Spring 2018 CS 32

Programming Assignment 3
Chetyris

Time due: 11:00 PM Thursday May 24

Introduction

You have been hired by SmallSoft, the world's second-largest producer of tacky video games, to produce a new video game called Chetyris. Chetyris is a knock-off of the world-famous Tetris game (with an allusion to Tetris's Russian origin — четыре, transliterated as *chetyre*, means *four*).

In Chetyris, your job as the player is to manage a well that is 10 units wide by 18 units deep. At the top of the well, pieces of various shapes are dropped into the well. The pieces are made up of one or more small blocks in various geometric configurations. For example, here are some of the blocks (in their initial orientation) that will be dropped, along with their conventional names:

```
 #      #   #     ##     ##   ##
###   ###   ###   ##    ##     ##   ####

 T     L     J     O     S     Z     I
```

At the start of each game, the well starts out entirely empty. As each piece falls it can be rotated clockwise or moved horizontally so that every space in the well is filled. If and when a horizontal row of the well is completely filled with blocks, that row is vaporized and all of the blocks above fall into the void created by the vaporized row of blocks. In addition, every time you vaporize a row, you get points. On the other hand, if a row is covered by one or more pieces before it is completely filled with blocks, then the unfinished row simply takes up space in the well. The more such incomplete rows that the player leaves, the higher the pieces stack within the well, reducing the space in which falling pieces can be manipulated. If the blocks ever reach the top of the screen, preventing pieces from being dropped, then the game ends.

The game is divided into levels, so that the player can take a rest after each level. During the first level, the player must complete and vaporize five full rows of blocks before advancing to the next level. At the second level, the player must complete ten full rows; on the third level, fifteen full rows, etc. At the end of a level, all remaining blocks in the well are cleared out, so that each successive level starts with an empty well.

With each succesive level, gravity increases so that pieces fall faster than during the previous level, giving the player less time to manipulate the pieces before they come to rest at the bottom of the well. The game continues (potentially indefinitely) until the well fills to the top with pieces.

Here is a screen shot of the Chetyris game, showing an S piece coming down:

```
@           @
@           @
@           @
@      ##   @    Next piece:
@     ##    @     #
@           @    ###
@           @
@           @
@           @    Score:           0
@           @    Rows left:       5
@           @    Level:           1
@           @
@           @
@           @
@ $         @
@$$         @
@$$$        @
@$$   $$$$ @
@@@@@@@@@@@@
```

To play the game, the player may hit any of the following keys:

- To move the current piece to the left, the left arrow key or 'a' or '4'.
- To move the current piece to the right, the right arrow key or 'd' or '6'.
- To rotate the piece clockwise, the up arrow key or 'w' or '8'.
- To move the current piece down one row (to speed things up), the down arrow or 's' or '2'.
- To move the current piece down as far as it will go (to speed things up more), the space bar.
- To quit the game, 'q' or 'Q'.

The player may hit these keys one or more times to shift the current piece left or right as it falls into the well. If a piece is falling slowly enough, it is possible to shift a piece two or more times (e.g. twice to the left) before the piece advances down a single row in the well.

In addition to the seven classic Tetris shapes shown above, Chetyris has three more pieces:

The VaporBomb: `##`

Once the vapor bomb comes to rest on other blocks in the well or at the bottom of the well, it will vaporize itself and also obliterate the square of four blocks above the vapor bomb and the square of four blocks below the vapor bomb. This is the special action of the VaporBomb. They are sent down occasionally to help the player zap blocks that are in the way. (See the Game Details section below for more details.)

The FoamBomb: `#`

Once the foam bomb comes to rest on other blocks in the well or at the bottom of the well, it will immediately inflate, much like exploding whipped cream, to fill out the void surrounding and including the landing spot of the foam bomb. This is the special action of the FoamBomb. This can help to fill hard-to-reach voids in the well. (See the Game Details section below for more details.)

The CrazyShape: `# #` / `##`

The CrazyShape doesn't look all that crazy, but responds in an opposite fashion to the player's keystrokes. To shift the CrazyShape right, the player must press the left arrow key, and to shift the CrazyShape left, the player must press the right arrow key. This can be very confusing during high-speed play. (See the Game Details section below for more details.)

Game Details

Here is a functional specification for the Chetyris game:

