Notes on Ray Tracing in One Weekend

Junyi Li

October 2022

1 Image Format

The book uses the PPM image format as a simple way of producing images from RGB values. This isn't that convenient in the long run, so I used SDL2 to create a window to view the image. There are three main entities in this process:

1. The SDL2 window and renderer
2. An *Image* which wraps a int buffer containing RGBA data
3. A *Scene* which contains objects in our world

The scene and its objects perform the actual ray tracing and write pixel colors to an image. That image buffer is then sent to SDL2 to be displayed. Everything is kept inside a main *App* class, with member variables *window_*, *renderer_*, *image_*, and *scene_*. The snippet below suggests how each component is initialized.

```
1 SDL_Init(SDL_INIT_EVERYTHING);
2 window_ = SDL_CreateWindow("Yet Another Ray Tracer", SDL_WINDOWPOS_CENTERED,
3                            SDL_WINDOWPOS_CENTERED, 1280, 720, 0);
4 renderer_ = SDL_CreateRenderer(window_, -1, 0);
5 image_.Init(1280, 720, renderer_);
```

On line 5, we are resizing the image's internal vector, and passing it a pointer to the SDL2 renderer. The latter is necessary because the Image class actually holds two buffers – a simple int vector for itself, and another *SDL2 texture*, which needs the renderer to be initalized. This texture is pretty much also an int vector, except we can't write to it directly. Note that the scene is not explicitly initialized here, as a default constructor suffices.

1.1 Pixel Data

Pixels are represented in either RGBA or ABGR format, depending on your platform. Each pixel is a 32-bit integer, with 8 bits per channel; thus an int vector suffices to store the entire image. Assuming we already have the values of each channel, we can write the pixel using the correct format like so:

```
#if SDL_BYTEORDER == SDL_BIG_ENDIAN
    uint32_t pixelColor = (r << 24) + (g << 16) + (b << 8) + a;
#else
    uint32_t pixelColor = (a << 24) + (b << 16) + (g << 8) + r;
#endif
```

The snippet below shows how to send the pixel data to SDL2. The main thing to note is that we pass it a pointer to the pixel vector.

```
SDL_UpdateTexture(texture_, nullptr, pixels_.data(), w_ * sizeof(uint32_t));
SDL_RenderCopy(renderer_, texture_, nullptr, nullptr);
SDL_RenderPresent(renderer_);
```

> The pixel buffer should be distinguished from the *viewport*. The latter is a virtual canvas of unit square "pixels" through which we aim our rays into the world. More on this later.

1.2 Coordinate Systems

To keep things simple, we only need to care about two coordinates.

1. **World coordinates.** This is the usual xyz coordinates represented by vec3.
2. uv **coordinates.** These are coordinates on the viewport of our raytracer.

World coordinates are straightforward so not much will be said about that. The uv coordinates can be defined in a few different ways, and which one you use is simply a matter of personal preference, I think. They can be thought of as xy axes on the viewport, and are depicted in Figure 1.

Note that the values for u and v fall within $[-1, 1]$. What we easily have are the dimensions of our image, say, $m \times n$. Thus we can view each pixel of the image as a cell in a $m \times n$ matrix. The scaling required to get u and v is simply

$$u \leftarrow \frac{2j}{n} - 1 \text{ and } v \leftarrow \frac{2i}{m} - 1.$$

We typically also store a $\hat{u}$ and $\hat{v}$ vector. The length of these vectors should be half the width and height respectively, and its easy to see why. Given a (u, v) coordinate, and O the center of our viewport *in world space*, we can compute the world coordinates of the (u, v) coordinate using $(O + u\hat{u}, O + v\hat{v})$.

2 Spheres

The equation for a point on the sphere can be described as

$$(P - C) \cdot (P - C) = r^2$$

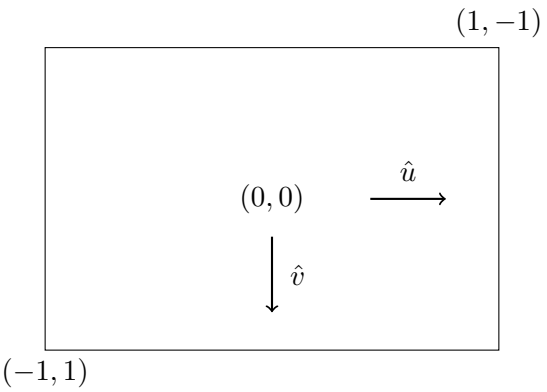

Figure 1: The viewport, with its origin at the center and directions of the u and v vectors shown.

where P is the point on the sphere, C the center, and r its radius. Note that P and C are vectors here. To check if a ray intersects a sphere, we just need to substitute p with the equation of the ray.

Recall that a ray is just a line, defined like $A + tb$, where A is its origin, b its direction vector, and t any real number. Thus we get the substitution

$$(A + tb - C) \cdot (A + tb - C) = r^2.$$

A short expansion gives us

$$b \cdot bt^2 + 2(A - C) \cdot bt + (A - C) \cdot (A - C) - r^2 = 0,$$

which is just a quadratic equation in t. Solving this equation will tell us the point(s) of intersection, if they exist.

> It is also useful to compute the normal at the intersection point. For a sphere, this is just $I - C$, where I is the intersection point and C the center.

3 Antialiasing

The objects in our scene exist in $\mathbb{R}^3$. But our rays go from camera to viewport in a discrete fashion, from pixel to pixel; in fact, each ray is going through the center of the pixel. [1] The result is an image which, when zoomed in, reveals jagged edges, since the color of each pixel is determined solely by the color found at its very center.

Antialiasing is a way to fix this visual defect. Instead of shooting just one ray through each pixel, we shoot $n > 1$ rays to random points within that pixel, with the final color of that pixel being the average across all n rays.

[1] This depends on how the uv coordinate system in the viewport is defined. Peter Shirley defines the lower left corner as $(0, 0)$, so each ray will travel through the lower left corner of each pixel. In my attempt I defined it in the same way as shadertoy.com, using the viewport's center as $(0, 0)$ instead.

Suppose our pixel was represented by an ij coordinate (kinda like a matrix), then the snippet below describes how to compute the uv coordinates. Note that the `scale_u()` and `scale_v()` functions perform the scaling described in Section 1.2.

```
// n is the number of samples per pixel
for (int s = 0; s < n; i++)
{
    // rand() produces a random double in [0, 1)
    double u = scale_u(i + rand());
    double v = scale_v(j + rand());
    // --snip--
}
```

4 Diffuse Materials