

CPSC 217: Assignment #3

Due: Friday November 22, 2019 at 11:55pm

Weight: 7%

Sample Solution Length: Approximately 60 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. 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 or ViperImages.gif because we already have them (but it won't hurt anything if you include it).

Acknowledgements:

The game that you will create in this assignment is based on Viper AGA, originally developed by Nathaniel Myhre and Robert Drennan in 1994 (which I remember playing on a friend's Amiga 4000).

I'd like to thank Ray Zegrer for creating the art assets for the game.

Assignment Description:

There are many variations of Snake – a game where the player controls the direction of travel of a continuously moving snake that grows as the game progresses. In most variations of the game the player loses if the snake crashes into the boundary of the screen, the snake itself, an obstacle (if present), or an opponent snake (if present). The game that you create will follow these established conventions.

The variant of Snake that you will create for this assignment, which we will refer to as Viper, includes between 2 and 4 snakes, 1 controlled by a human player and 1 to 3 controlled by the computer. Our game consists of multiple rounds. In each round the player and AI snakes all try to survive for as long as possible. Each time a snake crashes all of the snakes that have not yet crashed are awarded one point. The round ends when only one un-crashed snake remains. A new round starts shortly after the previous round ends.

The overall goal of the game is to earn the most points. The game ends at the end of the round where one or more players has reached 10 points, with the player with the highest score being declared the winner. A tie can occur when multiple players have the same score when the game ends.

All of the code needed for the computer-controlled snakes and scoring has been provided. Your task is to implement the code needed for the human player by completing the parts of the assignment that follow. Running the provided viper_start.py file (with ViperImages.gif in the same folder/directory) should allow you to select 1 to 3 AI snakes and then show those snakes competing against each other

without allowing the human player to move. Somewhat amusingly, the human player always wins in this version of the game because the human's inability to move means that the human cannot crash.

Part 1: A Moving Dot

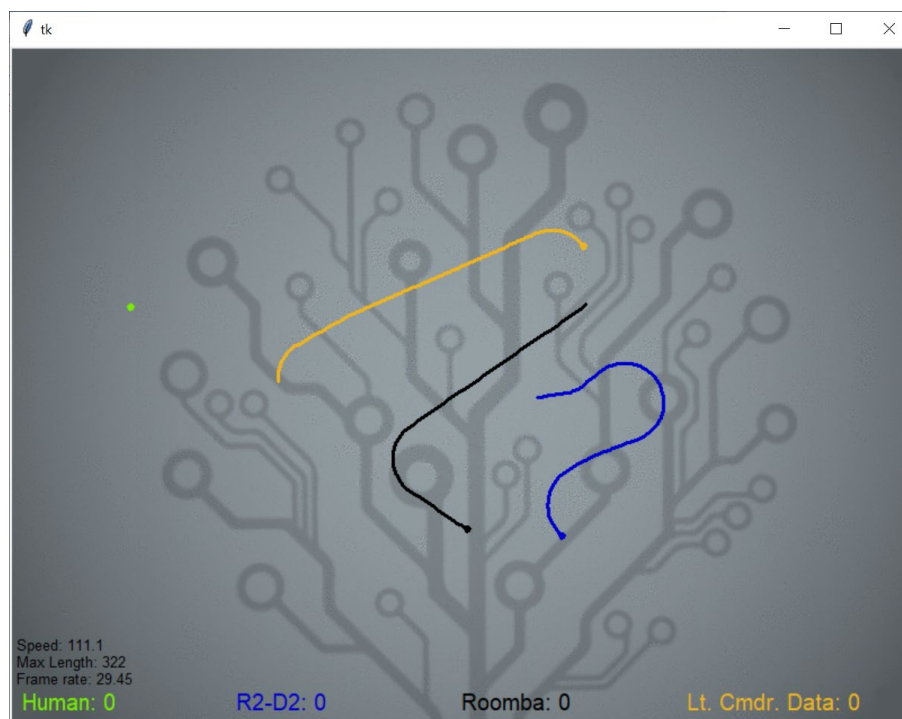
The provided code has assigned initial values to variables named `p1_x` and `p1_y` which are the screen coordinates of the head of the player's snake. These values, which are randomly selected, always start the player in the upper left quadrant of the screen. A small green circle is drawn showing the player's location. However, the provided code does not move the circle. You will add the code needed for this capability in this part of the assignment.

