CPSC 217: Assignment #3

Due: Friday, March 27, 2026, at 11:00pm
Weight: 7.5%
Sample Solution Length: Approximately 100 lines, including blank lines and lots of comments, but not the provided code.

Individual Work

All assignments in this course are to be completed individually. Use of large language models, such as ChatGPT, and/or other generative AI systems is prohibited. Students are advised to read the guidelines for avoiding plagiarism located on the course website. Students are also advised that electronic tools may be used to detect plagiarism.

Late Penalty

Late assignments will not be accepted.

Submission Instructions:

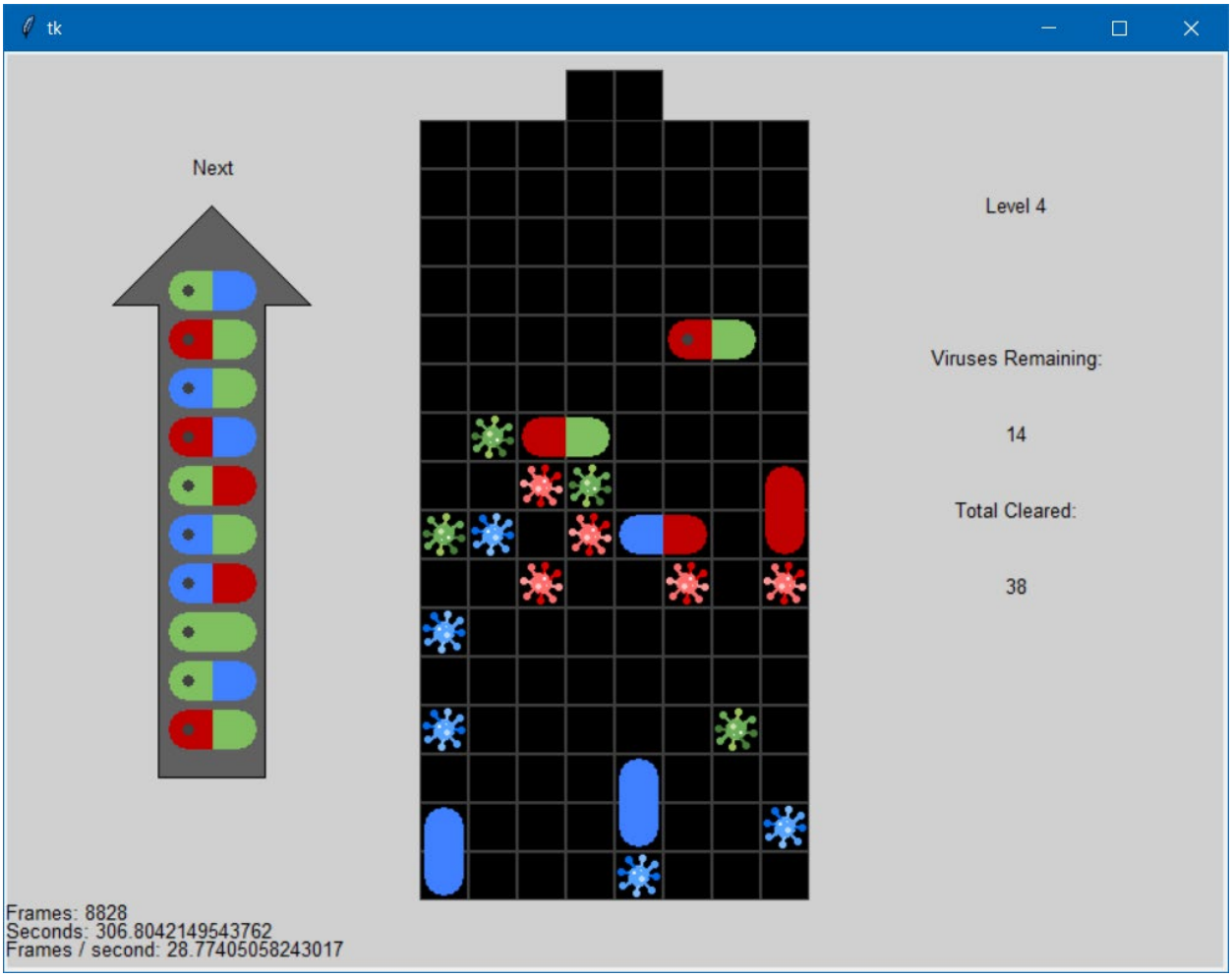
Your program must be submitted electronically to the Assignment 3 drop box in D2L. You don't need to submit SimpleGraphics.py because we already have it (but it won't hurt anything if you include it).

