CPSC 217 Assignment #2: Optical Illusions

Due: Friday, October 23, 2026, at 11:00pm
Weight: 7.25%
Sample Solution Length: Approximately 260 lines (including reasonable comments and the A+ portion of the assignment). Skipping the A+ portion of the assignment reduces its length to approximately 220 lines.

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:

Submit your .py file electronically to the Assignment 2 drop box in D2L. You don't need to submit SimpleGraphics.py or Smiley_248.png – we already have them, but it won't cause any problems if you include them.

Acknowledgements:

The general idea for this assignment was developed by Faan Tone Liu at the University of Denver, as was the suggestion to include the gradient illusion, Hering's illusion and the hidden image illusion in the collection of images that are displayed.

The bulging squares and animated dots illusions are variations of illusions posted on Reddit by user Ebonystealth. It's not clear whether these are original illusions created by that user or if they are reproductions of illusions developed by someone else.

Description

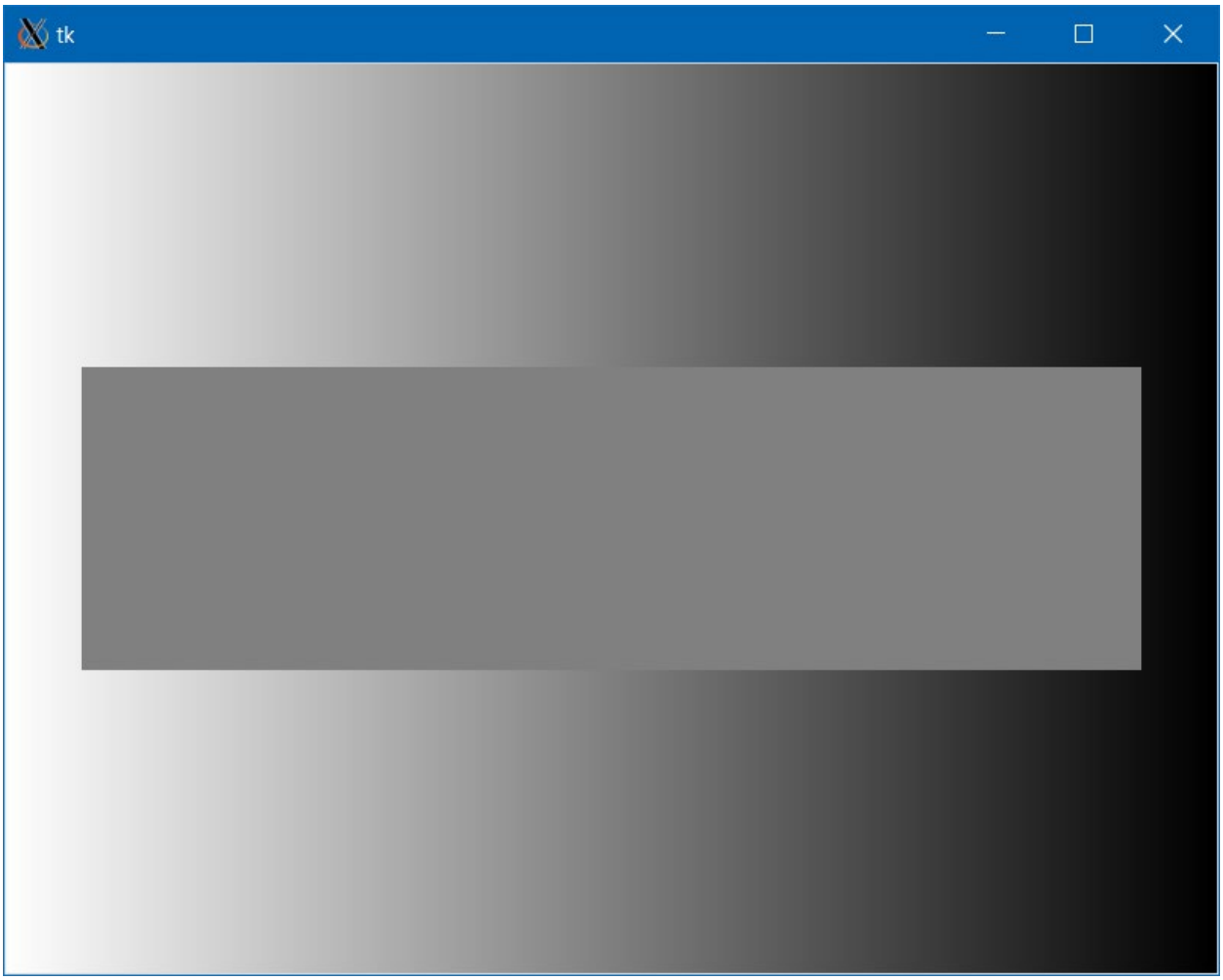
An optical illusion is a visual perception that differs from what is actually present. In this assignment, you will create a program that draws images that contain optical illusions of increasing complexity. Your program will allow the user to select which illusion will be displayed, and it will continue to display illusions until the user chooses to quit the program.

