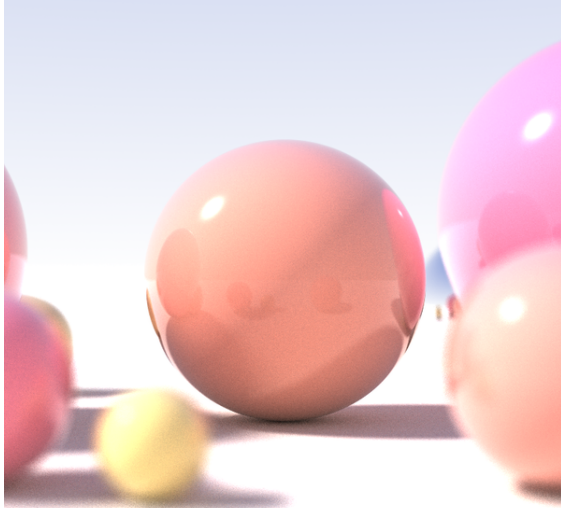


Ray tracing (graphics)

Not to be confused with Ray casting or Ray tracing (physics).

In computer graphics, **ray tracing** is a technique for

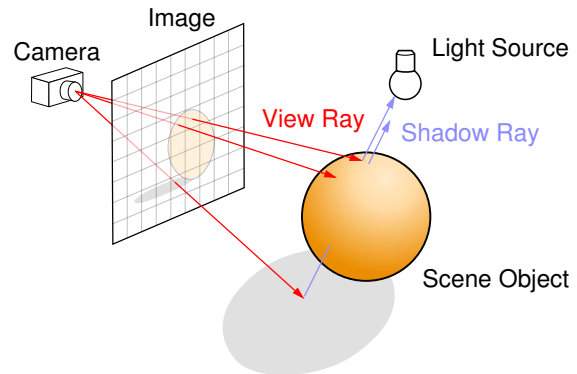


This recursive ray tracing of a sphere demonstrates the effects of shallow depth of field, area light sources and diffuse interreflection.

generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television visual effects, and more poorly suited for real-time applications like video games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and dispersion phenomena (such as chromatic aberration).

1 Algorithm overview

Optical ray tracing describes a method for producing visual images constructed in 3D computer graphics environments, with more photorealism than either ray casting or scanline rendering techniques. It works by tracing a path from an imaginary eye through each pixel in a virtual screen, and calculating the color of the object visible



The ray tracing algorithm builds an image by extending rays into a scene

through it.

Scenes in ray tracing are described mathematically by a programmer or by a visual artist (typically using intermediary tools). Scenes may also incorporate data from images and models captured by means such as digital photography.

Typically, each ray must be tested for intersection with some subset of all the objects in the scene. Once the nearest object has been identified, the algorithm will estimate the incoming light at the point of intersection, examine the material properties of the object, and combine this information to calculate the final color of the pixel. Certain illumination algorithms and reflective or translucent materials may require more rays to be re-cast into the scene.

It may at first seem counterintuitive or “backwards” to send rays away from the camera, rather than into it (as actual light does in reality), but doing so is many orders of magnitude more efficient. Since the overwhelming majority of light rays from a given light source do not make it directly into the viewer’s eye, a “forward” simulation could potentially waste a tremendous amount of computation on light paths that are never recorded.

Therefore, the shortcut taken in raytracing is to presuppose that a given ray intersects the view frame. After either a maximum number of reflections or a ray traveling a certain distance without intersection, the ray ceases to travel and the pixel’s value is updated.

2 Detailed description of ray tracing computer algorithm and its genesis

2.1 What happens in nature

In nature, a light source emits a ray of light which travels, eventually, to a surface that interrupts its progress. One can think of this “ray” as a stream of **photons** traveling along the same path. In a perfect vacuum this ray will be a straight line (ignoring **relativistic effects**). Any combination of four things might happen with this light ray: **absorption**, **reflection**, **refraction** and **fluorescence**. A surface may absorb part of the light ray, resulting in a loss of intensity of the reflected and/or refracted light. It might also reflect all or part of the light ray, in one or more directions. If the surface has any **transparent** or **translucent** properties, it refracts a portion of the light beam into itself in a different direction while absorbing some (or all) of the **spectrum** (and possibly altering the color). Less commonly, a surface may absorb some portion of the light and fluorescently re-emit the light at a longer wavelength colour in a random direction, though this is rare enough that it can be discounted from most rendering applications. Between absorption, reflection, refraction and fluorescence, all of the incoming light must be accounted for, and no more. A surface cannot, for instance, reflect 66% of an incoming light ray, and refract 50%, since the two would add up to be 116%. From here, the reflected and/or refracted rays may strike other surfaces, where their absorptive, refractive, reflective and fluorescent properties again affect the progress of the incoming rays. Some of these rays travel in such a way that they hit our eye, causing us to see the scene and so contribute to the final rendered image.