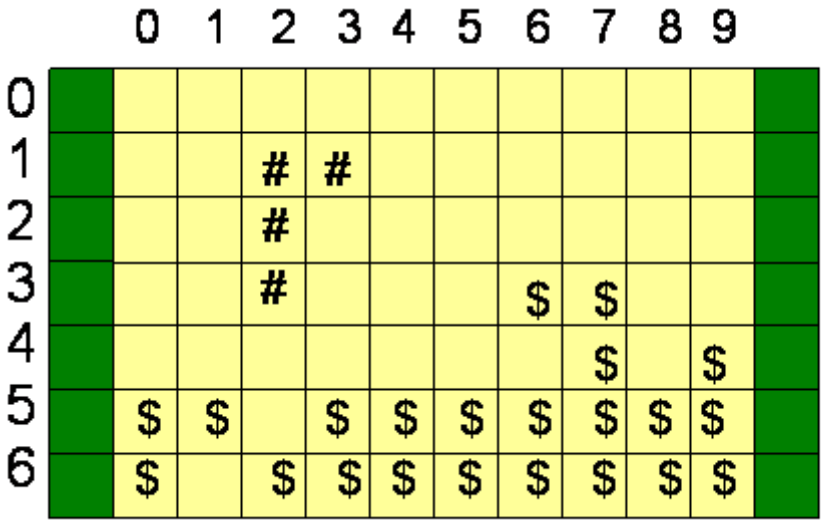
Gameplay Rules

1. There are an unlimited number of levels in the Chetyris game. The first level number is Level #1 (not level #0).

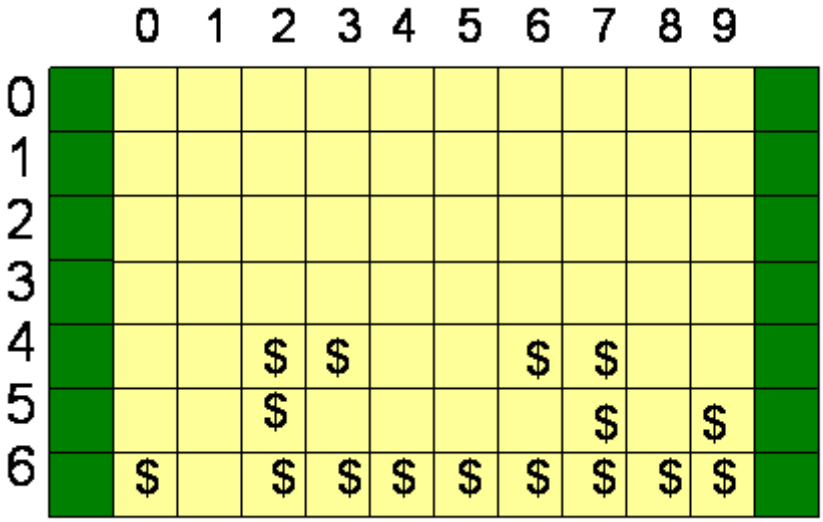
2. At the start of each level, the well is empty.

3. To complete a given level n, the player must vaporize 5*n rows by filling 5*n rows full of blocks (all 10 blocks must be filled in a row to vaporize the row). Once 5*n rows have been vaporized, the well's contents will be cleared, and the game will prompt the player to start the next level.

   The following diagram shows a J piece as it is about to complete, and thus vaporize, a row. Once the bottom end of the piece settles to rest in position (2,5), row #5 will be completed and should be vaporized. (Notice the coordinate system we are using: The first coordinate is the x coordinate, which grows toward the right; the second is the y coordinate, which grows downward.)



   And here is the resulting well, after row 5 has been vaporized:



4. At the start of the game, the first piece will be placed at the top of the well (see details below) in its default orientation (orientation #0) and then will proceed to fall a row at a time until it reaches the bottom of the well.

5. The game should provide the player with a different random sequence of pieces each time the game is played.

6. Once a normal piece (one without a special action) has come to rest at the bottom of the well or on top of another piece (see details below), the game will determine if one or more new rows were entirely filled. If a row was entirely filled then it will be removed from the well, shifting all rows above the removed row down by one. If more than one row was completed at the same time, then all these rows will be removed and the rows above will be shifted down by the number of rows that were removed.

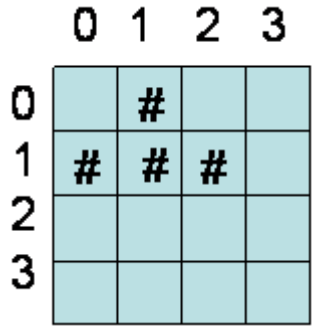
7. Once a special piece (e.g. a VaporBomb) comes to rest, the program will apply the special action of the piece (e.g. vaporizing blocks above and below the piece). Next, the program must determine if one or more new rows were entirely filled due to the special action of the piece. If a row was entirely filled then it will be removed from the well, shifting all rows above the removed row down by one. If more than one row was completed at the same time, then all these rows will be removed and the rows above will be shifted down by the number of rows that were removed.

8. After a piece comes to rest and any completed rows have been removed, the next piece begins to fall from the top of the well. Each piece always starts in its default orientation (Orientation #0)

9. If, when a new piece is to be dropped into the well, there is no room for this piece (i.e. one or more of the blocks comprising the piece would immediately overlap another block which is already in the well from a previous piece), then the game is over. However, if the new piece can be placed in the well without overlapping any other blocks in the well, then the game continues (even if the newly-placed piece happens to be sitting one row above other

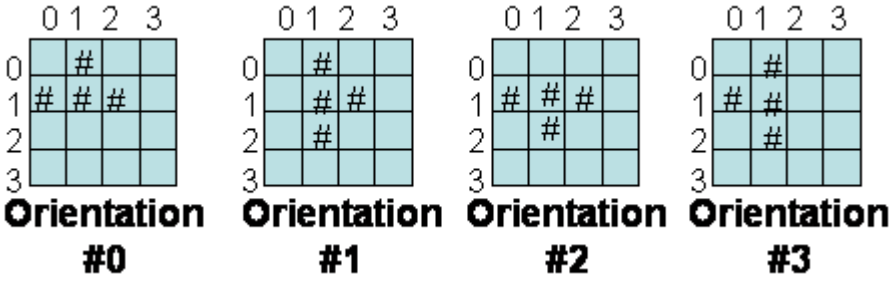
blocks, since the player may be able to quickly shift the new piece to the right or left, possibly permitting it to descend further in the well).

10. All pieces are considered to be inscribed within a 4x4 *bounding box*. For example, consider the T piece below:

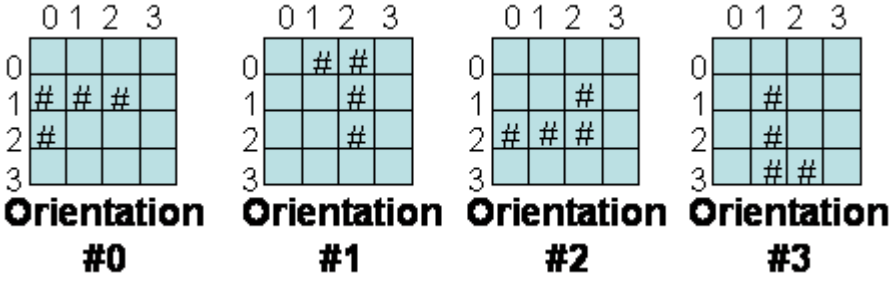


The following specifications show how each Chetyris piece *must* be considered inscribed within its 4x4 bounding box, with # signs representing the *blocks* in each piece:

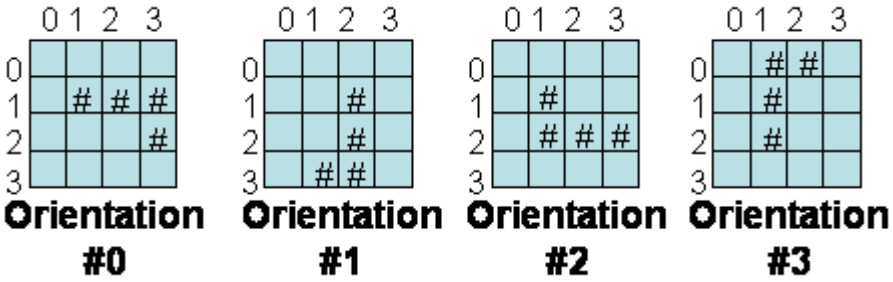
The T Piece



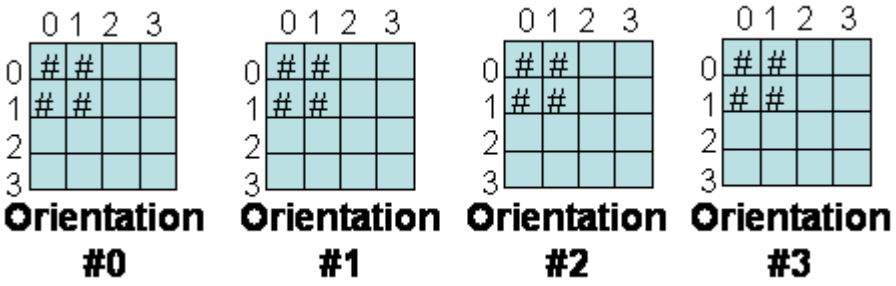
The L Piece



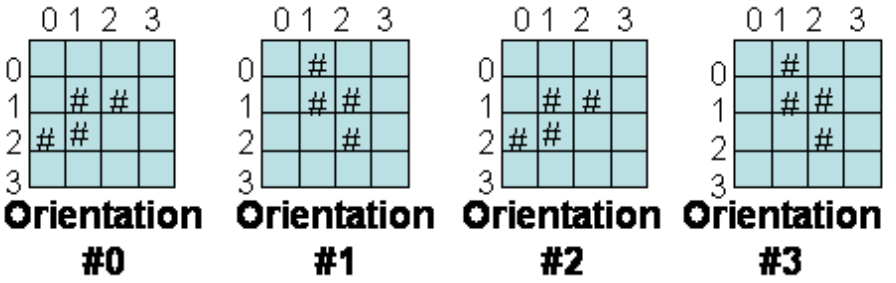
The J Piece



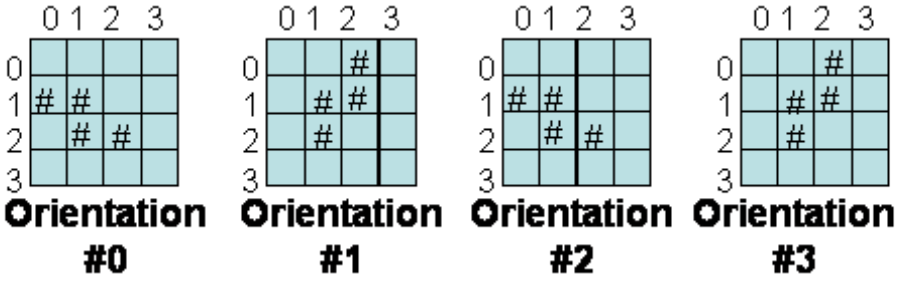
The O Piece



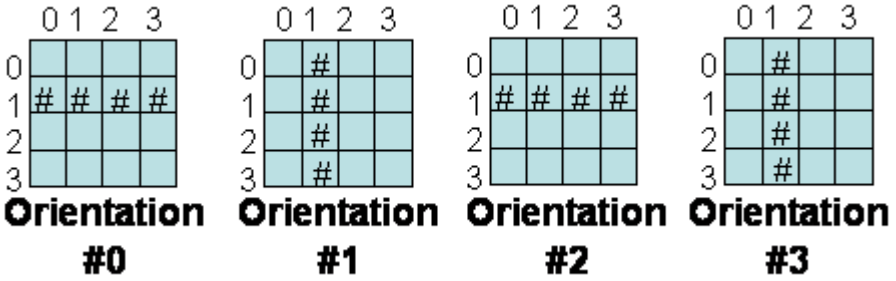
The S Piece



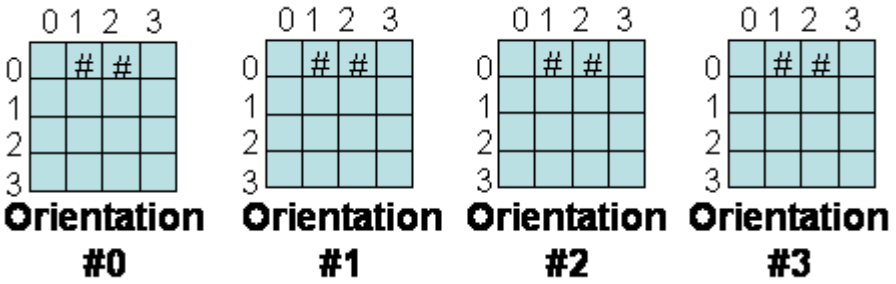
The Z Piece



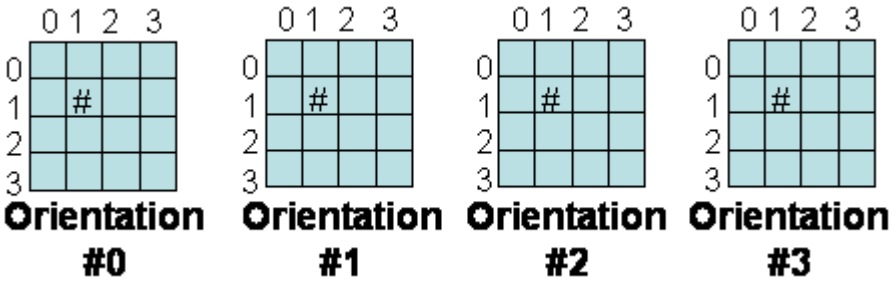
The I Piece



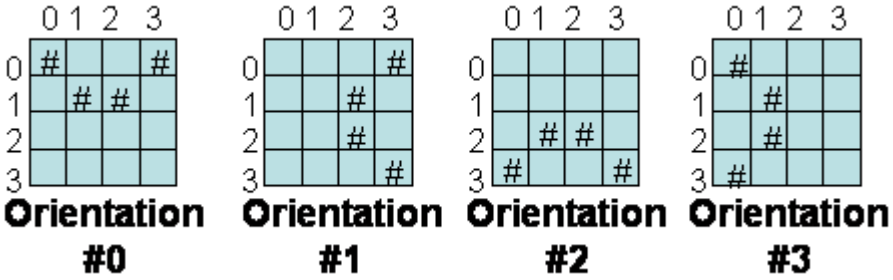
The VaporBomb



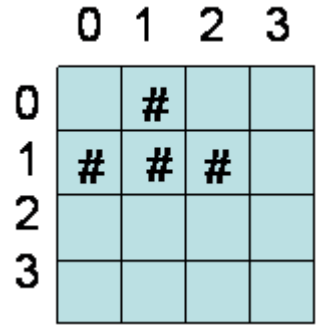
The FoamBomb



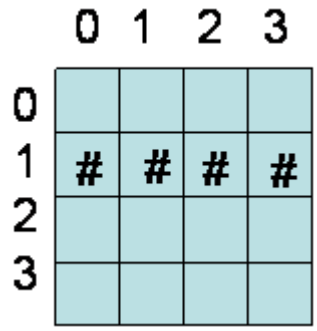
The CrazyShape



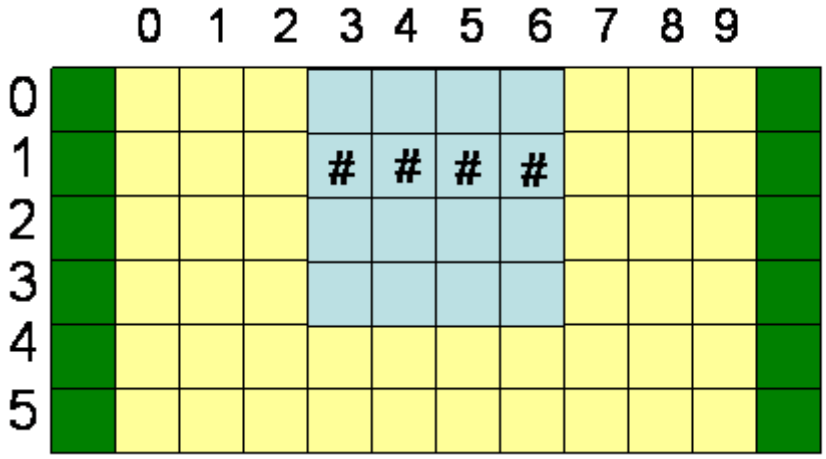
11. When a new piece is initially placed at the top of the well, the upper-left corner of its bounding box **must** be placed at offset X=3, Y=0 within the well. For example, consider the T shape in its default orientation (orientation #0):



In this case, the first # sign on the top row of the grid above would be placed at X=4, Y=0 within the well. In contrast, consider the I piece below:



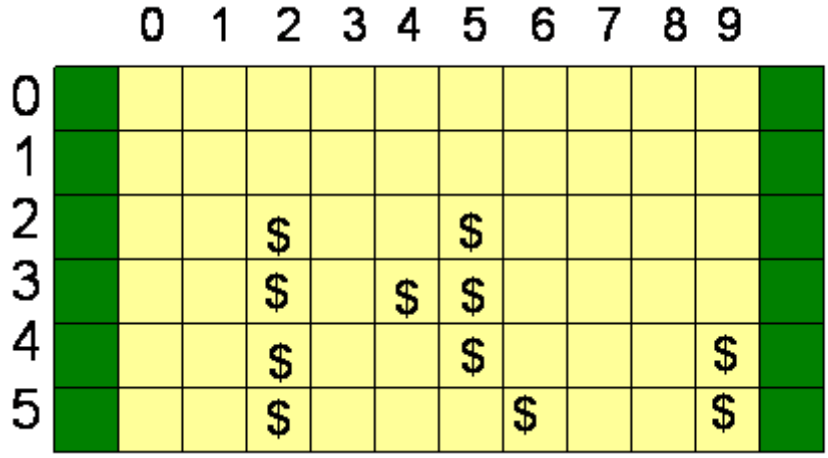
Again, the upper-left corner of the bounding box will be placed at X=3, Y=0. In this case, since the bar, in its default orientation, occupies the second row of blocks, the left-most block of the bar would initially be placed at X=3, Y=1 of the well. The second left-most block would be placed at X=4, Y=1 of the well, as shown below:



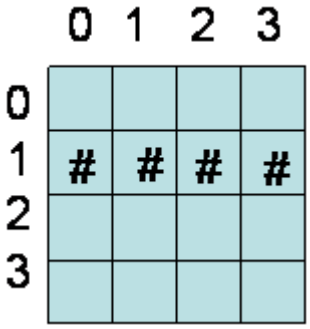
12. When a new piece has been placed in its initial starting position in the well, there are two possibilities:

    a. The new piece, in its initial position, already directly overlaps an existing block within the grid. In this case, the game is over.
    b. The new piece, in its initial position, does not directly overlap any existing block within the grid. In this case, the play continues. Note: overlap does not mean "sits immediately above".

