Estructura de datos y algoritmos geométricos de sistemas gráficos
2023-2024

Ray Tracing Geometry

Estudiante
Dmitry Ivanov (divanov@correo.ugr.es)
Profesor
Juan Manuel Jurado Rodríguez (jjurado@ujaen.es)

UNIVERSIDAD
DE GRANADA

25 de abril de 2024

1. Introduction

When we say ray tracing, the only geometrical object to pay attention to is the Ray. It is partly true because in Ray-Tracing we apply different Ray behaviours interacting with different parts of the scene we created. On the other hand, everything on the scene is created with the help of computational geometry, describing geometrical objects of many kinds, and we need to understand how they behave. Another topic where computational geometry involves 100 per cent is optimizations where we can meet Convex Hulls, Bounding Volumes and Hierarchies. In the present work, I want to describe the ray-tracing basics from the geometrical standpoint and how computation geometry helps to create astonishingly realistic renderings.

2. Ray

Even though ray tracing technology utilizes Ray as the basic geometric term, ray tracing engines work with a parametric line. Peter Shirley et. al. describe [HAM19] the Ray as a parametric line:

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

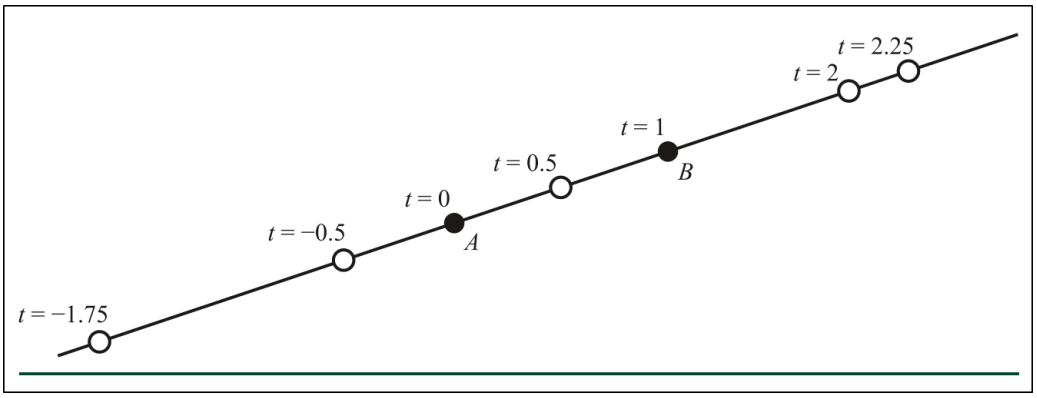

Figure 1: How changing values of t gives different points on the line.

Instead of two points, choosing a point and a direction is better. We can define point B as d (*direction*) and point A as O (*origin*). For various computation reasons, like computing cosines with dot products, it's preferable to use a normalized vector as a direction:

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

We may select any values of t and the point P moves continuously along the line. When we use a normalized vector as a direction, the value of t represents the signed distance from the origin.

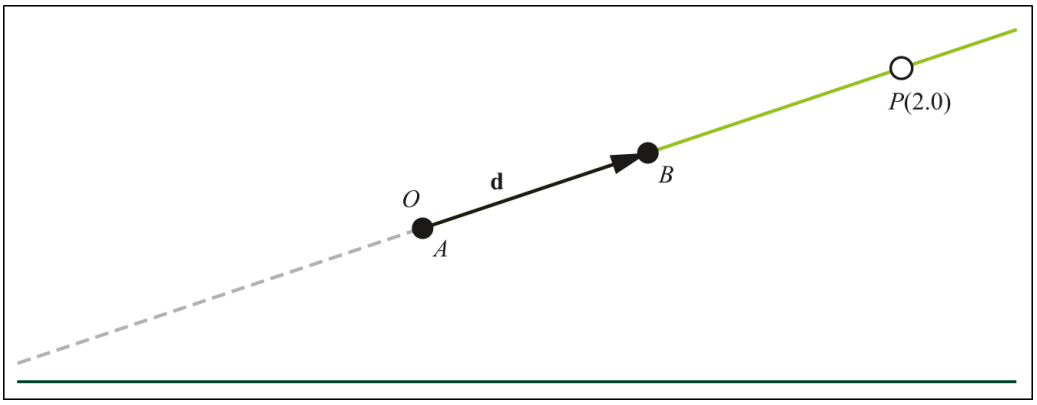

Figure 2: A Ray, described by an origin O and direction vector $\hat{d}$. The points are in front of the origins, i.e. $t > 0$. The dashed line represents points behind the origin.

