

GAM 695 Research II

Image Effects with Compute Shaders
GDImageProcessor

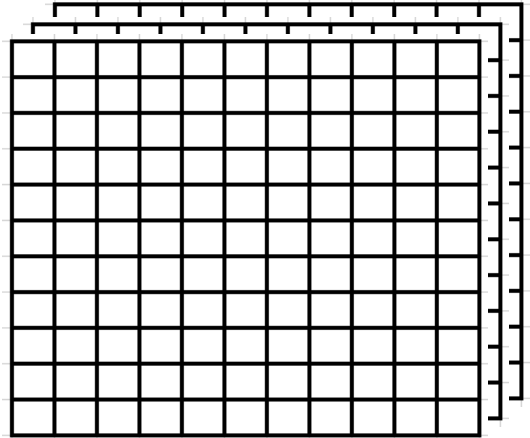
By: Kerwin Ghigliotty Rivera

DepaulID: 1938268

Initial Thoughts

My initial idea was to get some more exposure to compute shaders, the idea was to create an application in which I could use compute shaders to apply different effects to an image, be it bloom, edge detection, invert or other effects.

Taking advantage of the ability to define a x, y, z grid and the fact that a texture is x wide and y tall we can simply set `glDispatchCompute(textureWidth, textureHeight, 0)` that way for each effect we have a compute shader that is handling a specific pixel.



Prototype Design

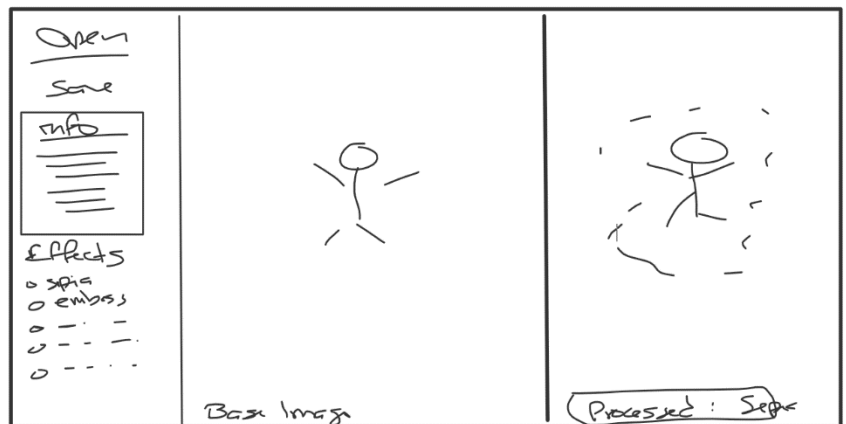
Initial concept

The user should be able to load an image file by clicking the Open button.

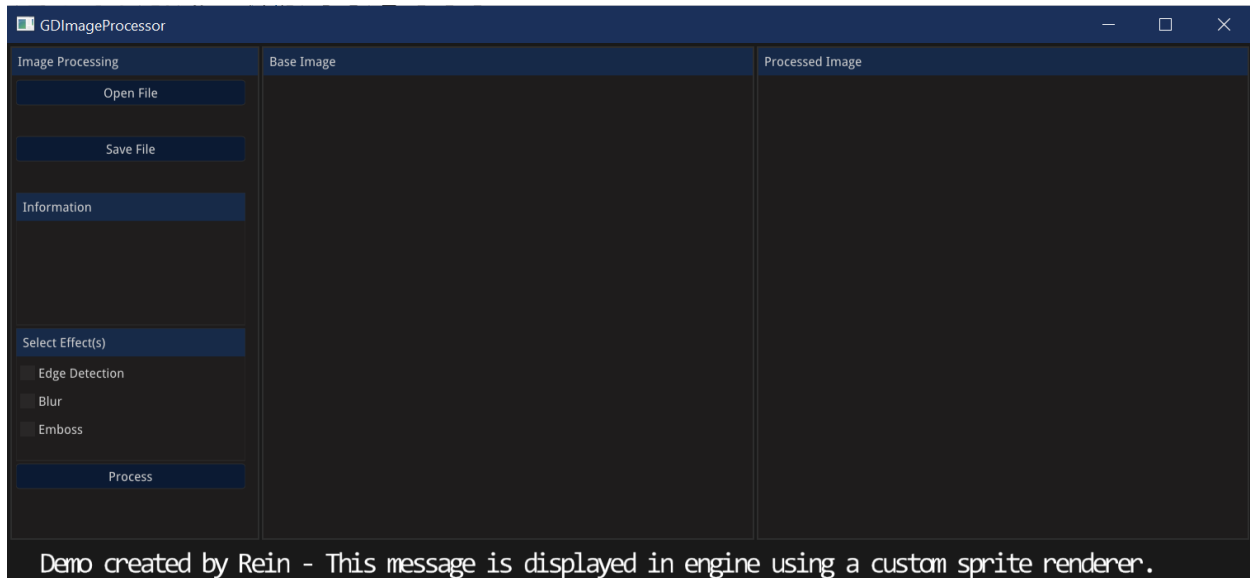
Once it is loaded the Base image should be populated with the image data and some Metadata should be available for the user in the Info section.

