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: 12/05/2022 Horário: 23:59 Local: Entrega eletrônica via SIGAA

Instruções para entrega do trabalho: Os arquivos contendo o código do projeto devem ser salvos em um diretório com o nome `RT3`. Faça o *upload* no SIGAA antes do horário limite indicado acima.

Ray Tracer - Phase 2 - Rays and Cameras

1 Introduction

In this programming project you will expand your ray tracing project, adding rays and a camera.

At the core of a ray tracer (RT), as the name implies, is the entity ray. A ray encodes information of a line segment, which usually originates at the eye (or origin) of a camera with direction determined by the grid of pixels at the image plane. In other words, the RT shoots rays through pixels of an image into a scene. Therefore, the role of the camera in a RT is to generate the rays that must interact with the scene.

The camera model we need to incorporate to the RT follows the *pinhole camera model*, with the difference that the image plane is placed in front of the eye of the camera, rather than behind it. This change does not alter the final result and makes the implementation easier.

If we want to expand the possibilities of images that our RT might create, it is important to offer different types of cameras. The cameras we need to implement are perspective and orthographic. Both cameras are projective cameras, with the perspective camera shooting rays from a single point to every pixel area into

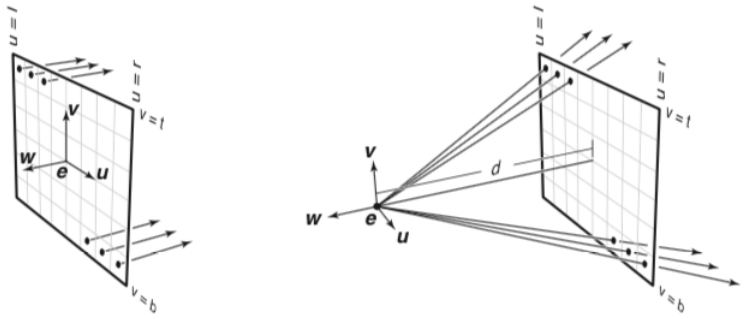

Figure 4.9. Ray generation using the camera frame. Left: In an orthographic view, the rays start at the pixels' locations on the image plane, and all share the same direction, which is equal to the view direction. Right: In a perspective view, the rays start at the viewpoint, and each ray's direction is defined by the line through the viewpoint, **e**, and the pixel's location on the image plane.

Figura 1: Schematics of the two different projection types used, parallel (left) and perspective (right) for a right-handed coordinate system.

the scene, whereas the orthographic shoots rays from each pixel to a direction (usually) perpendicular to the image plane into the scene.

The Figure 1 shows the two types of projective cameras describes so far (original source: Shirley 2009).

Another important decision we have to make is regarding the camera handedness. We are following the left-hand rule to determine the camera's frame. Note, however, that the frame of the cameras shown in Figure 1 follow the *right-hand rule*. Because we have adopted the *left-hand* rule instead, the $\hat{\mathbf{w}}$ should be pointed in the other direction, towards the image plane.

Usually the camera is positioned at the origin looking down at the $\mathbb{Z}^+$ axis. Later on we will introduce the tools to move the camera around. The same mechanism, called geometric transformations, shall be used to place and/or modify other components of the scene in an upcoming project.

2 Raster Image

Before we continue our journey to find the right math machinery to produce rays, it is important to understand the concept of **raster image**. Recall from

our previous classes that the rendering equation is a mathematical model that, if solved for a given point on a virtual image $(x, y) \in \mathbb{R}^2$, it yields a color value for that point. However, we need to sample this continuous (solution) domain so that we may produce pixel-based images, which is called a raster image. This means that a pixel value produced by the (estimation of) the rendering equation is a local average of the color of the virtual image, and it's called a point sample of an image. So, a value v of a pixel is the value of the virtual image in the vicinity of this pixel or grid point.

From here on, a raster image is indexed by a pair (i, j) indicating column (i) and row (j) of the pixel, counting from the bottom left corner of the raster image. If an image has n_x columns by n_y rows of pixel, the bottom left is $(0, 0)$ and the top right is $(n_x - 1, n_y - 1)$.

