Writing a Basic Raycaster

Vassillen Chizhov

April 2019

1 Introduction

This is a tutorial that is meant to walk the reader through programming a basic ray tracer at the novice level. Topics which are covered include: outputting an image in a very simple format (PPM), creating a stationary camera, intersecting a collection of primitives, and shading. This document comes paired with an example implementation in C++ of the ideas and techniques discussed.

1.1 Prerequisites

The reader is expected to have some fundamental programming skills as well as have some basic knowledge from linear algebra and analytic geometry. The code throughout this tutorial will be in C++, and only basics such as control structures, arrays, and classes will be employed for the most part. The focus will be on simplicity and clarity rather than performance. The mathematical knowledge required includes: the notion of 3-dimensional vectors, the dot product, the cross product, and solving quadratic equations.

2 Ray tracing a sphere

In this section we aim to set the fundamentals on which the later parts will build upon. The end result of what seems like a lot of work is a rather unimpressive flat rendering of a sphere. However, all that effort will pay off in the later parts, when we have the technicalities out of the way and can focus on the more interesting aspects. Readers familiar with the basics should feel free to skip this part.

2.1 A simple 3D vector library

Throughout the rest of the tutorial performing operations with 3D vectors will be commonplace. And while it is possible to type those out for each component every time they arise, it is neither practical nor efficient, so we will make a slight detour to build a very simple 3D vector library. The structure representing our 3-dimensional vector, that we will refer to as *vec*3, will be made of 3 floating

point fields: e_0, e_1, e_2. If we have a vector $\vec{v}$ we will use the notation v_0, v_1, v_2 for its first, second, and third component respectively, alternatively $v(0), v(1), v(2)$, or $v = (v_1, v_2, v_3)$. We define the componentwise operations of addition $(+)$, subtraction $(-)$, multiplication $(\cdot)$, and division $(/)$ on pairs of vectors $\vec{u}, \vec{v}$ respectively as:

$$\vec{u} + \vec{v} \coloneqq (u_0 + v_0, u_1 + v_1, u_2 + v_2)$$

$$\vec{u} - \vec{v} \coloneqq (u_0 - v_0, u_1 - v_1, u_2 - v_2)$$

$$\vec{u} \cdot \vec{v} \coloneqq (u_0 v_0, u_1 v_1, u_2 v_2)$$

$$\frac{\vec{u}}{\vec{v}} \coloneqq (\frac{u_0}{v_0}, \frac{u_1}{v_1}, \frac{u_2}{v_2})$$

We additionally define multiplication and division with a scalar:

$$\vec{u} \cdot s \coloneqq (u_0 s, u_1 s, u_2 s)$$

$$\frac{\vec{u}}{s} \coloneqq (\frac{u_0}{s}, \frac{u_1}{s}, \frac{u_2}{s})$$

Finally we define the dot and cross product as:

$$\langle \vec{u}, \vec{v} \rangle \coloneqq u_0 v_0 + u_1 v_1 + u_2 v_2$$

$$\vec{u} \times \vec{v} \coloneqq (u_1 v_2 - u_2 v_1, u_2 v_0 - u_0 v_2, u_0 v_1 - u_1 v_0)$$

The length (magnitude) of a vector $\vec{v}$ will be denoted as $\|\vec{v}\|$, and is computed as:

$$\|\vec{v}\| = \sqrt{\langle \vec{v}, \vec{v} \rangle}$$

We also define a utility operation *clamp* as:

$$clamp(s, a, b) = \min(\max(s, a), b)$$

$$clamp(\vec{u}, a, b) = (clamp(u_0, a, b), clamp(u_1, a, b), clamp(u_2, a, b))$$

We recommend that readers that have never before written a vector class, take the time to implement the structure and operations defined above as an exercise. The corresponding reference code can be found in *vec.h*.

