Program 3: Graphics Primitives Library ...

Due date: Friday, December 4, 11:59pm.

Purpose

To write and test a collection of functions.

Programming Environment. This is a team programming assignment. Feel free to organize in the teams of four (one team will be of size five) as you prefer.

Programming Style. All submitted code must adhere to the programming style described in detail at

http://users.csc.calpoly.edu/∼cstaley/General/CStyle.htm

When graded, the programs will be checked for style. Any stylistic violations are subject to a 10% penalty. Significant stylistic violations, epsecially those that make grading harder, may yield stricter penalties. Also note the the Lab 2 requirement for the content of the header comment in each file you submit applies **to each assignment** (lab, programming assignment, homework) in this course.

Assignment Description

For this assignment, you will build **your own implementation** of the graphics primitives from the `image.h` header file.

You have used these graphics primitives in Labs 6 and 7 to build more complex functions and programs. This time, you will work on implementing the "back end" of the graphics software you have developed in those labs.

The `image.h` header file contains declarations of **fifteen (15)** functions. Some of these functions (`getMin()`, `getMax()`, `getAngle()`) play supporting roles withing the `image.h` library, but most other functions are responsible for either changing the contents of an `image` array, or for outputing the contents of an `image` array to standard output.

The following section reprises the description of the functions in the `image.h` library. The description is enhanced with some implementation comments when they are needed.

Instructor's Graphics Library `image.h`

The image primitives library can be used to produce Portable Pixel Map files of predefined size. The size of the image is defined as a pair of symbolic constants, `HEIGHT` and `WIDTH` in the `image.h` file.

An *graphics primitive* is a C function that draws a single simple shape. For example, `image.h` header file declares functions that draw a single point, a line, a circle, a rectangle and an ellipse, among others.

Most functions declared in the `image.h` header file take as input (via the call-by-reference mechanism) a 3dimensional array representing the color assignment to `WIDTH`× `HEIGHT` pixels. This array is similar to the ones you used in your Lab 5 assignments.

Each graphics primitive works by changing the colors of a specific set of points (determined by the type of the function) from its input image array. For example, a function that draws a line between two points, computes which pixels along the way lie on the line between two points specified as function parameters, and colors these pixels according to the color, another parameter of the function.

For simplicity, all graphics primitives take as an input parameter an `char` array of size 3, that represents a triple of RGB color intensities. Each graphics primitive uses only one color to "paint" the image, namely, the color passed to it as a parameter using such an array.

The `image.h` file contains declarations of a number of auxillary functions. These functions may not be needed for you, but they are used by other functions in the `image.h` library.

image.h library description

`image.h` file declares a collection of symbolic constants and a list of functions loosely partitioned into three categories: graphics primitives, work with PPM format, and auxillary functions. All of these are described below.

image.h Symbolic Constants

The following symbolic constants are defined in the `image.h` file:

```
#define COLORS 3       /* number of colors components in an RGB color */
#define HEIGHT 400     /* height of the image produced                 */
#define WIDTH  600     /* width of the image produced                  */
#define PI 3.14159265  /* PI                                           */
```

At present, the `image.h` library is designed to produce images of size 600×400. In general, this can be modified by changing the values of `HEIGHT` and `WIDTH` constants.

Additionally, the `image.h` file declares the following symbolic constants designed to represent RBG colors:

```
#define BLACK 0,0,0
#define WHITE 255, 255, 255
#define RED 255, 0, 0
#define DARK_RED 128,0,0
#define BLUE 0,0,255
#define YELLOW 255,255,0
#define BROWN 140,70,20
#define CYAN 0,255,255
#define ORANGE 255, 128,0
#define GREEN 0,255,0
#define DARK_GREEN 0,128,0
#define MAGENTA 255,0,255
#define GRAY 128, 128, 128
#define LIGHT_GRAY 196, 196, 196
#define DARK_GRAY 64, 64, 64
```

Note: Some instructor's examples use different colors. Similarly, you are NOT restriced in your selection of colors for these assignments. The colors above are specified for your convenience.

image.h Functions

The following functions are declared in the `image.h` header file. For each function we provide its declaration, explain the meaning of all arguments and specify what the function does.