The provided code also assigns a value to a variable named `p1_heading`. This variable holds the direction that the player is travelling (in radians) and is updated when the user presses the left and right arrow keys. Its initial value is set so that the player is moving toward the center of the screen.

Find the location marked "Part 1: A Moving Dot..." in the provided code (it's at approximately line 770). Add the following two lines immediately after that comment (indented the same amount as the comment):

```
p1_x = p1_x + cos(p1_heading) * speed * elapsed
p1_y = p1_y + sin(p1_heading) * speed * elapsed
```

These lines of code use a little bit of trigonometry to update the x and y location of the player's dot (which will be a snake in the future) based on the player's current heading, speed, and the amount of time that has elapsed since the last time that the player's was drawn. (The speed and elapsed variables are also both assigned values by the provided code and are used by both the player's snake and the AI snakes). When you run the program you should be able to "drive" the dot using the left and right arrow keys. Don't move on to the next part of the assignment until you have accomplished this goal.



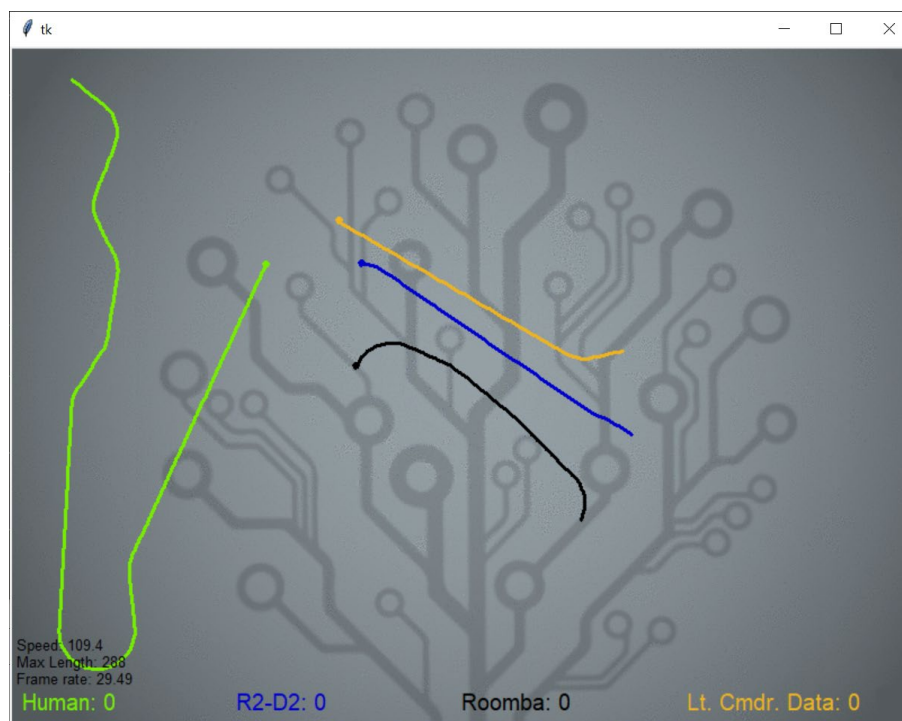
Part 2: A Growing but Permanent Line

A dot is not a particularly good representation of a snake. What we really need is a line, and for this game it needs to be a line that gets longer over time. As a result, we need a data structure that can hold several, even many, x and y locations and can grow as the game progresses. We'll use a list for this purpose.

