Programming Assignment #4

CS 200, Fall 2015

Due Tuesday, September 29

Programming task #1

I will provide you with a header file named `RasterUtilities.h`, which contains the following function declarations.

```
#include "Raster.h"

void FillRect(Raster& r, int x, int y, int W, int H);
void DrawRect(Raster& r, int x, int y, int W, int H);
```

You are to implement each of these functions, which are described below. I will also provide you with the `Raster.h` which defines the elementary raster operations discussed in class (see also the documentation at the end of this handout).

`FillRect(r,x,y,W,H)` — fills in a solid axis aligned rectangle, using the current foreground color. The coordinates (x, y) give the screen coordinates of the lower left hand corner of the rectangle. The rectangle should be W pixels wide and H pixels tall.

`DrawRect(r,x,y,W,H)` — draws the boundary of an axis aligned rectangle, using the current foreground color (the interior is not filled in). The coordinates (x, y) give the screen coordinates of the lower left hand corner of the rectangle. The rectangle should be W pixels wide and H pixels tall.

In both functions, the figure is drawn to the frame buffer associated with the `Raster` class instance `r`.

The implementations of these functions will be graded on correctness as well as efficiency and simplicity of coding. In particular, you should use the `IncrementX`, `IncrementY`, `DecrementX`, and `DecrementY` functions when possible, rather than using `GotoPoint`. For full credit, the functions should not write any duplicate pixels.

Your submission for this part of the assignment should consist of a single source file, named `RasterUtilities.cpp`. You may not include any header files except for the file `RasterUtilities.h` (note that this file includes `Raster.h`).

Programming task #2

The header file `RasterUtilties.h` also contains the following function declaration for drawing a line segment to the frame buffer associated with `Raster` class instance `r`:

```
void DrawLine(Raster& r, const Point& P, const Point& Q);
```

This function will scan convert the line segment from point P to point Q in screen coordinates. Note that unlike the `FillRect` and `DrawRect` functions in the first programming task, the coordinates are floating point values, and not necessarily pixels.

Your implementation of the `DrawLine` function will be graded on efficiency as well as correctness. This means that you should use the DDA algorithm. In particular, there should be no *explicit* multiplications inside of any loop.

Your submission for this part of the assignment should consist of a single file, named `DrawLine.cpp`. You may only include the header file `RasterUtilities.h`.

The `Raster` header file

The file `Raster.h` contains the following declarations.

```
typedef unsigned char byte;
Raster(byte *rgb_data, int W, int H, int S);
void GotoPoint(int x, int y);
void SetColor(byte r, byte g, byte b);
void WritePixel(void);
void IncrementX(void);
void DecrementX(void);
void IncrementY(void);
void DecrementY(void);
```

`Raster(rgb_data,W,H,S)` — (constructor) creates an instance of the `Raster` class associated to the frame buffer starting at address `rgb_data`. The frame buffer is W pixels wide, and H pixels tall, with each scan line taking up S bytes (the stride of each scan line). The caller of constructor is responsible for allocating the memory for the frame buffer, and the memory is expected to persist over the lifetime of the class instance.

`GotoPoint(x,y)` — sets the current point to pixel location (x, y). No data is written to the frame buffer.

`SetColor()` — sets the current foreground color.

`WritePixel()` — writes a pixel to the frame buffer at the current point in the current foreground color. See the comments below.

`IncrementX()` — moves the current point one pixel to the right.

`DecrementX()` — moves the current point one pixel to the left.

`IncrementY()` — moves the current point one pixel upwards.

`DecrementY()` — moves the current point one pixel downwards.