The user should be able to select the effects they would like to apply to the base image.

Concept Prototype v0.1



Prototype v0.1



The main concern in my implementation is how to render 3 different viewports on screen

In prototype v0.1 there is a top layer viewport for UI elements, in which the Text “Demo created by Rein – This message is displayed in engine using a custom sprite renderer” is kept.

The Nuklear UI is drawn on top of this viewport

```
CameraNode* camUI = CameraMan::Find(CameraBase::CAM_NAME::CAMERA_UI);  
camUI->GetCamera()->SetViewState();  
GameObjectMan::DrawBatch(0);  
UIContext::Draw();
```

As in the example above I managed to implement a batch system to just draw what elements I design with a particular ID, in this case UI elements are set to 0

And the order of operations is as follows

Get the UI Camera

Set its viewport into context

Draw elements in batch 0

By setting the view state we are simply calling the viewport by its preset coordinates as well as enabling a Scissor test.

```
glViewport(this->viewport_x, this->viewport_y, this->viewport_width, this->viewport_height);  
glScissor(this->viewport_x, this->viewport_y, this->viewport_width, this->viewport_height);  
glEnable(GL_SCISSOR_TEST);
```

Now if we do these for all 3 viewports, we get this

```
void Game::Draw()
{
    /// <summary>
    /// Camera Batches
    /// 0 = UI Elements
    /// 1 = Base Image Viewport
    /// 2 = Processed Image Viewport
    /// </summary>

    CameraNode* camUI = CameraMan::Find(CameraBase::CAM_NAME::CAMERA_UI);
    camUI->GetCamera()->SetViewState();
    GameObjectMan::DrawBatch(0);
    UIContext::Draw();

    CameraNode* camBase = CameraMan::Find(CameraBase::CAM_NAME::CAMERA_0);
    camBase->GetCamera()->SetViewState();
    GameObjectMan::DrawBatch(1);

    CameraNode* camProcessed = CameraMan::Find(CameraBase::CAM_NAME::CAMERA_1);
    camProcessed->GetCamera()->SetViewState();
    GameObjectMan::DrawBatch(2);
}
```

And the result of this grants us 3 viewports

One for the UI, one for Base Image and one for Processed Image



Now we can proceed with the next step which is to render the full image loaded on its viewport and then process it.

Compute Shader

With this out of the way I started working on my basic compute shader (bloom effect for now)

(Some snippets from RenderDoc)

Here I am specifying my input and output image values (this will be common for all the shaders used)

```
layout(binding = 0, rgba8)  readonly uniform image2D img_input;
layout(binding = 1, rgba32f) writeonly uniform image2D img_output;
```

As for the bloom effect, I am using the method described in this article which was extremely useful in understanding how blur is applied: [Efficient Gaussian blur with linear sampling](#)

I have my offset and weights set

```
uniform float offset[3] = float[](0.0, 1.3846153846, 3.2307692308);
uniform float weight[3] = float[](0.3162162162, 0.2270270270, 0.0702702703);
```

This is the way the weights are being used (numbers indicate indices)

		2		
		1		
2	1	0	1	2
		1		
		2		

The closer to the center the bigger the weight value will be, that means more color will be considered for that pixel, the farther from the center the less color is considered.