A list named `p1_queue` has been initialized to the empty list in the provided code, but no x and y locations are ever added to it. The provided code also draws a line connecting all of the points in this list (when multiple points are present in it).

Add each x and y value that you computed in the previous part of the assignment to the `p1_queue` list. You can add them to either the front of the list or the back of the list – just ensure that the x value always comes immediately before the y value that it is paired with.

My solution to Part 2 is only two lines of code (without any comments). You should have a snake that grows over time when you have this part of the assignment completed successfully. However, your snake will quickly become much longer than the AI snakes because it will grow at its head without shrinking at its tail. Do not move on to the next part of the assignment until you have a growing snake.



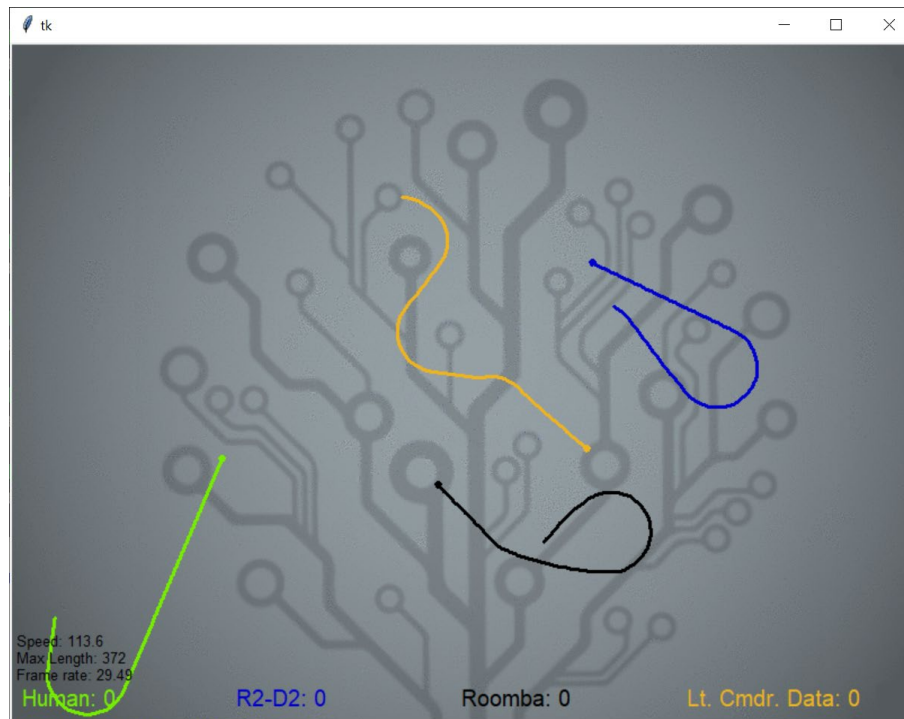
Part 3: A Moving Snake

The provided code includes a variable named `max_length` which is the maximum length that any snake (player or AI) is allowed to have. The value of this variable increases as the game progresses and its value is shown near the lower left corner of the screen. The snake that you created in Part 2 only added points to its list (points were never removed). As a result, the length of the snake quickly exceeded the value stored in the `max_length` variable.

In this part of the assignment you will write a function that takes a list of points as its first parameter, along with a maximum length as its second parameter. If the total distance from point to point along

the path represented by the list is less than or equal to the maximum length then your function should return without modifying the list. If the total distance from point to point along the path represented by the list exceeds the maximum length then sufficient points should be removed from the list so that it no longer exceeds the maximum length. When points are removed from the list the oldest points should be removed first. Whether this involves removing points from the beginning or the end of the list will depend on which end of the list you added the points to in Part 2.

Call the function described in the previous paragraph to restrict the length of your snake to the value contained in the `max_length` variable. Once you have completed this part of the assignment your snake should have the same length as the AI snakes and move over time. My solution to this part of the assignment consisted of one line of code inserted into the main program and a function that was approximately a dozen lines of code (without considering blank lines or comments).



