CHAPTER 2

What is a Ray?

Peter Shirley, Ingo Wald, Tomas Akenine-Möller, and Eric Haines
NVIDIA

ABSTRACT

We define a ray, show how to use ray intervals, and demonstrate how to specify a ray using DirectX Raytracing (DXR).

2.1 MATHEMATICAL DESCRIPTION OF A RAY

For ray tracing, an important computational construct is a three-dimensional ray. In both mathematics and ray tracing, a *ray* usually refers to a three-dimensional half-line. A ray is usually specified as an interval on a line. There is no implicit equation for a line in three dimensions analogous to the two-dimensional line $y = mx + b$, so usually the parametric form is used. In this chapter, all lines, points, and vectors are assumed to be three-dimensional.

A parametric line can be represented as a weighted average of points A and B:

$$P(t) = (1-t)\,A + tB. \tag{1}$$

In programming, we might think of this representation as a function $P(t)$ that takes a real number t as input and returns a point P. For the full line, the parameter can take any real value, i.e., $t \in [-\infty, +\infty]$, and the point P moves continuously along the line as t changes, as shown in Figure 2-1. To implement this function, we need a way to represent points A and B. These can use any coordinate system, but Cartesian coordinates are almost always used. In APIs and programming languages, this representation is often called a `vec3` or `float3` and contains three real numbers x, y, and z. The same line can be represented with any two distinct points along the line. However, choosing different points changes the location defined by a given t-value.

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_2

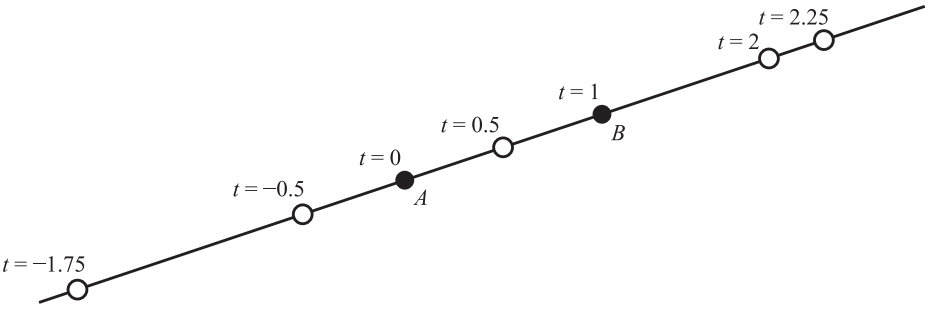

Figure 2-1. *How varying values of t give different points on the ray.*

It is common to use a point and a direction vector rather than two points. As visualized in Figure 2-2, we can choose our *ray direction* **d** as $B - A$ and our *ray origin* O as point A, giving

$$P(t) = O + t\mathbf{d}. \tag{2}$$

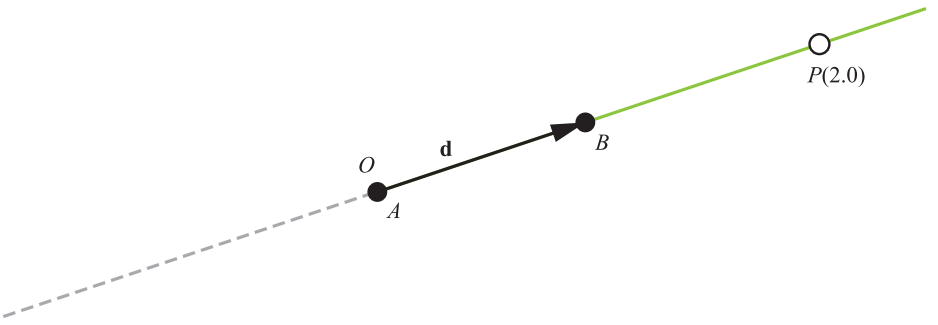

Figure 2-2. *A ray $P(t) = O + t\mathbf{d}$, described by an origin O and a ray direction $\mathbf{d}$, which in this case is $\mathbf{d} = B - A$. We often are interested in only positive intersections, i.e., where the points found are in front of the origin ($t > 0$). We depict this limitation by drawing the line as dashed behind the origin.*

For various reasons, e.g., computing cosines between vectors via dot products, some programs find it useful to restrict **d** to be a unit vector $\hat{\mathbf{d}}$, i.e., *normalized*. One useful consequence of normalizing direction vectors is that t directly represents the signed distance from the origin. More generally, the difference in any two t-values is then the actual distance between the points,

$$\left\| P(t_1) - P(t_2) \right\| = \left| t_2 - t_1 \right|. \tag{3}$$

For general vectors **d**, this formula should be scaled by the length of **d**,

$$\left\| P(t_1) - P(t_2) \right\| = \left| t_2 - t_1 \right| \left\| \mathbf{d} \right\|. \tag{4}$$

2.2 RAY INTERVALS

With the ray formulation from Equation 2, our mental picture is of a ray as a semi-infinite line. However, in ray tracing a ray frequently comes with an additional interval: the range of t-values for which an intersection is useful. Generally, we specify this interval as two values, t_{min} and t_{max}, which bound the t-value to $t \in [t_{\text{min}}, t_{\text{max}}]$. In other words, if an intersection is found at t, that intersection will not be reported if $t < t_{\text{min}}$ or $t > t_{\text{max}}$. See Figure 2-3.