Part 1: Gradient Illusion

The gradient illusion consists of a smooth color gradient from white at the left edge of the window to black on the right edge of the window and a gray rectangle that occupies a little less than a third of the window. While all of the pixels in the rectangle are a middle gray color (128, 128, 128), most people perceive that the left side of the rectangle is darker than the right side.

Requirements for the gradient illusion:

- Resize the window so that it is 800 pixels by 600 pixels by calling the `resize` function. (This won't be necessary if this illusion is drawn first, but it will be necessary if this illusion is selected after another illusion that uses a different window size).
- Clear the window so that any previous illusion is removed before drawing this one by calling the clear function.
- Draw the color gradient using vertical lines.
- The rectangle should be gray (128, 128, 128), 700 pixels wide, 200 pixels tall, and centered in the window.



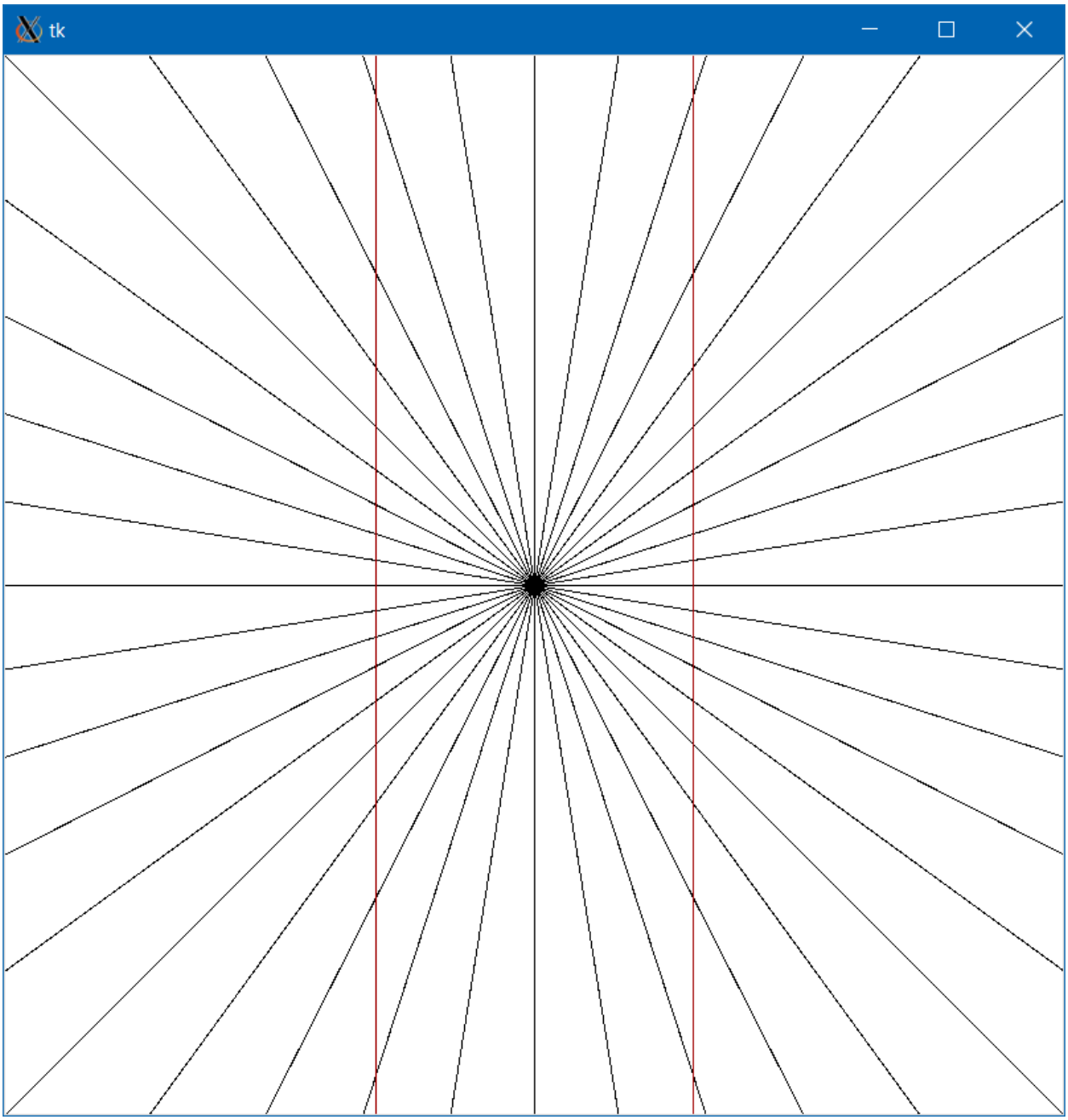
My solution to the gradient illusion was less than 10 lines of code (without any comments).

