

# Programming Assignment #5

CS 200, FALL 2024

*Due Friday, October 25*

I will provide you with the interface of a class for managing bitmaps and textures. Specifically, the file `Texture.h` contains the following declaration.

```
namespace cs200 {
    class Bitmap {
    public:
        explicit Bitmap(unsigned W=64, unsigned H=64);
        Bitmap(const char *bmp_file);
        unsigned char* data(void) { return &bmp_data[0]; }
        const unsigned char* data(void) const { return &bmp_data[0]; }
        unsigned width(void) const { return bmp_width; }
        unsigned height(void) const { return bmp_height; }
        unsigned stride(void) const { return bmp_stride; }
        unsigned offset(int i, int j) const;
    private:
        std::vector<unsigned char> bmp_data;
        unsigned bmp_width, bmp_height, bmp_stride;
    };

    unsigned computeStride(unsigned W);
    void reverseRGB(Bitmap &b);
    glm::mat4 bitmapToTextureTransform(const Bitmap &b);
    glm::mat4 textureToBitmapTransform(const Bitmap &b);
    glm::vec3 getColor(const Bitmap &b, float u, float v);
}
```

You are to implement the class and functions of this package. Here are the details.

`Bitmap::Bitmap(W,H)` — (constructor) creates a  $W \times H$  bottom-up bitmap object. 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  $(u,v)$  are the texture coordinates corresponding to the bitmap coordinates  $(x,y)$ . The bitmap must be aligned on 4-byte boundaries, and data should be stored in RGB order.

`Bitmap::Bitmap(bmp_file)` — (constructor) creates bitmap object from an image file (.BMP) with name `bmp_file`. The image file is assumed to be a 24 bit color uncompressed standard bitmap image. If the file is not a valid bitmap image file, a `runtime_error` exception should be thrown. Although bitmap images are typically stored in BGR format, do not reverse the byte order. The user is expected to call the `reverseRGB` function after the image is loaded.

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.

Be aware that some applications 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.

`Bitmap::data()` — returns a pointer to the bitmap data stored by the class. [implemented]

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

`Bitmap::height()` — returns the height of the bitmap image, in pixels. [implemented]

`Bitmap::stride()` — returns the number of bytes per scan line of the image. Note that scan lines are aligned on 4-byte boundaries. [implemented]

`Bitmap::offset(i,j)` — returns the offset into the bitmap data `bmp_data` corresponding to bitmap coordinates  $(i,j)$ . If the coordinates  $(i,j)$  fall outside of the image, a standard `out_of_range` exception should be thrown.

`computeStride(W)` — returns the stride (in bytes) for a bitmap of width  $W$  pixels that is aligned along 4-byte boundaries.

`reverseRGB(b)` — reverses the byte order (from RGB to BGR and vice versa) of the bitmap object  $b$ .

`bitmapToTextureTransform(b)` — returns the transformation that maps bitmap coordinates  $(U,V)$  to texture coordinates  $(u,v)$  for the bitmap object  $b$ .

`textureToBitmapTransform(b)` — returns the transformation that maps texture coordinates  $(u,v)$  to bitmap coordinates  $(U,V)$  for the bitmap object  $b$ .

`getColor(b,u,v)` — returns the color of the pixel in bitmap object  $b$  that is nearest to the texture coordinates  $(u,v)$ . Texture coordinates wrapping should be used.

## 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.

## What to turn in

Your submission for this assignment should consist of a single source file named `Texture.cpp`. You may only include `Texture.h`, `Affine.h`, and any standard C++ header file (in particular, make sure to include `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 bytes 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 be aligned on *4-byte boundaries*; i.e., there are 0–3 additional bytes that are appended 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,  $p_{00} = (R_0, G_0, B_0)$ ,  $p_{01} = (R_1, G_1, B_1)$ ,  $\dots$ ,  $p_{12} = (R_5, G_5, B_5)$ . Since the nearest multiple 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++ 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 the RGB data for the pixel at location  $(i, j)$ :

```

struct RGB { unsigned char r, g, b; };
RGB color;
int index = j*stride + 3*i;
color.r = data[index+2];
color.g = data[index+1];
color.b = data[index+0];

```