2.2 Ray casting algorithm

Main article: [Ray casting](#)

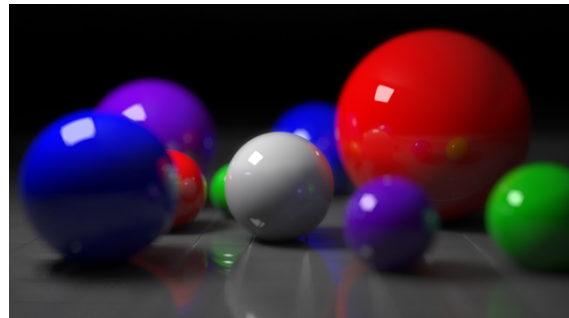
The first ray tracing algorithm used for rendering was presented by **Arthur Appel**^[1] in 1968. This algorithm has since been termed “ray casting”. The idea behind ray casting is to shoot rays from the eye, one per pixel, and find the closest object blocking the path of that ray. Think of an image as a screen-door, with each square in the screen being a pixel. This is then the object the eye sees through that pixel. Using the material properties and the effect of the lights in the scene, this algorithm can determine the **shading** of this object. The simplifying assumption is made that if a surface faces a light, the light will reach that surface and not be blocked or in shadow. The shading of the surface is computed using traditional 3D computer graphics shading models. One important advantage ray casting offered over older **scanline algo-**

rithms was its ability to easily deal with non-planar surfaces and solids, such as **cones** and **spheres**. If a mathematical surface can be intersected by a ray, it can be rendered using ray casting. Elaborate objects can be created by using **solid modeling** techniques and easily rendered.

2.3 Recursive ray tracing algorithm

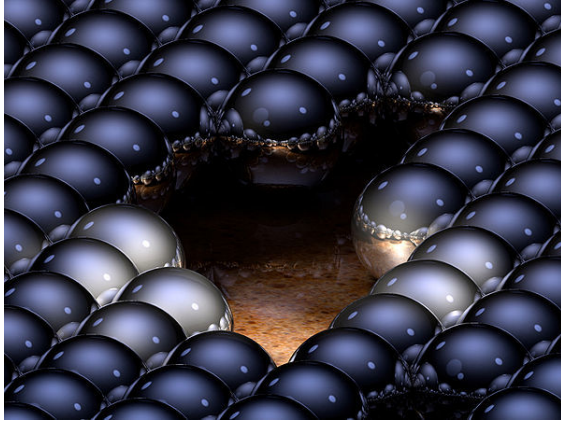


Ray tracing can create realistic images.

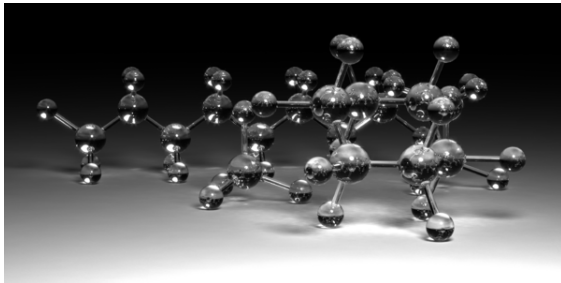


*In addition to the high degree of realism, ray tracing can simulate the effects of a camera due to depth of field and aperture shape (in this case a **hexagon**).*

The next important research breakthrough came from **Turner Whitted** in 1979.^[2] Previous algorithms traced rays from the eye into the scene until they hit an object, but determined the ray color without recursively tracing more rays. Whitted continued the process. When a ray hits a surface, it can generate up to three new types of rays: reflection, refraction, and shadow.^[3] A reflection ray is traced in the mirror-reflection direction. The closest object it intersects is what will be seen in the reflection. Refraction rays traveling through transparent material work similarly, with the addition that a refractive ray could be entering or exiting a material. A shadow ray is traced toward each light. If any opaque object is found between the surface and the light, the surface is in shadow and the light does not illuminate it. This recursive ray tracing added more realism to ray traced images.



