Lab 6: Part 2: Functional Decomposition ...

Due date: Wednesday, February 23, 11:59pm.

Lab Assignment

Assignment Preparation

Lab type. This is an **pair programming lab**. For this lab, you get to select your own partner. The only rule is that your partner **must be different** than your Lab 5 partner.

Purpose. This lab is designed to teach you simple functional decomposition skills.

Programming Style. All submitted C programs 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.

Testing and Submissions. Any submission that does not compile using the

```
gcc -ansi -Wall -Werror -lm
```

compiler settings will receive an automatic score of 0.

Please, make sure you test all your programs prior to submission! Feel free to test your programs on test cases you have invented on your own.

Assignment Overview

For this lab assignment, you will work in pairs on using a library of graphics primitives provided to you by the instructor to build a number of Portable Pixel Map images.

Unlike other labs, for this lab, you will concentrate on understanding how a specific image (a crescent, a ying-yang symbol, a house, etc.) was constructed using the functions from the instructor's library. You will then write a program that attempts to replicate the image. The exact match of the image you produce to the instructor's sample image is not required, however, the image that you produce must be of similar complexity (i.e., contain a similar number of elements in it) and must represent the same object or set of objects as the instructor's image.

Instructor's Graphics Library `image.h`

The instructor's graphics library is provided to you in two files:

- `image.h`: is the header file for the library. It contains **function declarations** for all functions implemented by the instructor. Additionally, it contains a number of symbolic constant definitions.
- `image.o`: is the **object file** which contains the binaries of instructor's implementation of all functions declared in `image.h`. In order to use instructor's functions, you will **link** the `image.o` file to your code during compilation time. Instructions are given below.

Image Primitives Library: overview

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 which 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`$\times$ `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.

`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.

`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.

`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.

`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.

`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.

`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).

`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:

`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.

`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.

void putArc(char image[][WIDTH][COLORS], int centerY, int centerX, int radius1, int radius2, int start, int end, char color[]); This function takes the 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` ≤ `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`≤ `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.

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.

`void putLine(char image[][WIDTH][COLORS], int fromY, int fromX, int toY, int toX, char color[])`. **This is an extra credit assignment.** If you do not want to complete it, or, if you were unable to get it right, **include in your `image.h` file a stub of this function**, i.e., a function that does nothing (doing so, allows me to run all tests, including extra credit tests on all submissions).

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 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.

Use of image.h library in your code

Source code. As examples above show, the only thing you need to do in the source code of your programs in order to gain access to the functions from the `image.h` library is to add

```
#include "image.h"
```

preprocessor directive in your programs. Note, that because `image.h` is NOT a standard C library, we enclose `image.h` in double quotes ”’.

Please ensure that `image.h` file is located in the same directory as your C programs that use it.

Compilation. `image.h` file does not, by itself, have any code implementing the declared functions. This is done in a C program written by your instructor. Because your future task may be to write some of these functions, the source code implementing `image.h` functions is **not released**. Instead, you are provided with a **binary object file** `image.o` — the result of compiling instructor's code.

In order to successfully create executables for your C programs that use `image.h` library, you need to **link** `image.o` to your program. This is done by adding `image.o` to the list of parameters for the `gcc` compiler.

Example. For a C program `boo.c` that uses `image.h` functions, the compilation command will look as follows:

```
> gcc -ansi -Wall -Werror -lm -o boo boo.c image.o
```

The result of this command (assuming no compilation errors) is an executable file `boo` whose code includes the code for the necessary functions from the image library.

Please note, that `image.o` file must be located in the same directory as the C program you are compiling.

Assignment

You will produce 10 programs that use the graphics primitives declared in `image.h` to draw various pictures. Of the 10 programs, eight (8) shall attempt to reproduce images generated by the instructor, and the remaining two (2) will be designed by you.

All programs you submit are expected to consist of a single `main()` function (albeit, you may, in some cases, declare other functions and use them in `main()` in turn).

Instructor's images

All instructor's images are posted to the Lab 6 data page,

http://www.csc.calpoly.edu/∼dekhtyar/101-Winter2011/labs/lab6.html

You are expected to produce images that are **similar** to the instructor's and capture and, possibly, improve the objects depicted on them. You are not expected to produce images that are exactly the same as instructor's. (e.g., being off by a few pixels in drawing a smiley face is ok, failing to correctly depict the smile is NOT OK).

Red Cross and Red Crescent symbol (redcross.c). The first image is the symbol of the International Red Cross and Red Crescent Society. As follows from the name of the society, its symbol consists for two parts: a red cross and a red crescent.

