DigiPen INSTITUTE OF TECHNOLOGY EUROPE - BILBAO

CS200 | Computer Graphics I
Lab 1 | The Frame Buffer

Topics

- Introduction to CS200 framework.
- Frame Buffer.

Goals

- Getting familiar with the CS200 framework.
- Implement our own frame buffer class.

Grading:

- There's no grading for this lab.

Description

A Visual Studio project is provided for this assignment. Although there are a lot of files in the 'src' folder, all your work for this lab will be in FrameBuffer.cpp. We will be coding all the functions except for the Save/Load functions which are provided for you.

The FrameBuffer class

The frame buffer is essentially an interface around a 2D array of pixels in 32-bit RGBA format. It provides functions to allocate/free the pixel data as well as basic functionalities such as getting/setting a pixel at location x, y and ***clearing*** the frame buffer, which means that we are setting a color to ALL its pixels.

Static Class

Additionally, the FrameBuffer class is programmed as a ***pure static class***. This can be seen by the keyword static prepending all member methods and variables of the class. Doing so means that there's no *per-instance data* and only *static data* defined for the FrameBuffer class.

This is an implementation of the SINGLETON programming pattern, which is used when one wants to make a class unique in the code base. In our case, this is appropriate, because there can only be one frame buffer where we send the results of the draw calls.

Calling a static class functions

Pure static classes behave very much like namespaces (with the exception of having private and public variables). The static functions are for all intents and purposes, ***global*** functions and are called with the following syntax:

<class_name>::<function_name>([arguments]).

For example, calling the FrameBuffer Present function we would write:

FrameBuffer::Present();

Defining static class variables

Very much like static class member functions, static class member variables are similar to global variables, which follow the same access rules (private/public/protected) as regular classes. However, unlike regular classes, the static member variables must be ***defined*** in a compiled file (i.e. a cpp file). This is why at the top of FrameBuffer.cpp, you will see the FrameBuffer variables defined as so:

```
u8 *    FrameBuffer::frameBuffer = NULL;
u32     FrameBuffer::frameBufferWidth = 0;
u32     FrameBuffer::frameBufferHeight = 0;
```

It's good practice to initialize the variables to default values.

The Present function

Another important function is **Present()**. This function is responsible of sending the contents of the frame buffer to the window. Normally, this process is done by calling a Win32 function, but in our case, we will rely on the Alpha Engine to do so.

Instead, we will create a temporary quad and texture to hold the pixels of the frame buffer and then draw it, applying a scale equal to the window dimensions.

About the Colors

You will see that the SetPixel and GetPixel functions interface with the **Color** class, declared in Color.h. This class stores the color component (RGBA) in normalized float format (in the range [0,1]). However, our frame buffer stores the pixel data as a 1 dimensional array of unsigned characters. In this case, we will have to convert to and from normalized color.

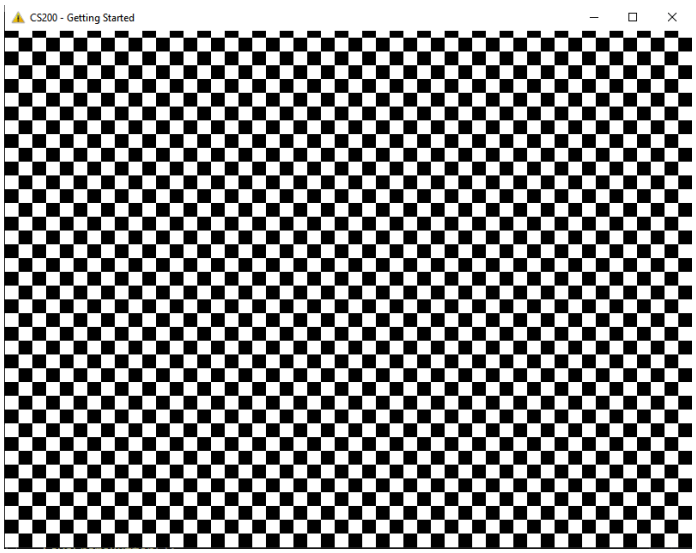
Extra Challenges

When you are done with the basic functions for the framebuffer, try implementing the following code challenges:

Code Challenge 1: Checkerboard pattern.

Description

Implement the following function that will clear the frame buffer with a checkerboard pattern.



Setup

Add the following declaration in the FrameBuffer class in FrameBuffer.h

```
static void ClearCheckerboard(u32 colors[2], u32 size);
```

Add the corresponding function definition in FrameBuffer.cpp

```
void FrameBuffer::ClearCheckerboard(u32 colors[2], u32 size)
```

Parameters

The function takes 2 parameters:

`u32 colors[2]:` the two colors to use in the checkerboard pattern.

`u32 size:` the size of each cell in the board.

Test yourself

To test your code, comment the previous call to clear and add the following lines in Level 1.cpp so that it looks like this:

```
void Level_1_Render()
{
	// Clear the frame buffer.
	//FrameBuffer::Clear(Rasterizer::Color().FromU32(0xFFFFFFFF));
	u32 colors[] = { 0xFFFFFFFF, 0xFF000000 };
	FrameBuffer::ClearCheckerboard(colors, 16);
```

```
		// Send content of frame buffer to
		FrameBuffer::Present();

		// Debug
		SaveFrameBuffer();
	}
```

Code Challenge 2: Fill the frame buffer with a png image

Description

Implement the following function that loads a png image from file and copies it into the frame buffer. To do so, you will need to use the Alpha Engine function AEGfxLoadImagePNG:

```
bool AEGfxLoadImagePNG(const char* filename, u8*& outPixels, u32& outWidth, u32&
outHeight);
```

As you can see, the function takes a u8 pointer as an ouptut parameter. You DON'T NEED TO ALLOCATE MEMORY FOR THIS ARRAY. The Alpha Engine will allocate memory for the pixel data and store the image width and height in the parameters you provide. If the image is loaded successfully, then it will return true. YOU WILL NEED TO FREE THE PIXEL MEMORY AFTER COPYING IT.