The number of reflections a “ray” can take and how it is affected each time it encounters a surface is all controlled via software settings during ray tracing. Here, each ray was allowed to reflect up to 16 times. Multiple “reflections of reflections” can thus be seen. Created with Cobalt



The number of refractions a “ray” can take and how it is affected each time it encounters a surface is all controlled via software settings during ray tracing. Here, each ray was allowed to refract and reflect up to 9 times. Fresnel refractions were used. Also note the caustics. Created with Vray

2.4 Advantages over other rendering methods

Ray tracing’s popularity stems from its basis in a realistic simulation of lighting over other rendering methods (such as scanline rendering or ray casting). Effects such as reflections and shadows, which are difficult to simulate using other algorithms, are a natural result of the ray tracing algorithm. The computational independence of each ray makes ray tracing amenable to parallelization.^{[4][5]}

2.5 Disadvantages

A serious disadvantage of ray tracing is performance. Scanline algorithms and other algorithms use data coherence to share computations between pixels, while ray tracing normally starts the process anew, treating each eye ray separately. However, this separation offers other advantages, such as the ability to shoot more rays as needed to perform spatial anti-aliasing and improve image quality where needed.

Although it does handle interreflection and optical effects such as refraction accurately, traditional ray tracing is also not necessarily photorealistic. True photorealism occurs when the rendering equation is closely approximated or fully implemented. Implementing the rendering equation gives true photorealism, as the equation describes every physical effect of light flow. However, this is usually infeasible given the computing resources required.

The realism of all rendering methods can be evaluated as an approximation to the equation. Ray tracing, if it is limited to Whitted’s algorithm, is not necessarily the most realistic. Methods that trace rays, but include additional techniques (photon mapping, path tracing), give far more accurate simulation of real-world lighting.

It is also possible to approximate the equation using ray casting in a different way than what is traditionally considered to be “ray tracing”. For performance, rays can be clustered according to their direction, with rasterization hardware and depth peeling used to efficiently sum the rays.^[6]

2.6 Reversed direction of traversal of scene by the rays

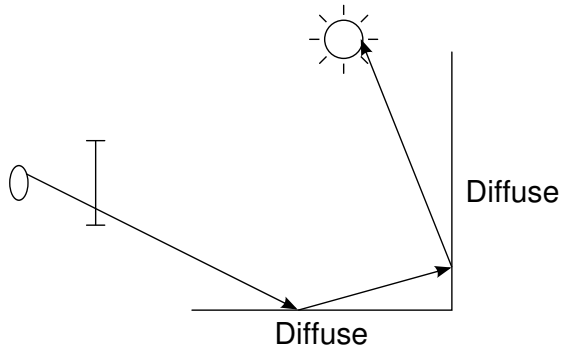
The process of shooting rays from the eye to the light source to render an image is sometimes called *backwards ray tracing*, since it is the opposite direction photons actually travel. However, there is confusion with this terminology. Early ray tracing was always done from the eye, and early researchers such as James Arvo used the term *backwards ray tracing* to mean shooting rays from the lights and gathering the results. Therefore it is clearer to distinguish *eye-based* versus *light-based* ray tracing.

While the direct illumination is generally best sampled using eye-based ray tracing, certain indirect effects can benefit from rays generated from the lights. Caustics are bright patterns caused by the focusing of light off a wide reflective region onto a narrow area of (near-)diffuse surface. An algorithm that casts rays directly from lights onto reflective objects, tracing their paths to the eye, will better sample this phenomenon. This integration of eye-based and light-based rays is often expressed as bidirectional path tracing, in which paths are traced from both the eye and lights, and the paths subsequently joined by a connecting ray after some length.^{[7][8]}

Photon mapping is another method that uses both light-based and eye-based ray tracing; in an initial pass, energetic photons are traced along rays from the light source so as to compute an estimate of radiant flux as a function of 3-dimensional space (the eponymous photon map itself). In a subsequent pass, rays are traced from the eye into the scene to determine the visible surfaces, and the photon map is used to estimate the illumination at the visible surface points.^{[9][10]} The advantage of photon mapping versus bidirectional path tracing is the ability to achieve significant reuse of photons, reducing computa-

tion, at the cost of statistical bias.