Part 2: Hering's Illusion

Hering's illusion consists of 40 straight black lines radiating from the center of the window, and two vertical red lines. While the vertical lines are straight, many people perceive that those lines are curved, with them being closer together at the top and bottom of the window, and further apart near its middle.

Requirements for Hering's illusion:

- Resize the window so that it is 800 pixels by 800 pixels by calling the `resize` function.
- Clear the window so that any previous illusion is removed and the background is white by calling the clear function.
- Draw 40 lines radiating from the center of the window using a loop. These can either be drawn with the `line` function and a small amount of trigonometry, or they can be drawn without any trigonometry by using the `pieSlice` function that SimpleGraphics provides.
- The vertical lines should be a reasonably bright red color. I used (160, 0, 0). You don't need to use exactly this color, but it should be something similar.
- The vertical lines should be drawn at x = 280 and x = 520.



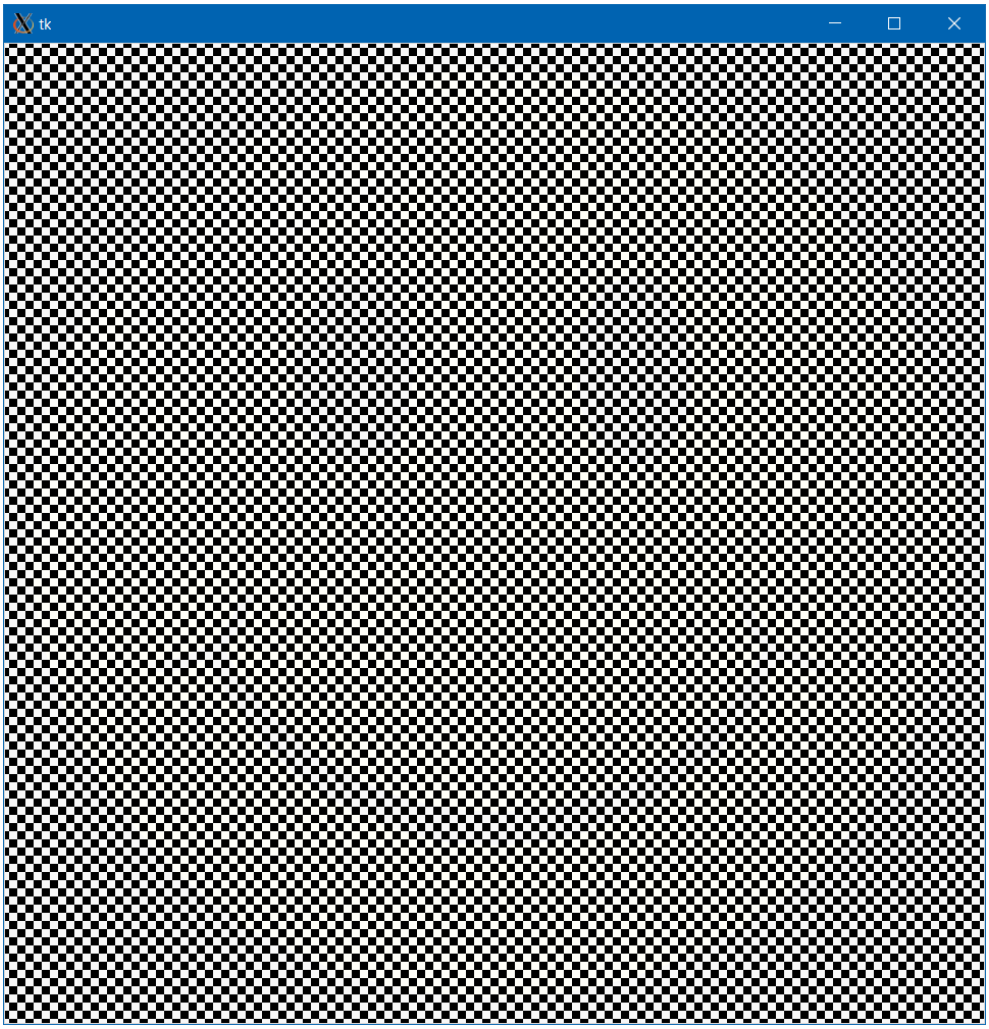
My solution to Hering's illusion was approximately 15 lines of code (without any comments).