Assignment Description

Dr. Mario was a puzzle game for the Nintendo Entertainment System and Gameboy that was released in 1990. It combined elements of Tetris (1984) and Connect 4 (1974) to provide engaging gameplay. The goal of the game was to eliminate viruses on the game board by forming horizontal or vertical lines of at least 4 elements of the same color. Capsules, which consist of two halves that might differ in color, fell from the top of the game board. These could be moved left or right and rotated clockwise or counterclockwise. Each capsule stopped falling when it reached either a non-empty square or the bottom of the game board. If a line of four or more pieces of the same color was formed when the capsule stopped, then all of the pieces in that line were removed from the board (which would typically be a combination of capsules and viruses). Each level ended when all of the viruses were removed from it, and then a new level was created with gameplay continuing in the same manner.

In this assignment you will create a game that mimics many of the features of Dr. Mario. I have written most of the code needed for the game, including the visual elements and user interface. You will implement several functions that are needed to complete aspects of the game's logic, as described in the following sections.

An image of the completed game is shown below. When you run the provided starter code the game board will be empty and the game will report "Game Over" immediately after it starts because the game logic that you need to create is not yet available.



Game Controls

Capsules can be moved left and right using the arrow keys. The down arrow key can also be used to speed a capsule's descent when desired. Capsules are rotated with z and x, with z rotating the capsule counterclockwise and z rotating it clockwise. The game can be closed by pressing the escape key, or by closing the window with the mouse.

Part 1: Populating the Board

The provided code creates a new, empty board, where every element is initialized to "0" (which is the EMPTY constant in the provided code), but in order to have an engaging game, the board needs to be populated with viruses at random positions that the player will eliminate.

Write a function named `populateBoard` (note the capitalization) that populates the board with viruses with random positions and colors. The board to modify and the number of viruses will be the function's only two parameters, with the board being the first parameter and the number of viruses being the second. Viruses are denoted by the strings "1", "2", and "3" which represent the green, red and blue viruses respectively. They only appear in the bottom 10 rows of the board, as indicated by the VIRUS_ROWS constant in the provided code.

Your function will modify the board provided as a parameter. It does not need to return a result (and any result that is returned will be ignored by the provided code). The location where your `populateBoard` function should be placed within the provided code is clearly marked by a comment. (It's at approximately line 150 in the provided code, or you can search for "Insert your code below this comment" in the provided code).

The provided code will test your `populateBoard` function. It will only report a warning if the `populateBoard` function doesn't exist, but will report an error and quit if it takes the wrong number of parameters. Warnings are generated for any tests that populate the board with the wrong number of viruses. I'd suggest that you refrain from moving past this step until you have the `populateBoard` function working correctly (meaning that it doesn't report any warnings). However, if you are unable to get the `populateBoard` function working successfully, you can move on to the next parts of the assignment without it as the provided code will only issue warnings, not errors, if the populate board fails some of its tests.

Hint: You'll need to ensure that your implementation of `populateBoard` doesn't inadvertently try to place two viruses on top of each other. If this happens, then the board will have fewer viruses than required, and some of the tests performed on `populateBoard` will likely fail.

My implementation of `populateBoard` is approximately 10 lines (ignoring comments).

Part 2: Counting the number of Viruses

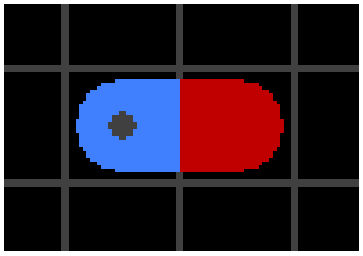
The game needs to be able to count the number of viruses that are currently on the board. This functionality is necessary to determine when a level has been completed. The number of viruses that remain is also displayed as part of the user interface.