Setup

Add the following declaration in the FrameBuffer class in FrameBuffer.h

```
static void LoadFromImageFile (const char* filename);
```

Add the corresponding function in FrameBuffer.cpp. Use this template as a starting point:

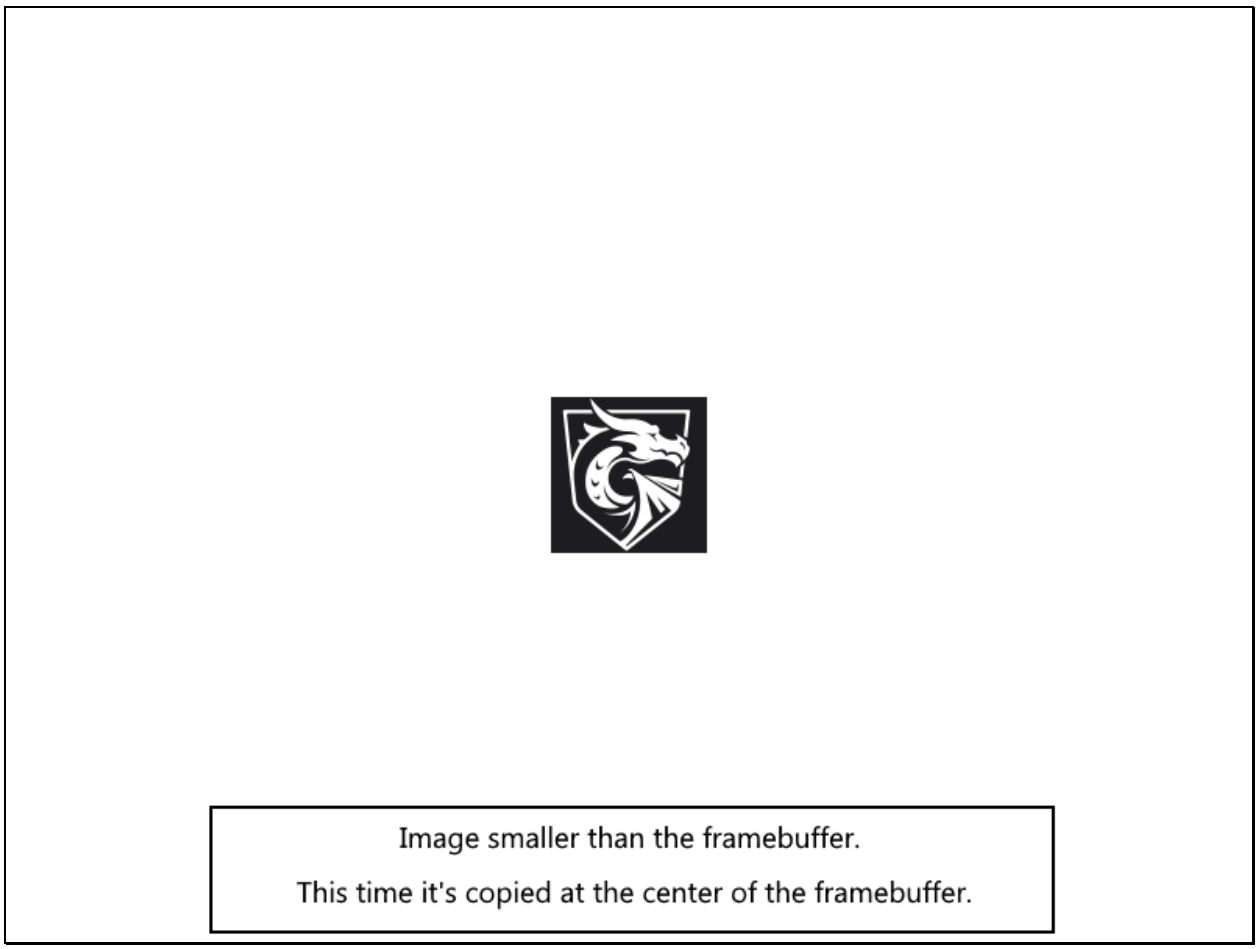
```
void FrameBuffer::LoadFromImageFile(const char* filename)
{
	// load the file using the alpha engine
	u8* imgPixels = 0;
	u32 imgWidth = 0, imgHeight = 0;
	if (AEGfxLoadImagePNG(filename, imgPixels, imgWidth, imgHeight))
	{
		// TODO: copy the data to the framebuffer

		// cleanup
		delete[] imgPixels;
	}
}
```

Hints:

It is highly likely that the picture you load doesn't have the same dimensions as the framebuffer. In this case, you have to be careful not to go out of bounds when copying the image pixels into the frame buffer.


Loading an image bigger than the frame buffer.
Copy as is. Ideally, we'd scale the image to fit

Image smaller than frame buffer. Copied as is means it will appear in the lower-left corner of the frame buffer.



Code Challenge 3: Combination of Challenge 1 and 2

Implement the following function to clear the frame buffer with an image in a checkerboard pattern. Since every other image needs to be different from the original, we will invert the image color.

```
void FrameBuffer::CheckerboardImage(const char* filename, u32 size);
```

The parameters are the same as the previous two functions:

- Filename: path to the image file to load.
- Size : size of a square in the checkerboard.


Challenge 3: Combine challenges 1 & 2.
Invert colors every time.