A bare-bones ray tracer
Functional Programming 2018

Jesper Bengtson Patrick Bahr

March 6, 2018

Contents

1 Introduction

This assignment is designed to get you started with the Ray Tracer for the second year project. You are to set up a rendering engine that renders a single sphere of a given radius, with a given colour, with its center at the origin of a cartesian coordinate system. The camera is implemented such that it can be moved around the scene. We will not cover the basics of vectors and points in this assignment, as you should have had that from previous assignments. We will, however, delve into cartesian coordinate spaces and a few things that we need to think about when building any form of rendering engine.

2 Assignment

The task is to implement the following function that renders a coloured sphere to an image file:

```
let renderSphere
      (radius : float)    // The radius of the sphere
      (colour : Color)    // The colour of the sphere
      (position : point)  // The position of the camera
      (lookat : point)    // The point that the camera is looking at
      (up : vector)       // The up-vector for the camera
      (zoom : float)      // Distance to the view plane
      (width : float)     // Width of the view plane in units
      (height : float)    // Height of the view plane in units
      (resX : int)        // The horizontal resolution of the view plane
      (resY : int)        // The vertical resolution of the view plane
      (fileName : string) // Name of the file to save the rendered image
        : unit
  = ...
```

The first two arguments specify the colour and the radius the sphere (cf. Section 5 below). The next 8 arguments specify the camera (cf. Section 4 below), and the final argument specifies the file where the rendered image should be saved.

The accompanying file `Raytracer.fs` provides a skeleton where you have to implement the above function. The file also contains a main function that tests your implementation. Reference images that show the intended output of these tests are in the accompanying folder `reference`.

3 Cartesian coordinate spaces

A three-dimensional cartesian coordinate system allows us to uniquely specify points in three dimensions by providing their coordinates along three mutually perpendicular axes – the x-, the y-, and the z-axis. Coordinates are given using floating point numbers where the origin is at the point [0.0, 0.0, 0.0]. This is similar to the coordinate systems you are

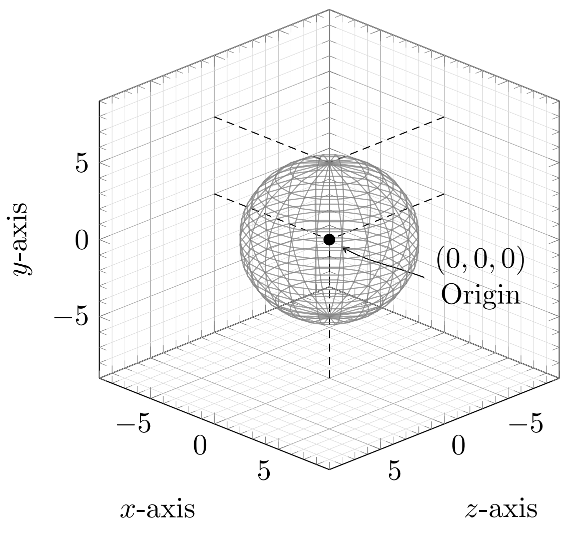


Figure 1: A wire frame of a sphere with a radius of five and with its center at the origin of a cartesian coordinate system.

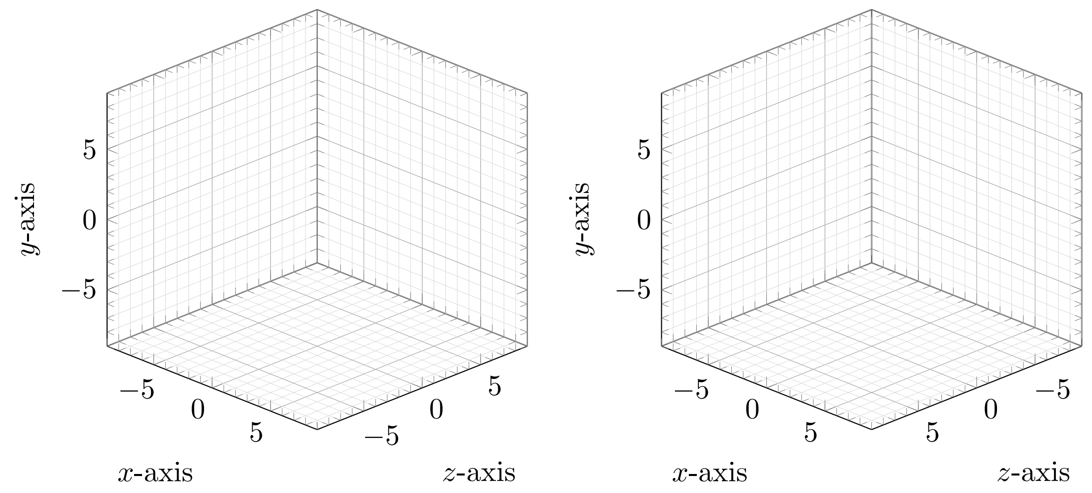
used to from your high-school calculus classes, although they will mostly have focussed on two-dimensional coordinate systems. We are just adding one more dimension.