Part 4: Colliding with the walls

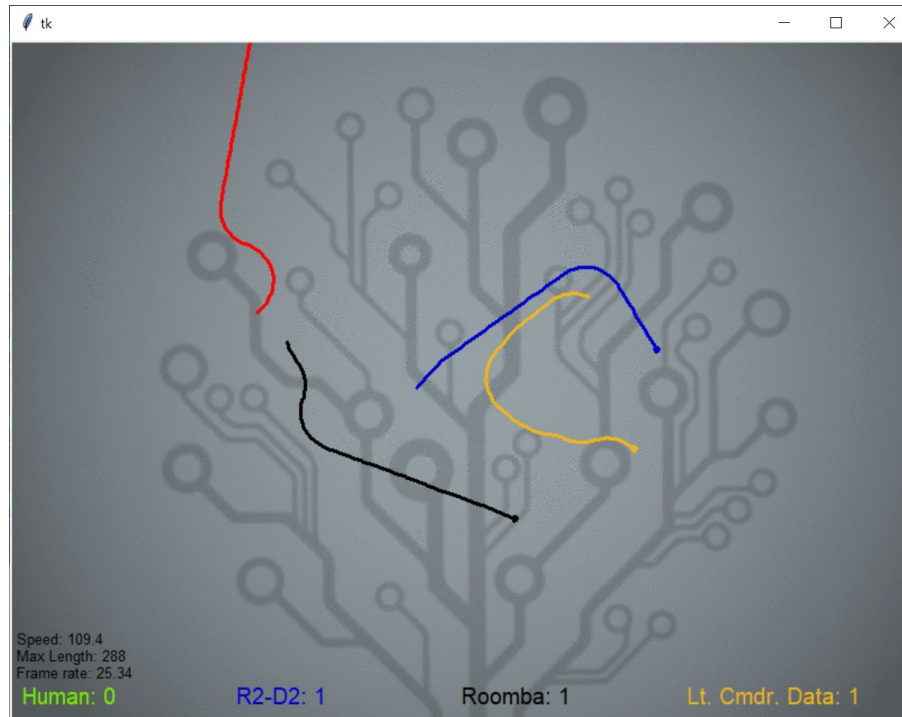
Right now there is no way for the human player to lose because the game doesn't detect collisions between the human player and the edge of the screen, itself, or the AI snakes. We'll address collisions with the edge of the screen in this part of the assignment, and collisions with the other objects in the parts of the assignment that follow.

Update the game so that it sets the `p1_lost` variable to `True` if the head of the snake has gone outside of the screen (meaning that the snake must have crashed into the boundary of the screen). It's ok if the snakes go over the scores or the values displayed near the lower left corner of the screen. Be careful with your conditions – the player should be able to use any position on the screen from (0, 0) up to and including (799, 599).

When the `p1_lost` variable is set to `True` the provided code will draw the player's snake in red instead of green. In addition, the player's snake should stop moving once it has crashed (but the provided code

does not take care of this). Update the code that you wrote in Parts 1, 2 and 3 so that they only compute a new location for the snake's head, append it to the list and truncate the list to the maximum length when the player has not lost the game.

My solution to this part of the assignment was 9 lines of code, though it is possible to write it more compactly if you would like to.



Part 5: Colliding with Yourself

In most variants of Snake the player loses the game if they collide with themselves. This game is no exception, and you will add this functionality in this part of the assignment.

To determine if the player has collided with themselves you need to check and see if the line segment most recently added to the snake intersects with any of the other line segments in the snake, except for its immediate predecessor. (The immediate predecessor must be excluded because the most recently added segment will always intersect with its immediate predecessor because they have a common end point).

