UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
DIM0451 - COMPUTAÇÃO GRÁFICA

PROF. BRUNO MOTTA DE CARVALHO

Projeto Ray Tracer - 1ª Etapa - **Trabalho em grupo com 2 participantes**
Entrega: 28/04/2022 Horário: 11:59 Local: Entrega eletrônica via SIGAA

Instruções para entrega do trabalho: O programa deve ser salvo em um arquivo com o nome `raytracer.c`. Faça o *upload* no SIGAA antes do horário limite indicado acima.

Ray Tracer - Phase 1

In this programming project you will begin writing your own ray tracer. We've learned from the lessons so far that a basic ray tracing algorithm works by shooting rays through pixels of an image into a scene; every time a ray hits something in the scene, the algorithm triggers some form of shader (aka reflectance model) to determine the color of that pixel. By shader we mean another class or procedure, physically based or otherwise, that might shoot other rays, stop on the first hit (i.e. a ray casting), compute the contribution of the lights in the scene, for example, to determine and return a rgb color (see Figure 1, from Chapter 4 of "Fundamentals of Computer Graphics", 3rd ed., Shirley and Marschner).

At this point, several of the ray tracing components mentioned above are to be considered black boxes. For now, our major concern is to understand the "big picture" and lay down the foundations of our ray tracing, which, in turn, will be extended and experimented with along the next projects.

Requirements

We'll start with the basic elements of a ray tracer, namely:

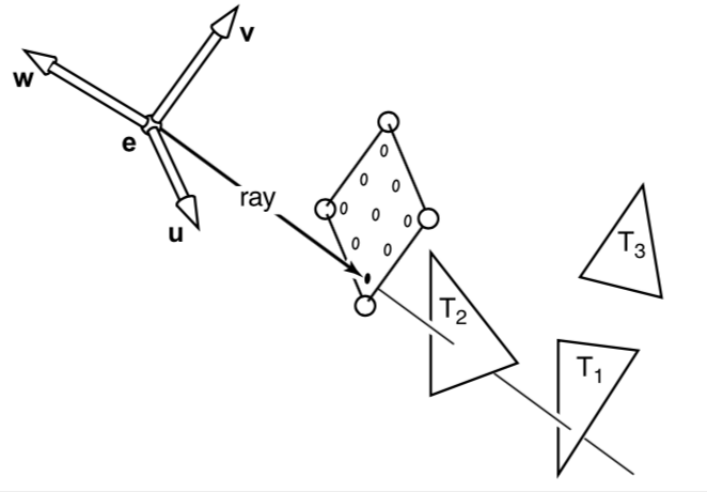

Figura 1: Basic work of a Ray tracer.

1. A class `Film` that stores pixels values as an image color buffer (a matrix). This allows the ray tracer to save the color buffer to an image file in PPM or PNG format. The class is named `Film` because it plays a role similar to a film in an analog camera, or sensor in a modern digital camera.

2. A class `Background` that is responsible for returning a color each time the primary ray misses any object in the scene (i.e. hits nothing). In this project, this class will receive a raster coordinate (i, j) and return the corresponding background color.

3. A set of classes to handle the math operations on vectors and matrices. In this case you might either implement your own library (`Vector3`, `Point3`, `Mat3x3`, etc.) based on the one provided in "Ray Tracing in One Weekend", or adopt other math libraries such as **OpenGL Mathematics**, or **cyCodeBase**.

4. The API class which is a singleton that can be coded as a static class that will instantiate and keep track of all object that we will need to run the ray tracer. In this project, for instance, this class should hold an instance of Background, and Film, and provide a method `render()` in which resides the main loop described next.

5. The main loop of the ray tracing algorithm, which should traverse the

image pixels and shoot rays into the scene. At this stage, the main loop only traverses the image and samples colors from the Background object. (No rays are shot just yet.)

6. A set of functions that parses, i.e. reads in and validates a scene description file in **XML** with the format explained below, extracts the information necessary to set up the other classes, and sends these to the API class via its static methods.

The suggested flux of information for the project inside the `main()` function is the following:

1. You program reads the command line arguments and stores them into a `struct RunningOptions`. This struct may be defined in a globally visible header file, such as `rt3.h`. The command line arguments are:

```
Usage: rt3 [<options>] <input_scene_file>
    Rendering simulation options:
        --help                      Print this help text.
        --cropwindow <x0,x1,y0,y1> Specify an image crop window.
        --quick                     Reduces quality parameters to render image quickly
        --outfile <filename>        Write the rendered image to <filename>.
```

   - `input_scene_file`: Name of the input scene file. [required]
   - `cropwindow`: this arguments defines a sub-window to be rendered instead of the full image. [optional]
   - `quick`: this is a Boolean flag that triggers the rendering of a image with 1/4 of it original resolution. [optional]
   - `outfile`: Name of the image output file (this overwrites the file informed in the XML scene file). [optional]

2. Calls the method `API::init_engine(const RunningOptions &opt)`. This method stores the running options and initialize the API's internal states (graphics state). [Not necessary in this project.]

