Lab 4 : Creating an RTI Viewer using Fragment Shaders

Reflectance Transfer Imaging (RTI) files encode the reflectance information for a scene. Think of this as an image that supports relighting from any incident lighting angle. Play around with the **QtRTI** example provided in the Lab 4 page to get an understanding of what this implies. It loads a dataset with a coin image and allows you to relight it from different angles by using the relative positions of your mouse clicks on the window.

Your objective in Lab 4 is to implement a similar (or preferably better) viewer, but this time use the GPU (i.e. fragment shader code) to do the per-pixel relighting calculation.

What's the data format?

The coefficients for relighting (which are stored in the .rti file) are loaded by the loadHSH() method in rtiutil.h. This file also has comments for the global variables and each function which you should read before continuing.

The contents of the RTI file are returned from the loadHSH() method as a large multi-dimensional float array. The ordering of dimensions are (rtiheight * rtiwidth * bands * terms) where **bands** is the number of color channels in the image (3 for RGB, which all our datasets are) and **terms** is the number of coefficient layers in the file. This can take a value of 4, 9 or 16. Note that in the code we keep track of terms by the global variable **order** where terms = order*order *(order of the hemi-spherical harmonics function that is used for the reflectance calculation, don't worry about this though, you can compute the reflectance without knowledge of hemi-spherical harmonics)*

There is a convenience function getindex(int h, int w, int b, int o) which takes in the y and x co-ordinates of a pixel (as h, w), the color channel (as b) and the term index (as o) and returns the coefficient value for that pixel for that color channel and band. Use this function (or use it as a guide) when copying data off the multi-dimensional RTI data array to your own arrays / textures.

How is the relighting calculation done?

To understand how the relighting calculation is done, take a look at the function renderImageHSH() in rtiutil.h. This is important as this is the functionality you will be porting to the shader program.

The per-pixel, per-color channel relighting calculation can be written as,

color_value = weight[0]*coef[0] + weight[1]*coef[1] + weight[2]*coef[2] + …….. + weight[n]*coef[n]

a) **n** is the number of **terms**.
b) **coef** array represents the coefficients for that pixel and that color channel :

   For example, say we are at the pixel (x=12, y=5) , color channel 2 (green), you have a maximum of 9 terms (n = 9). We can access the respective coefficients by using getindex(12,5,2,**0**) … getindex(12,5,2,**8**).

c) **weight** array represents the weights calculated depending on the normalized lighting direction *(lx, ly, lz)*.

The weights need only to be calculated once per the whole image when the lighting direction changes and you can find the calculations at the beginning of the renderImageHSH() function.

The whole relighting calculation can be found in the innermost loop of the renderImageHSH() function. I'll quote it here,

```
// The computation for a single color channel on a single pixel
double value = 0;
// Multiply and sum the coefficients with the weights.
// This evaluates the polynomial function we use for lighting
for (int q=0;q<order*order;q++)
{
        value += hshimage[getindex(j,i,b,q)]*weights[q];
}
value = min(value,1.0);
value = max(value,0.0);
// Set the computed pixel color for that pixel, color channel
Image[j*rtiwidth*bands+i*bands+b]=(unsigned char)(value*255);
```

Here **value** is the computed **color_value** for that pixel for that color channel. The result image is calculated this way pixel by pixel by color-channel.

How can I use a shader program to do this computation?

A key feature of the relighting calculation is that it's strictly done on a per-pixel basis. There is no dependence on the relighting results of the neighboring pixels. Therefore rather than doing the computation in a linear manner on the CPU we can use the GPU to do it in parallel (and get the result much faster!).

By this point hopefully you have checked out all 3 examples QtRTI, QtES2Example and QtSimpleTexture. The QtSimpleTexture is the easiest to use as a starting point for your own code. You will also have to use multiple textures to pass information into the shader, of which a simple example is given in code. Given that, here's a step-by-step approach to the shader-based implementation.

1) Construct 2D polygon that fills the entire screen (already done for you in QtSimpleTexture)
2) Load the RTI file (an example call can be found in the constructor of GLWidget). Let the user specify the file either through command line or a 'File Open' dialog box.
3) Construct an OpenGL texture from each layer of the RTI coefficients arry. For example for a 9 term (3rd order) RTI, you will need 9 textures.
    a. Copy the respective values from the RTI float array (hshimage) to a float array of your own (of size rtiheight*rtiwidth*3) for each term.
    Note : OpenGL texture images are defined from bottom to top (i.e. 0,0 is at bottom left), whereas the RTI data (and other normal images) is defined from top to bottom (i.e. 0,0 is at

top left). Even I missed this for the QtRTI Example. Be sure to flip the data as you copy it over. It becomes evident with examples other than 'coin.rti'.

b. Construct an OpenGL texture from each float array. Your glTexImage2D call would be similar to

```
glTexImage2D(GL_TEXTURE_2D, 1, GL_RGB32F_ARB, rtiwidth, rtiheight, 0,
GL_RGB, GL_FLOAT, firsttermarray);
```

Where **firsttermarray** represents the float array containing the coefficients for the first term. We need as many of these as there are terms. (In the given example 9)

The QtSimpleTexture gives an example with a single channel array (which uses GL_LUMINANCE instead of GL_RGB).

Note : For the assignment, you need to be able to load 3rd order (9 term) RTIs. However, you may find it easier to get started with the vase_2ndorder.rti sample RTI file which only has 4 terms.

4) Pass the textures into the shader. Once again the code provides an example with two textures. Be sure to activate each texture by using glActivateTexture.
5) Once the user changes the lighting direction (it's up to you on how to implement this interactively, use the simple mouse position / an hemi-sphere on the side of the main area / etc) recalculate the **weights .** Use the first section of renderImageHSH() as your reference.

   The lighting direction vector (lx, ly, lz) is defined over the upper-hemisphere, and needs to be normalized before being passed to the renderImageHSH(). Specifying either a non-normalized vector / having a direction in the lower hemisphere will result in lighting values that are out of range.

6) Pass the weights into the shader as Uniform variables.
7) Inside the fragment shader, carry out the weights*coefs multiplication and sum them up for each color channel. Assign the result to the gl_FragColor!

That's it! You should now have your GPU based RTI viewer.

Troubleshooting

1) If your shader program fails to compile, you'll simply get a blank opengl window. Go back to QtCreator and checkout the 'Application Output' window to figure out what the error is.
2) If you're having problems with OpenGL / shader support check out the related forum post. You need OpenGL 2.0 or above to have programmable shaders.
3) It may be easier to work with small textures first (like in QtSimpleTexture) where you can understand what's going on. You can selectively copy a few pixels from the large RTI for this

purpose. Construct your multi-texturing setup and make sure everything makes sense before using the full layer as your texture.