An additional problem occurs when light must pass through a very narrow aperture to illuminate the scene (consider a darkened room, with a door slightly ajar leading to a brightly lit room), or a scene in which most points do not have direct line-of-sight to any light source (such as with ceiling-directed light fixtures or *torchieres*). In such cases, only a very small subset of paths will transport energy; *Metropolis light transport* is a method which begins with a random search of the path space, and when energetic paths are found, reuses this information by exploring the nearby space of rays.^[11]



To the right is an image showing a simple example of a path of rays recursively generated from the camera (or eye) to the light source using the above algorithm. A diffuse surface reflects light in all directions.

First, a ray is created at an eyepoint and traced through a pixel and into the scene, where it hits a diffuse surface. From that surface the algorithm recursively generates a reflection ray, which is traced through the scene, where it hits another diffuse surface. Finally, another reflection ray is generated and traced through the scene, where it hits the light source and is absorbed. The color of the pixel now depends on the colors of the first and second diffuse surface and the color of the light emitted from the light source. For example if the light source emitted white light and the two diffuse surfaces were blue, then the resulting color of the pixel is blue.

2.7 Example

As a demonstration of the principles involved in raytracing, let us consider how one would find the intersection between a ray and a sphere. This is merely the math behind the *line-sphere intersection* and the subsequent determination of the colour of the pixel being calculated. There is, of course, far more to the general process of raytracing, but this demonstrates an example of the algorithms used.

In *vector notation*, the equation of a sphere with center \mathbf{c} and radius r is

$$\|\mathbf{x} - \mathbf{c}\|^2 = r^2.$$

Any point on a ray starting from point \mathbf{s} with direction \mathbf{d} (here \mathbf{d} is a *unit vector*) can be written as

$$\mathbf{x} = \mathbf{s} + t\mathbf{d},$$

where t is its distance between \mathbf{x} and \mathbf{s} . In our problem, we know \mathbf{c} , \mathbf{r} , \mathbf{s} (e.g. the position of a light source) and \mathbf{d} , and we need to find t . Therefore, we substitute for \mathbf{x} :

$$\|\mathbf{s} + t\mathbf{d} - \mathbf{c}\|^2 = r^2.$$

Let $\mathbf{v} \stackrel{\text{def}}{=} \mathbf{s} - \mathbf{c}$ for simplicity; then

$$\|\mathbf{v} + t\mathbf{d}\|^2 = r^2$$

$$\mathbf{v}^2 + t^2\mathbf{d}^2 + 2\mathbf{v} \cdot t\mathbf{d} = r^2$$

$$(\mathbf{d}^2)t^2 + (2\mathbf{v} \cdot \mathbf{d})t + (\mathbf{v}^2 - r^2) = 0.$$

Knowing that \mathbf{d} is a unit vector allows us this minor simplification:

$$t^2 + (2\mathbf{v} \cdot \mathbf{d})t + (\mathbf{v}^2 - r^2) = 0.$$

This *quadratic equation* has solutions

$$t = \frac{-(2\mathbf{v} \cdot \mathbf{d}) \pm \sqrt{(2\mathbf{v} \cdot \mathbf{d})^2 - 4(\mathbf{v}^2 - r^2)}}{2} = -(\mathbf{v} \cdot \mathbf{d}) \pm \sqrt{(\mathbf{v} \cdot \mathbf{d})^2 - (\mathbf{v}^2 - r^2)}$$

The two values of t found by solving this equation are the two ones such that $\mathbf{s} + t\mathbf{d}$ are the points where the ray intersects the sphere.

Any value which is negative does not lie on the ray, but rather in the opposite *half-line* (i.e. the one starting from \mathbf{s} with opposite direction).

If the quantity under the square root (the *discriminant*) is negative, then the ray does not intersect the sphere.

Let us suppose now that there is at least a positive solution, and let t be the minimal one. In addition, let us suppose that the sphere is the nearest object on our scene intersecting our ray, and that it is made of a reflective material. We need to find in which direction the light ray is reflected. The laws of *reflection* state that the angle of reflection is equal and opposite to the angle of incidence between the incident ray and the *normal* to the sphere.