3. Intersections

Once we generate a ray, we need to wind its intersections with various geometrical objects. Different geometries can behave differently depending on the shape or material of an object. Historically, each ray training engine starts with ray-sphere interactions [Hai89]:

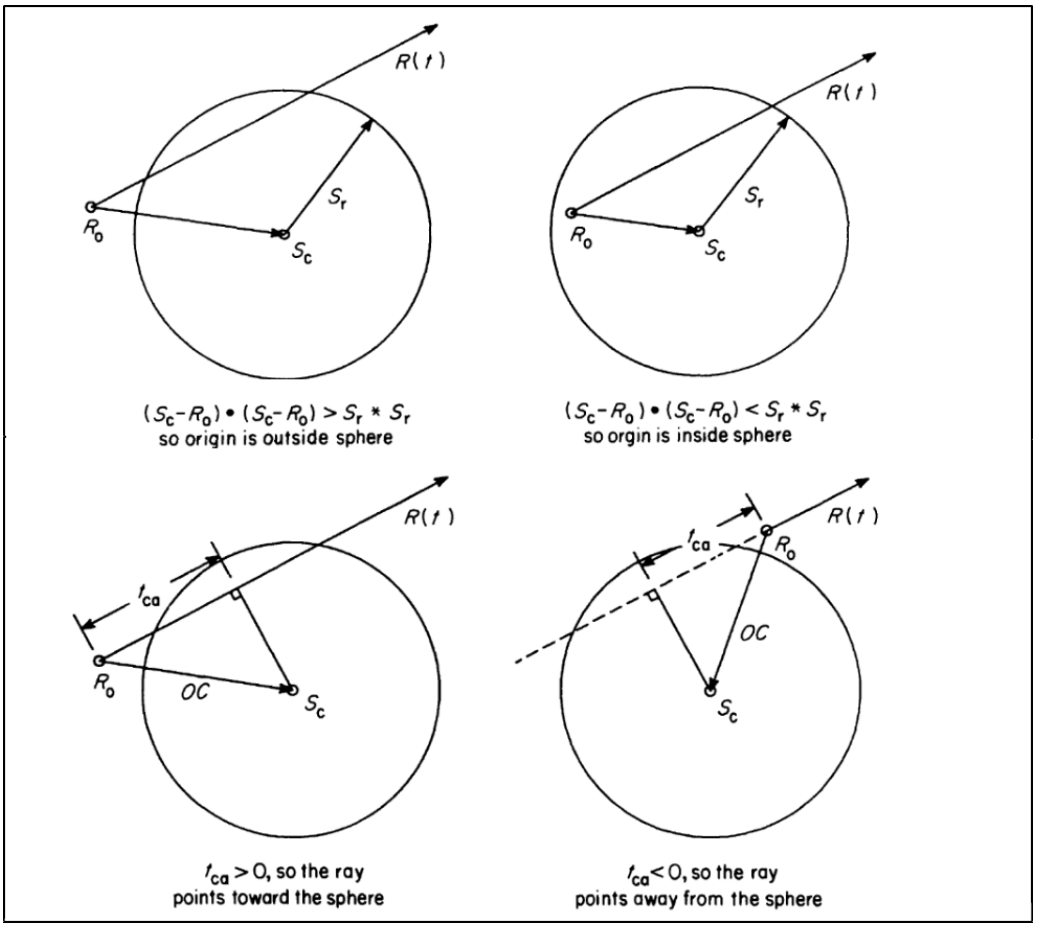

Figure 3: The ray origin with respect to sphere location.

After defining intersections with spherical geometries, the ray tracing engine should get ray-plane intersections for constructing quadrilaterals and further usage in ray-polygon intersection algorithms.

