1 Ray Tracing

In the introduction of the book "Ray Tracing from the Ground Up" [1] the author loosely defines two classes of renderers: projective algorithms and image-space algorithms. The former nowadays is the most popular because it's based on the graphics pipeline model and can easily take advantage of modern graphics cards. The basic idea is to project the objects on the scene into a 2D plane and then compute the appearance of the objects that can be seen. The image-space algorithms start from the pixel and try to figure out where the light came form for that pixel. Following the light's path we can compute hard shadows, soft shadows, reflection, refraction, depth of field, motion blur, caustics, ambient occlusion, indirect lighting and other effects. But following every light ray's bounce it's extremely costly, especially if there are a lot of light sources in the scene, and that's why this approach historically wasn't the one to be used for interactive application. Instead, given the high quality and fidelity that this method can achieve, it's been widely used to create movies, clips and images, because in these application, after an initial low-res prototyping, the computer can elaborate the final product even for days.

say that the rt is being used also in real time application, blended with standard pipeline

1.1 Rays and Cameras

As we said before, the idea behind a ray tracer is to generate a ray for each pixel and check whether the ray hits a surface. If so it has also to find out if a light source lighten the point or if it is in shadow.

The easiest way to generate such rays is the following: imagine to put a 2D rectangle into a 3D scene; draw a square grid on the rectangle and for each square project rays perpendicular to the rectangle's faces; if a ray hits an object color the corresponding square with the surface color; if a ray hits no objects then color the pixel with black (or a default color). Because the rays are parallel to one another this method gives us a really simple orthogonal camera.

Orthogonal cameras are useful but in general prospective cameras are preferred because they resemble more the human eye. To model a perspective camera all we have to do is to put a point behind the rectangle with the grid and generate rays that start from the point and pass through the grid's squares. The common choice is to make the rays pass through the squares' center. The model just described is called a pinhole camera, shown in Figure 1, because mimics the real life cameras. One trick here is that we put che plane before the point so we don't have to flip the image before showing it to the user.

A ray is defined as a point and a direction, as shown in Figure 2, in formulas:

$$r(t) = O + tR$$

where $O, R \in \mathbb{R}^3$ and R is a unit vector. At each time t we can compute a point position on the ray, in this way the ray tracing algorithm can check for each t

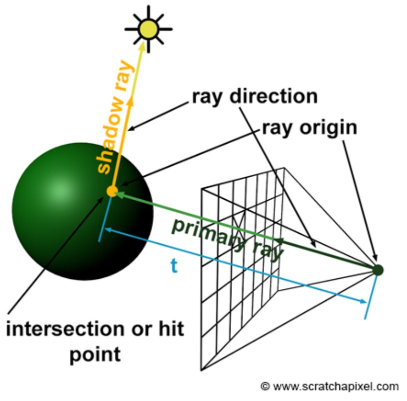

Figure 1: A pinhole camera inside a scene with a sphere and a light source

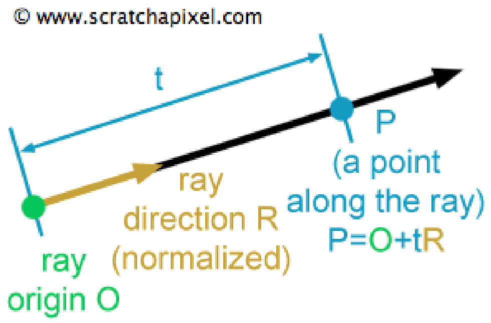

Figure 2: Ray model

whether the ray intersected a surface or not.

When generating rays we have to take in consideration several factors: camera position and orientation are the more obvious, but also the field of view and the image aspect ratio. First of all we need to derive the rays' direction with the pixel's centers on the image plane (the rectangle with a grid) and the camera origin (also called *eye*). But these two values refers to different coordinate systems because the pixel position refers to the top left corner of the image plane and the camera origin is defined in world coordinates. Here with world coordinates we mean the coordinate system with which the object (and the camera) are placed on the scene. One way to solve this problem is to produce the rays as if the camera was placed in the origin looking towards $-z$ and then transate and rotate the rays according to the true camera position and orientation.

In order to do so we need to remove the dependency from the actual output image size. Given two coordinates in raster space we normalize them by dividing respectively by the image width and height. Now the coordinate are in Normalized Device Coordinates (NDC) space. Actually the precise operation is the following, in which we also move the coordinate in pixels center:

$$PixelNDC_x = \frac{(Pixel_x + 0.5)}{ImageWidth}$$
$$PixelNDC_y = \frac{(Pixel_y + 0.5)}{ImageHeight}$$

Now we have to map this coordinates form $[0,1]$ into $[-1,1]$ because the eye of the camera is at the origin, so the center of the image plane (also called screen) is on the z-axis. In practice we are aligning the sign of the four quadrants of the xy-plane in world coordinates with the image plane.

$$PixelScreen_x = (2 * PixelNDC_x - 1) * AspectRatio$$
$$PixelScreen_y = 1 - 2 * PixelNDC_y$$

Remember that we want a right handed system so we map: the top left corner, $(0, 0)$ in raster space, to $(-1, 1)$ in screen space; the bottom right corner, $(ImageWidth, ImageHeight)$ in raster space, to $(1, -1)$ in screen space; and the last two accordingly.

The $AspectRatio$ used in the formula equals $ImageWidth/ImageHeight$ and we must consider it when the output image is not squared. In fact we need to produce more pixel in one direction otherwise the generated image will look squished.

One of the last thing to consider is the field of view (fov), the view angle of the camera. Assuming that the distance between the camera eye and the image plane equals to 1, we want to know by how much enlarge or shrink the image plane horizontally and vertically. Knowing the fov angle α we just need to multiply the screen coordinates by $\tan(\alpha/2)$.

The last thing to take in consideration is that out camera may not be always positioned in the world origin and look toward $-z$. To do so there are several options the one explained here is also

explain like if we are moving a point with a normal direction

References

[1] Kevin Suffern. *Ray Tracing from the Ground Up*. A K Peters, Ltd., 2016.