2.2 Image representation and saving as PPM

One can represent rendered images as either *vec*3 valued 2D arrays (the first, second, and third component correspond respectively to red, green, blue) with dimensions $width \times height$, or one-dimensional arrays in row- or column-major order of size $width \times height$. We only require a structure that exposes an accessor, which given indices (x, y) in the range $[0, width) \times [0, height)$ returns the cell at (x, y). Throughout this tutorial we will use the notation $A_{x,y}$ or $A(x, y)$ for accessing the (x, y) element of an image named A. In order to save such an image in plain PPM format, we write out the plain PPM header, which is made up of a magic number describing the file type $P3$, followed by a

whitespace, the width as an integer value, followed by a whitespace, the height as an integer value, followed by a whitespace, the maximum color value which can be at most $2^{16} - 1 = 65535$. An example of such a header for an image of dimensions 1024×768, with a maximum color value of $2^8 - 1 = 255$ looks like this:

```
P3
1024 768
255
```

After we have written out the header, we can proceed to write out our image in row-major order:

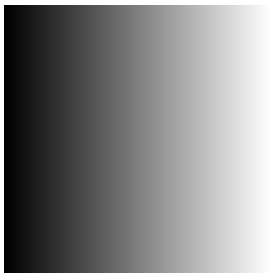
```
for(int y=0;y<height;++y)
{
    for(int x=0;x<width;++x)
    {
        vec3 clampedColor =
        clamp(image(x,y) * 256.0, 0.0, 255.0);

        file << (int)clampedColor(0) << "\t"
        << (int)clampedColor(1) << "\t"
        << (int)clampedColor(2) << "\t\t";
    }
    file << "\n";
}
file.close()
```

The clamping is done in order to keep values in the range$[0, 255]$.

Finally, we provide an algorithm to render a horizontal gradient:

```
for(int y=0;y<height;++y)
{
    for(int x=0;x<width;++x)
    {
        image(x,y) = vec3((float)x/(float)(width-1));
    }
}
```

The output for a 100×100 image is presented below:



A reference implementation of a structure representing an image and a method to save a PPM file can be found in *image.h*. The readers are encouraged to modify it or implement their own variant.

2.3 Camera and generating rays

A ray in 3D is defined by its origin $\vec{0}$ and a direction vector $\vec{d}$. All points on the ray are given by $\vec{r}(t) = \vec{o} + t\vec{d}, t \geq 0$. We will exclusively use normalized ray directions ($\|\vec{d}\| = 1$) since it saves a lot of trouble down the line normalizing them for shading calculations. Additionally if the direction vector is normalized, then the point:

$$\vec{r}(t) = \vec{o} + t\vec{d}$$

is exactly at a distance t from $\vec{o}$, meaning that $\|\vec{r}(t) - \vec{o}\| = t$, which will prove useful later on where we get to shading.

Figure 1: Figure of a ray, add a link to an interactive app letting the user modify t

We will also define a very simple camera, which given a pixel's coordinates (x, y) will generate a ray passing through the center of said pixel. There are numerous ways to define a camera, but here we define it by its origin $\vec{o}$, its right vector $\vec{e_0}$, its up vector $\vec{e_1}$, and its forward vector $\vec{e_2}$. Initially those will be set to $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ respectively, i.e. the 3 directions will coincide with the X, Y, Z axes, and the camera's origin will be at the origin of the coordinate system. When we generate a ray, its origin will coincide with the origin of the camera. Given pixel coordinates (x, y) $in[0, width) \times [0, height)$ we will first map those to $[-r, r] \times [1, -1]$ in order for the central image pixel to be at $(0, 0)$ (which provides a nice symmetry), where $r = \frac{width}{height}$ is the aspect ratio. We usually want the virtual camera film to have the same aspect ratio as our target resolution as to not stretch out the image, that's why we have $[-r, r] \times [1, -1]$ rather than $[-1, 1]x[1, -1]$. The flip of the y coordinate (that is using $[1, -1]$ rather than $[-1, 1]$) is done to account for the fact that in screen/image coordinates the y coordinate increases downwards and not upwards, which is the opposite of the case for our virtual world where $(0, 1, 0)$ points up.

