 Graphics I – Lab 1

# Objective

The objective of today’s lab will be to implement what was learned in lecture. You will be directly manipulating the color array by inserting color values into specific spots in the array. Before we begin with the lab specifics, first create a new **WIN32 CONSOLE APPLICATION**. Make it an empty project and create a main.cpp. By now you should know how to define a main function, so go ahead and do so. At this time please also include the **RasterSurface.h**. Inside main we will use three functions that are a part of the RasterSurface class.

* bool RS\_Initialize( unsigned int \_width, unsigned int \_height );
* bool RS\_Update( const unsigned int\* \_xrgbPixels, const unsigned int \_numPixels );
* bool RS\_Shutdown( void );

Initialize and Shutdown will only be called one time. Initialize at the start of main and Shutdown at the end of main. Update needs to be called at least one time per frame. Meaning this function needs to be called at least one time per cpu cycle. Since we do not know how many times we will be calling this function, which appropriate loop iteration keyword should we use here? As a side note, Initialize will create and open a **WIN32 WINDOW**. RS\_Update will continue to return true until the WIN32 Window closes. Once the window is closed, RS\_Update will return false. Now onto the usage of this RasterSurface class.

* Let’s initialize our RasterSurface to a window size of 500 by 500.
  + 500 pixels wide by 500 pixels tall
* We should also create an unsigned int array with the total number of pixels that we created our window with.
  + 1D array and 2D array both work just fine.
* RS\_Update will take this array and it will need to take the total number of pixels as well.

You will also be writing a few helper functions to assist you in this process. Here are a few example functions you might need.

* A function to clear the color buffer to a solid color of your choice.
* A function to convert 2 dimensional coordinates to a 1 dimensional coordinate.
* A function to draw a pixel (fill a certain pixel with a specific color)
* A function to BLIT (Block Image Transfer)

Additionally, we give you an executable called **TGA\_POW2\_32BIT\_TO\_TEXTUREARRAY.EXE** that converts any 32bit uncompressed .tga file to a .h file. The usage of this executable lets us easily access the color values of each pixel of the .tga file. In order to use this executable properly, just drop it in your texture folder and double click the executable. It will convert all .tga files inside the folder to their respective .h counterpart. You should include the new header file so you can access its data members. If you do not have a .tga file, you can create one by dragging and dropping the texture into Visual Studio. Afterwards go to **FILE > SAVE AS > CHANGE FILE EXTENSION TO TGA > CHOOSE A LOCATION.**

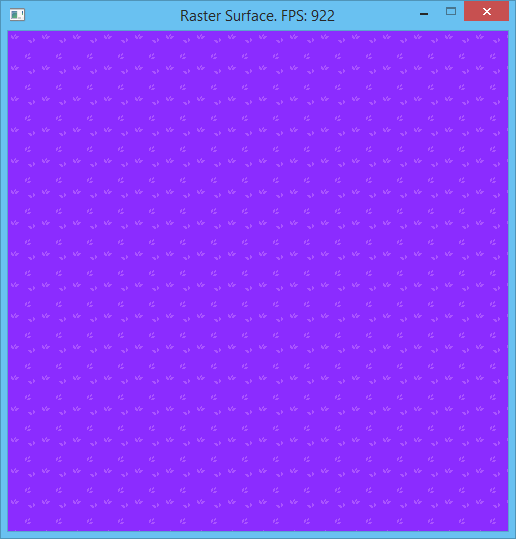
# Grading Breakdown

* 25% - Get a tiled background showing on the entire color array
  + The color values will be strange but this is fine for this section.
* 50% - Swap the color bits from **BGRA** to **ARGB** and get 10 random objects in the color array.
  + Some bit manipulation will be needed in order to fix the colors
  + If you were able to get this far, you should be able to draw 10 random “tiled” objects onto the color array as well.
* 75% - Get at least 1 “cell” of an animation rendering properly with blending. Ensure that the cell properly blends with what is behind it.
  + Depending on the alpha value of the cell, some form of interpolation will be needed to handle properly blending the background with the cell.
* 100% - Play an animation with a high speed timer ( The XTime class is great for this)
  + A minimum of 30 frames a second playing for the animation.

# Guidance

1. Once you setup the rendering loop, it is time to start drawing a pixel. Call your function that draws a pixel and make sure it is drawing in the correct location. Contrasting colors are best to test with.
2. Now that you can draw a pixel, we should now attempt to draw a tile (square set of pixels) or BLIT an image.
   1. For the tile’s height
      1. For the tile’s width
         1. Find the location into the image array. (2D to 1D)
         2. Draw a pixel at this location iterating through the width and the height
3. Once you are able to draw a BLIT / tiled image, we need to fill our color buffer with the same image. You are essentially going to repeat this BLIT horizontally and vertically as well.
   1. For the number of evenly divisible tile heights
      1. For the number of evenly divisible tile widths
         1. Call your function to draw a tile.
4. At this point you should notice a border around the edge of your right and bottom side of the screen. This is because the remaining pixels did not constitute enough space to draw another BLIT. What needs to be done to your draw function to adjust for this?
5. Getting this far you should have a strong understanding on how to BLIT. Place 10 BLITs randomly placed throughout the color buffer. These can be anything of your choosing. Obvious choices would be something large enough to notice.
6. Now let us introduce 1 “cell” of an animation. Be careful here because this is where you need to interpolate through color values since the animation might have alpha values lower than 255.
7. As soon as you can get a properly rendering BLIT with alpha transparency, solve how to iterate through the animation over time. Make sure the animation plays smoothly at 30 fps.

# Example Images





# Submission

Labs will be turned into student vfiler in the format of a zip file. The naming convention for the file will be ***Lastname.Firstname.Lab1.zip***. The contents of your zip file should only contain source files and shaders you have written. We would like these submissions to be as small as possible in order to accommodate space for everyone. If you fail to submit properly *you will lose points* on your grade. Even if it was a visual check-off.

# Above & Beyond

1. Convert the alpha blending code to use ONLY ***fixed point*** integer math as an optimization. You are allowed to research this topic online.
2. Write a BLIT variant that supports drawing a ***rotated*** bitmap. This routine should accept a 2x2 rotation matrix as well as a point to rotate about.

# Frequently Asked Questions