I am retrieving the current pixel using `gl_GlobalInvocationID.xy` to get the current position as it is a 1x1x1 layout and `imageLoad` to get the information then I apply the base weight to the center pixel.

```
int loops = 3;

ivec2 pos = ivec2(gl_GlobalInvocationID.xy); //get the current position
vec4 color = texelFetch(img_input, pos,0) * weight[0]; //get the pixel at that position and apply the base weight

for (int i=1; i<loops ; i++) //increase this for more blur (need to add more weights as well)
{
    //Vertical pass
    ivec2 offsetAdjustY = ivec2(0.0, offset[i]);
    color += texelFetch(img_input, pos + offsetAdjustY,0 ) * weight[i]; //apply the weight to the above pixels
    color += texelFetch(img_input, pos - offsetAdjustY,0 ) * weight[i]; //apply the weight to the below pixels

    //Horizontal pass
    ivec2 offsetAdjustX = ivec2(offset[i], 0.0);
    color += texelFetch(img_input, pos + offsetAdjustX,0 ) * weight[i]; //apply the weight to the right pixels
    color += texelFetch(img_input, pos - offsetAdjustX,0 ) * weight[i]; //apply the weight to the left pixels
}

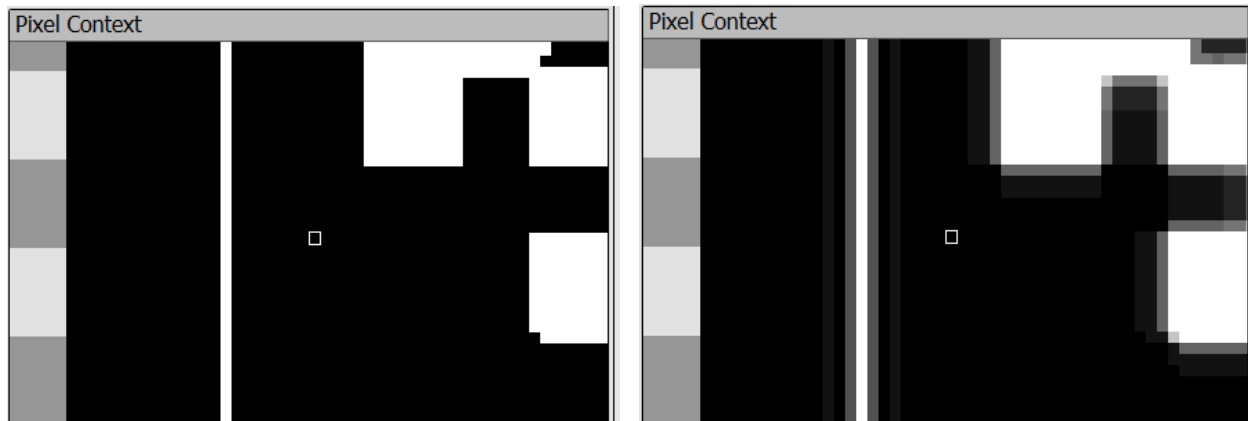
imageStore(img_output, pos, color);
```

I am performing a loop for both the Vertical and Horizontal bloom and then I am applying the color to the output image using `imageStore`

Using this shader, I can see the results immediately for a 738x576 image file

Input

Output



Current problem (Solved below)

Earlier I mentioned that this was my input and output images

```
layout(binding = 0, rgba8)  readonly uniform image2D img_input;  
layout(binding = 1, rgba32f) writeonly uniform image2D img_output;
```

Currently for binding 0 which is the input I am specifying that the image is in rgba8 format, but that can change depending on what is the input image.