Create a function named `numViruses` (note the capitalization). This function will take a game board as its only parameter and return the number of viruses on it as its only result. Viruses are represented by "1", "2" and "3" on the game board. As a result, the number of viruses can be determined by counting all occurrences of those three strings on the game board. Note that capsules on the board are represented by a variety of other strings. As a result, one must check for the strings "1", "2" and "3". One **cannot** simply check for strings that are not "0" (which represents empty spaces).

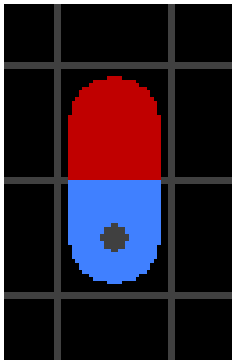
The `numViruses` function is tested by the provided code. It will report warnings for any tests that fail. I'd suggest that you refrain from moving past this step until you have the `numViruses` function working correctly (meaning that it doesn't report any warnings). My implementation of `numViruses` is less than 10 lines (ignoring comments).

Part 3: Allowing the Capsule to Drop

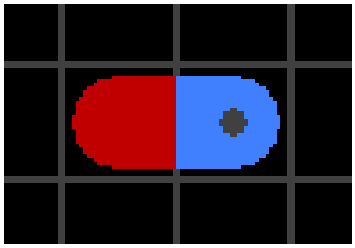
In order for the game to progress, the capsules that start at the top of the board must move down over time. Whether or not a capsule can drop is based on its location on the board, whether the spaces immediately below it are occupied, and its rotation. A capsule can be rotated by 0, 90, 180 or 270 degrees. The position of the capsule, stored as a row and column, is denoted by the gray dot on one of its halves.



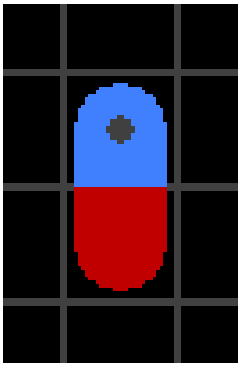
0 degrees



90 degrees



180 degrees



270 degrees

The canDrop function takes 4 parameters, the first of which is the board, and the other three of which are the capsule's row, column and rotation respectively. The canDrop function will return True when the capsule is able to move down, and False otherwise.

You will likely find the following points helpful when creating your function:

- If the capsule has reached the bottom of the board, then it cannot drop.
    - When the piece is rotated by 0, 90 or 180 degrees, the capsule's row will be equal to the bottom row in the board.
    - When the piece is rotated by 270 degrees, the capsule's row will be equal to the second from the bottom row in the board (because the other half of the capsule is below the row reported for the capsule).
- If the capsule is rotated by 90 degrees, and the board location immediately below the capsule's row and column is non-empty, then the capsule cannot drop.
- If the capsule is rotated by 270 degrees, and the board location two rows below the capsule's row and column is non-empty, then the capsule cannot drop.
- If the capsule is rotated by 0 degrees, and either the board location immediately below the capsule's row and column is non-empty, or the board location immediately below and one column to the right of the capsule's column is non-empty, then the capsule cannot drop.
- If the capsule is rotated by 180 degrees, and either the board location immediately below the capsule's row and column is non-empty, or the board location immediately below and one column to the left of the capsule's column is non-empty, then the piece cannot drop.
- In all other cases, the capsule can drop.

Create the canDrop function so that it returns True or False based on the rules described in the previous bullet points. My implementation of canDrop was around 40 lines of code (about half of which were comments).

Hint: My implementation of canDrop contained lots of if statements, but it didn't include any loops. You should, similarly, be able to complete this function without writing any loops.

The canDrop function is tested by the provided code, and warnings are reported for any test cases that fail.