The normal to the sphere is simply

$$\mathbf{n} = \frac{\mathbf{y} - \mathbf{c}}{\|\mathbf{y} - \mathbf{c}\|},$$

where $\mathbf{y} = \mathbf{s} + t\mathbf{d}$ is the intersection point found before. The reflection direction can be found by a **reflection** of \mathbf{d} with respect to \mathbf{n} , that is

$$\mathbf{r} = \mathbf{d} - 2(\mathbf{n} \cdot \mathbf{d})\mathbf{n}.$$

Thus the reflected ray has equation

$$\mathbf{x} = \mathbf{y} + u\mathbf{r}.$$

Now we only need to compute the intersection of the latter ray with our **field of view**, to get the pixel which our reflected light ray will hit. Lastly, this pixel is set to an appropriate color, taking into account how the color of the original light source and the one of the sphere are combined by the reflection.

3 Adaptive depth control

This means that we stop generating reflected/transmitted rays when the computed intensity becomes less than a certain threshold. You must always set a certain maximum depth or else the program would generate an infinite number of rays. But it is not always necessary to go to the maximum depth if the surfaces are not highly reflective. To test for this the ray tracer must compute and keep the product of the global and reflection coefficients as the rays are traced.

Example: let $K_r = 0.5$ for a set of surfaces. Then from the first surface the maximum contribution is 0.5, for the reflection from the second: $0.5 * 0.5 = 0.25$, the third: $0.25 * 0.5 = 0.125$, the fourth: $0.125 * 0.5 = 0.0625$, the fifth: $0.0625 * 0.5 = 0.03125$, etc. In addition we might implement a distance attenuation factor such as $1/D^2$, which would also decrease the intensity contribution.

For a transmitted ray we could do something similar but in that case the distance traveled through the object would cause even faster intensity decrease. As an example of this, Hall & Greenberg found that even for a very reflective scene, using this with a maximum depth of 15 resulted in an average ray tree depth of 1.7.

4 Bounding volumes

We enclose groups of objects in sets of **hierarchical bounding volumes** and first test for intersection with the **bounding volume**, and then only if there is an intersection, against the objects enclosed by the volume.

Bounding volumes should be easy to test for intersection, for example a sphere or box (slab). The best bounding volume will be determined by the shape of the underlying object or objects. For example, if the objects are long and

thin then a sphere will enclose mainly empty space and a box is much better. Boxes are also easier for hierarchical bounding volumes.

Note that using a hierarchical system like this (assuming it is done carefully) changes the intersection computational time from a linear dependence on the number of objects to something between linear and a logarithmic dependence. This is because, for a perfect case, each intersection test would divide the possibilities by two, and we would have a binary tree type structure. Spatial subdivision methods, discussed below, try to achieve this.

Kay & Kajiya give a list of desired properties for hierarchical bounding volumes:

- Subtrees should contain objects that are near each other and the further down the tree the closer should be the objects.
- The volume of each node should be minimal.
- The sum of the volumes of all bounding volumes should be minimal.
- Greater attention should be placed on the nodes near the root since pruning a branch near the root will remove more potential objects than one farther down the tree.
- The time spent constructing the hierarchy should be much less than the time saved by using it.

5 In real time

The first implementation of a “real-time” ray-tracer was credited at the 2005 **SIGGRAPH** computer graphics conference as the REMRT/RT tools developed in 1986 by **Mike Muuss** for the **BRL-CAD** solid modeling system. Initially published in 1987 at **USENIX**, the BRL-CAD ray-tracer is the first known implementation of a parallel network distributed ray-tracing system that achieved several frames per second in rendering performance.^[12] This performance was attained by means of the highly optimized yet platform independent LIBRT ray-tracing engine in BRL-CAD and by using solid implicit **CSG** geometry on several shared memory parallel machines over a commodity network. BRL-CAD’s ray-tracer, including REMRT/RT tools, continue to be available and developed today as **Open source** software.^[13]

Since then, there have been considerable efforts and research towards implementing ray tracing in real time speeds for a variety of purposes on stand-alone desktop configurations. These purposes include interactive 3D graphics applications such as **demoscene productions**, **computer and video games**, and image rendering. Some real-time software 3D engines based on ray tracing have been developed by hobbyist **demo programmers** since the late 1990s.^[14]