The entire scene that we render will be modelled in this coordinate system. For example, a sphere is modelled using a point for its center and its radius; cameras will have a position and a direction they are facing. Figure 1 illustrates what a cartesian coordinate system can look like along with a sphere with its center at the origin. We will, for the most part, not explicitly mark the origin in the future but the axes will be clearly marked when necessary.

In this section we introduce the mathematics required to solve the assignment.

3.1 Right-handed coordinate systems

When building a scene one question you must be able to answer is "*What directions correspond to increasing and dereasing x-, y-, and z-coordinates?*". Does the y-axis go up like on two dimensional graphs? Does the z-axis go into the scene or out of the scene? As it turns out the answer to this question depends on who you ask. If you were to ask an architect she would most likely say that positive x is right, positive y is forward, and positive z is up in the enormous skyscraper that she is curently modelling. If you were to ask a 3D games programmer they would most likely (but not always) say x is right, y is up, and z is going into the scenes so that when you run your character down a corridor the z-coordinates increase. Finally, if you were to ask a phycisist or a CAD-programmer they would most likely look at whatever they are building from above in a God's view-like scenario where x is right (or east), y is up (or north), and z is towards the user (increasing altitude).



(a) Left-handed coordinate system (b) Right-handed coordinate system

Figure 2: In computer graphics, a left-handed coordinate system has the z-axis pointing into the screen, while right-handed system has it pointing out of the screen.

There are no right or wrong answer to these questions. In computer graphics we frequently talk about left- or right-handed coordinate systems as illustrated in Figure 2. To visualise this, hold your right hand in front of you and let your thumb go along an imaginary x-axis and your index finger along the y-axis. You will find that your middle finger points towards you. This means that as z increases, points move towards you (the God's view approach). If you try the same thing with your left hand you will find that the positive z-axis points into the scene.

The only objective (and then only slightly so) reason to use one coordinate system over another is that the cross product is calculated for right-handed coordinate systems. Converting between the two is however trivial (you just invert the z-axis) but you have to do so consistently or strange bugs will start appearing. The simple ray tracer we are building in this assignment uses a right-hand coordinate system.

3.2 Rays

Rays are, not surprisingly, a central concept when building a ray tracer. A ray has an origin point, and a direction vector. More precisely, the ray equation is written as

$$o + td$$

where o is the origin of the ray, d is the direction that the ray fires in and t is a scalar constant that uniquely determines points along the ray.

3.3 Orthonormal coordinate spaces

Behind this fancy term lies a powerful concept that we will use when we set up our camera before we render a scene. Orthonormal, as the name suggests, is a conjunction of orthogonal and normalised. A three-dimensional orthonormal coordinate space consists of three mutually orthogonal and normalised vectors (one for the x-, one for the y-, and one for the z-axis) that form a right-handed coordinate system. The case you will be familiar with is our standard world coordinate space that is created by the vectors $\langle 1.0,\ 0.0,\ 0.0\rangle$, $\langle 0.0,\ 1.0,\ 0.0\rangle$, and $\langle 0.0,\ 0.0,\ 1.0\rangle$. There are, however, infinitely many other orthonormal coordinate spaces and a key observation that we will demonstrate shortly is that it is very simple to transfer a vector from one space into another.

In Section 4 we will create a coordinate space for our camera that we generate dependent on its location and the direction the camera is facing. The intution is that the origin of the camera is exactly the center of the image that you want to render for the x- and the y- coordinate, and the positive distance to the view plane (since the view plane is in front of the camera) for the z-axis. You can then set up the vectors for the rays that fire through each pixel very easily without having to worry about what corresponding world coordinates they have. In order to actually render the scene, however, we need to translate these vectors into coordinates in our world coordinate space. As it turns out, this is very easy. Assuming that our camera has the orthonormal coordinate space (a, b, c) then we can translate any vector v^c in this coordinate space to world coordinates v^w by using the following equation.

$$v^w \triangleq \left\langle v^c_x a,\ v^c_y b,\ v^c_z c\right\rangle$$

Setting up an orthonormal coordinate system is typically done using a direction vector d, and an *up* vector u. The direction vector is typically the z coordinate axis, like the direction that the camera is looking at, and the up vector is typically a vector that represents what is considered “up” in the scene (normaly $\langle 0,\ 1,\ 0\rangle$). An orthonormal coordinate space (a, b, c) is set up in the following way

$$\begin{aligned} c &= \widehat{d} \\ b &= \widehat{u \times c} \\ a &= c \times b \end{aligned}$$

where we use the cross product to make sure that a, b, and c, are orthogonal to each other. The notation $\widehat{d}$ means that we take the normalised vector of d. Note that b and c are normalised to ensure that all axes are normalised. We do not explicitly have to normalise a as the cross product of two normalised vectors is also normalised.

