University of Tübingen
Prof. Dr.-Ing. Hendrik P.A. Lensch
Chair of Computer Graphics
Faezeh Zakeri (faezeh-sadat.zakeri@uni-tuebingen.de)
Lukas Ruppert (lukas.ruppert@uni-tuebingen.de)
Raphael Braun (raphael.braun@uni-tuebingen.de)
Andreas Engelhardt (andreas.engelhardt@uni-tuebingen.de)
Philipp Langsteiner (philipp.langsteiner@student.uni-tuebingen.de)

31. January 2023

Computer Graphics
Assignment 12

Submission deadline for the exercises: 07. February 2023, 0:00

Source Code Solutions

- Upload **only** the files that you changed.

  Do NOT include your build folder, results, meshes, ext, or exercise sheet in the zip file you upload.

- Zip them first and upload the **.zip** file on ILIAS.

12.1 Spectral Images (20 + 20 + 10 + 10 + 10 = 70 Points)

When capturing multispectral (or hyperspectral) images, the result is one grayscale image per captured wavelength, e.g. one image per each wavelength in the visible spectrum (400nm-700nm) at a 10nm interval. A dataset of such images can be found here: https://www.cs.columbia.edu/CAVE/databases/multispectral/.
In order to display such an image more naturally on a regular RGB display, we can convert the image into RGB if we consider the contribution of each wavelength to the RGB color channels.
Measured data necessary for such a conversion is available if we take an intermediate step through the XYZ color space. For the final display, we also need to apply gamma correction to the linear RGB values to produce the correct input for a low dynamic range display.

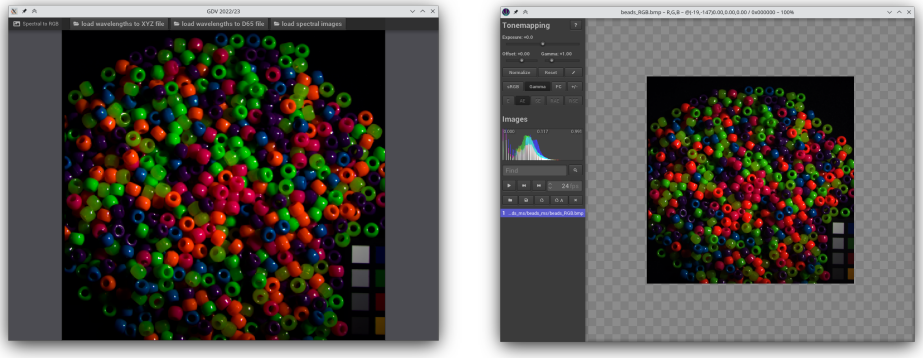

Figure 1: Spectral to RGB conversion result. The reference image needs to be viewed without further gamma correction, i.e. using gamma 1.0, e.g. in TEV.

a) First, we need to load the conversion table from spectral intensity to XYZ.

Download the tables for the CIE XYZ color matching functions (CMF) from http://cvrl.ioo.ucl.ac.uk/index.htm. Select the *NEW CIE XYZ (from LMS) functions* and download the *2-deg XYZ CMFs transformed from the CIE (2006) 2-deg LMS cone fundamentals* with 5 nm stepsize. The downloaded file contains the response of the three types of human photoreceptors on certain wavelengths.

Implement the function

```
void SpectralToRGB::loadWavelengthToXYZ(const std::string_view filename)
```

To load the CSV file, parse the XYZ values per wavelength and store them in

```
std::map<uint32_t, Vector3D> wavelengthToXYZ;
```

For parsing the CSV file, you can use `std::ifstream` (especially its `operator>>`), or alternatively the C-style `fscanf`.

To add values to a `std::map`, you can `insert` a pair of values, e.g. `wavelengthToXYZ.insert({400, Vector3D{0.02214302f, 0.002452194f, 0.109609f}})` or simply use the `operator[]`, e.g. `wavelengthToXYZ[400] = Vector3D{0.02214302f, 0.002452194f, 0.109609f}`.

b) In order to properly reconstruct the image, we also need to account for its illumination. For the CAVE dataset, all scenes are illuminated with the CIE standard illuminant D65.

Like for the wavelength to XYZ conversion, you will have to download the *CIE_std_illum_D65.csv* file and read its contents into the

```
std::map<uint32_t, float> wavelengthToD65;
```

Implement this in

```
void SpectralToRGB::loadWavelengthToD65(const std::string_view filename)
```

Here, every wavelength has a single value which describes how bright a white pixel should be. We can make use of this information to normalize the spectral images to the correct brightness. Without this step, some images might be too dark due to the absorption by the spectral filters applied to the camera.

