CUDA Lab 4. CUDA OpenGL Interoperability & Image processing

1. Learn how to load an image using CUDA SDK
2. Understand how to use OpenGL textures in a CUDA kernel
3. Learn how to edit an image by writing a kernel function
4. Understand the basic principle of smoothing an image
5. Understand how to implement the bilinear and cubic image smoothing filters.

Exercise 1. Create an OpenGL-CUDA program based on a CUDA SDK sample

1. Starting from the CUDA template program created in VS2019. Replace the default "kernel.cu" CUDA program with "ImageProcess_cuda.cpp" and "ImageProcess_cuda.cu" provided under the Week4's module section, which are the simplified version of "bicubicTexture.cpp" and "bicubicTexture_cuda.cu" provided in the CUDA Samples folder. This simple framework provides you a framework for loading an image to GPU and processing it using CUDA. If you do not have any knowledge with OpenGL API, please following the tutorial provided from the link below to gain a basic understanding how OpenGL API interoperate with CUDA: https://www.3dgep.com/opengl-interoperability-with-cuda/#Download_the_Source

2. Edit VS2019 VC++ directories
    a. Include directories
    "C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.5\common\inc"
    b. Lib directories
    C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.5\common\lib\x64
    c. Input the following filenames under the Linker folder
    "freeglut.lib" and "glew64.lib"
    d. Copy the files " freeglut.dll", "glew64.dll", "FreeImage.dll" from folder "C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.5\bin\win64\Release" to the folder containing your compiled program.
3. Download the test image "lena_bw.pgm" to place it in your project folder.
4. Compile and observe if the project can be compiled and run properly.

Exercise 2. Understand pixel colour

1. An image is simply a 2D array of pixels. Each pixel has a colour value which can be digitally represented as a list of numbers, depending on the data format adopted. In the framework, the colour of each pixel is represented in RGBA format using 4 integers, each of which ranging from 0 to 255. Open *ImageProcess_cuda.cu* and go to the method *d_render*( ), modify the 4 numbers shown in *make_uchar4*(…, …, …, …) in the following line:

    d_output[i] = make_uchar4(c * 0xff, c * 0xff, c * 0xff, 0);

    say,

    ```
    d_output[i] = make_uchar4(0xff, 0, 0, 0);
    ```

    and then

```
d_output[i] = make_uchar4(0, 0xff, 0, 0);

d_output[i] = make_uchar4(0, 0, 0xff, 0);
```

2. The original image is a grey value image, the pixel intensity at a pixel position at (u,v) is read using

```
float c = tex2DFastBicubic<uchar, float>(texObj, u, v);
```

where c is in [0, 1].

3. Now modify the value d_output[i] using image pixel value *c* read from image location at (u, v) with the following colour and observe how the image colour is changed.

```
d_output[i] = make_uchar4(0, 0, c*0xff, 0);
```

Exercise 3. Image transformation

As an image is simply a 2D array of colours, it can be processed by employing a collection of threads arranged in two-dimensional manner. So, each pixel can be identified directly based on the thread indices and block indices, as well as block size. Rewrite the lines on the definition of x, y and i variables in the form you are familiar with:

```
uint x = blockIdx.x*blockDim.x + threadIdx.x;
uint y = blockIdx.y*blockDim.y + threadIdx.y;
uint i = y* width + x;
```

The __umul24 (unsigned int x, unsigned int y) used in the original code is a function defined in math.h, which calculates the least significant 32 bits of the product of the least significant 24 bits of two unsigned integers.

1. Translate the image.
   a. Define a translation as a 2D vector, say
      float2 T={20, 10};
   b. Translate (x, y) with vector T:
      x +=T.x;
      y +=T.y;
   c. Read pixel colour with translated coordinates *x*, *y*:
      float c = tex2D<float>(texObj, x, y);
   d. Compile the run your program and observe if the image is translated according to your wish.
   e. Observe how the image is transformed by defining different translation vectors.
2. Scale the image
   a. Define a scaling transformation as a 2D vector, say
      float2 S= {1.2, 0.5};
   b. Scale (x, y) with vector S:
      x *=S.x;
      y *=S.y;
   c. Read pixel colour with scaled coordinates *x, y*:
      float c = tex2D<float>(texObj, x, y);
   d. Compile the run your program and observe if the image is scaled according to your wish.
   e. Observe how the image is scaled by defining different scaling vectors.

