Programming Assignment #5

CS 200, Fall 2017

Due Wednesday, October 11

Your task: implement a texture map class

I will provided you with the interface of a class for managing a texture map. Specifically, the file `Texture.h` contains the following declaration.

```
class Texture {
  public:
    explicit Texture(unsigned W=64, unsigned H=64);
    Texture(const char *bmp_file);
    ~Texture(void);
    unsigned char* RGBData(void) { return data; }
    unsigned width(void)         { return bmp_width; }
    unsigned height(void)        { return bmp_height; }
    unsigned stride(void)        { return bmp_stride; }
    Hcoord uvToRGB(float u, float v);
    Point UVTouv(float U, float V);
    Point uvToUV(float u, float v);
    enum { R=0, G=1, B=2 };
  private:
    unsigned char *data;
    int bmp_width, bmp_height, bmp_stride;
    Affine uv_to_UV, UV_to_uv;
};
```

(the header file `Affine.h` has been included). You are to implement this class. The member functions of this class are detailed below.

`Texture(W,H)` — (constructor) creates a texture whose underlying bitmap is a bottom–up image that is W pixels wide and H pixels high. The bitmap should be initialized so that the pixel at coordinate (x, y) has color value

$$(r,g,b) = \begin{cases} (100+155u, 0, 100+155v) & \text{if } (u-\frac{1}{2})^2 + (v-\frac{1}{2})^2 > 0.16 \\ (100+155v, 0, 100+155u) & \text{if } (u-\frac{1}{2})^2 + (v-\frac{1}{2})^2 < 0.09 \\ (0, 255, 0) & \text{otherwise} \end{cases}$$

where (u, v) is the normalized texture coordinate corresponding to the (unnormalized) bitmap coordinate (x, y). The bitmap must be aligned on 4–byte boundaries, and data should be stored in RGB order.

`Texture(bmp_file)` — (constructor) creates a texture from the bitmap image file (.BMP) with name `bmp_file`. The bitmap file is assumed to be a 24 bit color uncompressed image. If the file is not a valid bitmap image file, a `runtime_error` exception should be thrown.

Recall that the bitmap file data is aligned on 4–byte boundaries, and stored in BGR order. However, data stored by the texture class should be in RGB order, so you will need to reverse the byte order of each pixel.

Also, some paint programs will write a bitmap file whose `SizeOfBitmap` (at offset 34 in the header) field is set to zero! In this case, you will need to compute the size of the data directly from the `Width` and `Height` fields.

`~Texture()` — (destructor) deallocates the bitmap data stored by the class.

`RGBData()` — returns a pointer to the bitmap data stored by the class. The data is aligned on 4–byte boundaries and stored in RGB order. [implemented]

`width()` — returns the width of the image, in pixels. [implemented]

`height()` — returns the height of the image, in pixels. In the bitmap image file format, the height value may be negative, indicating a top–down image. For this assignment, you do **not** need to convert to a bottom–up image. However, you will need to store a positive value in the `bmp_height` field of the class. [implemented]

`stride()` — returns the number of bytes per scan line of the image. As noted above, scan lines are aligned on 4–byte boundaries. [implemented]

`uvToRGB(u,v)` — returns the color of the pixel that is nearest to the point (U, V), where (U, V) are the fractional bitmap coordinates corresponding to the given normalized texture coordinates (u, v). Texture wrapping should be used.

`UVTouv(U,V)` — returns the normalized texture coordinates (u, v) corresponding to the fractional bitmap coordinates (U, V). Texture wrapping should *not* be used.

`uvToUV(u,v)` — returns the fractional bitmap coordinates (U, V) corresponding to the normalized texture coordinates (u, v). Texture wrapping should *not* be used.

Usage remark. The class is intended to be *light–weight:* you should **not** make copies of class objects, and you should not overwrite class objects. E.g., the statements

```
Texture texture1("fred.bmp");
Texture texture2 = texture1;  /* BAD: attempting to make a copy */
Texture texture3(512,512);
texture2 = texture3;          /* BAD: attempting to overwrite */
```

will lead to undefined behavior. Pointers to `Texture` class objects should be used instead.

File reading remark. In the non–default constructor where a bitmap file is to be read–in, it is possible that the file either does not exist, or is in an unexpected format. An exception should be thrown in both cases. To ensure that the file is in the expected format, you will need to check a few things in the file header: (1) the `FileType` field is the correct value (indicating an actual bitmap file), (2) the `BitPlanes` field is set to 1, (3) the `BitsPerPixel` field is set to 24, and (4) the `Compression` field is set to 0.

Your submission for this assignment should consist of a single source file named `Texture.cpp`. You may only include the header file `Texture.h` (which includes `Affine.h`, and consequently the standard `cmath` header file) and the standard header files `fstream` and `stdexcept`.

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 (although some programs will generate bitmap files with differing header sizes). 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   = <offset to start of data>
[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 a top–down image: the first 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 a bottom–up image: 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.

Practical matters

On 32 bit and 64 bit machines (like the ones we use), fields in a structure are typically aligned along 4–byte boundaries. This means that if we use a C/C++ structure for the bitmap file header, we will run into trouble, since the header has fields that are aligned on 2–byte boundaries, and the compiler will often add padding bytes to enforce the 4–byte alignment of the fields. For this reason, it is better to access the fields of the header using offsets and recasting.

Here are some useful code recipes for dealing with 24 bit color uncompressed bitmap files.

- Open a bitmap file and read in its header:

```
fstream in(filename,ios_base::binary|ios_base::in);
char header[54];
in.read(header,54);
```

- Extract the size of the bitmap data, and the offset to the data from the header:

```
unsigned data_size = *reinterpret_cast<unsigned*>(header+34),
         data_offset = *reinterpret_cast<unsigned*>(header+10);
```

  Warning: some programs write a value of 0 for `data_size`. In this case, you will need to compute the data size from the image pixel width and height.

- Read in the raw bitmap data:

```
unsigned char *data = new unsigned char[data_size];
in.seekg(data_offset,ios_base::beg);
in.read(reinterpret_cast<char*>(data),data_size);
```

- Extract the image width and height from the header:

```
int width = *reinterpret_cast<int*>(header+18),
    height = *reinterpret_cast<int*>(header+22);
```

  Note that the value of `height` may be negative. You should compute the stride of the image from the width and height values.

- Read in the RGB data for the pixel at location (i, j):

```
enum { R=0, G=1, B=2 };
unsigned char rgb[3];
int index = j*stride + 3*i;
rgb[R] = data[index+B];
rgb[G] = data[index+G];
rgb[B] = data[index+R];
```