We know a point is on the surface of a plane if it satisfies the plane equation:

$$\begin{aligned} Plane = Ax + Bx + Cz + D = 0 \\ where : A^2 + B^2 + C^2 = 1 \end{aligned} \tag{3}$$

If no point along the ray satisfies the plane equation, the ray and plane do not intersect [Sza17]:

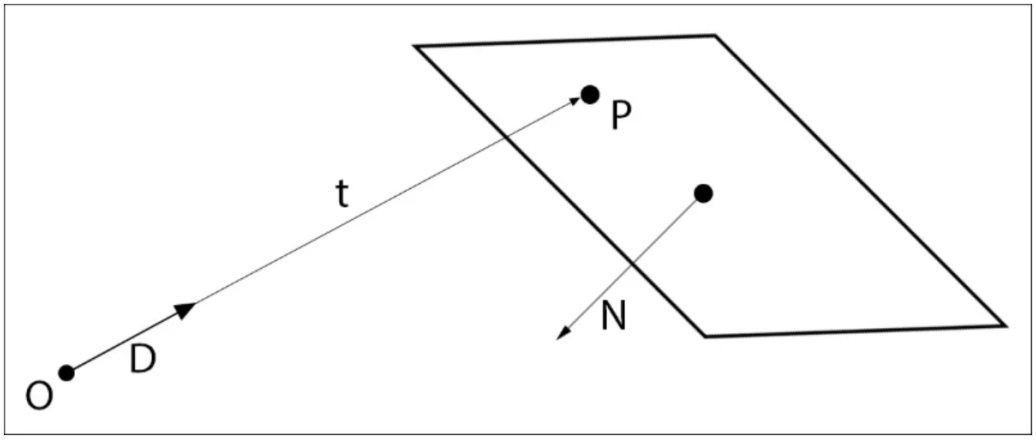

Figure 4: Ray-Plane intersection diagram.

Once the ray-plane intersection is defined, the ray-polygon intersection can be performed. Eric Haines presented [Hai89] one of many methods for testing the location of points (inside or outside), known as the Jordan curve theorem:

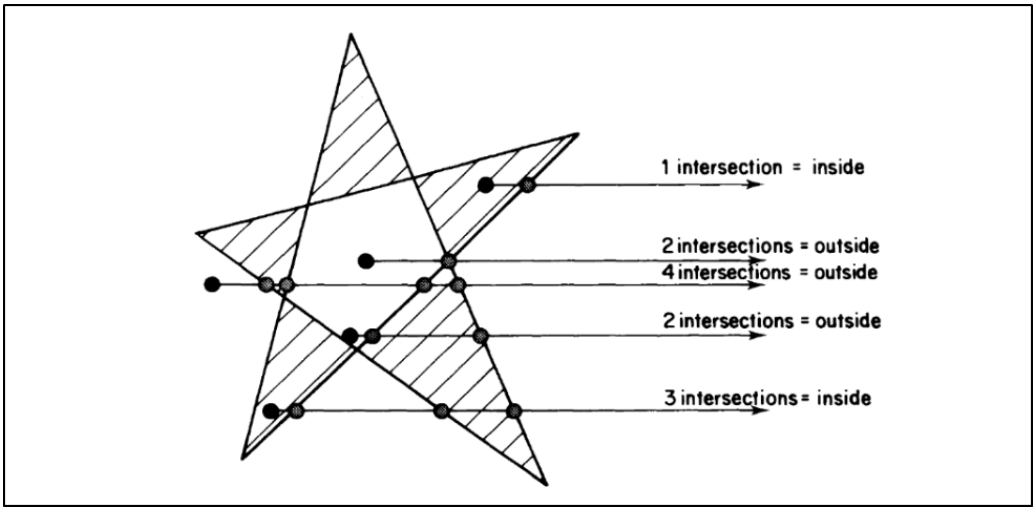

Figure 5: Ray-Plane intersection diagram.

This algorithm works by shooting rays in an arbitrary direction and counting the number of intersections with the polygon. If the number is odd, the point is inside the polygon; otherwise, it is outside. The Jordan curve theorem is a fundamental concept in computational geometry. Once the ray-polygon interacting achieved, the ray tracing engine can construct more complex geometries like triangles quadrilaterals, and other polygons.