The **OpenRT** project includes a highly optimized software core for ray tracing along with an **OpenGL**-like API in order to offer an alternative to the current rasterisation based approach for interactive 3D graphics. **Ray tracing hardware**, such as the experimental **Ray Processing Unit** developed at the **Saarland University**, has been designed to accelerate some of the computationally intensive operations of ray tracing. On March 16, 2007, the University of Saarland revealed an implementation of a high-performance ray tracing engine that allowed computer games to be rendered via ray tracing without intensive resource usage.^[15]

On June 12, 2008 Intel demonstrated a special version of *Enemy Territory: Quake Wars*, titled *Quake Wars: Ray Traced*, using ray tracing for rendering, running in basic HD (720p) resolution. ETQW operated at 14-29 frames per second. The demonstration ran on a 16-core (4 socket, 4 core) Xeon Tigerton system running at 2.93 GHz.^[16]

At SIGGRAPH 2009, Nvidia announced **OptiX**, a free API for real-time ray tracing on Nvidia GPUs. The API exposes seven programmable entry points within the ray tracing pipeline, allowing for custom cameras, ray-primitive intersections, shaders, shadowing, etc. This flexibility enables bidirectional path tracing, Metropolis light transport, and many other rendering algorithms that cannot be implemented with tail recursion.^[17] Nvidia has shipped over 350,000,000 OptiX capable GPUs as of April 2013. OptiX-based renderers are used in Adobe AfterEffects, Bunspeed Shot, Autodesk Maya, 3ds max, and many other renderers.

Imagination Technologies offers a free API called **OpenRL** which accelerates tail recursive ray tracing-based rendering algorithms and, together with their proprietary ray tracing hardware, works with Autodesk Maya to provide what 3D World calls “real-time raytracing to the everyday artist”.^[18]

6 Computational Complexity

Various complexity results have been proven for certain formulations of the ray tracing problem. In particular, if the decision version of the ray tracing problem is defined as follows^[19] - given a light ray's initial position and direction and some fixed point, does the ray eventually reach that point, then the referenced paper proves the following results :-

- Ray tracing in 3D optical systems with a finite set of reflective or refractive objects represented by a system of rational quadratic inequalities is undecidable.
- Ray tracing in 3D optical systems with a finite set of refractive objects represented by a system of rational linear inequalities is undecidable.

- Ray tracing in 3D optical systems with a finite set of rectangular reflective or refractive objects is undecidable.
- Ray tracing in 3D optical systems with a finite set of reflective or partially reflective objects represented by a system of linear inequalities, some of which can be irrational is undecidable.
- Ray tracing in 3D optical systems with a finite set of reflective or partially reflective objects represented by a system of rational linear inequalities is PSPACE-hard.
- For any dimension equal to or greater than 2, ray tracing with a finite set of parallel and perpendicular reflective surfaces represented by rational linear inequalities is in PSPACE.

7 See also

- Beam tracing
- Cone tracing
- Distributed ray tracing
- Global illumination
- Gouraud shading
- List of ray tracing software
- Parallel computing
- Phong shading
- Progressive refinement
- Rendering (computer graphics)
- Shading
- Specular reflection

8 References

- [1] Appel A. (1968) *Some techniques for shading machine renderings of solids*. AFIPS Conference Proc. 32 pp.37-45
- [2] Whitted T. (1979) *An improved illumination model for shaded display*. Proceedings of the 6th annual conference on Computer graphics and interactive techniques
- [3] Tomas Nikodym (June 2010). “Ray Tracing Algorithm For Interactive Applications”. *Czech Technical University, FEE*.
- [4] J.-C. Nebel. A New Parallel Algorithm Provided by a Computation Time Model, Eurographics Workshop on Parallel Graphics and Visualisation, 24-25 September 1998, Rennes, France.