Part 3: Hidden Image Illusion

This illusion is constructed by drawing a faint background image, and then a checkerboard of black squares over that image. Most people can't see the background image when the image is still, but are able to see the background image when it is moving because the window containing the illusion is being dragged or a document containing the image is being scrolled.

Requirements for the hidden image illusion:

- Resize the window so that it is 960 pixels by 960 pixels by calling the `resize` function.
- Clear the window so that any previous illusion is removed.
- Load and draw the background image which is stored in Smiley_248.png at (0, 0) in the graphics coordinate system.
- Draw the black squares using nested loops. Each black square is 8 pixels by 8 pixels. Draw the squares so that they are offset by 4 pixels from the edge of the window resulting in a collection of half squares around the perimeter of the window.



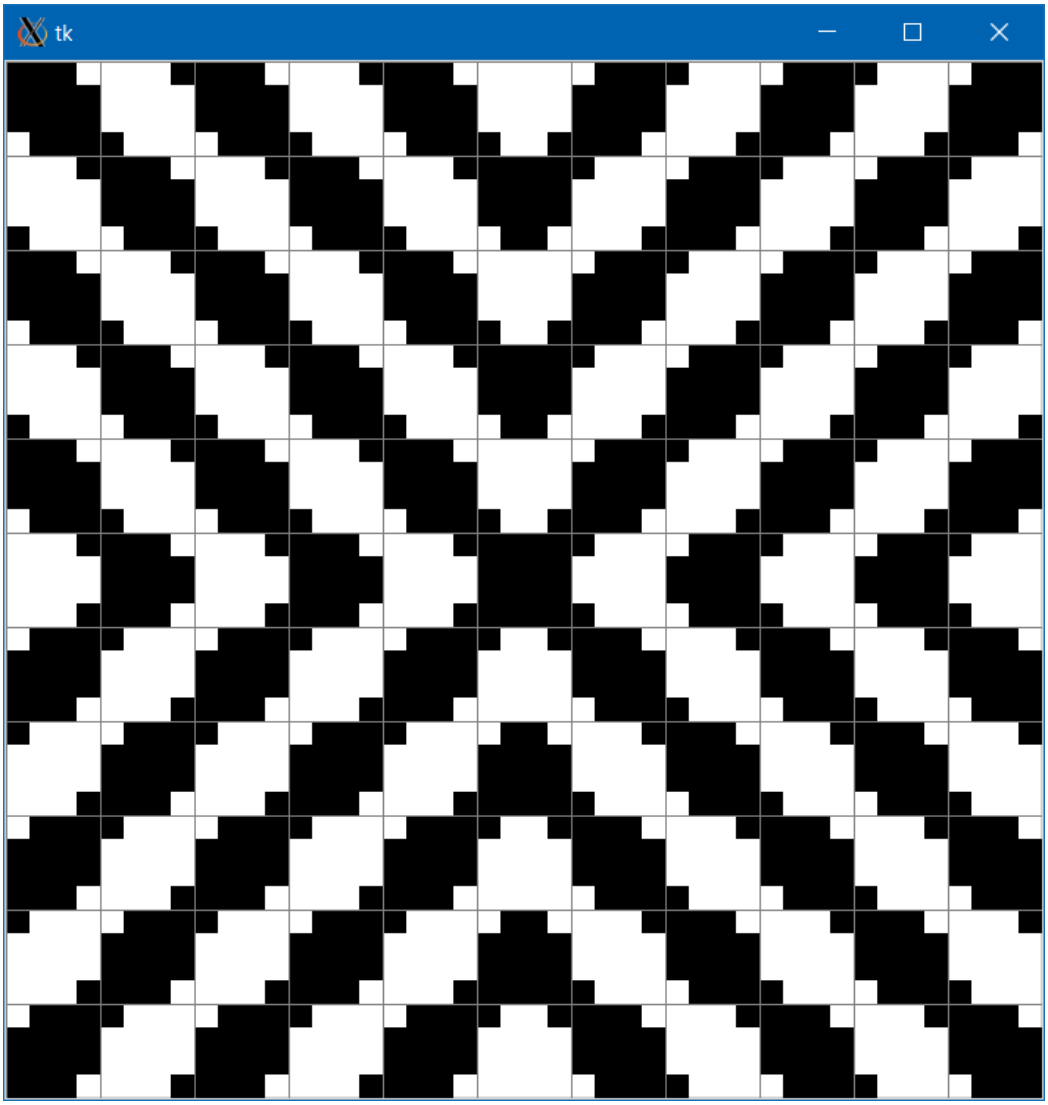
My solution to the hidden image illusion was less than 20 lines of code (without any comments).

Part 4: Bulging Squares

All of the lines in this illusion are either parallel or perpendicular, but most people perceive that the lines are slightly curved and that the center square in the pattern is bulging upward.

Requirements for the Bulging Squares Illusion:

- Resize the window so that it is 704 pixels by 704 pixels by calling the `resize` function.
- Clear the window so that any previous illusion is removed.
- Nested loops must be used to construct this illusion.
- Nine different cases need to be considered: Four quadrants (upper left, upper right, lower left and lower right), four lines (top, left, right and bottom), and the central black square. Whether you write one set of nested loops and identify the correct case inside the body of the loop, or you write several sets of loops to handle the regions described previously is up to you – both approaches will work.
- A grid of gray lines (128, 128, 128) must be drawn in addition to the black and white pattern shown below. The grid of gray lines separates the board into squares that are 64 pixels by 64 pixels



My solution to the bulging squares illusion was over 100 lines of code (including decent comments), but shorter solutions are certainly possible.

Additional Requirements:

Your program should begin by displaying a menu that allows the user to select the illusion that will be drawn. The user will enter 1 to draw the gradient illusion, 2 to draw Hering's illusion, 3 to draw the hidden image illusion, and 4 to draw the bulging squares illusion. If you complete the A+ portion of the assignment, the user will enter 5 to draw the animated dots illusion.