Solution

By changing the layout for the input image to be a sampler2D instead of a image2D I am able to get away with not specifying the input format

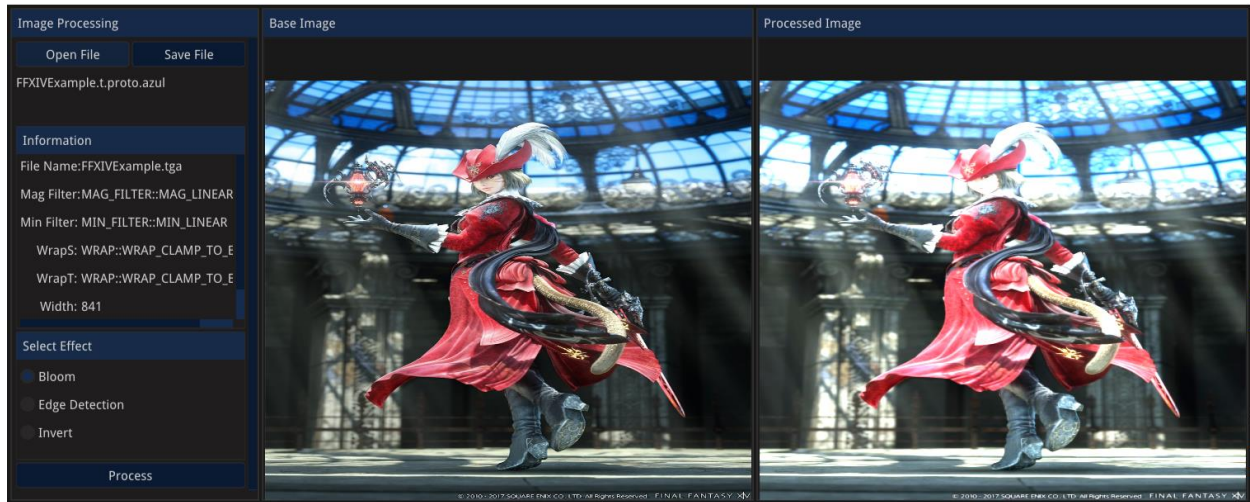
```
layout(binding = 0) uniform sampler2D img_input;  
layout(binding = 1, rgba32f) uniform image2D img_output;
```

This enables me to use any image as input and the output will be set to rgba32f

Granted some adjustments were needed aside from that, instead of calling imageLoad I had to now call texelFetch to get the specific pixel I needed.

```
vec4 color = texelFetch(img_input, pos,0) * weight[0]; //get the pixel at that position and apply the base weight
```

Results of Bloom



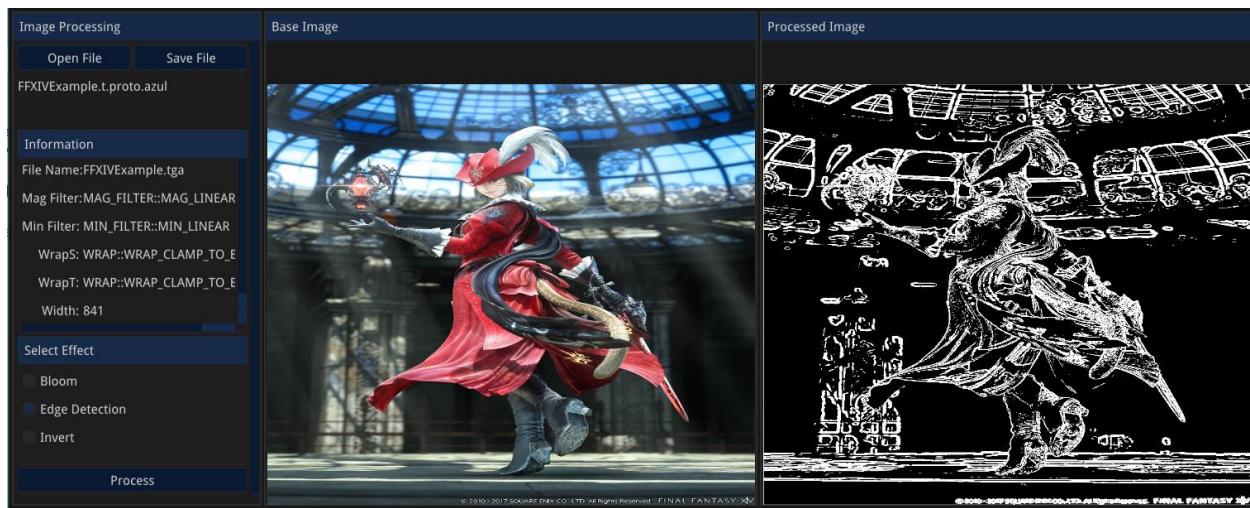
With this step completed the next step was to work on the edge detection as well as the invert effect