PPM Image functions.

```
void drawImage(char image[][WIDTH][COLORS]);
void printHeader(int w, int h);
```

Graphics Primitives.

```
void blankImage(char image[][WIDTH][COLORS], char color[]);
void putPixel(char image[][WIDTH][COLORS], int i, int j, char color[]);
void putRectangle(char image[][WIDTH][COLORS], int topY, int topX,
                   int height, int width, char color[]);
void putCircle(char image[][WIDTH][COLORS], int centerY, int centerX,
                   int radius, char color[]);
void putPie(char image[][WIDTH][COLORS], int centerY, int centerX,
                   int radius, int start, int end, char color[]);
void putRing(char image[][WIDTH][COLORS], int centerY, int centerX,
                   int radius1, int radius2, char color[]);
void putArc(char image[][WIDTH][COLORS], int centerY, int centerX,
                   int radius1, int radius2, int start, int end, char color[]);
void putEllipse(char image[][WIDTH][COLORS], int centerY, int centerX,
                   int radius1, int radius2, char color[]);
void putLine(char image[][WIDTH][COLORS],
             int fromY, int fromX, int toY, int toX,
     char color[]);
```

Auxillary functions.

```
void setColor(char color[], char r, char g, char b);
int getMin(int i, int j);
int getMax(int i, int j);
float getAngle(int i, int j);
```

These functions are briefly summarized in the table below:

Function name	Meaning
`drawImage()`	print contents of an image array
`printHeader()`	output the PPM file header info
`blankImage()`	fill image with chosen color
`outPixel()`	draw a single pixel of the image array
`putRectangle()`	draw a rectangle with given coordinates
`putCircle()`	draw a circle with given center and radius
`putPie()`	draw a segment of a circle
`putRing()`	draw a ring with given center and radii
`putArc()`	draw a segment of a ring
`putEllipse()`	draw an ellipse with a given center and radii
`putLine()`	draw a line connecting two points
`setColor()`	set the values in the input color array
`getMin()`	return the smaller of two numbers
`getMax()`	return the larger of two numbers
`getAngle()`	return an angle (in degrees) given a point in space

Detailed descriptions of the functions are below.

`void drawImage(char image[][WIDTH][COLORS])`. This function takes as input the image array, and outputs to *standard otput* the PPM image file representing the image in the array.

Implementation note. Remember the order in which pixels need to be output in a PPM file. Also, do not forget to call `printHeader()`.

`void printHeader(int w, int h)`. This function takes as input two numbers specifying the size of a PPM image, and prints to the standard output the first three lines of the binary PPM image, i.e., the magic number (`"P6"`), the width and height of the image, and the range of color intensity values (255).

PLEASE NOTE: this function **is called from** `drawImage()`, so there is no need for you to use it in your code. Its description is given for the sake of completeness.

`int getMin(int i, int j)`. This function returns the smaller of the two of its input parameters.

`int getMax(int i, int j)`. This function returns the larger of the two of its input parameters.

`float getAngle(int i, int j)`. This function returns the angle between the $x = 0$ line and the line from $(0, 0)$ to the point (j, i). The angle is measured in degrees and should have the value between 0 and 360.

Implementation notes. First find the angle in radians. Then convert it to degrees.

To find the angle in radians, you need to be careful because depending on the quadrant your point (j, i) is in, the computation may be different. `math.h` has `asin()`, `acos()` and `atan()` functions that may be helpful. You need to understand what values they return and what values you need.

`void setColor(char color[], char r, char g, char b).` This function takes as input the "color" array and three color intensities for the red (`char r`), green (`char g`) and blue (`char b`) components. The function assigns the appropriate color intensity values to the elements of the `color` array.

This function mostly exists for your (and my) convenience. In C programs that use the library, I can now write statements of the sort:

```
char black[COLORS], red[COLORS], yellow[COLORS];

setColor(black, 0,0,0);
setColor(red, 255, 0, 0);
setColor(yellow, 255,255,0);
```

or, even:

```
char black[COLORS], red[COLORS], yellow[COLORS];

setColor(black, BLACK);
setColor(red, RED);
setColor(yellow, YELLOW);
```

The color arrays can then be used when calling the remaining functions from the library.

Implementation Note. Straightforward.

`void blankImage(char image[][WIDTH][COLORS], char color[]).` This function takes as input the image array and the color array. It sets the color of each pixel in the image array to be the color represented by the `color` array.