- [5] A. Chalmers, T. Davis, and E. Reinhard. Practical parallel rendering, ISBN 1-56881-179-9. AK Peters, Ltd., 2002.
- [6] GPU Gems 2, Chapter 38. High-Quality Global Illumination Rendering Using Rasterization, Addison-Wesley
- [7] Eric P. Lafortune and Yves D. Willems (December 1993). “Bi-Directional Path Tracing”. *Proceedings of Compu-graphics '93*: 145–153.
- [8] Péter Dornbach. “Implementation of bidirectional ray tracing algorithm”. Retrieved 2008-06-11.
- [9] Global Illumination using Photon Maps
- [10] Photon Mapping - Zack Waters
- [11] <http://graphics.stanford.edu/papers/metro/metro.pdf>
- [12] See Proceedings of 4th Computer Graphics Workshop, Cambridge, MA, USA, October 1987. Usenix Association, 1987. pp 86–98.
- [13] “About BRL-CAD”. Retrieved 2009-07-28.
- [14] Piero Foscari. “The Realtime Raytracing Realm”. *ACM Transactions on Graphics*. Retrieved 2007-09-17.
- [15] Mark Ward (March 16, 2007). “Rays light up life-like graphics”. BBC News. Retrieved 2007-09-17.
- [16] Theo Valich (June 12, 2008). “Intel converts ET: Quake Wars to ray tracing”. TG Daily. Retrieved 2008-06-16.
- [17] Nvidia (October 18, 2009). “Nvidia OptiX”. Nvidia. Retrieved 2009-11-06.
- [18] “3DWorld: Hardware review: Caustic Series2 R2500 ray-tracing accelerator card”. Retrieved 2013-04-23. 3D World, April 2013
- [19] “Computability and Complexity of Ray Tracing”. <https://www.cs.duke.edu/~{ }reif/paper/tygar/raytracing.pdf>

9 External links

- What is ray tracing ?
- Ray Tracing and Gaming - Quake 4: Ray Traced Project
- Ray tracing and Gaming - One Year Later
- Interactive Ray Tracing: The replacement of rasterization?
- A series of tutorials on implementing a raytracer using C++
- Tutorial on implementing a raytracer in PHP
- The Compleat Angler (1978)
- Writing a Simple Ray Tracer (scratchapixel)

10 Text and image sources, contributors, and licenses

10.1 Text

- **Ray tracing (graphics)** *Source:* [http://en.wikipedia.org/wiki/Ray%20tracing%20\(graphics\)?oldid=630713809](http://en.wikipedia.org/wiki/Ray%20tracing%20(graphics)?oldid=630713809) *Contributors:* Brion VIBBER, Timo Honkasalo, The Anome, Taral, DrBob, Stevertigo, Dhilvert, Spiff, Patrick, Michael Hardy, Blueshade, Wapcaplet, Ixfd64, SebastianHelm, Hashar, Nohat, Furrykef, Ed g2s, Altenmann, Lupo, Jleedev, Giflile, Ravn, Hugh2414, Tom-, Jpkoester1, Jawed, Zfr, Nerd65536, Shen, Andreas Kaufmann, Qef, Imroy, Slady, Lumrs, Rich Farmbrough, Caesar, Pmsyyz, ArnoldReinhold, RJHall, Bdoseror, Etimbo, Femto, O18, ToastieIL, Brlcad, VBGfscJUn3, Minghong, Anteru, HasharBot, Japsu, Arthena, TheRealFennShysa, Bjorke, Mattbrundage, Gmaxwell, Erich666, GregorB, Joke137, Deltabeignet, Abstracte, Nneonneo, Phrood, Reedbeta, Arnero, Carrionluggage, Viznut, Srieffler, Toxygen, Kri, Whosasking, Wavelength, Mosquitopsu, Anetode, TakingUpSpace, Scs, Voidxor, Claygate, Themunkee, Osmaker, Tom Morris, Rcronk, SmackBot, RDBury, Timrb, Henrikb4, Davepape, Cozdas, Kolibri, Thumperward, Al Hart, Jdh30, Tamfang, H2oski2liv, Wrayal, Jumping cheese, Cybercobra, Abmac, Ryan Roos, Daniel.Cardenas, BrownHairedGirl, Soumyasch, Jim.belk, Pflatau, Dicklyon, Kvng, Hetar, JoeBot, Lasneyx, GDallimore, Pinbucket, Powerslide, FatalError, SkyWalker, Mikiemike, JonesMI, Deon, Skadge, Requestion, Sir Lothar, Fph, Simeon, MoritzMoeller, Cydebot, Alanbly, Chrislk02, CobbSalad, The machine512, Thijs!bot, OnionKnight, Pjvpjv, Davidhorman, Greg L, Stannered, GGGregory, Gioto, Rusty432, ForrestVoight, Jj137, Vendettax, D V S, Ingolfson, RubyQ, 22, Coastline, Penubag, Domsau2, SwiftBot, Jodi.a.schneider, Cadience, Geekrecon, Cdecoro, Uvainio, Ku7485, Lclacer, Yonaa, K.brewster, Camtomlee, Vanished user 342562, Ryan Postlethwaite, Mimigu, Vette92, Fountains of Bryn Mawr, R.cabus, Graphicsguy, Regenspaziergang, Priceman86, Zeno333, Mun206, ColinSSX, KungfuJoe1110, Jesin, Gjelebbink, NimoTh, SieBot, YonaBot, Hertz1888, Darathin, Soler97, Paolo.dL, Dhatfield, Samuelalang, Randomblue, C0nanPayne, Sebastian.mach, Martarius, ClueBot, Luke490, JustinTalbot, Washboardplayer, Excirial, Alexbot, Goodone121, Mate2code, Elizium23, Tired time, Colimaharaj, Ubardak, DumZiBoT, InternetMeme, XLinkBot, Cowpip, Delicious carbuncle, Addbot, ToolmakerSteve, Fgnievinski, Jeancolasp, Yobot, AnomieBOT, Rubinbot, Xqbot, Gidoca, Juhame, Phresnel, Solphusion, VitruV07, Benindigo, FrescoBot, Tuomari, Anonymous the Editor, Gryllida, TobeBot, 0x394a74, Devendermishra, Diannaa, Gregwhitfield, Samjameshall, EmausBot, WikiWriteyWeb, WikitanvirBot, GoingBatty, Niky cz, Dllu, ClueBot NG, Versatranitonlywaytofly, Chellmuth, Helpful Pixie Bot, Wikiedit555, Iskander HFC, Badgerlovestumbler, YFdyh-bot, Tmfs10, Dexbot, Rty3hree, Paulexyn0, Marc saint ourens, ScotXW, Jcnebel and Anonymous: 222