Figure 2: Add an illustration of the above

Novice readers that are not very comfortable with mathematics should feel free to skip the derivation below and just use the results from it readily. We derive the linear mapping from $[0, width) \times [0, height)$ to $[-r, r] \times [-1, 1]$. This is done by finding linear functions: $L_x(x) = a_x x + b_x, L_y(y) = a_y y + b_y$, such that $L_x(0) = -r$, $L_x(width) = r$, $L_x(0) = 1$, $L_x(height) = -1$.

$$L_x(0) = a_x 0 + b_x = b_x = -a$$

$$L_x(width) = a_x width + b_x = a_x width - r = r \implies a_x = 2\frac{r}{width}$$

$$L_x(x) = \frac{2r}{width}x - r = r\left(\frac{2}{width}x - 1\right)$$

$$L_y(0) = a_y 0 + b_y = b_y = 1$$

$$L_y(height) = a_y height + b_y = a_y height + 1 = -1 \implies a_y = -\frac{2}{height}$$

$$L_y(y) = -\frac{2}{height} y + 1$$

There's a final minor detail to account for, which is that (x, y) really constitutes a pixel's upper left corner, so an offset by $(0.5, 0.5)$ is necessary for the ray to pass through the center of the pixel and not its lower left corner (lower due to the y flip). Thus the final mapping from pixel coordinates to normalized screen coordinates is:

$$L_x(x) = r\left(\frac{2}{width}(x + 0.5) - 1\right)$$

$$L_y(y) = -\frac{2}{height}(y + 0.5) + 1$$

Finally, to generate the ray direction passing through a given pixel we use the normalized screen coordinates to pick a point on the virtual film of the camera:

$$\vec{d} = L_x(x)\vec{e}_0 + L_y(y)\vec{e}_1 + e_2$$

Figure 3: Add an illustration of the above

We normalize the direction after that for the reasons outlined in the beginning of the subsection. A reference implementation for generating a ray from pixel coordinates can be found below, as well as in *camera.h* in the accompanying code:

```
float u = (float)width/(float)height *
(2.0f * ((float)x + 0.5f) / (float)width - 1.0f);
float v = -2.0f * ((float)y + 0.5f) / (float)height + 1.0f;
vec3 rayOrigin = camera.origin;
vec3 rayDirection = normalize(u * camera.e0 + v * camera.e1 +
    camera.e2);
```

2.4 Sphere intersection

We finally get to the last part of rendering our sphere - actually defining the intersection of a ray with the sphere, which lies at the heart of ray casting - finding intersections between rays and geometry. A sphere is defined mathematically as the set of points that are at the exact same distance (R the radius) from a certain point ($\vec{c} = (c_x, c_y, c_z)$ the center of the sphere). Thus for a point $\vec{p} = (x, y, z)$ lying on the sphere, the following formal (canonical) equation holds:

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = R^2$$

or equivalently:

$$\|\vec{p} - \vec{c}\|^2 = \langle \vec{p} - \vec{c}, \vec{p} - \vec{c} \rangle = R^2$$

In the previous section we saw that a ray is defined (parametrically) as:

$$\vec{r}(t) = \vec{o} + t\vec{d}$$

For a point q to be in the intersection of the ray and the sphere, it simply must fulfill both equations. We can satisfy that by setting $\vec{p} = \vec{r}(t)$ and solving for t:

$$\langle \vec{r}(t) - \vec{c}, \vec{r}(t) - \vec{c} \rangle = R^2$$

$$\langle \vec{o} + t\vec{d} - \vec{c}, \vec{o} + t\vec{d} - \vec{c} \rangle = R^2$$