Now we need a 2D real screen coordinates to specify pixel positions. The pixels will be located at integer coordinates, as show in the image below. Thus, the rectangular continuous domain for this continuous image with n_x width and n_y height is $R = [-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$ and the pixels integer coordinates place them at the center of the pixel area.

3 The Problem

At the heart of this phase of the project we are trying to solve the following problem:

How to generate a ray for each pixel of the image, according to the type of the camera specified in the scene file?

To solve this problem, we are given (input) the following information:

1. The pixel coordinate (i, j), we want to generate the ray for. (See Figure 2.)

2. The screen space dimensions "`l r b t`" (left, right, bottom, top), read from the input scene file, OR the vertical field of view angle, `fovy`, and the aspect ratio = (image width/image height), that works only for perspective cameras.

3. A point in world space where the camera is located at.

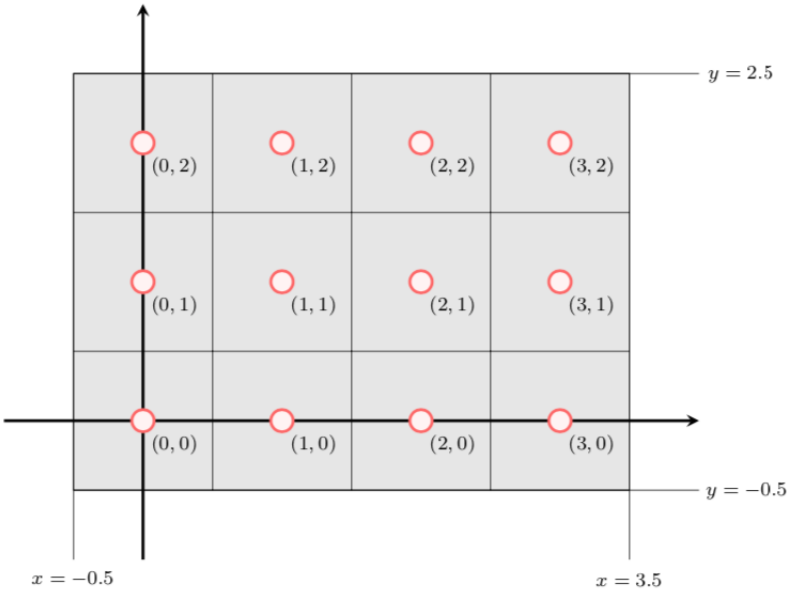

Figure 3: Coordinates of a $n_x = 4 \times n_y = 3$ pixel screen. Note that pixels are located at the center of a *pixel area*. The rectangle domain of this $n_x \times n_y$ image is $R = [-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$.

Figura 2: Grid of the underlying raster image.

4. A point in world space where the camera is pointed at.

5. A vector in world space that indicate the up direction of the camera.

The output of the camera algorithm should be the `ray` object.

If you examine carefully the images in Figure 1 above, you will identify that each ray, regardless of the camera type, has an origin and a direction. So, our mission is to develop the math machinery that enable us to determine these two elements that define a ray.

Our first objective is to identify a correspondence between the **image space** (the pixels, e.g. 1920×1080) and the screen space (which is called "image plane" in the Figure 1. Usually, the screen space is a 2D normalized frame of reference. The dimensions of the screen space comes from the input scene file explicitly or are deduced based on the `Film` dimensions. Note that for both the orthographic and perspective cameras, the bounds of the screen space should be provided in the scene file under the attribute `screen_window` as a list of 4 values, as in `screen_window=-1.777 1.777 -1.0 1.0"` (This represents a 16:9 proportion found in a full HD resolution). By default, the values for this attribute is [-1,1] along the shorter image axis and is set proportionally along the other (longer) axis.

For the perspective camera case, it might be preferable to specify the values `l r b t` indirectly; this can be done by defining a `fovy` angle (field of view in Y). This angle enables us to determine the height of the screen window (the near plane) and from this value it is possible to find the desired width by keeping the same aspect ratio defined by the film dimensions. The image below tries to show the relationship between the vertical angle `fovy` and the screen's height, assuming a focal distance equal to 1.

