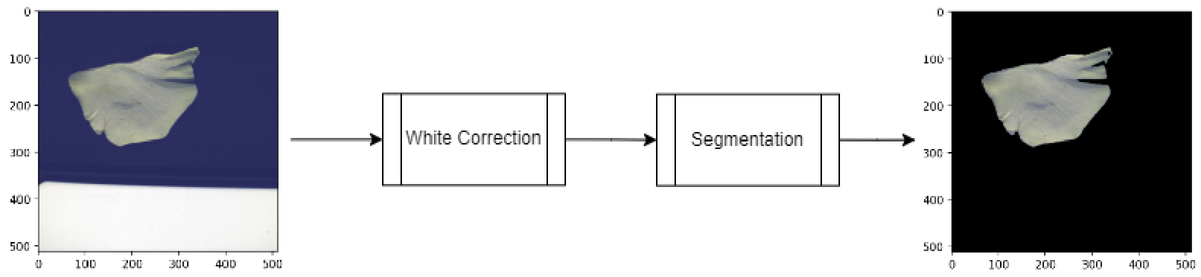
Data Acquisition:

A hyperspectral image database of flower petals was built using the SpecimIQ. Each captured image has a size of 512 x 512 x 204 and their information are stored in “hdr” and “raw” files.

Data Preprocessing:

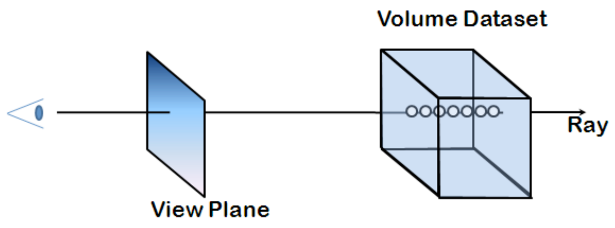
Firstly, the hyperspectral image was white corrected, thereafter, the petal was segmented from the hyperspectral image **by selecting only spectrums similar to the petal’s spectra**.



The resulting hyperspectral cube was then stored as a grid of textures in a single png image file. This was due to WebGL 1.0’s limitations, it’s impossible to store the textures of hyperspectral image as an array of textures which can be passed to a shader. The only way we could handle this was to store all the textures as a single image file which could then be passed and sampled in a shader. Besides, storing all the hyperspectral image as a single png image file reduces the number of images loaded and speedup the web application loading time.

Rendering hyperspectral image as a volume in web application:

The hyperspectral image was rendered on a web application by implementing a volume rendering algorithm known as the ray casting algorithm in the fragment shader of a volume (cube). The basic idea behind this algorithm involves tracing rays from the camera into the cube, and computing rendering integral along the rays as shown below.



For each fragment that would be rendered on the screen, a ray is cast from the camera into the cube. The hyperspectral image data corresponding to the fragment position are then sampled at discrete positions along the ray. The accumulation of these data properties (color and opacity) gives the final property of the fragment.

Implementing this algorithm in WebGL to render the hyperspectral image as a volume consists of two main steps (also two main fragment shaders).

1. First rendering pass:
   In this step, we store the fragment’s world space coordinate as the fragment’s output. Thereafter, the resulting texture from this process is then passed through a second shader for the second rendering pass process.

2. Second rendering pass:
   The implementation of the volume rendering algorithm was done at this stage. The vertex shader of this process outputs two coordinates. The world space coordinate which was used as the entry point of the ray, and the projected coordinate which was used to sample the texture from the first rendering pass (back face coordinate of the cube).

   In the fragment shader of this process, firstly, the ray exit point and direction were computed. Thereafter, the ray travels through the cube along its direction for a certain step size. For each step, the hyperspectral image texture was sampled, thereafter, a transfer function (for color), alpha and gamma correction were applied on the resulting intensity value. The final color from each step were then accumulated, and the resulting accumulated color is then returned as the color the fragment.

During this rendering process, in order to differentiate each wavelength from another, visible wavelength colors were used in place of the transfer function. Also, in order to view the inner structure of the rendered petal, clipping parameters that clips the model on the spatial and spectral direction were added to the implementation.

References

[1] L. R. Barbagallo, "WebGL Volume Rendering made simple," [Online]. Available: http://web.archive.org/web/20160402041316/http://www.lebarba.com/blog/. [Accessed May 2022].

[2] R. Marques, P. S. Luís , L. Peter and P. Céline , "GPU Ray Casting," 2009.