For example, the following C program:

```
#include "image.h"
int main() {
   char image[HEIGHT][WIDTH][COLORS];
   char color[COLORS];

   setColor(color, 255,0,0);
   blankImage(image,color);
   drawImage(image);
   return 0;
}
```

outputs an all-red PPM image.

Implementation Note. Also straightforward.

`void putPixel(char image[][WIDTH][COLORS], int i, int j, char color[])`
This function takes as input the following parameters:

`char image[][WIDTH][COLORS]`	image array
`int i, int j`	the row and the column of a pixel
`char color[]`	color array

It sets the color of pixel `(j,i)` in the `image` array to be the one represented by the `color` array.

Note: in all functions whenever coordinates of a pixel are presented, the first argument will be the **row** and the second argument will be the **column** of the pixel. (at the same time, standard mathematical notation is (column, row)).

For example, the following C program:

```
#include "image.h"
int main() {
   char image[HEIGHT][WIDTH][COLORS];
   char black[COLORS], color[COLORS];

   setColor(black, 0,0,0);
   setColor(color, 255,0,0);
   blankImage(image,black);
   putPixel(image, 200,300,color);
   drawImage(image);
   return 0;
}
```

outputs a PPM image of a single red pixel at coordinates (300,200) on the all-black background.

Implementation Note. As long as you heed the note above, this is the easiest function to implement.

`void putRectangle(char image[][WIDTH][COLORS], int topY, int topX, int height, int width, char color[])`. This function takes the following parameters:

`char image[][WIDTH][COLORS]`	image array
`int topY, int topX`	the top left corner of the rectangle
`int height, int width`	the height and the width of the rectangle
`char color[]`	color of the rectangle

This function changes the pixels of the `image` array to show a rectangle with top right corner at `(topX,topY)` (row `topY`, column `topX`), of witdth `width` and height `height` using the color specified by the `color` array.

For example, the following program:

```
#include "image.h"
int main() {
   char image[HEIGHT][WIDTH][COLORS];
   char black[COLORS], color[COLORS];

   setColor(black, 0,0,0);
   setColor(color, 255,0,0);
   blankImage(image,black);
```

```
    putRectangle(image, 50, 50, 200,300,color);
    drawImage(image);
    return 0;
}
```

outputs a PPM image showing a red rectangle on black background.

Implementation Note. Make sure that the rectangle starts at the right location and has correct dimensions.

`void putCircle(char image[][WIDTH][COLORS], int centerY, int centerX, int radius, char color[]).` This function takes as input the following parameters:

`char image[][WIDTH][COLORS]`	image array
`int centerY, int centerX`	center of the circle
`int radius`	radius of the circle
`char color[]`	color of the circle

This function changes the pixels of the `image` array to show a solid circle of radius `radius` centered at `(centerX, centerY)` (column `centerX`, row `centerY`). using the color specified by the `color` array.

For example, the following program:

```
#include "image.h"
int main() {
    char image[HEIGHT][WIDTH][COLORS];
    char white[COLORS], color[COLORS];

    setColor(white, 255,255,255);
    setColor(color, 255,0,0);
    blankImage(image,white);
    putCircle(image, 200,300,100, color);
    drawImage(image);
    return 0;
}
```

outputs a PPM image similar to the flag of Japan (a red circle in the center of a white field).

Implementation Note. The equation of a (filled) circle is

$$(x - x_0)^2 + (y - y_0)^2 \leq r^2,$$

where (x_0, y_0) are the coordinates of the center and r is the radius. Note the $\leq$ sign - the circle circumfence itself needs to be colored[1]

`void putPie(char image[][WIDTH][COLORS], int centerY, int centerX, int radius, int start, int end, char color[]).` This function takes as input the following parameters:

[1] Using $<$ instead of $\leq$ removes four points at the N,S,E, and W ends of the circle from the image of the circle.

`char image[][WIDTH][COLORS]`	image array
`int centerY, int centerX`	center of the segment's circle
`int radius`	radius of the segment's circle
`int start`	the starting angle of the segment (in degrees)
`int end`	the ending angle of the segment (in degrees)
`char color[]`	color of the circle

This function changes the pixels of the `image` array to show a solid circular segment (a pie slice (-:) with the radius `radius` centered at `(centerX, centerY)` (column `centerX`, row `centerY`) and located between the `start` and `end` angles using the color specified by the `color` array.