4. Optimization

Another important area in ray tracing is performance optimizations and acceleration techniques. Using different geometry structures allows for the reduction of complicated computations, increasing the rendering time of a scene. The figure 6 presents the broad classification of various optimization approaches described by Arvo and Kirk [AK89].

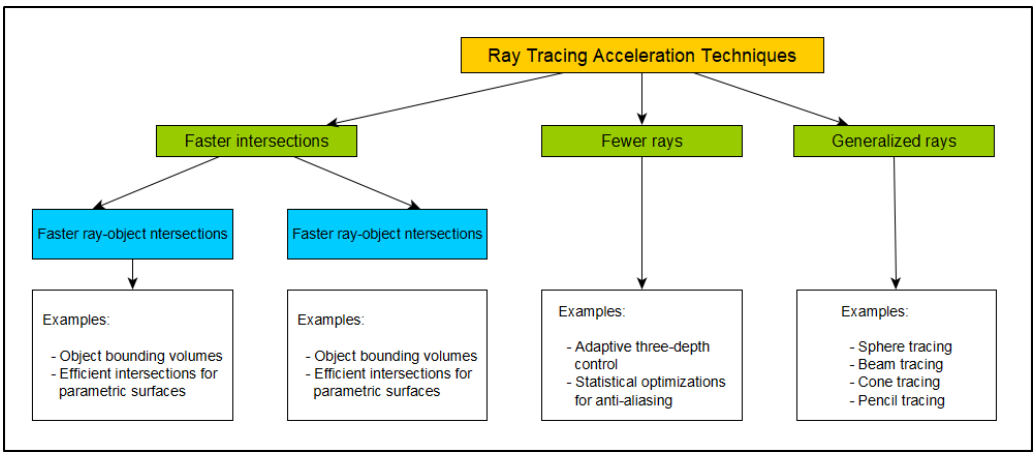

Figure 6: A broad classification of acceleration techniques.

BVH trees are the most important optimization technology in ray tracing. A scene can consist of thousands of objects. Some of them are visible, some of them are not. If an object invisible, it could be traceable or not traceable. BVH constructions and bounding box hit-checking algorithms help to the time of initial scene construction and ray-tracing computations. The figures 7, 8, 9, 10, 11 show the increasing number of objects on the ray traced scene. Each of the spheres wrapped into a bounding box and the whole scene exists as one BVH construction.

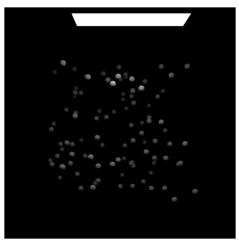

Figure 7:
100 objects
Time: 5s

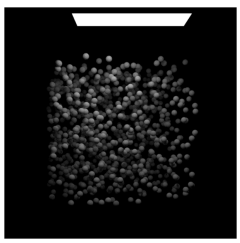

Figure 8:
1000 objects
Time: 27s

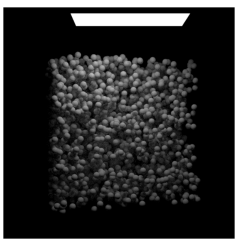

Figure 9:
5000 objects
Time: 106s

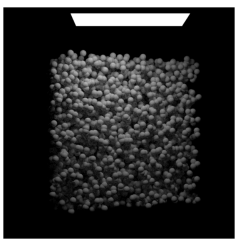

Figure 10:
10000 objects
Time: 112s

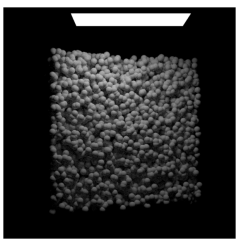

Figure 11:
20000 objects
Time: 149s

As was shown in the figures above, the computation time for 100 samples per pixel doesn't increase too much when sphered starting to overlap each other.

Another interesting optimization approach proposed by Alexander Reshetov [Res19]. Their GARP method (Geometric Approach to Ray/bilinear Patch intersections) is trying to find a balance between the simplicity of triangles and the richness of such smooth shapes as subdivision surfaces, NURBS, and Bézier patches.