Part 4: Finding Lines

Empty squares on the board are represented by "0" (and there is a constant, EMPTY, representing this value in the provided code that you should use). Filled squares are represented by a string of one or more characters. The first character in a filled square will be "1", "2" or "3", with "1" representing green, "2" representing red and "3" representing blue. If the string is only one character in length, then the square is occupied by a virus of the indicated color. Strings that are more than one character in length represent capsules or spaces that have been marked for deletion because they are part of a line, with the characters that follow the "1", "2" or "3" in the string indicating the orientation of the capsule or the deletion marking. The specifics of what characters are used for these markings isn't important for finding lines – you only need to identify lines based on the first character of each location in the line being the same.

Begin by writing a function named `findHorizontal` (note the capitalization) that finds horizontal lines. It will take the game board as its only parameter. It will return two integers as its result, which are the row and the column of the left edge of any horizontal line of 4 or more elements of the same color on the board. (If there is more than one such line than your function should return the left edge of any one line – it doesn't matter which one). If no such line exists, then the function will return -1 for both the row and the column to indicate such. Your function should not make any changes to the board passed to it as an argument.

Write a second function named `findVertical` (note the capitalization) that finds vertical lines. It will also take the game board as its only parameter, and it will return two integers as its result which are the row and column of the top (smallest row number) of any vertical line of 4 or more elements of the same color on the board. If no such line exists, then the function will return -1 for both the row and the column to indicate such. Your function should not make any changes to the board passed to it as an argument.

My implementations of these functions are approximately 10 lines of code each (ignoring comments). Once you have these functions implemented successfully (and you have completed the other parts of the assignment) you will be able to play the game.

Additional requirements

- Ensure that your name and ID number appear within your submission, either at the top of the file, or at the top of what you added to the provided code.
- Your program must implement the functions described in the previous sections. A solution that works but does not implement the functions described will not receive credit. All of the lines of code that you write should be inside a function (exception for the def lines that begin each function definition).
- Your functions must be named as indicated in this document (including the capitalization).
- You may write additional functions if doing so helps you separate the problem into logical steps, reduces the amount of repeated code, or otherwise results in a better solution to the problem, but there is no requirement to write additional functions.
- Do **not** define one function inside another function.
- Your program must **not** use global variables (except for constant values that are never changed).

- Do **not** modify the provided code, other than to add your functions at the indicated location.
- Do **not** import any modules that have not already been imported by the provided code.
- Your program must use good programming style. This includes things like appropriate variable names, good comments, minimizing the use of magic numbers, etc. **Each function should begin with a comment that describes its purpose, parameters (if any) and return values (if any).**
- Break and continue are generally considered bad form. As a result, you are **NOT** allowed to use them when creating your solution to this assignment. In general, their use can be avoided by using a combination of if statements and writing better conditions on your while loops.

For an A+

When the game board is populated with viruses, it is possible that one color will be much more strongly represented than another. This can be frustrating, particularly if the capsules that are provided to the player aren't similarly proportioned. Improve your populateBoard function so that it guarantees at least 25% of the viruses are green, at least 25% of the viruses are red, and at least 25% of the viruses are blue. While this won't result in boards that are perfectly even, it will prevent the boards from being extremely heavily skewed to any color. Note that these constraints can't be met on boards that contain 1, 2 or 5 viruses. Ensure that your program comes as close as possible to meeting the constraint in these cases, noting that it won't actually reach the 25% threshold for all colors.

Grading

This assignment will be graded on a combination of functionality and style. A base grade will be determined from the general level of functionality of the program (Does it populate the board correctly? Does it count the viruses correctly? Does it correctly determine when a capsule can drop? Does it correctly identify the locations of any lines that have been formed?). The base grade will be recorded as a mark out of 12.

Style will be graded on a subtractive scale from 0 to -3. For example, an assignment which receives a base grade of 12 (A), but has several stylistic problems (such as magic numbers, missing comments, etc.) resulting in a -2 adjustment will receive an overall grade of 10 (B+). Fractional marks will be rounded to the closest integer.

Total Score (Out of 12)	Letter Grade
12	A
11	A-
10	B+
9	B
8	B-
7	C+
6	C
5	C-
4	D+

3	D
0-2	F