The `start` and `end` angles are expected to be between 0 and 360, and it is expected that `start` <= `end`.

For example, the following program:

```
#include "image.h"
int main() {
   char image[HEIGHT][WIDTH][COLORS];
   char white[COLORS], color[COLORS];

   setColor(white, 255,255,255);
   setColor(color, 255,0,0);
   blankImage(image,white);
   putPie(image, 200,300,100,0,90, color);
   drawImage(image);
   return 0;
}
```

outputs one (**South-East**) quarter of a circle.

Note: Not every possible circular segment can be displayed using this function — only those, that *do not cross the positive side of the X axis.* This is mostly due to the fact that the function works only for `start`≤ `end`. Interpretation of situations when `end`< `start` is somewhat difficult, and may require extra help from users, so it is left outside the scope of the `image.h` library.

Implementation Notes. Use the circle equation from above to determine if a point is inside the circle. To determine if it is inside the "pie slice" you need to compute the angle of the point (in degrees), given its x and y coordinates. Implement `getAngle()` correctly and use it!

`void putRing(char image[][WIDTH][COLORS], int centerY, int centerX, int radius1, int radius2, char color[]).` This function takes as input the following parameters:

`char image[][WIDTH][COLORS]`	image array
`int centerY, int centerX`	center of the ring
`int radius1`	outer radius of the ring
`int radius2`	inner (smaller) radius of the ring
`char color[]`	color of the ring

This function changes the pixels of the `image` array to show a ring with outer radius `radius1` and inner radius `radius2` centered at `(centerX, centerY)` (column `centerX`, row `centerY`). using the color specified by the `color` array.

For example, the following program:

```
#include "image.h"
int main() {
   char image[HEIGHT][WIDTH][COLORS];
   char white[COLORS], color[COLORS];

   setColor(white, 255,255,255);
   setColor(color, 255,0,0);
   blankImage(image,white);
   putRing(image, 200,300,100, 60, color);
   drawImage(image);
   return 0;
}
```

outputs a PPM image with a red ring of width 40 (100 - 60) on the white background.

Implementation Note. Equation of a ring is

$$r \leq (x - x_0)^2 + (y - y_0)^2 \leq R^2,$$

where (x_0, y_0) is the center, r is the smaller radius and R is the larger radius. Please remember that you are not allowed to just paint two circles, one on top of another: the inside of the ring **must remain unchanged** after the ring is drawn.

`void putArc(char image[][WIDTH][COLORS], int centerY, int centerX, int radius1, int radius2, int start, int end, char color[]).` This function takes as input the following parameters:

`char image[][WIDTH][COLORS]`	image array
`int centerY, int centerX`	center of the ring segment
`int radius1`	outer radius of the ring segment
`int radius2`	inner (smaller) radius of the ring segment
`int start`	the starting angle of the ring segment (in degrees)
`int end`	the ending angle of the ring segment (in degrees)
`char color[]`	color of the ring

This function changes the pixels of the `image` array to show a ring segment (an arch) with the radi `radius1` (outer) and `radius2` (inner) centered at `(centerX, centerY)` (column `centerX`, row `centerY`) and located between the `start` and `end` angles using the color specified by the `color` array.

The `start` and `end` angles are expected to be between 0 and 360, and it is expected that `start` $\leq$ `end`.

For example, the following program:

```
#include "image.h"
int main() {
   char image[HEIGHT][WIDTH][COLORS];
   char white[COLORS], color[COLORS];

   setColor(white, 255,255,255);
   setColor(color, 255,0,0);
   blankImage(image,white);
   putArc(image, 200,300,100, 60, 0, 90color);
   drawImage(image);
```

```
   return 0;
}
```

outputs a PPM image of a **South-East quarter** of a red ring of width 40 (100 - 60) on the white background.

Note: Not every possible ring segment can be displayed using this function — only those, that *do not cross the positive side of the X axis*. This is mostly due to the fact that the function works only for `start`$\leq$ `end`. Interpretation of situations when `end`$<$ `start` is somewhat difficult, and may require extra help from users, so it is left outside the scope of the `image.h` library.

Implementation notes. `putArc()` to `putRing()` is what `putPie()` is to `putCircle()`. Use this observation to develop code for `putArc()` appropriately.