The intersection point could be computed as either $X_r = R(t)$ or as $X_q = Q(u, v)$ using the found parameters t, u, and v. The two-step GARP process dynamically reduces a possible error in each step. In the first step, we find the best estimation for u. On the second step, using the found $u - aim$, minimizing the total error.

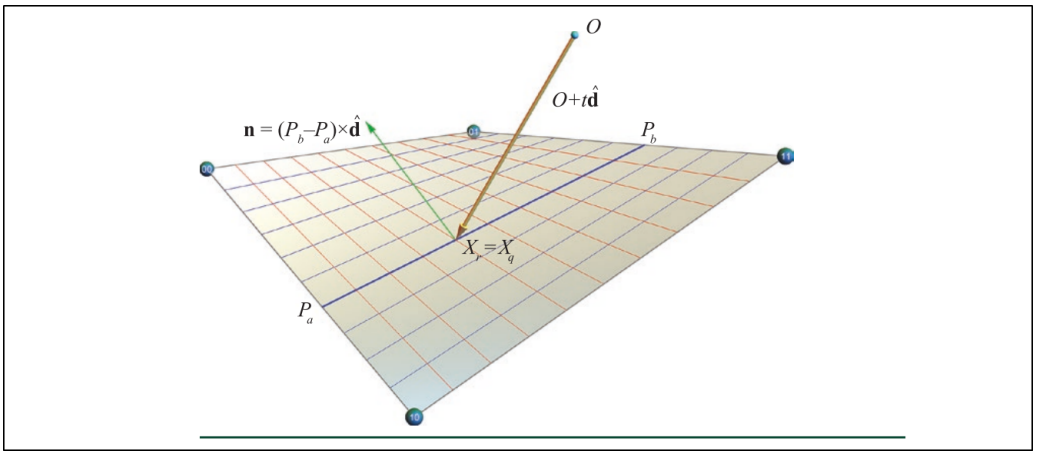

Figure 12: Finding ray/patch intersections.

The figure 13 shows the performance measurements performed by the author by counting the total number of rays processed per second.

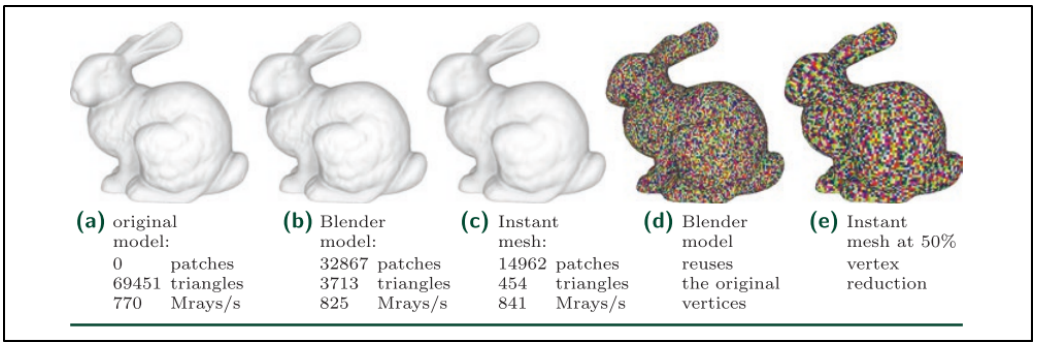

Figure 13: Finding ray/patch intersections.

5. Conclusion

As was shown, the computational geometry is the essential part of the ray tracing ecosystem that works in different areas: scene construction, object relations, and optimizations.

References

[AK89] James Arvo and David Kirk. *A survey of ray tracing acceleration techniques*, page 201–262. Academic Press Ltd., GBR, July 1989.

[Hai89] Eric Haines. *Essential ray tracing algorithms*, pages 33–77. Academic Press Ltd., GBR, July 1989.

[HAM19] Eric Haines and Tomas Akenine-Möller, editors. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, Berkeley, CA, 2019.

[Res19] Alexander Reshetov. *Cool Patches: A Geometric Approach to Ray/Bilinear Patch Intersections*, page 95–109. Apress, Berkeley, CA, 2019.

[SM09] Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., USA, 3rd edition, June 2009.

[Sza17] Gabor Szauer. *Game Physics Cookbook*. Packt Publishing Ltd., UK, 2017.