One key observation here is that this procedure will fail if d and u are parallel as b will not be uniquely defined. One way around this, other than constantly checking for parallelism and that works well in practice, is to jitter u ever so slightly and use, for instance, $\langle 0.00034,\ 1.0,\ 0.00086\rangle$ in stead.

4 The camera

We require a bit of information in order to set up a camera. We need a position and a direction that it is looking in, we need to know the distance to the view plane, and we need the resolution of the image as well as the size of the pixels. One important thing to keep in mind is that even though our end result is a set of pixels, the image itself is actually a very small part of the actual rendering process. Try to think of everything in terms of world and camera coordinates – the camera has a position in world coordinates, so does the view plane and even the individual pixels can be seen as small squares suspended in the scene. The camera will fire a ray through the center of every pixel and paint the pixel the colour of any shape it hits (taking shading and reflection into consideration).

$$\begin{array}{lll} \mathit{camera} & \triangleq & (\mathit{position} : \mathit{point}) \quad (\mathit{lookat} : \mathit{point}) \\ & & (\mathit{up} : \mathit{vector}) \quad (\mathit{zoom} : \mathbb{R}) \\ & & (\mathit{width} : \mathbb{R}) \quad (\mathit{height} : \mathbb{R}) \\ & & (\mathit{resX} : \mathbb{Z}) \quad (\mathit{resY} : \mathbb{Z}) \end{array}$$

A camera is given by its position, the center of the view plane (*lookat*), a vector that points upwards, a zoom value that represents the distance to the view plane, the width and height of the view plane, and finally the resolution of the view plane (i.e. the width and height of the view plane in terms of pixels).

From these values we can calculate the width and the height of each pixel

$$\begin{aligned} \mathit{pixelWidth} &= \frac{\mathit{width}}{\mathit{resX}}, \text{ and} \\ \mathit{pixelHeight} &= \frac{\mathit{height}}{\mathit{resY}}. \end{aligned}$$

The basic ray-tracing algorithm works by shooting rays from the position of the camera through the pixels of the view plane – one ray through each pixel. The colour of a pixel is then determined by checking which object the corresponding ray hits (if any).

4.1 Calculating the rays

The camera is set up using an orthonormal basis constructed by the direction vector between the camera position and the center of the view plane, and the up vector in the following manner

$$\begin{aligned} w &= \widehat{\mathcal{D}\ \mathit{lookat}\ \mathit{position}} \\ u &= \widehat{\mathit{up} \times w} \\ v &= w \times u, \end{aligned}$$

where $\mathcal{D}\ u\ v$ is the distance vector from u to v (note that since we are in a right-handed coordinate system this vector points backwards behind the camera).

To target the center of a pixel (x, y) we can use the following algorithm

$$\begin{aligned} p_x &= pixelWidth * \left(x - \frac{resX}{2} + 0.5\right) \\ p_y &= pixelHeight * \left(y - \frac{resY}{2} + 0.5\right) \end{aligned}$$

Finally, the ray will have the origin *position* and the direction

$$p_x * u + p_y * v - zoom * w$$

in world coordinates (not camera coordinates).

5 Rendering a sphere

To render any shape requires a hit function. For this exercise we will only do the bare minimum – determine wether a ray hits a sphere centered at the origin or not.

The hit function for the sphere can be described by an equation, namely

$$p_x^2 + p_y^2 + p_z^2 = r^2$$

This equation describes a sphere with radius r and [0, 0, 0] as its centre. Recall from Section 3.2 that a ray has the equation

$$o + td$$

and plugging this ray equation into the equation for a circle gives us

$$(o_x + td_x)^2 + (o_y + td_y)^2 + (o_z + td_z)^2 = r^2$$

which after simplification, which is what your second ray tracing assignment in Fuctional Programming was about, you get the following equation

$$(d_x^2 + d_y^2 + d_z^2)t^2 + 2(o_xd_x + o_yd_y + o_zd_z)t + o_x^2 + o_y^2 + o_z^2 - r^2.$$

This is, in fact, just a second degree polynomial with t as the only unknown as we know where each ray originates and which direction they are fired in. As you may remember form high-school, a second degree polyniomal has the form

$$a \cdot t^2 + b \cdot t + c = 0,$$

where in our case

$$\begin{aligned} a &= d_x^2 + d_y^2 + d_z^2 \\ b &= 2 \cdot (o_x \cdot d_x + o_y \cdot d_y + o_z \cdot d_z) \\ c &= o_x^2 + o_y^2 + o_z^2 - r^2, \end{aligned}$$

and can be solved using the formula

$$t = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

which depending on the value of the discriminant (the expression under the square root)

$$d = b^2 - 4 \cdot a \cdot c$$

has either zero (if $d < 0$), one (if $d = 0$), or two (if $d > 0$) real-valued solutions. A negative solution (i.e. $t < 0$) will mean that the hit points are both behind us, and we can discard them. In case of two positive solutions, we pick, of course, the closest one, i.e. the smallest positive t.

Assuming that the ray indeed does hit, we return the sphere colour, otherwise we return the background colour, which in our case is black.