`void putEllipse(char image[][WIDTH][COLORS], int centerY, int centerX, int radius1, int radius2, char color[]).` This function takes as input the following parameters:

`char image[][WIDTH][COLORS]`	image array
`int centerY, int centerX`	center of the ellipse
`int radius1`	the vertical radius of the ellipse
`int radius2`	the horizontal radius of the ellipse
`char color[]`	color of the circle

This function changes the pixels of the `image` array to show a solid ellipse centered at point `(centerX, centerY)` (column `centerX`, row `centerY`) with vertical radius `radius1` and horizontal radius `radius2` using the color specified by the `color` array.

For example, the following program:

```
#include "image.h"
int main() {
   char image[HEIGHT][WIDTH][COLORS];
   char white[COLORS], color[COLORS];

   setColor(white, 255,255,255);
   setColor(color, 255,0,0);
   blankImage(image,white);
   putEllipse(image, 200,300,100,50 color);
   drawImage(image);
   return 0;
}
```

outputs a PPM image of a red ellipse enlongated along the vertical axis.

Note. This function can only produce ellipses that are co-alinged with the major coordinate axes.

Implementation notes. The equation of an ellipse is

$$\frac{(x-x_0)^2}{r_h^2} + \frac{(y-y_0)^2}{r_v^2} \leq 1,$$

where (x_0, y_0) is the center of the ellipse, r_h is its horizontal radius and r_v is its vertical radius.

`void putLine(char image[][WIDTH][COLORS], int fromY, int fromX, int toY, int toX, char color[]).` This function takes as input the following parameters:

`char image[][WIDTH][COLORS]`	image array
`int fromY, int fromX`	the starting point of the line
`int toY, int toX`	the ending point of the line
`char color[]`	color of the line

The function paints pixels of the `image` array to show a line of color specified by the `color` array, connecting pixels `(fromX,fromY)` and `(toX,toY)`.

When implementing this function, please note the following:

- Any pair of pixels can be connected by a line. The choice of the start and the end pixel of the line is arbitrary. That is, given two pixels: `(y1,x1)` and `(y2,x2)`, `putLine()` can be called in two different ways:

  ```
  putLine(image,y1,x1,y2,x2,color);
  ```

  and

  ```
  putLine(image,y2,x2,y1,x1,color);
  ```

  to produce the line between them.
- Any line on a pixelated raster is an approximation of a real straight line between two points in space. Thus, lines may and will look pixelated.
- It may be a good idea to make the line at least two pixels wide to make it appear smoother.
- The two-point form of the line equation is:

  $$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1),$$

  where (x_1, y_1) and (x_2, y_2) are the two points through which the line passes.
- Notice that $x_2 = x_1$ is a special case, which needs to be handled separately.
- The formula above describes a line. You need only a line fragment between (x_1, y_1) and (x_2, y_2). This can be done in a number of ways. One way is to limit your consideration to only points that are "between" (x_1, y_1) and (x_2, y_2) (but make sure that you order x_1, x_2 and y_1, y_2 properly in this case).

Image.c

You will put all implementations of the functions in the `image.h` into the `image.c` file. The file should contain the `#include "image.h"` directive, and should `#include` any other necessary standard C libraries (e.g., `<math.h>`). After that, `image.c` shall consist of definitions of functions from `image.h`.

Testing

To test your implementations, you can use any of your programs from Lab 6-2 and Lab 7 and incorporate your implementations of the `image.h` functions instead of the instructor's.

In particular, to compile your `image.c` file use the following command:

```
gcc -ansi -Wall -Werror -c image.c
```

This will create an `image.o` object file from your code. You can then compile any of your programs using **your** `image.o` file:

```
gcc -ansi -Wall -Werror -o car car.c image.o
```

Submission.

Files to submit. You need to submit only one file:

```
image.c
```

Submission procedure. You will be using `handin` program to submit your work. The procedure is as follows:

- `ssh` to `vogon` (`vogon.csc.calpoly.edu`).

```
> handin dekhtyar program03  image.c
```

`handin` is set to stop accepting submissions 24 hours after the due time.

Testing and Grading

Any submitted program that does not compile earns 0 points.

I will use my versions of the Lab 6 and Lab 7 programs to test your `graphics.c` implementations.