    For instance, consider the top six rows of the following well, which has already had a number of pieces played (shown as $ signs):

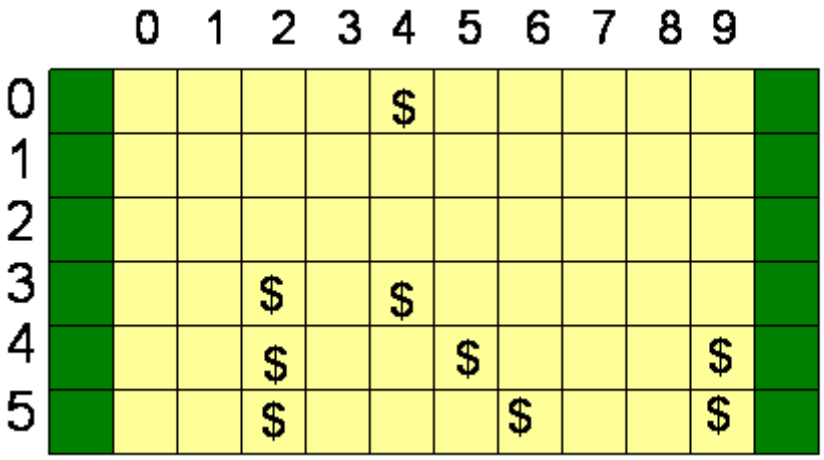


    If the I piece



were to be dropped as the next piece into the well, the blocks of the piece would be placed at offsets (3,1), (4,1), (5,1), and (6,1). Since none of these slots in the well are occupied, this shape would not be considered to overlap an existing object. Even though the piece sits immediately above the block at position (5,2), this is not considered to be overlapping. The player could potentially quickly shift the piece to the right so that its left–most block is in position (6,1), thereby avoiding the obstacle at (5,2) before the piece advances to the next row.

13. There may be unusual situations where a new piece is actually placed below existing blocks that are located in row 0 of the well. For example, consider what would happen if an I piece were dropped into the well given this layout (which is possible, given the effect of VaporBombs):



The I piece in its initial position would still start at row 1 in the well, from X=3 to X=6. But since the bar would *not* overlap any blocks (including the block at (4,0)) this would not be considered a game–ending scenario. Instead, the bar would continue to fall below the block at (4,0) until it came to rest below. (This is not necessarily the way the original Tetris game works, but is a simplification for this assignment.)

14. Once a new piece appears in the well, the game will allow the player exactly T = maximum(1000–(100*(L–1)), 100) milliseconds to perform commands (e.g. move the piece left one or more times) before advancing the shape down one row in the well, where L is the current level number. Each subsequent time the piece shifts down, the player will have T milliseconds to perform commands.

15. The player may hit any of the command keys (e.g. the arrow keys) as many times as desired during this T milliseconds to shift or rotate the piece. In other words, the player might shift the object 3 times to the left, then 7 times to the right, all before T milliseconds elapse and the piece is shifted down by one row.

16. If the player hits the 'q' or 'Q' key, the game should end immediately and display the Game Over message (see the Display Rules section for more information).

17. When rotating a piece, the piece should cycle from Orientation #0 to Orientation #1, to Orientation #2, to Orientation #3, to Orientation #0, etc. When rotating a piece, you must ensure that the bounding box enclosing the shape stays at the same X,Y coordinate. The X,Y coordinate of a piece should only change if the player moves the piece left or right, or if the piece advances down a row in the well.

18. If the act of rotating a piece or shifting a piece either left or right would cause one or more blocks comprising the piece to overlap with blocks already in the well (from previous pieces) or to be outside the top, bottom, left, or right boundary of the well, then the rotation/shifting must fail and no change should be made to the piece.

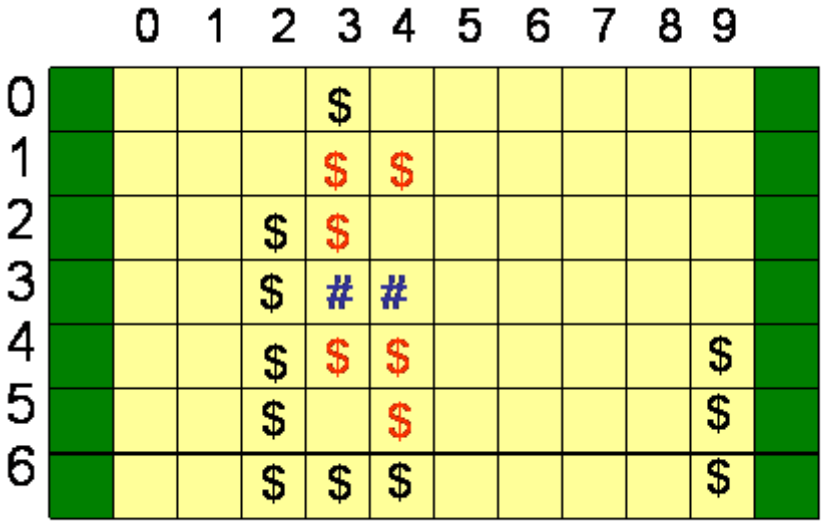
19. When a piece is *about* to shift down to the next row (due to the T milliseconds elapsing, or due to the player hitting the down arrow key to manually advance the piece), if any block within the piece would overlap with an existing block within the well or if any block within the piece would descend past the bottom row of the well, then the piece should *not* be shifted down further as it has reached the bottom of its descent. This piece has come to rest.

20. When the player hits the space bar, the piece immediately descends to the position it would end up at if the user had instead hit the down arrow key repeatedly. It has come to rest.

21. When a piece has come to rest,

    a. If the piece is a normal piece without a special action (i.e., one of the seven classic Tetris shapes or the CrazyShape), the piece should become part of the well at its current position and orientation.
    b. If the piece is one with a special action (i.e., a VaporBomb or FoamBomb), then the bomb's "payload" should go off in the piece's current position. For example, the VaporBomb will vaporize itself and all blocks up to 2 below and 2 above it.
    c. After either (a) or (b) above has been performed, the game must check for any fully completed rows. The contents of all such fully completed rows should be removed by shifting the rows above them to replace them.
    d. Finally, the next piece will begin to fall from the top of the well.

