancement to the core projects. Keep the following rules and guidelines in mind when planning enhancements:

- It is far more important to have your core project working well, than to have a poorly done project with extras. Remember, the enhancements are only worth 10%. Your documentation and design are also more valuable than your enhancements, so please take time to do these well.
- Your program must be able to run the core project, without enhancements, as well as with them. You are not allowed to submit two programs for grading — one with enhancements and one without. You are not allowed to recompile your program with different compiler flags to produce versions with and without enhancements. You may select or deselect your enhancements via flag arguments on the command line. (e.g., `./project -enablebonus`, or something similar). Even better would be if you could turn your enhancements on and off dynamically, as the program is running.
- One enhancement that is available for projects 2 and 3 is offered as a challenge: complete the entire project, without leaks, and without explicitly managing your own memory. In other words, handle all memory management via STL vectors and smart pointers. If you do this, there should be no delete statements in your program at all, and very few raw pointers (the only raw pointers you would be permitted to have are those that are not meant to express ownership). Successful completion of this challenge will earn you 4 of the available 10 bonus marks. If your program leaks, these marks cannot be earned. Also note that these 4 marks are not reserved for this feature; it is theoretically possible to get all 10 bonus marks without implementing this challenge.

3.6 Project 2 Deliverables

Submit a Project Plan (a.k.a. plan of attack or Work Breakdown Structure). This should include **A) a UML diagram that describes how you anticipate your system to be structured** (even if you complete project 3 by the due date of project 2, the UML diagram you submit must be the one that you built prior to starting the project).

Your UML diagram should show the classes that make up your project and the relationships between them. You only need to show public methods (i.e., you can leave out private fields and protected / private methods, unless you need to show them to illustrate a point, e.g., a design pattern). Do not show the SMFs, or any other constructors, accessors, or mutators. You will not be graded on the degree to which you adhere to this model, but you will be asked to account for any differences that arise between this model and your final submission file — `uml.pdf`.

In addition, your project plan must include **B) a breakdown of the project (Work Breakdown Structure, WBS)**, indicating what you plan to do first, what will come next, and so on and so forth. Include estimated completion dates and which partner will be responsible for which parts of the project. You should try to stick to your plan, but you will not be graded by the degree to which you adhere to it. Your initial plan should be realistic, and you will be expected to explain why you had to deviate from your plan (if you did).

Finally, your initial project plan must include **C) answers to all the Design Questions con-**

tained in the project description above. You should answer in terms of how you would anticipate solving these problems in your project, even though you are not strictly required to do so.

If your answers turn out to be inconsistent with your final design submitted in Project 3, you will have an opportunity to submit revised answers on the Final Assessment.

Your plan should be no more than 5 pages long.

3.7 Project 3 Deliverables

On the due date of project 3, you must submit the implementation portion of the project:

- All of your code to Marmoset, together with a `Makefile`, such that issuing the command `make` builds your project. Your executable should be called `quadris`.
- Your code will be assessed on both correctness as well as styling in the same way as the assignments.

3.8 Late Policy

We strongly discourage you from making last-minute changes to your code. The risk of your code not compiling, or otherwise not working, is too high. We recommend writing no new code in the last six hours before your code is due. Save those hours for priming your final documentation, and for emergency bug fixes.

Late submissions for Project 2 and Project 3 will not be accepted.