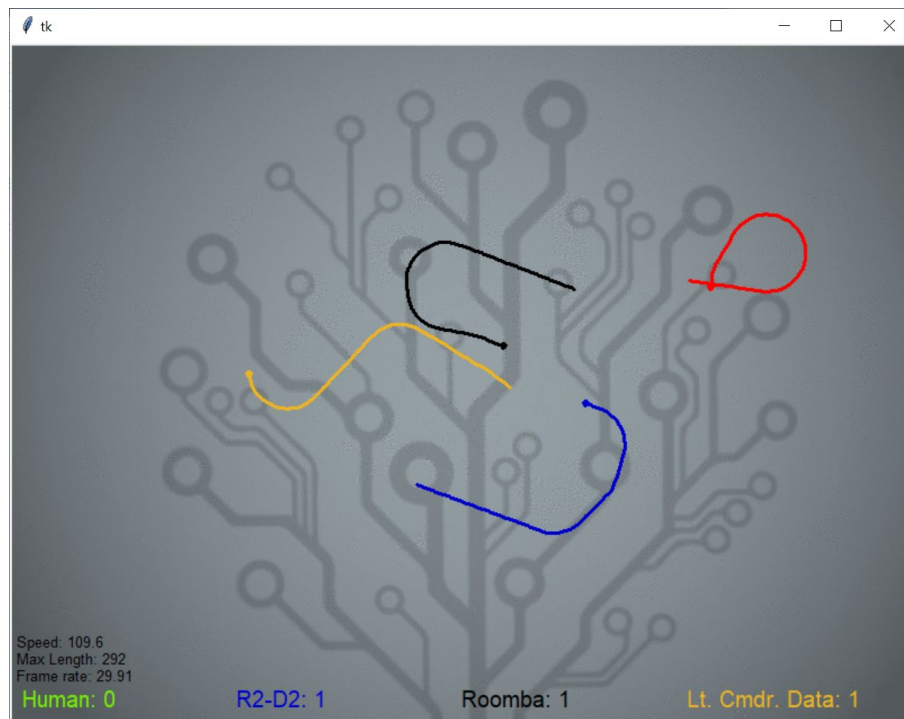
I have provided a function named `doIntersect` which determines whether or not two line segments intersect. This function takes 8 parameters which are the end points of each line segment. The first segment has end points (ax, ay) and (bx, by) (which are the first four parameters to the function) while the second segment has end points (cx, cy) and (dx, dy) (which are the function's last four parameters). The function returns a Boolean value that is True if the segments intersect. Otherwise it returns False.

Write a function that takes a line segment (described by its end points) and a list of points as its parameters. Your function will return True if the line segment intersects with any of the line segments that connect adjacent points in the list of points. Otherwise it will return False. The function that you

write should call the `doIntersect` function described in the previous paragraph. Then you should update the main program so that it uses your function to determine if the player has collided with themselves. If so, set the `p1_lost` variable to `True` so that the player is drawn in red and is no longer able to move.

Hint: You can use list slicing to create a copy of your snake that excludes the most recently added segment (or the two most recently added segments) and then pass this copy of the snake to the function that you write.