22. Once a VaporBomb has come to rest (on top of other blocks or at the bottom of the well), its special action must be executed. The VaporBomb's special action is to remove all blocks that are up to two rows directly above or below the VaporBomb, plus the blocks of the VaporBomb itself. Since the VaporBomb is exactly two blocks wide, it will clear up to a 2–wide by 5–high column when it comes to rest. For example, consider the following scenario:

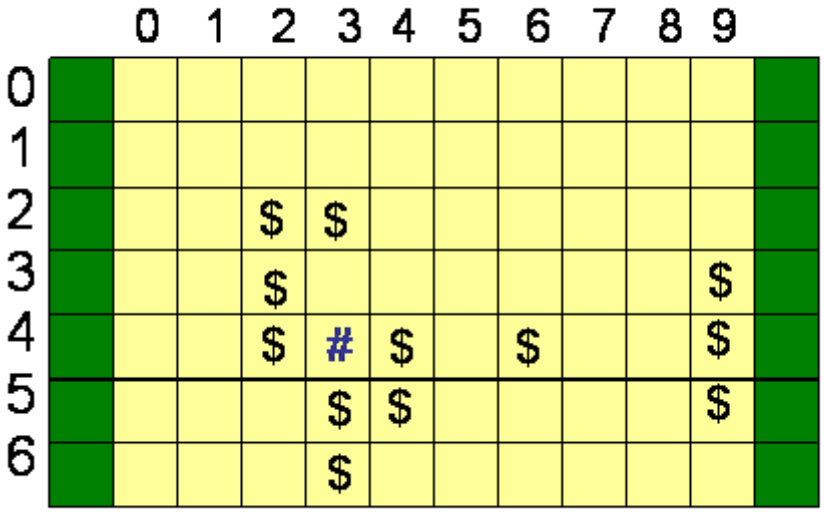


Suppose the VaporBomb, shown by # signs, were to come to rest at position (3,3) as shown above (presumably because it was dropping in columns 5 and 6, say, and at row 3, the player quickly shifted it left into columns 3 and 4). It would vaporize all of the dollar signs indicated in red, but would not remove the dollar signs at positions (3,0), (3,6), or (4,6).

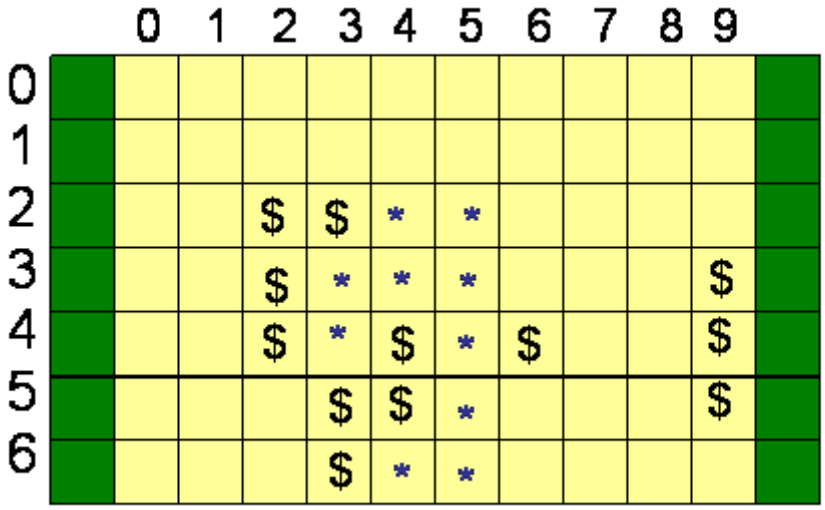
23. Once a FoamBomb has come to rest (on top of other blocks or at the bottom of the well), its special action must be executed. Assuming the FoamBomb comes to rest at position (X,Y), its special action is to fill with foam (indicated by *) all empty slots that are in the 5x5 block bounding box (X−2,Y−2) to (X+2,Y+2), subject to the constraint that they are *reachable* from the FoamBomb:

    a. All empty slots within the 5x5 bounding box that are directly adjacent, either vertically or horizontally (but not diagonally), to the FoamBomb should be filled with a * block.
    b. All empty slots within the 5x5 bounding box that are directly adjacent, either vertically or horizontally, to slots defined in (a) should also be filled with a * block.
    c. Step (b) should be recursively repeated for all blocks that are directly adjacent, either vertically or horizontally, to empty slots within the 5x5 bounding box as defined in (b) or in (a) — they should also be filled with a * block.

    For example, consider the board shown below with a FoamBomb coming to rest at position (3,4) (presumably because it was dropping in column 4, say, and at row 3, the player quickly shifted it left into column 3):



Once the foam bomb explodes, the screen would look like this:



Notice that positions (2,5), (2,6), (1,2), (1,3), (1,4), (1,5), and (1,6) are not filled in because they cannot be reached via horizontal or vertical adjacent squares from the starting position of the FoamBomb at (3,4). Other than their being displayed as a '*' character, once added to the well, FoamBomb blocks are the same as blocks from any other piece and are vaporized along with other blocks when a row is completely filled.

24. The CrazyShape is like any normal piece with one exception: To shift the CrazyShape to the right, the player must press the left arrow key, and to shift the CrazyShape left, the player must press the right arrow key. Once the CrazyShape comes to rest, like any other normal piece, it should be added to the well and the game should check for completely filled–in rows.

25. If the player hits the down arrow key to move the current piece down (for example, because the player has already positioned the piece in a good X location and wants to drop it quickly into place), the piece should shift down one row (assuming that it wouldn't overlap with another piece by being moved down by one row). If the shift down is successful and the piece has not come to rest, the timer should be reset, allowing the player a full T milliseconds to type commands before the piece is automatically advanced to the next row in the well.