Edge Detection

For this shader I consulted a really good programming cookbook

[OpenGL 4 Shading Language Cookbook - Third Edition by David Wolff](#)

Chapter 6 of this book goes step by step into creating a basic Edge Detection compute shader, with the basics of my implementation already done I used this shader and it worked surprisingly well.



Invert

The invert compute shader was very straightforward

```
layout(local_size_x = 1, local_size_y = 1) in;

layout(binding = 0) uniform sampler2D img_input;
layout(binding = 1, rgba32f) uniform image2D img_output;

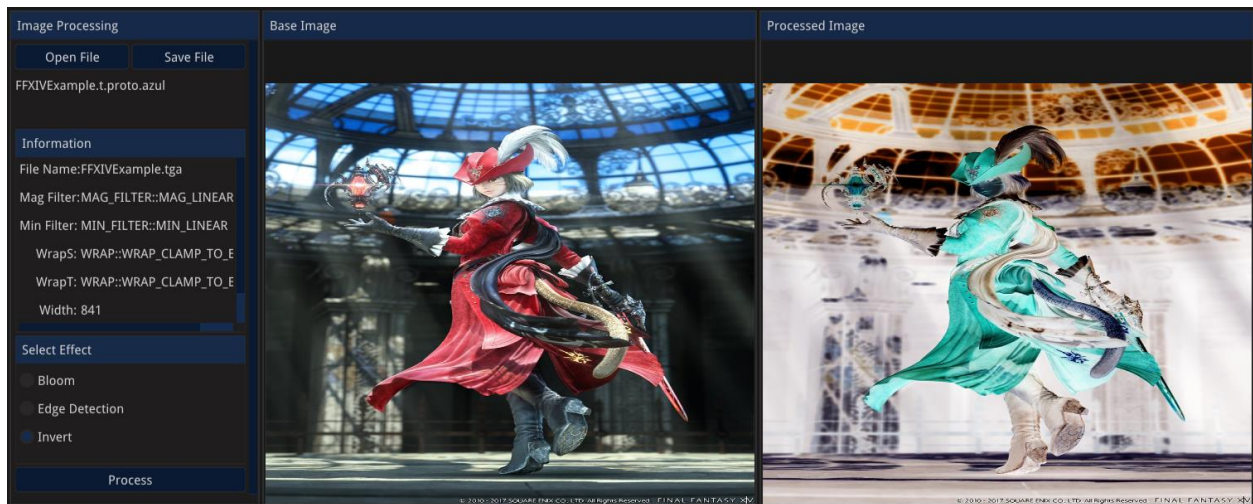
void main()
{
    // Acquire the coordinates to the texel we are to process.
    ivec2 texelCoords = ivec2(gl_GlobalInvocationID.xy);

    // Read the pixel from the input texture.
    vec4 pixel = vec4(1 - (texelFetch(img_input, texelCoords, 0)).rgb, 1.0);

    // Now write the modified pixel to the second texture.
    imageStore(img_output, texelCoords, pixel);
}

// --- End of File ---
```

And the result is this



Export to TGA

The next big step was to export the texture into TGA format

My idea was to take the texture id of the new image and use `glGetTexImage` to retrieve the data and store it in a file. I used my `SaveFile` method created for my `GDConverter` project to select the folder I wanted to save to and depending on the choice the user selected I would append the effect name to the file. This worked well, but the real problem was handling the data itself. I started by binding the texture I was about to retrieve, then creating a buffer to store the data I needed. After performing `glGetTexImage` I would get the data into the buffer, so far so good.

The next step was to create a header for the TGA format.

I found a basic TGA header and tried it.

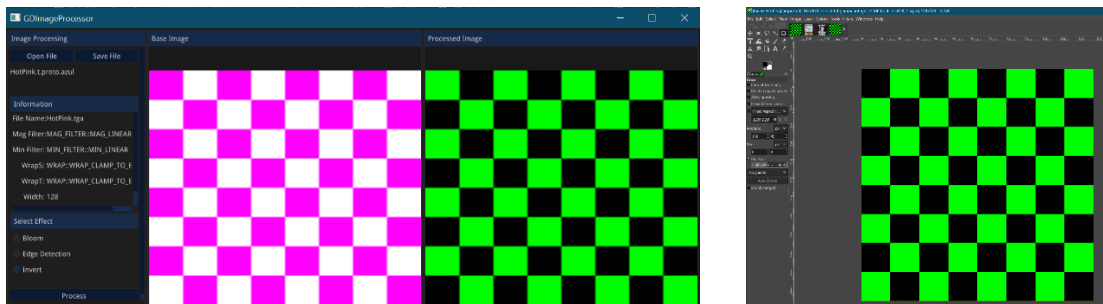
```
short header[] = {0, 2, 0, 0, 0, 0, (short) windowHeight, (short) windowWidth, 24};
```

I used my custom File Library and wrote the Header and the Databuffer into file

```
error = File::Write(fh.outputHandle, buffer, &header, inSize: sizeof(header));  
assert(error == File::Error::SUCCESS);  
error = File::Write(fh.outputHandle, databuffer, inSize: size);  
assert(error == File::Error::SUCCESS);  
error = File::Close(& fh.outputHandle);  
assert(error == File::Error::SUCCESS);
```

```
delete[] databuffer;  
return true;
```

Result



There is a major problem here, and it is that my output image is flipped, the reason for this is because while OpenGL's coordinate systems starts with its origin (0,0) being bottom left, TGA (and a lot of other systems) use top left as the origin it was only after reading [the TGA format specification doc](#) that I managed to figure out that to flip it, I needed to specify the image descriptor byte to suit this requirement.

```
17 | 1 | Image Descriptor Byte.  
| | | Bits 3-0 - number of attribute bits associated with each  
| | | pixel.  
| | | Bit 4 - reserved. Must be set to 0.  
| | | Bit 5 - screen origin bit.  
| | | 0 = Origin in lower left-hand corner.  
| | | 1 = Origin in upper left-hand corner.  
| | | Must be 0 for Truevision images.  
| | | Bits 7-6 - Data storage interleaving flag.  
| | | 00 = non-interleaved.  
| | | 01 = two-way (even/odd) interleaving.  
| | | 10 = four way interleaving.  
| | | 11 = reserved.  
| | | This entire byte should be set to 0. Don't ask me.
```

I then decided to create my own TGAHeader struct to handle this

```
struct TGAHeader
{
    byte ImageIdentificationLength = 0; //0
    byte ColorMapType = 0; //1
    byte ImageType = 0; //2
    byte ColorMapOrigin = 0; //3 - 4 //this should be short but for some reason it doesnt work
    short ColorMapLength = 0; //5 - 6
    byte ColorMapEntrySize = 0; //7
    short xOrigin = 0; //8 - 9
    short yOrigin = 0; //10 - 11
    short Width = 0; //12 - 13
    short Height = 0; //14 - 15
    byte PixelSize = 0; //16
    byte ImageDescriptor = 0; //17 0x20 for BottomLeft Origin
};
```

With this done I tested and still had some issues, and I found that the ColorMapOrigin while it says in the document that it should be 2 bytes it was not working until I set it to be one byte, still looking into why this gave me issues.

And now my export WriteToTGA function looks like this




```
File::Handle outputHandle;
File::Error error = File::Open(&#h;outputHandle, #fileName:outputFile, File::Mode::WRITE);
assert(error == File::Error::SUCCESS);

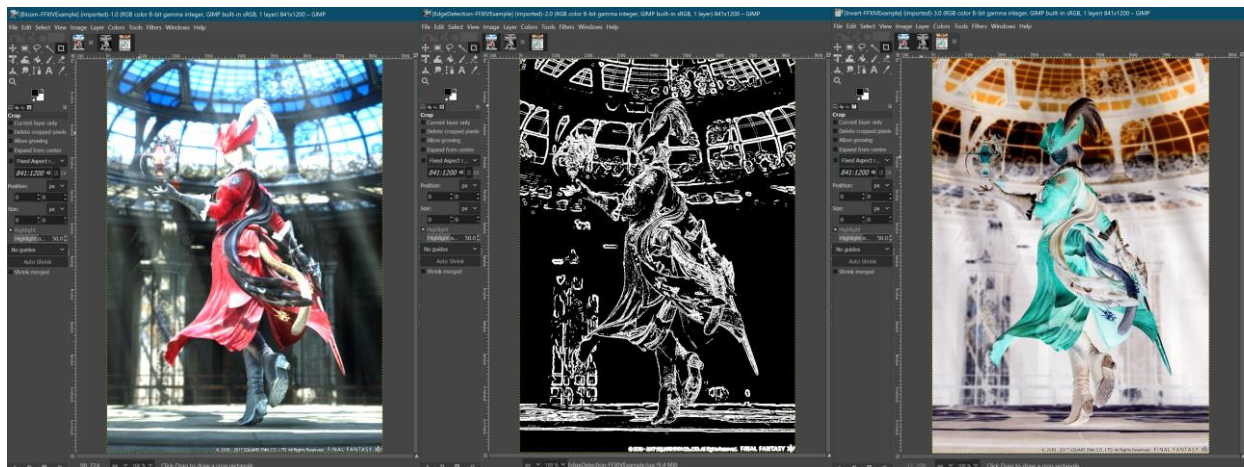
TGAHeader header;
header.ImageType = 2;
header.Width = windowWidth;
header.Height = windowHeight;
header.PixelSize = 24;
header.ImageDescriptor = 0x20;

error = File::Write(&#h;outputHandle, buffer:&header, inSize:sizeof(TGAHeader));
assert(error == File::Error::SUCCESS);
error = File::Write(&#h;outputHandle, databuffer, inSize:size);
assert(error == File::Error::SUCCESS);
error = File::Close(&#h;outputHandle);
assert(error == File::Error::SUCCESS);

delete[] databuffer;
```

The result of the Export Function

Name	Date modified	Type	Size
 Bloom-FFXIVExample.tga	8/7/2022 1:38 PM	GIMP 2.10.32	2,957 KB
 EdgeDetection-FFXIVExample.tga	8/8/2022 9:55 PM	GIMP 2.10.32	2,957 KB
 Invert-FFXIVExample.tga	8/7/2022 12:41 PM	GIMP 2.10.32	2,957 KB



Post-Mortem

Had I more time with this project, I would try to implement Canny Edge Detection instead of regular Edge Detection. Also working with UI frameworks made me start writing up my own wrapper classes to make it easier to implement features into the UI (like sliders to set location of elements, etc...) I enjoyed working with this project, mostly because I wanted to use compute shaders for something other than skinning models, I think compute shaders are very powerful specially with how efficient new GPUs have been.

I'm grateful to have learned so much during this and for having such supportive advisors and colleagues. My next steps would be to redo my engine from scratch and try to simplify UI and Tool creations and maybe take a break for a bit after working so hard on my master's degree.