So, the correspondence we need is just a couple of equations that takes a (i, j) pixel coordinate in the image space and finds out a corresponding (u, v) location on the screen space. The next section describes the steps to find out these equations.

3.1 Mapping Pixels to Screen Space

For both types of the camera we need to solve this basic sub-problem:

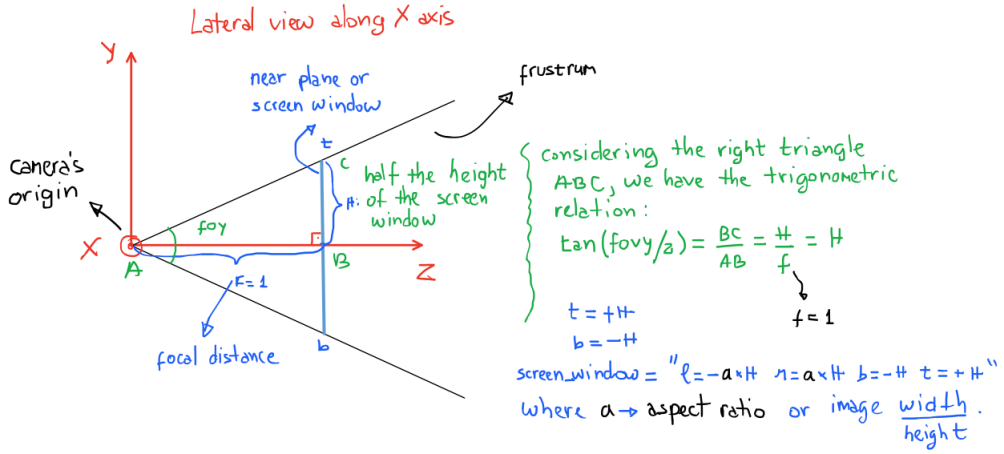

Figura 3: Lateral view along the x axis of a camera and its frustum setup.

"How does a pixel (i, j) on the image space maps to a coordinate (u, v) on the screen space?"

Recall that the camera needs this information to generate the rays, depending of the type of the camera. For the orthographic camera, the (u, v) defines the ray's origin, whereas for the perspective camera the (u, v) and the origin $\mathbf{e}$ define the direction of the ray.

In both cases first we need to map the image space to the screen space, $[-0.5; n_x - 0.5] \times [-0.5; n_y - 0.5] \mapsto [l; r] \times [b; t]$, where n_x and n_y are, respectively, the horizontal and vertical number of pixels of the image we want to create. This is a linear mapping done in each direction. Let us determine the equation only for the X dimension. For a given i value on the image space we want to find the corresponding u value on the screen space, so we have the following relation

$$\frac{i - (-0.5)}{u - l} = \frac{n_x}{r - l}, \tag{1}$$

and the unknown is u. Rearranging Equation 1 we get $(i+0.5)(r-l) = n_x(u-l)$, then $(u - l) = (r - l)(i + 0.5)/n_x$, and finally $u = l + (r - l)(i + 0.5)/n_x$, $i \in [-0.5; n_x - 0.5]$. For both dimensions we have:

$$\begin{cases} u &= l + (r - l)(i + 0.5)/n_x \\ v &= b + (t - b)(j + 0.5)/n_y \end{cases}$$

So far, we are able to identify the corresponding (u, v) point on the screen space given the first two input information we were provided with. To complete our task and solve the original problem, we now need to determine the 3D frame that defines the camera space. If we have this frame we have all the ingredients we need to create a ray, which is our final objective. So, the next section shows how to define the camera frame.

3.2 Determining the Camera Frame

Basically, what we need to do is to determine the camera orthonormal frame $[\mathbf{e}, (\hat{\mathbf{u}}, \hat{\mathbf{v}}, \hat{\mathbf{w}})]$.