26. When one or more rows are vaporized, the player is awarded points. It is possible that by placing a single piece, the player can fill more than one row in a single move. If, once a piece comes to rest:

    a. One row is fully completed, the player gets 100 points.
    b. Two rows are fully completed simultaneously, the player gets 200 points.
    c. Three rows are fully completed simultaneously, the player gets 400 points.
    d. Four rows are fully completed simultaneously, the player gets 800 points.
    e. Five rows are fully completed simultaneously, the player gets 1600 points.

Display Rules

This section describes exactly how your game should display information to the screen.

1. Here are the coordinates of all key items on the display:

    a. The upper-left corner of the left wall of the well should be displayed at offset X=0,Y=0 of the screen. The well should be surrounded by @ signs on its left, right and bottom sides. Therefore, the character displayed at X=0,Y=0 should be an @ sign. Similarly, the character displayed at X=11,Y=0 should also be an @ sign.
    b. The text "Next piece:" should be displayed at (16,3).
    c. The next piece should be displayed with the upper-left corner of its bounding box at (16,4).
    d. The current score should be displayed at (16,8) in the form "Score:" followed by five spaces followed by the score right-justified in a field seven characters wide.
    e. The number of rows left that must be vaporized before the current level is completed should be displayed at (16,9) in the form: "Rows left: " followed by one space, followed by the number of rows left before the level is completed, right-justified in a field 7 characers wide.
    f. The current level number should be displayed at (16,10) in the form "Level:" followed by five spaces, followed by the current level number, right-justified in a field seven charcaters wide.
    g. All prompts (e.g., "Press the Enter key to begin playing Chetyris!") should be displayed at (0,20) and should overwrite any previous prompts already printed to the screen (even if the previous prompt is longer than the current prompt). The player must hit the Enter key to acknowledge a prompt and continue.

2. Game display characters:

   a. The current falling piece should be drawn using # characters.
   b. All blocks comprising previously fallen pieces that have come to rest should be drawn using $ characters to differentiate them from falling pieces.
   c. All foam blocks created by a foam bomb should be drawn using * characters to differentiate them from falling pieces and the blocks of previously fallen pieces.

3. When the game begins, it should display the empty board, the current score (0), the number of remaining rows (5), and the current level (1). Then the game should prompt the player with "Press the Enter key to begin playing Chetyris!". Once the player presses the Enter key, the game should begin.

4. Immediately, the current piece should be displayed within the well and the next piece should be displayed to the right, under the text "Next piece:" showing the player what to expect next.

5. The game should re-display the well each time it or its contents change for any reason, such as:

   a. The player hits the left or right arrow keys to shift the current piece to the left or right.
   b. The piece advances down a row in the well.
   c. The piece comes to rest and fully fills in a row, causing one or more rows to vaporize.
   d. A VaporBomb vaporizes one or more blocks in the well.
   e. A FoamBomb adds foam blocks to the well (and possibly causes one or more rows to vaporize in the process).

6. Any time the score changes, the game should update the score display to the right of the well.

7. Any time the number of remaining rows changes, the game should update the rows left display to the right of the well.

8. If, when a new piece is about to be played, that piece will cause the game to end (because it overlaps with existing blocks in the well), you should still update the display to show how the piece would fit in the well.

9. When the player completes a level, the game should display the following in the prompt area of the screen (overwriting other text that may already be there): "Good job! Press the Enter key to start next level!". Once the player presses the Enter key, the next level should begin with an empty well and updated score, rows left, and level number.

10. If the player quits or if the well fills up, the game should display the following in the prompt area of the screen (overwriting other text that may already be there): "Game Over! Press the Enter key to exit!" Once the player presses the Enter key, the program should terminate.

What We Provide You

To help you understand the requirements of this project, we have provided Windows, macOS, and Linux executables that you can run. If there is a detail that is not specified explicitly the Game Details, you can use these executables to determine the intended behavior.

To help you get started, we have provided askeleton that contains the beginning of a solution. It contains the following files:

chetyris.cpp
: This little file contains the main routine. When we build your solution, we will ignore any main routine you provide and will use this file instead, so **your program must build correctly with this file**. During your program development, you might want to reduce the size of the well to make the game take less time when you're testing it.

Game.h and Game.cpp
: The Game class manages the game. You are free to make whatever changes and additions to these files that you like, provided that they build with the provided chetyris.cpp.

Well.h and Well.cpp
: In order for the skeleton code we provided to build, we provided a minimal class representing the well. You may modify these as you wish.

Piece.h and Piece.cpp
: These files are where code related to pieces should go. Also, at the point in the game where the next piece to be dropped must be chosen, you **must** call the chooseRandomPieceType function declared in this header. For example, you could say

```
switch (chooseRandomPieceType())
{
  case PIECE_I:
    ... // make the next piece an I piece
    break;
  case PIECE_L:
    ... // make the next piece an L piece
    break;
  ...
}
```

: The implementation we provided picks a piece type at random. For some of our tests of your program, we will build your program with a different implementation of chooseRandomPieceType, one that supplies a predetermined sequence of pieces with each successive call. We put the implementation of chooseRandomPieceType in chetyris.cpp so that we can easily substitute a different one when we run those tests.

UserInterface.h
: This file provides an interface for writing to the screen, reading from the keyboard, and measuring time. Use these functions for all of your input and output; don't write to cout or read from cin. We describe these functions below.

UserInterfaceWindows.cpp and UserInterfaceUnix.cpp
: These are two implementations of the interface in UserInterface.h. You will use exactly one of these files on a given platform. Use UserInterfaceWindows.cpp on a Windows system,

and UserInterfaceUnix.cpp on a macOS or Linux system.

User Interface Routines

Here is the functionality provided by the user interface library we supply.

The Screen class has member functions for writing to the screen. The gotoXY member function positions the cursor at the x and y coordinates supplied as parameters; text subsequently written appears starting at that point. The upper left corner of the screen is (0,0); x increases as you move rightward, and y increases as you move downward.

The printChar and printString member functions write a single character and a string, respectively. printStringClearLine writes a string, but also clears the rest of the characters on the current line after the string (in case any non-blanks were left over from a previously written longer string). The clear member function clears the screen.