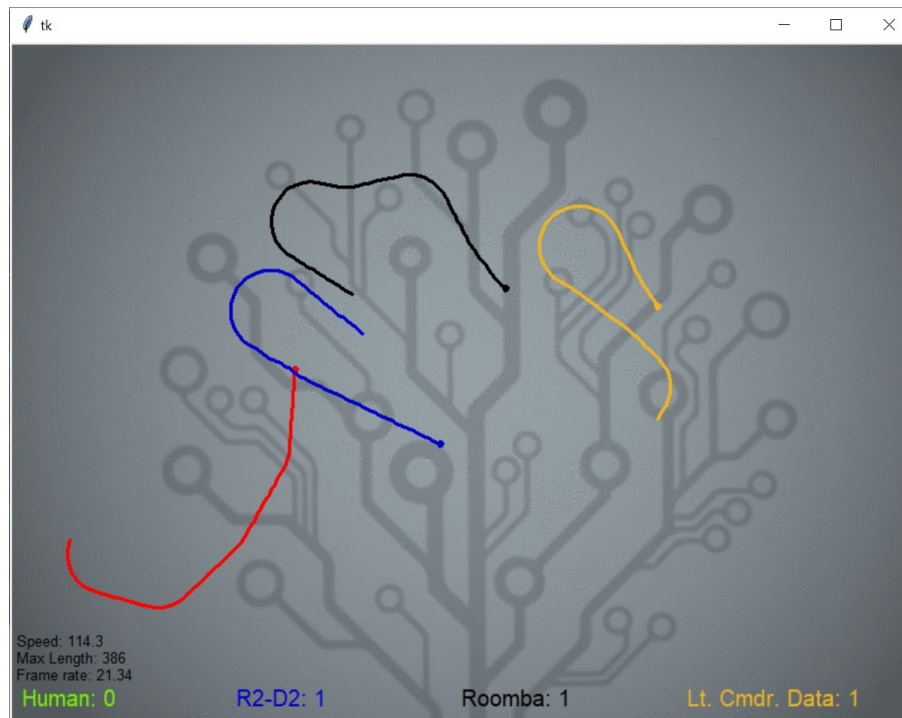
My solution to this part of the assignment was approximately 10 lines of code (not including comments). Approximately half of those lines were in the function that needed to be written while the other half was the code that called the function to determine if the player has collided with themselves.



Part 6: Colliding with Other Snakes

The provided code includes a variable named `e_queues` which is a list of lists. Each list in `e_queues` is a list of points that describes the location of one of the AI snakes. The `e_queues` list will contain 1, 2 or 3 lists of points (depending on the number of AI snakes selected when the game was started) so your solution to this part of the assignment should **not** assume there will be some fixed number of AI snakes.

Use the function that you wrote in the previous part of this assignment to determine whether or not the player has collided with any of the AI snakes. Set `p1_lost` to `True` when such a collision occurs.



Additional Requirements:

- Include a comment at the top of your file that includes your name and student number.
- Do not use map, reduce, partial, lambda or zip anywhere in your code. I have used these functions / keywords in the code that I have provided to improve its performance but we won't be discussing these Python features in this course. (If you'd like to learn more about these topics please consider enrolling in CPSC 449).
- Do not call the fastCollides or closestCollision functions that are included in the provided code. Using one or both of these functions to complete any part of the assignment, other than the bonus for an A+, will result in a mark of 0 for that part of the assignment.
- You must write the functions described in Parts 3 and 5. Do **not** implement this functionality in the main program.
- Include appropriate comments for each of your functions. Specifically, all of your functions should begin with a comment that briefly describes the purpose of the function, along with a description of every parameter and every return value. Functions that do not return a value should be explicitly marked as such.
- Your program must **not** use global variables (except for values that are treated as constants).
- Your program must use good programming style. This includes things like appropriate variable names, good comments, minimizing the use of magic numbers, etc.
- Do **not** define one function inside of another function.
- 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.
- Do **not** modify the provide code outside of the marked areas unless you are attempting to implement the bonus item described below.

Hint: You may find the 3 second countdown frustrating when debugging your program. You can remove this delay by setting COUNTDOWN_DURATION to 0 near the top of the provided code. You can quickly select the number of AI players by pressing 1, 2 or 3 on the keyboard instead of clicking on the appropriate box.

For an A+:

Looking for an A+? Add multiple randomly positioned barriers to the playing field. The random generation needs to be “reasonable” – a player should never start out on top of an obstacle (or so close to it that they can’t possibly avoid it), they should be a reasonable size, etc. Update the AI so that it considers the obstacles when making its decisions about where to go, and ensure that both the player and AI snakes crash when they collide with the obstacles. Do **not** attempt this challenges until you have all 6 parts of the assignment working correctly.

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 draw the player snake? Does the player snake move over time? Does the player snake collide with the screen boundary, itself and the AI snakes?). 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