The code for computing the normalization is already given in the template. The normalization will be applied once you click on any pixel in the image. You should select the center of the brightest gray pixel in the color chart to normalize it to 0.5. (Implement the remaining tasks first).

c) Next, we need to convert a set of greyscale images per wavelength to one image in XYZ color space.

Implement the function

```
Texture SpectralToRGB::computeXYZTexture(Pixel referencePoint) const
```

where you are given a `std::vector` of monochrome images along with their wavelengths via the class member variable

```
std::vector<std::pair<uint32_t, Texture>> textures;
```

The result should be another `Texture` with the same resolution containing `Vector3D` values per pixel that sum up the monochrome images multiplied by the matching XYZ response for their associated wavelength.

The given template already initializes the result texture to 0 at every pixel and computes a scale factor `scale` that normalizes the result image.

You need to multiply that `scale` onto every pixel you read from the input.

d) From XYZ, we next need to convert to linear RGB values. This is a simple linear transformation expressed by the following matrix (see https://color.org/chardata/rgb/sRGB.pdf):

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} +3.2406255 & -1.537208 & -0.49862686 \\ -0.9689307 & +1.8757561 & +0.0415175 \\ +0.0557101 & -0.2040211 & +1.0569959 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

After the conversion, RGB values less than 0 or greater than 1 need to be clipped to 0 and 1.

Implement the function

```
Vector3D SpectralToRGB::xyzToLinearRGB(Vector3D c)
```

to perform the conversion. You can use a `Matrix3D` to do the conversion. Note that it is constructed from column vectors.

e) Finally, we need to apply the conversion from linear RGB to sRGB with gamma 2.2:
For each channel $C \in \{R, G, B\}$,

$$C_{\text{sRGB}} = \begin{cases} 12.92 C_{\text{linear}}, & \text{if } C_{\text{linear}} \leq 0.0031308 \\ 1.055 C_{\text{linear}}^{1/2.4} - 0.055, & \text{otherwise} \end{cases}$$

Implement the function

```
Vector3D SpectralToRGB::linearRGBToSRGB(Vector3D c)
```

to perform the conversion accordingly.

Once all is implemented, you should be able to use the GUI to first load the CSV file for wavelength to XYZ conversion, the CSV file for wavelength to D65 conversion, and then load a set of spectral images. (The GUI will assume that the first image has a wavelength of 400nm and all following images have the following wavelengths at 10nm increments – sorted by filename). The sRGB result should then be displayed in the GUI. Click on the brightest pixel in the color chart to normalize the image.
You should get similar results to the reference images provided along with the CAVE dataset, e.g. Figure 1. However, note that you have to open their reference image in an image viewer that allows you to turn gamma down to 1.0. Otherwise, you will most likely see their result with gamma correction applied twice (it will appear much brighter).

12.2 Post Processing: Gaussian Blur (30 Points)

Your final task is to add post processing to the OpenGL rendering pipeline.
The given code has already been extended to render into a framebuffer instead of rendering directly into the screen. Additional off-screen framebuffers can now be used to post-process the image to apply screen space effects to it. Here, we want to add some blur for out-of-focus objects in the distance.
To get started, we use a framebuffer at half resolution. That way we have a lot less pixels to process and already blur the image quite a bit. Then, we iteratively blur the image with a 2D Gaussian blur. To further reduce the necessary computation, we will alternate between blurring horizontally and vertically (see Figure 2): We separate the 2D Gaussian blur into a horizontal and a vertical Gaussian blur. Now, rather than evaluating all pixels in a 9×9 square centered around each pixel, we only need to evaluate a 9×1 pixel rectangle and a 1×9 pixel rectangle (once horizontally and once vertically) to compute the same values.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \cdot \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{y^2}{2\sigma^2}} = G(x) \cdot G(y) \tag{1}$$

A similar approach is typically taken to apply Bloom effects, where only the bright parts of the image are filtered.
For a 1D Gaussian blur with a footprint of 9 pixels centered around position $p = x$, we use the following weights w to compute the filtered/blurred result:

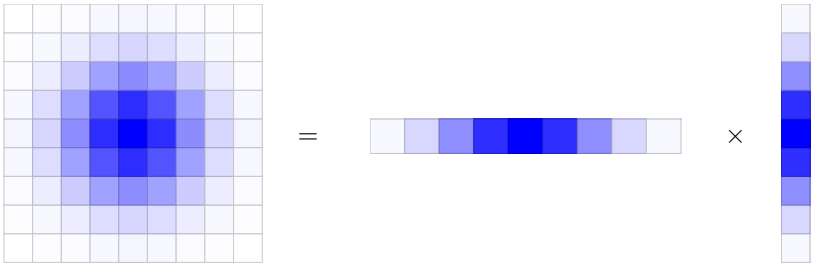

Figure 2: The 2D Gaussian blur can be separated into horizontal and vertical 1D Gaussian blurs.

p	$x-4$	$x-3$	$x-2$	$x-1$	x	$x+1$	$x+2$	$x+3$	$x+4$
w	0.016216	0.054054	0.1216216	0.1945946	0.227027	0.1945946	0.1216216	0.054054	0.016216

Note that these weights are symmetric around the center pixel x, and approximately sum up to 1.
To further optimize the computation of the filtered image, we can make use of the bilinear interpolation we are essentially getting for free in every `texture` lookup. We can combine the lookup for two neighboring pixels each by setting the lookup location in between the pixels relative to their filter weights. (This approach is known as linear sampling.)
For the center pixel at position x, we will still do a lookup at its center and keep its weight at 0.227027.
For the pixels at $x \pm 1$ and $x \pm 2$, we can use the combined lookup position

$$\pm\left(1 + \frac{0.1216216}{0.1945946 + 0.1216216}\right) \approx \pm 1.384615 \tag{2}$$

and combined weight $0.1945946 + 0.1216216 = 0.3162162$.
And for the pixels at $x \pm 3$ and $x \pm 4$, we can use the combined lookup position

$$\pm\left(3 + \frac{0.016216}{0.054054 + 0.016216}\right) \approx \pm 3.230767 \tag{3}$$

and combined weight $0.054054 + 0.016216 = 0.07027$.
So, for performing a 9×9 Gaussian blur, we only need to perform 10 texture lookups (5 for horizontal blurring, 5 for vertical blurring) per pixel as opposed to 81 in a naïve implementation.

Your task is to implement this separable Gaussian blurring approach in a fragment shader:

```
1  const std::string BlurShader::fragmentShaderBlur
```

Sum up the center pixel and its (`horizontal`ly or vertically) neighboring four pixels to each side using the lookup positions and weights given above relative to the center position already given `vec2 texCoord`. The size of each pixel in normalized coordinates ($(x, y) \in [0, 1)^2$) is given as `vec2 pixelSize`. As mentioned above, you should perform exactly 5 `texture` lookups per fragment.

The remaining code of the updated framework will apply your filtering code repeatedly and `mix` it with the unfiltered image to produce the final result using the new `CompositionShader`. The slider "blur" in the GUI (see Figure 3) allows you to choose how many blur iterations to perform and should thus result in a sharper or blurrier appearance of distant objects. The focus plane from which on blurring starts to take effect is based on the distance between the camera position and its target and can also be adjusted freely.
Note: If you're using a version of OpenGL that has troubles with running the geometry shader (e.g., on Mac), try using the "Simple" shader insted of the "PBR" shader. It only uses vertex and fragment shaders to compute a more basic shading of the scene but blurring should still work.

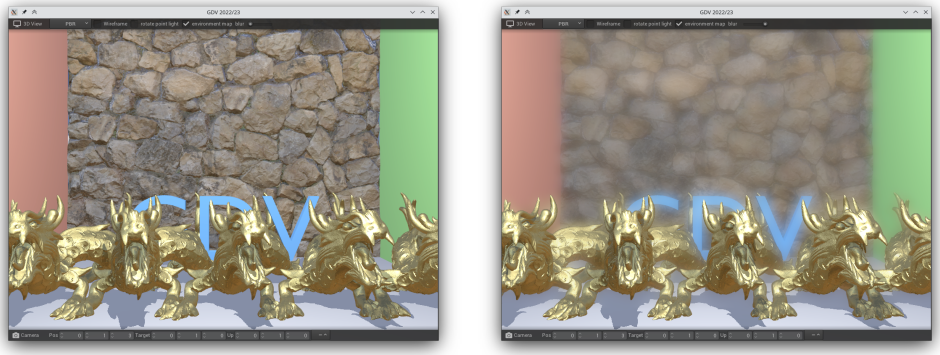

Figure 3: A scene rendered without and with depth-based blur. XYZ RGB Dragon model from the Stanford 3D Scanning Repository (mesh decimated in Blender).