10.2 Images

- **File:5-cell.gif** *Source:* <http://upload.wikimedia.org/wikipedia/commons/d/d8/5-cell.gif> *License:* Public domain *Contributors:* Originally from en.wikipedia; description page is/was here. *Original artist:* Original uploader was JasonHise at en.wikipedia
- **File:BallsRender.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/a/ae/BallsRender.png> *License:* CC-BY-3.0 *Contributors:* Own work (Original text: *I created this work entirely by myself.*) *Original artist:* Mimigu at English Wikipedia
- **File:Commons-logo.svg** *Source:* <http://upload.wikimedia.org/wikipedia/en/4/4a/Commons-logo.svg> *License:* ? *Contributors:* ? *Original artist:* ?
- **File:Edit-clear.svg** *Source:* <http://upload.wikimedia.org/wikipedia/en/f/f2/Edit-clear.svg> *License:* ? *Contributors:* The *Tango!* Desktop Project. *Original artist:* The people from the *Tango!* project. And according to the meta-data in the file, specifically: “Andreas Nilsson, and Jakob Steiner (although minimally).”
- **File:Glass_ochem.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/40/Glass_ochem.png *License:* CC-BY-SA-3.0 *Contributors:* Own work *Original artist:* Purpy Puppet
- **File:Glasses_800_edit.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/e/ec/Glasses_800_edit.png *License:* Public domain *Contributors:* <http://www.oyonale.com/modeles.php?lang=en&page=40> *Original artist:* Gilles Tran
- **File:PathOfRays.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/1/16/PathOfRays.svg> *License:* CC-BY-SA-2.5 *Contributors:* en:Image:PathOfRays.jpg *Original artist:* Traced by User:Stannered, original by en:user:Kolibri
- **File:Question_book-new.svg** *Source:* http://upload.wikimedia.org/wikipedia/en/9/99/Question_book-new.svg *License:* ? *Contributors:* Created from scratch in Adobe Illustrator. Based on Image:Question book.png created by User:Equazcion *Original artist:* Tkgd2007
- **File:Ray-traced_steel_balls.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/3/36/Ray-traced_steel_balls.jpg *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia *Original artist:* Original uploader was Greg L at en.wikipedia
- **File:Ray_trace_diagram.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/8/83/Ray_trace_diagram.svg *License:* GFDL *Contributors:* Own work *Original artist:* Henrik
- **File:Recursive_raytrace_of_a_sphere.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/3/32/Recursive_raytrace_of_a_sphere.png *License:* GFDL *Contributors:* Own work *Original artist:* Tim Babb

10.3 Content license

- Creative Commons Attribution-Share Alike 3.0