Recall the list of input information we always receive from the scene file. Now we are going to use the last three pieces of information from the scene file to calculate the camera's frame: the (1) **look from** point, the (2) **look at** point, and the (3) `vup` vector. The first point **look from** is the $\mathbf{e}$. The first two points together define the camera's gaze direction, while the last vector provides the camera orientation. If we normalize the gaze vector we got the $\hat{\mathbf{w}}$ vector. From that, we find a vector pointing to the right, perpendicular to the plane defined by the $\hat{\mathbf{w}}$ and **vup** plane, using cross product; this is the camera's $\hat{\mathbf{u}}$ vector (after normalization). Finally, based on the two previously found vectors, $\hat{\mathbf{w}}$ and $\hat{\mathbf{u}}$, we apply the cross product again to get the last vector $\hat{\mathbf{v}}$.

In summary, the steps are:

1. `Vector3 gaze = look_at - look_from; Vector3 w = normalize(gaze); // left-hand orientation`
2. `Vector3 u = normalize( cross( vup, w ) ); // The order inside cross matters. Can you guess why?`
3. `Vector3 v = normalize( cross ( w, u ) );`
4. `Point e = look_from;`

Finally, we now have all pieces of the puzzle. In the next section we describe how to generate the rays.

3.3 Generating the Rays

Assuming that the camera frame is $[\mathbf{e}, (\hat{\mathbf{u}}, \hat{\mathbf{v}}, \hat{\mathbf{w}})]$, as depicted in the Figure 1, we can define a camera ray as follows:

1. For the **orthographic** camera, we have:

   `ray.direction`$\leftarrow \hat{\mathbf{w}}$,

   `ray.origin` $\leftarrow \mathbf{e} + u\hat{\mathbf{u}} + v\hat{\mathbf{v}}$

2. For the **perspective** camera, we have:

   `ray.direction` $\leftarrow (fd)\hat{\mathbf{w}} + u\hat{\mathbf{u}} + v\hat{\mathbf{v}}$

   `ray.origin`$\leftarrow \mathbf{e}$

where (fd) is the focal distance that we may assume to be 1, and (u, v) are the image plane coordinates.

4 Requirements

You should implement the following classes and integrate then into your RT.

A class `Ray` that represents a single ray. Internal members are the ray's `origin` (a point in 3D) and the ray's `direction` (a vector in 3D). You may want to provide methods to normalize the ray, and a method that returns the value of a point given a parameter `t` to the ray, as in:

```
class Ray {
  public:
  Ray (const Point3& o, const Vector3& d ) : o{o}, d{d} {/*empty*/}
  private:
  Point3 o; //!< origin
  Vector3 d: //!< direction
  // ... the methods goes here
  Point3 operator()(float t) const { return o + d * t; }
};


Ray r{ Point3{0,0,0}, Vector3{3,2,-4} }; // creating a ray.
```

```
float t{2.3}; // a parameter for the ray.
Point3 p = r( t ); // overloading operator()(), so that it returns o + t*d.
```

A class `Camera` that is responsible for implementing the camera model described in class. The suggestion here is to create a virtual class first, with the method `generate_ray()` and derive the other camera classes from it. This method receives a pixel (or sub pixel) coordinates and returns a ray passing through that point.

```
class Camera{
  Ray generate_ray(int x, int y) = 0;
};
class PerspectiveCamera : public Camera {
  // ...
};
class OrthographicCamera : public Camera {
  // ...
};
```

5 The Scene Format

Here we provide a basic XML format with the camera tag included. Here is an example scene file with a orthographic camera:

```
<RT3>
<lookat look_from="0 7 0" look_at="0 0 0" up="0 0 1" />
<camera type="orthographic" screen_window="-5.3 5.3 -4 4"/>
<film type="image" x_res="2800" y_res="1800" crop_window="0 1 0 1"
filename="../images/gcodex_matte_ortho.png"
img_type="png"  gamma_corrected="yes" />

<world_begin/>
<background type="colors" color="0.14 0.19 0.26" />
<!-- Here goes the rest of the file, including scene definition. -->
```

```
<world_end/>
</RT3>
```

The attributes `look_from`, `look_at`, and up define the two vectors necessary to build the camera's frame: gaze (`look_at - look_from`), and `vup`. The goal here is to define the camera's orthonormal basis $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{w}}$.