Your program will continue reading input from the user and drawing illusions until the user enters Q to quit. If the user provides invalid input, then your program should display an appropriate error message before displaying the menu again, reading additional input, and displaying additional illusions. A sample of what the menu could look like is shown below.

```
What illusion would you like drawn?
  1) Gradient illusion
  2) Hering's illusion
  3) Hidden image illusion
  4) Bulging square illusion
  5) Animated dots illusion
  Q) Quit
```

In addition:

- All of the graphical output produced by your program must be drawn by calling functions in the SimpleGraphics module that can be downloaded from the course website.
- Do not import any modules other than SimpleGraphics, math (which may be needed to draw Hering's illusion), and time (which may be needed if you attempt the A+ portion of the assignment).
- An appropriate prompt / menu should be displayed before each input value is read. This prompt should include the name of each illusion and the number associated with it, as well as the fact that Q can be used to quit the program.
- Your solution must make appropriate use of loops. It is **not** acceptable to use long sequences of similar statements to draw the illusions. For example, the radiating lines in Hering's illusion must be drawn with a loop, as must the squares in the bulging squares illusion.
- Your program must use good programming style. This includes things like appropriate variable names and good comments.
- Minimize the use of magic numbers. In particular, values stated for each illusion such as the number of rows, the width of a line, size of a square, or size of the window (among others) should be named constants (variables with a name that is in all capital letters) so that names are used throughout the code instead of using the values directly. Changing the size of the window for a particular illusion should, ideally, only require you to change the values of a few constants near the beginning of your program.