In what was just described, for the Windows implementation, changes to the screen appear immediately. For the Unix implementation, no changes actually appear on the screen until you call the refresh member function. (By not refreshing after every single character change, there's less of a flicker as the cursor moves around.)

When it comes to keyboard input, there are three functions. waitForEnter pauses until the user presses the Enter key. discardPendingKeys throws away any keypresses that have not yet been processed by the program. The most important function is getCharIfAny. This function checks to see if the user has hit a key, and reports whether or not a key has been pressed; if one has, it also returns the character.

As for timing, the Timer class has a member function that returns the number of milliseconds that have elapsed since the Timer was created.

Let's see how these might be used. Suppose we wished to process as many keystrokes as we can in 2 seconds:

```
  // Start a timer
Timer timer;

  // Repeat as long as 2000 milliseconds have not yet elapsed
while (timer.elapsed() < 2000)
{
    char ch;
    if (getCharIfAny(ch))
    {
          // There was a character typed; it's now in ch
        switch (ch)
        {
          case ' ':
            ...
            break;
          case ARROW_LEFT:
            ...
            break;
          ...
```

```
            }
        }
    }
    // 2 seconds have elapsed
```

Project Setup

Here's what you should do to set up this project under Visual Studio.

1. Create a new project called, say, Chetyris.
2. Extract the contents of the chetyris-skeleton.zip file into the folder that already contains the .vcproj file. If your project is named Chetyris, this will be the folder Chetyris/Chetyris.
3. In the Solution Explorer window, right-click Header Files, and use Add Existing Item to add all the .h files.
4. In the Solution Explorer window, right-click Source Files, and use Add Existing Item to add all the .cpp files except UserInterfaceUnix.cpp.
5. In the course of your development, add new .h and .cpp files to the project.

Here's how to set up this project under Xcode.

1. Create a new project called, say, Chetyris.
2. In the middle panel, select the Build Settings tab, and in the search box on the line below that tab, type *linker flags* and hit Return. Click the right-pointing angle next to Other Linker Flags, then click the Debug that appears, then the + that appears. In the text box, type `-lncurses`. (The second character is a lower case ell, not a one.) Hit Return.
3. Extract the contents of the chetyris-skeleton.zip file into the folder that already contains the main.cpp file Xcode placed for you.
4. Right-click on the yellow folder icon with the name of your project and select Add Files to "*yourProjectName*" to add all the .h and .cpp files to the project except UserInterfaceWindows.cpp
5. Right-click on main.cpp and select Delete.
6. In the course of your development, add new .h and .cpp files to the project.
7. You can *not* launch the executable you build through the Go or Run commands, because Xcode does not provide a true terminal window. Instead, right-click on the executable under Products, select Show in Finder, and double-click on the executable in the Finder window.

Here's how to build from the command line under Linux or macOS.

1. Put all the .h and .cpp files in some directory — perhaps ~/Chetyris
2. Open a shell window from the Terminal utility.
3. Change directories into that directory and delete the unneeded Windows implementation file:

```
cd ~/Chetyris
rm UserInterfaceWindows.cpp
```

4. Execute one of these commands:

```
g32 -o chetyris *.cpp -lncurses              on a SEASnet Linux server

g++ -std=c++14 -o chetyris *.cpp -lncurses   on other Linux machines or a Mac
```

(The character after the hyphen is a lower case ell, not a one.) This should produce an executable file named `chetyris`. To run it, execute the command `./chetyris`

Don't know how or where to start?

When working on your first large object oriented program, you're likely to feel overwhelmed and have no idea where to start. It's important to attack your program piece by piece rather than trying to program everything at once.

We'd recommend that the first thing you do is figure out how to represent the well and how to display it, since that will let you see what's happening. Then pick one kind of normal piece, say an I or an O, and get it to drop in the well and stop at the bottom. Don't worry yet about letting the user shift or rotate it, and don't worry yet about overlaps with other blocks.

Do not proceed until you have this working. You will clear up some misconceptions you may have and will figure out some things about how the objects in the program interact with each other. From this basis, there are many ways you could choose to proceed next. You'll need to do all of these and more, but at this point you have more freedom in choosing the order.

- Consider overlaps with other blocks.
- Respond to player commands.
- Vaporize rows that fill up.
- Add just a second kind of normal piece; don't jump into adding all the kinds of pieces at once, or else any mistake in your design will probably be repeated ten times instead of two. Even with two pieces, you can discover what they have in common and how they are different to help you design appropriate classes.
- Add one of the pieces with a special action.

What to Turn In

For this project, you will turn in a zip file containing exactly these seven files:

Piece.h	contains all class definitions and constants relating to pieces
Piece.cpp	contains your implementations of the above classes
Well.h	contains all class definitions and constants relating to the well
Well.cpp	contains your implementations of the above classes
Game.h	contains all other class definitions and constants
Game.cpp	contains your implementations of the above classes
report.docx, report.doc, or report.txt	your report

Notice that you will *not* turn in chetyris.cpp or the UserInterface files, even though they are part of your project. Your program must build and function correctly with the versions we provided.

Your report should contain the following:

1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, "I chose to define a pure virtual version of the sing() function in my base Piece class because all Pieces sing, but each kind of piece sings in its own way."
2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. "My VaporBomb doesn't work correctly, so for now I just treat it like a normal piece." or "I couldn't get the recursive FoamBomb algorithm to work, so for now it just checks its four immediate neighbors to see if they can be foamed."
3. A list of other design decisions and assumptions you made (e.g., It was ambiguous what to do in situation X, so this is what I decided to do.)

Notice that for this project, your report does *not* have to include a list of test cases.

Good Coding Standards

As always, you're expected to use the best coding standards. This includes, but is not limited to:

1. Making sure to make your code readable and comment it appropriately.
2. Use of the const keyword where appropriate.
3. Use of the virtual keyword when appropriate.
4. Use of pure virtual functions when appropriate.
5. Eliminating duplicate code across classes (defining common code as far up the class hierarchy as possible).

Good luck!