The attribute `screen_window` of the camera tag defines the view plane dimensions by specifying its horizontal and vertical limits, respectively `l` (left), `r` (right), `b` (bottom), and `t` (top). These dimensions define a box that should encapsulate the entire scene we want to view. There is no need to define the limits for the Z axis, because there are no limits to the ray's reach into that dimension. These values, at this stage, should correspond to values in world coordinates. This means that a sphere S_1 with `radius=2` and `center = point3(0,1,3)` would be visible, whereas a sphere S_2 with the same radius but located at `center = point3(-10,1,3)` would not be visible. Another interesting fact is that a sphere S_3 with the same radius but located at `center = point3(-2, 2, 300)` would appear to have the same projected circle shape that of sphere S_1 on the final image: can you explain why?

Notice that the aspect ratio of the `screen_window` dimensions must be the same of the image dimension, if we want an image with undistorted objects (i.e. the sphere projection is a perfect circle). You may experiment with different aspect ratios in the next project and see the effect of this on the generated image.

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 the tags background to compose the scene. Later on we will add scene, light, material, etc.

Here is an example scene file with a perspective camera:

```
<RT3>
<lookat look_from="0 4 -11" look_at="0 1 0" up="0 1 0" />
<camera type="perspective" fovy="30" />
<!-- The screen_window will default to  "-1.555 1.555 -1 1" /> -->
<film type="image" x_res="2800" y_res="1800" crop_window="0 1 0 1"
filename="../images/gcodex_matte_ortho.png"
```

```
img_type="png"  gamma_corrected="yes" />

<world_begin/>
<background type="colors" color="0.14 0.19 0.26" />
<!-- Here goes the rest of the file, including scene definition. -->
<world_end/>
</RT3>
```

The attribute `fovy` corresponds to the vertical angle of the perspective viewing frustum expressed in degrees. The smaller the angle the more zoom we get into the scene. The aspect of the image is often determined by the image dimensions, however it may be overridden if the `frame_aspect` if specified. The attribute `focal_distance` is the distance of the view plane along the $\mathbb{Z}^+$ assuming the camera located at the origin with its frame aligned with the world: this is called the focal distance. Because we are not yet implementing a lens for our camera, the focal distance value is irrelevant and, thus, it might be omitted. In that case it defaults to 1. Next, we found a modified version of the main loop introduced in the previous project, illustrating a possible usage of the new classes.

```
\\ [1] Parser and load scene file
std::vector<Primitives> scene; // list of objects in the scene. Not yet implemented.
\\ [2] Instantiate the Camera and Background objects.
Camera camera{...};
\\ [3] Enter the ray tracing main loop
int main() {
  // 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 = h-1 ; j >= 0 ; j-- ) {
    for( int i = 0 ; i < w ; i++ ) {
      Ray r2 = camera.generate_ray( i, j );
```

```
      std::cout << "Ray1: " << r1 << ", Ray2: " << r2 << std::endl;
      std::cout << "Point at t=2, ray(2) = " << r1(2.f) << std::endl;
      // Rays are not hitting the scene just yet; so let us sample the background.
      auto color = background.sample( float(i)/float(w), float(j)/float(h), r1 ); // get b
      camera.film.add( Point2{i,j}, color ); // set image buffer at position (i,j), accord
    }
  }
  // send image color buffer to the output file.
  camera.film.write_image();
}
```

6 Recommendations

Read Chapters 3 and 4 of Shirley 2009. In particular, Section 3.2 (Chapter 3) talks about the pixel as an area; Section 4.2 (Chapter 4) introduces the types of projections, and Section 4.3 (Chapter 4) explains how to generate rays based on the chosen camera type.

References

- P. Shirley and S. Marschner., "Fundamentals of Computer Graphics", 3rd ed., 2009, A K Peters.