By default, the function `WritePixel` will cause the program to abort if the current point lies outside of the screen rectangle. Specifically, if (x, y) is the current point, then when `WritePixel` is called, then four comparisons

$$0 \leq x \quad \text{and} \quad x < \mathit{width} \quad \text{and} \quad 0 \leq y \quad \text{and} \quad y < \mathit{height}$$

are performed. If any of the comparisons are found to be false, the program will be aborted. This behavior can be modified in two ways. First, if the symbol `NDEBUG` is defined, then none of the above comparisons are performed, and the pixel will be written. If a point outside of the screen rectangle is written to, undefined behavior will resut. Second, if the symbol `CLIP_PIXEL` is defined, the above comparisons will be performed. If any of the comparisons are found to be false, the pixel will not be written to the frame buffer.

Simple bitmap file format

All bitmap (.BMP) files consist of a header followed by image data. Although the image data can be stored in a variety of ways, the simplest (and most common) is the 24 bit per pixel uncompressed color format. In this assignment, you are to assume that the bitmap images supplied to your program are always in this form. Let us examine each of the two parts of the 24 bit color format.

Bitmap file header

In the case of the 24 bit color image format, the header is a total of 54 bits in length. The fields of the header are as follows.

```
[00]  unsigned short FileType       = 'BM'
[02]  unsigned int   FileSize       = <size of file in bytes>
[06]  unsigned short Reserved1      = 0
[08]  unsigned short Reserved2      = 0
[10]  unsigned int   BitmapOffset   = 54
[14]  unsigned int   HeaderSize     = 40
[18]  int            Width          = <image width in pixels>
[22]  int            Height         = <image height in pixels (see below)>
[26]  unsigned short BitPlanes      = 1
[28]  unsigned short BitsPerPixel   = 24
[30]  unsigned int   Compression    = 0
[34]  unsigned int   SizeOfBitmap   = <size of image data in bytes>
[38]  unsigned int   HorzResolution = <ignore>
[42]  unsigned int   VertResolution = <ignore>
[46]  unsigned int   ColorsUsed     = <ignore>
[50]  unsigned int   ColorImportant = <ignore>
```

The value in square brackets on the left is the offset of the field (in bytes); the value on the right is the field value used in a 24 bit color bitmap.

Note that the `Width` and `Height` fields are both *signed* integers. The value of the width should always be positive, but the height value may actually be negative. A negative height value indicates that fist row of data corresponds to the *top* row of the image — as the convention in typical graphics screen coordinates. However, most bitmap images will have a positive height value, indicating that the first row of data corresponds to the *bottom* row of the image.

Bitmap data

The pixel data image will follow immediately after the header; the data consists of the pixel values stored in a row major manner: the bottommost row of pixels in the image are stored first, followed by the next row, and so on, with the topmost row stored last. With the 24 bit color format, the individual pixels within each row consist of 3 (unsigned) bytes: the first byte gives the *blue* component value, the second the *green* component, and the third the *red*

component (note the reverse order from the usual RGB representation of a pixel). However, each row must must be aligned on *4–byte boundaries;* i.e., there are 0–3 additional bytes that are appened to each row, so that the number of bytes that a row takes up is a multiple of 4. The total number of bytes per row is called the **stride** of the image.

For example, suppose we have a bitmap image that has a width of 3 pixels, and a height of 2 pixels. Letting p_{ij} denote the pixel in the i–th row and j–th column, the image looks like

p_{10}	p_{11}	p_{12}
p_{00}	p_{01}	p_{02}

(remember that row 0 is the bottommost row of the image). The data for the image would then be stored in the following form.

$$B_0, G_0, R_0,\ B_1, G_1, R_1,\ B_2, G_2, R_2,\ 0, 0, 0,\ B_3, G_3, R_3,\ B_4, G_4, R_4,\ B_5, G_5, R_5,\ 0, 0, 0$$

Where, in terms of RGB components, $p_{00} = (R_0, G_0, B_0)$, $p_{01} = (R_1, G_1, B_1)$, $\ldots$, $p_{12} = (R_5, G_5, B_5)$. Since the nearest multple of 9 is 12, each row is padded by 3 additional bytes, to give a stride of 12 bytes.