While it is fairly straightforward to see how the red cross part of the image can be built, the crescent requires a bit of trickery.

Name your program `redcross.c`.

Smiley Face. (smiley.c). The second image is the original iconic smiley face. Unlike the Red Crescent symbol, all elements forming the smiley face image are explicitly seen on the image, so creating it is just a matter of proper selection of input parameters to your function calls (circle and ellipse radii and centers, etc.).

Name your program `smiley.c`

Ying-Yang Symbol (yang.c). The third image is another iconic symbol: the **ying-yang**. The variant you are asked to produce uses the red-blue color scheme and the orientation of the ying and yang components found on the flag of South Korea.

Believe it or not, this image consists solely of circles (and pies).

Name your program `yang.c`

Hanging Traffic Light. (redlight.c). The fourth image is a simple picture showing a simplified traffic light hanging from an overhang beam by a couple of wires and flashing yellow.

You are welcome to improve the image by adding other elements/objects to it, but the following must be present:

- the traffic light itself, flashing yellow.
- the full background, separating "ground" from "air"/"sky".
- the traffic light must be affixed to something.

Name your program `redlight.c`.

Five-point Star. (star.c). The fifth image is a picture of a five-point star. Without revealing much, I can say that it is an exercise in drawing lines. The instructor's version puts the five points symmetrically, but the points are not located on the same circle, and therefore the star is somewhat imperfect. You may feel free to improve this image to produce a pefect star, or a pentagram image (a five-point start enclosed into a circle or a thin ring).

Name your program `star.c`.

A car on the road. (car.c). The sixth image is a picture of a car driving on a road. In the instructor's version the car is a rather lame sedan. You may try to match the instructor's design, or you may opt to create your own car design using the graphics primitives available to you. To qualify, your design must include:

- A split (road/sky) background.
- A car image that is constructed out of multiple primitive objects.
- Some elements on the car image must overlap (one element hiding a portion of the other).
- The front and the read of the car should be different from each other.

Name your program `car.c`.

A house. (house.c). Your seventh program is a picture of a house. The most devious aspect of the instructor's version is the triangular roof: recall, there is no graphics primitive for drawing triangles in our library. To qualify as fullfilling requirements your image must contain the following:

- The total count of all elements should be similar to the total count of elements (graphics primitives) on the instructor's version, or should exceed it.
- There should be full background split into "ground" and "sky" potions.
- The house should have a triangular (or trapezoidal) roof.

Name your program `house.c`.

A computer (computer.c). Your eighth program is a picture of a computer system consisting of a system block, a flat-panel monitor, a keyboard and a mouse on a mousepad. The instructor's version shows a "not-a-Mac" machine (e.g., the mouse has two buttons). The instructor's version lacks any wires, although it is a good extra credit exercise to include some wires (e.g., made out of connecting ring segment pieces) in the picture.

To qualify as fulfilling requirements your image must contain the following:

- There should be a full split (desk/"sky") background.
- Each part of the computer present in instructor's version must be present in yours.
- The total number of elements used to draw your image must be similar to or should exceed the number of elements on the instructor's image.

Mystery Images

The remaining two images each team has to produce are left to individual teams. Each team must submit two C programs, named `mystery1.c` and `mystery2.c`, which construct, using the graphics primitives from the `image.h` library images depicting **recognizable objects or scenes**.

The subject matter of the images is limited only by the standard rules of common sense and your imagination. As we will be organizing a gallery of these images. no offensive images, please! To qualify, the images built by the programs must have complexity (represented as the number of graphics primitives used to construct them) similar to or exceeding the complexity of the `computer.c`, `car.c` and `house.c` programs/images.

Extra points will be awarded for artistic creativity and all successful submissions will be featured on our course web site. Examples of the creative works from the prior year are available on the Fall 2009 CPE 101 web site (and are linked to from the Lab 6 data page). Please, try to come up with different ideas.

Submission.

Files to submit. Submit eleven files:

```
team.txt,
redcross.c,
smiley.c
yang.c
redlight.c
star.c
car.c
house.c
computer.c
```

Files can be submitted one-by-one, or all-at-once.

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

Section 01:

```
> handin dekhtyar lab06-2-01  <your files go here>
```

Section 09:

```
> handin dekhtyar lab06-2-09  <your files go here>
```

Testing and Grading

Any submitted program that does not compile earns 0 points. Please download an run instructor's test to see the exact output produced. Your programs are expected to match this output.

In general, the programs will be graded by compiling them, creating the PPM files and visually inspecting them.