3. Calls the method `API::run()`. This method just calls the the static function parse(`opt.file_name`). This function, in turn, does the following:

(a) Traverses the XML file, identifying each possible tag available in RT3;

(b) Collects each the tag's attributes and packing them into a bundle (a `ParamSet` class object);

(c) Sends the bundle object (parameters) to the proper API static method. Ex.: `API::background(...)`, and;

(d) When it finds the `world_end` tag it calls the `render()` method that implements the classical ray tracing loop.

4. Calls the method `API::clean_up()`, this frees all the resources previously allocated.

The Scene Format

Here we provide a basic XML format for describing the scene. Keep it in mind that upcoming projects will have additional tags that we will define when we need them. Here is an example scene file:

```
<RT3>
<camera type="orthographic" />
<film type="image" x_res="200" y_res="100" filename="simple_bkg.ppm" img_type="ppm" />
<world_begin/>
<background type="colors" color="153 204 255"/>
<world_end/>
</RT3>
```

The output may be find here. All elements are declared inside a RT3 tag. You might want to replace this tag with you ray tracer program name. At this stage, we only have a few tags, `background`, `camera`, `film`, `world_begin`, and `world_end`. Later on we will add `object`, `light`, `material`, etc. In this project, however, you may simply ignore the camera tag since we are not creating cameras yet.

In the example above, we (implicitly) asked for a solid background whenever a single color is provided (in this case, a light blue sky) for the entire background, via the color attribute. However, if the scene file provides 2 or more colors, it means your ray tracer needs to interpolate the colors to generate the background.

It is possible to specify a color for each of the four corners of the screen, as follows:

- `bl` - bottom-left corner of the screen;
- `tl` - top-left corner of the screen;
- `tr` - top-right corner of the screen, and;
- `br` - bottom-right corner of the screen.

You may assume a black color (RGB=(0,0,0)) for any unspecified/absent corners in the scene file, since we always need 4 colors. Given these color, your ray tracer should apply bilinear interpolation when sampling the background. You may want to provide a method with the following prototype:

```
rgb Background::sample( float i, float j );
// or
rgb Background::sample( const Point2f& raster );
```

where `rgb` means a color in the three-dimensional RBG space, and i and j are values in [0,1] corresponding to the normalized location of the pixel (i, j) in the color buffer. Here is another example, this time asking for a background generated through interpolation.

```
<RT3>
<camera type="orthographic" />
<film type="image" x_res="800" y_res="600" filename="interpolated_bkg.png" img_type="png"
<world_begin/>
<!-- This defines an interpolated background -->
<background type="colors" mapping="screen" bl="0 0 51" tl="0 255 51" tr="255 255 51" br="2
<world_end/>
</RT3>
```

This scene would generate the image shown on Figure 2. Note that the RGB colors components are specified as integers in the range [0,255]. However, they also may be specified as a real number in the range [0.0,1.0]. You program should handle both representations. The attribute mapping has the default value screen, which means the background is mapped to the pixels on the screen

Figura 2: Interpolated background.

window. An alternative mapping is spherical, in which the background is mapped ("glued") to a sphere, and the rays are shoot from the center of this sphere: This is called *spherical reflection map*. This type of mapping is particularly useful when you need to create a realistic surrounding for your scene based on a spherical image, which will make more sense when your ray tracer starts shooting rays, in the next project. Next, we provide a simple version of the main loop illustrating a possible usage of (some of) these classes together.

```
// [1] Parser and load scene file
// [2] Instantiate the Camera, Film, and Background objects.
// [3] Enter the ray tracing main loop, because parser has found the `world_end` tag.
int render() {
        // Perform objects initialization here.
        // The Film object holds the memory for the image.
        // ...
        auto w = camera.film.width(); // Retrieve the image dimensions in pixels.
        auto h = camera.film.height();
        // Traverse all pixels to shoot rays from.
        for ( int j = 0 ; j < h ; j++ ) {
                for( int i = 0 ; i < w ; i++ ) {
                        // Not shooting rays just yet; so let us sample the background.
```

```
                        auto color = background.sample( float(i)/float(w), float(j)/float(
                        camera.film.add( Point2{i,j}, color ); // set image buffer at posi
                }
        }
        // send image color buffer to the output file.
        camera.film.write_image();
}
```

Assignment Tasks

Your task in this assignment is to implement all the classes described in this document, and to produce a program capable of rendering the same images showed earlier based on the corresponding scene files.

Recommendations

Rather than updating (overwriting) the ray tracer program as we progress through the upcoming individual projects, I recommend that you create a separate folder for each new project. I understand that this will cause code replication, but the purpose of this recommendation is to make it easy to keep track of the individual steps throughout the semester. This, in turn, will allow me to provide progressive grading for the intermediate steps each time a due project is delivered, while keeping an easy-to-access evolution timeline of your ray tracing project over the semester.