- Close the graphics window when the program quits by calling the `close` function.
- Do not use break or continue anywhere in your program.

- Do not use lists, dictionaries, files or exceptions anywhere in your program. (These topics will be covered later in the course, but they are not required to solve this problem.)
- You are not required to define any functions when solving this assignment, but you may do so if you'd like to. Ensure that any functions that you create are thoroughly documented including a comment ahead of each function that describes its purpose, parameters (if any) and return value (if any).
- Your program should begin with a comment that includes your name, student number, and a brief description of the program.

Hints:

The fill color can be set to 'blank' by calling `setFill(None)`. You may find this particularly helpful if you want to draw Hering's illusion using the `pieSlice` function to avoid having to do any trigonometry.

Make sure that the illusions draw correctly no matter what order the user requests them.

You may find it helpful to draw primitives that extend outside the window; my solutions to Hering's illusion and the hidden image illusion both made use of this technique. SimpleGraphics will only draw the portion of the primitive that lies within the window.

Grading:

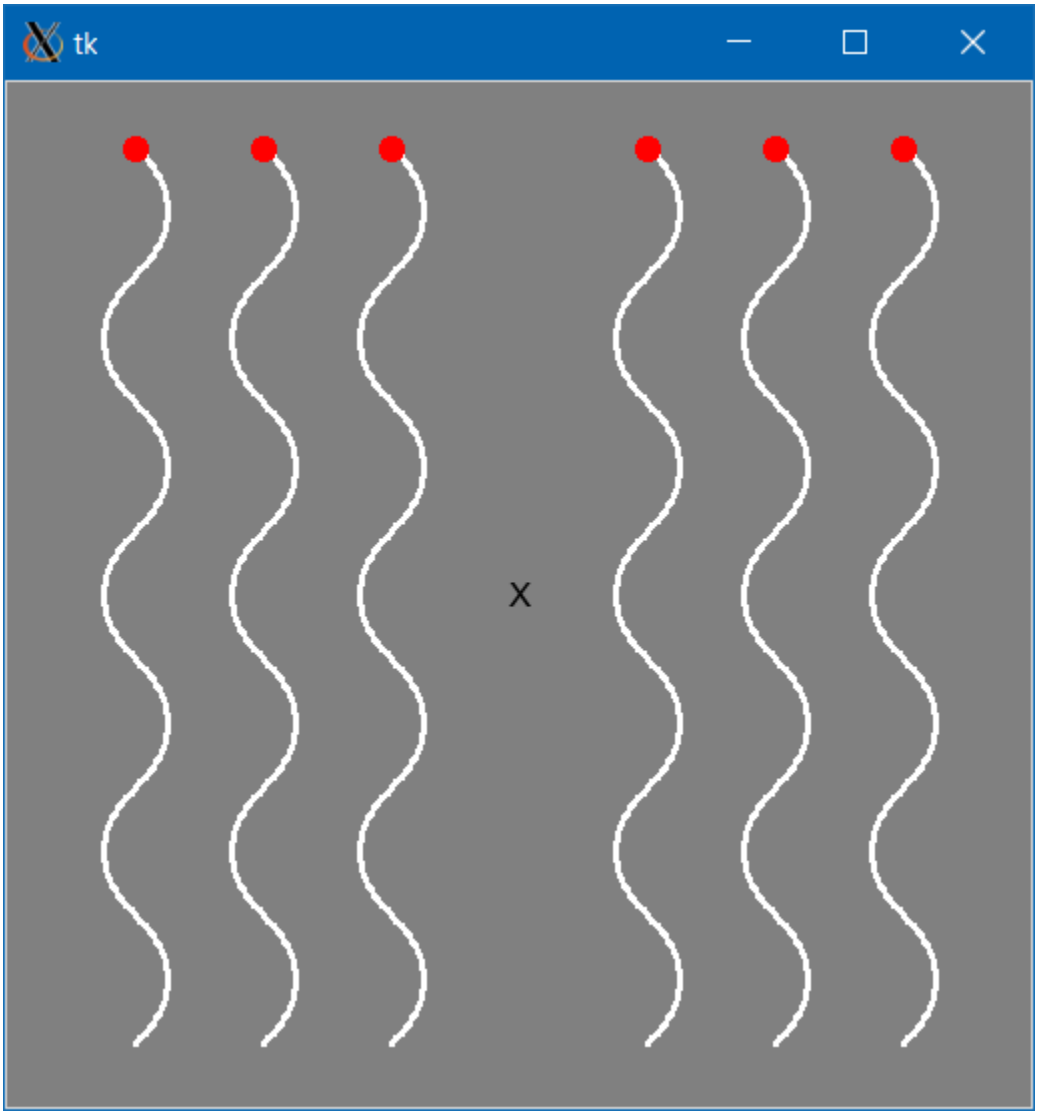
This assignment will be graded on a combination of functionality and style. A base grade will be determined from the general level of functionality of the program (Are each of the illusions drawn correctly? Does your program allow multiple illusions to be drawn? Does it correctly handle invalid input? Etc.). 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