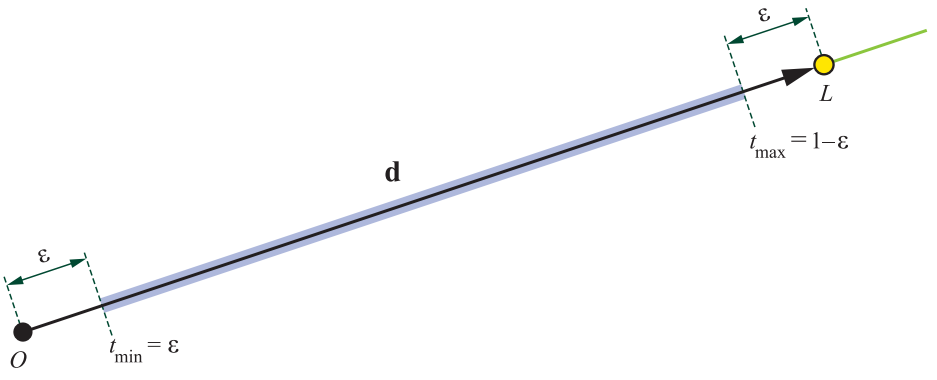

Figure 2-3. *In this example there is a light source at L and we want to search for intersections between only O and L. A ray interval $[t_{min}, t_{max}]$ is used to limit the search for intersections for t-values to $[t_{min}, t_{max}]$. To avoid precision problems, this restriction is implemented by setting the ray interval to $[\varepsilon, 1 - \varepsilon]$, giving the interval shown in light blue in this illustration.*

A maximum value is given when hits beyond a certain distance do not matter, such as for shadow rays. Assume that we are shading point P and want to query visibility of a light at L. We create a shadow ray with origin at $O = P$, unnormalized direction vector $\mathbf{d} = L - P$, $t_{\text{min}} = 0$, and $t_{\text{max}} = 1$. If an intersection occurs with t in $[0, 1]$, the ray intersects geometry occluding the light. In practice, we often set $t_{\text{min}} = \varepsilon$ and $t_{\text{max}} = 1 - \varepsilon$, for a small number ε. This adjustment helps avoid *self-intersections* due to numerical imprecision; using floating-point mathematics, the surface on which P lies may intersect our ray at a small, nonzero value of t. For non-point lights the light's primitive should not occlude the shadow ray, so we shorten the interval using $t_{\text{max}} = 1 - \varepsilon$. With perfect mathematics, this problem disappears using an *open interval*, ignoring intersections at precisely $t = 0$ and 1. Since floating-point precision is limited, use of ε fudge factors are a common solution. See Chapter 6 for more information about how to avoid self-intersections.

In implementations using normalized ray directions, we could instead use $O = P$, $\mathbf{d} = \frac{L-P}{\|L-P\|}$, $t_{\text{min}} = \varepsilon$, and $t_{\text{max}} = l - \varepsilon$, where $l = \|L - P\|$ is the distance to the light source L. Note that this epsilon must be different than the previous epsilon, as t now has a different range.

Some renderers use unit-length vectors for all or some ray directions. Doing so allows efficient cosine computations via dot products with other unit vectors, and it can make it easier to reason about the code, in addition to making it more readable. As noted earlier, a unit length means that the ray parameter t can be interpreted as a distance without scaling by the direction vector's length. However, instanced geometry may be represented using a transformation for each instance. Ray/object intersection then requires transforming the ray into the object's space, which changes the length of the direction vector. To properly compute t in this new space, this transformed direction should be left unnormalized. In addition, normalization costs a little performance and can be unnecessary, as for shadow rays. Because of these competing benefits, there is no universal recommendation of whether to use unit direction vectors.

2.3 RAYS IN DXR

This section presents the definition of a ray in DirectX Raytracing [3]. In DXR, a ray is defined by the following structure:

```
1 struct RayDesc
2 {
3     float3 Origin;
4     float  TMin;
5     float3 Direction;
6     float  TMax;
7 };
```

The ray type is handled differently in DXR, where a certain shader program is associated with each different type of ray. To trace a ray with the `TraceRay()` function in DXR, a `RayDesc` is needed. The `RayDesc::Origin` is set to the origin O of our ray, the `RayDesc::Direction` is set to the direction d, and the t-interval (`RayDesc::TMin` and `RayDesc::TMax`) must be initialized as well. For example, for an eye ray (`RayDesc eyeRay`) we set `eyeRay.TMin = 0.0` and `eyeRay.TMax = FLT_MAX`, which indicates that we are interested in all intersections that are in front of the origin.

2.4 CONCLUSION

This chapter shows how a ray is typically defined and used in a ray tracer, and gave the DXR API's ray definition as an example. Other ray tracing systems, such as OptiX [1] and the Vulkan ray tracing extension [2], have minor variations. For example, OptiX explicitly defines a ray type, such as a shadow ray. These systems have other commonalities, such as the idea of a *ray payload*. This is a data structure that can be defined by the user to carry additional information along with the ray that can be accessed and edited by separate shaders or modules. Such data is application specific. At the core, in every rendering system that defines a ray, you will find the ray's origin, direction, and interval.

REFERENCES

[1] NVIDIA. OptiX 5.1 Programming Guide. http://raytracing-docs.nvidia.com/optix/guide/index.html, Mar. 2018.

[2] Subtil, N. Introduction to Real-Time Ray Tracing with Vulkan. NVIDIA Developer Blog, https://devblogs.nvidia.com/vulkan-raytracing/, Oct. 2018.

[3] Wyman, C., Hargreaves, S., Shirley, P., and Barré-Brisebois, C. Introduction to DirectX RayTracing. SIGGRAPH Courses, Aug. 2018.