3. Rotate the image
    a. Define a rotation matrix for a certain rotation angle, say
        float angle = 0.5;
    b. Rotate (x, y) with rotation matrix defined below:

$$R = \begin{pmatrix} \cos(angle) & -\sin(angle) \\ \sin(angle) & \cos(angle) \end{pmatrix}$$

        float rx =x* cos(angle) -y*sin(angle);

        float ry =x*sin(angle) + y* cos(angle);

    c. Read pixel colour with scaled coordinates *uv*:
        float c = tex2D<float>(texObj, rx, ry);
    d. Compile the run your program and observe if the image is rotated according to your wish.
    e. Further observe how the image is rotated by defining different rotation angles.

4. Scaling by position (cx, cy)
   The scaling transformation considered above is about the coordinate origin (0, 0). Scaling image by a position (cx, cy) can be done in the following way:

```
float u = (x - cx)*scale + cx;
float v = (y - cy)*scale + cy;
```

   Show how to scale the image by its centre.

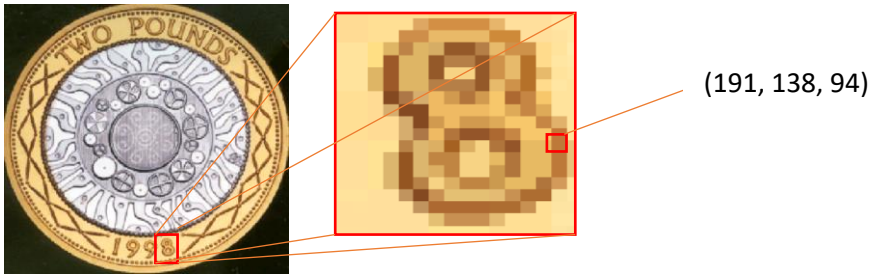
5. Rotate image by the image centre
   The rotation defined above is about the coordinate origin (0, 0). Modify the code given in step 3 to rotate the image by the image centre.
6. The following lines of code shown in the original sample code is a combination of both scaling and translation. What it does is to scale the image by location (cx, cy) and then translate the image with a vector (tx, ty):

```
float u = (x - cx)*scale + cx + tx;
float v = (y - cy)*scale + cy + ty;
```

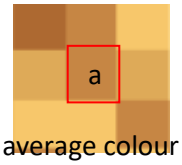
   Go the render( ) method and modify the values of tx, ty, scale, cx, cy and observe how the image is transformed..

Exercise 4. Image smoothing.

A 2D image is essentially a 2D array of colours, which can be understood as a matrix of colours. For instance, for the coin image shown below, a close-up look at the area of “8” is shown below, where each pixel element can be clearly seen.



To smooth an image, the basic idea is for each pixel, find its neighbour colours, and use the average colour to replace the original colour. For instance, the order one squared neighbour at pixel **a** contains 9 pixels



As the calculation of the average colour for each neighbouring box can be done pixel by pixel independently in parallel, CUDA can be used to solve the problem. Now modify the kernel function tex2DBicubic( ) to understand the basic principle of how an image can be smoothed.

1. For a pixel at location (px, py), replace its original colour with the average of the following five pixel colour (the original pixel colour + four neighbour pixels colours):

```
float centre= tex2D< float >(tex, px , py);
float left = tex2D< float >(tex, px - 1, py);
float right = tex2D< float >(tex, px + 1, py);
float up = tex2D< float >(tex, px, py + 1);
float down = tex2D< float >(tex, px, py - 1);
return (centre + left + right + up + down)/5;
```

2. Run your program and observe how image has been smoothed.
3. (Optional) Smooth the image using 16 squared neighbour colours.
4. Modify the thread configuration and observe the performance of your program.