For an A+:

Consider the image below. It contains a black X in its middle, six red dots, and six white 'squiggles', with three on each side of the X drawn in a window that is 512 pixels by 512 pixels. The image itself is not an optical illusion. However, when the red dots are animated so that they move in a straight line vertically from the top of each squiggle to its bottom an interesting effect is achieved: If one focuses on the red dots, they appear to move in straight vertical lines, but if one focuses on the black X the dots appear to move along (or at least close to) the paths indicated by the squiggles.



Requirements for the animated dots illusion:

- Resize the window so that it is 512 pixels by 512 pixels by calling the `resize` function.
- Clear the window so that any previous illusion is removed.
- Draw the squiggles with a line width of 2. They can be drawn with the curve primitive. I used the following coordinates for the left-most squiggle: (64, 32, 96, 64, 32, 128, 96, 192, 32, 256, 96, 320, 32, 384, 96, 448, 64, 480). Use a loop to draw the 6 squiggles, not 6 separate calls to `curve`.

- The background color for the window is (128, 128, 128).
- When the user selects this illusion, animate all six dots so that they move straight down from the top of the squiggle to its bottom (in a straight line, not following the curves), and then return to the top of the squiggle, again following a straight line. It should take between 1.0 and 1.5 seconds for the dot to move from one end of the squiggle to the other. Your dots should complete three round trips and then allow the user to select another illusion.
- You may want to import the time module and make use of its sleep function to slow down the movement of the red dots. If you need to speed up the movement of the red dots, you can do so by having them move down or up by more than one pixel at a time.
- Every time you want to draw the dots in a new position, you'll also need to draw the background and squiggles, but you won't want to see that process happening because it will result in a lot of flickering. To remove the flicker, turn off graphics updates by calling `setAutoUpdate(False)`, remove all of the existing graphics primitives by calling `clear()`, draw all of the graphical elements for the current frame of the animation, and then call the `update()` function. This will cause SimpleGraphics to display the updated image all at once, instead of displaying each element as its function is drawn, resulting in a smooth, flicker-free animation. You might want to turn automatic updates back on at the end of the animation by calling `setAutoUpdate(True)` so that your other illusions behave as expected when they are drawn after the animated dots illusion.

My implementation of the animated dots illusion was approximately 35 lines of code.