$$\langle \vec{d}, \vec{d} \rangle t^2 - 2\langle \vec{d}, \vec{c} - \vec{o} \rangle t + \langle \vec{c} - \vec{o}, \vec{c} - \vec{o} \rangle - R^2 = 0$$

$$At^2 + -2Bt + C = 0$$

$$A = \langle \vec{d}, \vec{d} \rangle, B = \langle \vec{d}, \vec{c} - \vec{o} \rangle, C = \langle \vec{c} - \vec{o}, \vec{c} - \vec{o} \rangle - R^2$$

$$D = B^2 - AC$$

In step 3 we have used the linearity of the dot product and its distributive property. We have a quadratic polynomial with respect to t on the left side. The solution for t is well known, and is real only when the discriminant D satisfies: $D \geq 0$. If $D = 0$, there is a single solution for t corresponding to the ray grazing the sphere in that single point, if $D > 0$, there are 2 solutions for t corresponding to the points where the ray enters the sphere and where it leaves the sphere. Before we proceed further, we will note that $A = 1$, as long as the ray direction is normalized, which we stated that we will uphold, thus we get the simplified solutions:

$$t_1 = B - \sqrt{D}, t_2 = B + \sqrt{D}$$

Figure 4: Add figure to illustrate the 3 cases

The last detail that we are missing, is the fact that $\vec{r}(t)$ is part of the ray only for $t \geq 0$. Thus we should throw away intersections that do not satisfy this criteria, as they will be effectively "behind" the ray's origin.

Figure 5: Add figure to illustrate what behind the ray and camera means

When tracing the ray through the scene we want to return the first intersection, as obviously anything behind that would be occluded. In this case where we have only one sphere, and two possible intersections, we know which is the closer one: t_1 since we subtract a non-negative value from B. If the closer intersection is not smaller than 0 then it is in front of the ray, and we should pick

it as the closest one. If it is behind the ray, we can now pick the second intersection t_2 as long as it is > 0. If both are behind the ray origin ($t_1 < 0, t_2 < 0$) we treat it as there being no intersections. We will additionally extend this to arbitrary ranges $[t_{\min}, t_{\max}]$, so that we can have a "far distance" and a "near distance" if we so wish. The only difference in that case is that we perform a check $t_{\min} < t_1 < t_{\max}$ rather than just $t_1 \geq 0$, and similarly for t_2.

The reference implementation can be found in *sphere.h* in the accompanying code. Note that we treat the grazing case as having no intersection there, since it's irrelevant and in most cases caused by numerical error (due to the finite precision of computers):

2.5 Putting it all together

All we need to do now is iterate over all our pixels, generate rays through their centers, check for an intersection with our sphere, and if there's an intersection write a white color in the respective pixel of our image, otherwise write a black color.

```
int main()
{
  Image image;
  image.init(640,480);

  Camera camera;

  Sphere sphere(vec3(0, 0, 3), 1);

  float aspectRatio = (float)image.w() / (float)image.h();

  // iterate over all image pixels in row-major order
  // for each image pixel shoot a ray and check for an intersection
  // set white if there's an intersection, and black if there's
    none
  for (u32 y = 0; y < image.h(); ++y)
  {
    for (u32 x = 0; x < image.w(); ++x)
    {
      // map [0,width]x[0,height] to
      // [-aspectRatio,aspectRatio] x [1,-1]
      // multiply by the aspect ratio to stretch/squeeze the
      // virtual film size to match the screen's aspect ratio
      float u = aspectRatio *
      (2.0f * ((float)x + 0.5f) / (float)image.w() - 1.0f);
      float v = -2.0f *
      ((float)y + 0.5f) / (float)image.h() + 1.0f;

      Ray ray = camera(u, v);

      image(x, y) = vec3(float(0.0f<sphere.intersect(ray)));
    }
  }
  image.savePPM("out.ppm");
  return 0;
}
```

The above code produces the following image: