A concise summary of modern light transport algorithms

S. Bork

Table of Contents:

Introduction to ray-tracing using Stratified-Monte-Carlo-Integration:

The most well known light-transport algorithm is probably ray-tracing. Here the behavior of light is simulated as closely as applicable without jeopardizing performance. Here rays are cast out from all light sources with an internal color parameter first defined by the color of the light.

Ray intersections with the geometry are then computed and upon contact the internal color of the ray is updated using the absorption of the material and the ray is randomly scattered with the scattering function of the material hit. When a ray hits the virtual camera sensor, its color becomes part of the image at the point of contact with the virtual sensor. This however is wildly inefficient as almost no rays actually hit the camera before the energy of the ray is fully absorbed. This is perfectly realistic, but an algorithm with the same fidelity is desired without requiring the casting of many non-illuminating rays.

The solution is to do the algorithm in reverse. Rays are cast out from the camera instead of the light sources, and the rays scatter away from the camera keeping track of the material properties they hit. This in and of itself does not improve render efficiency,

however this allows some optimization making sure that no ray data goes unused. The trick is to favor scattering directions that take the ray closer to light sources making it more likely to be relevant the scene and then weighting the rays based on how likely their scattering angle would be, based on the scattering probability density function, in the form of a stratified Monte-Carlo-Integration. This means that substantially more rays work to illuminate the virtual camera sensor.
Some approximations of the light transport always have to be made, even if performance is not an issue, as some ray-tracing problems have been shown to be impossible to compute on any pure turing maschine[1] (turing machines, not unlike ZF/ZFC, have been shown to be at least either incomplete or inconsistent[2]).

Bi-Directional path tracing:
Casting rays solely from the virtual camera can create new issues in scenes where the main, very bright, light source is not visible from the point of view of the camera and is instead behind geometry that requires the light to reflect of other objects in the scene before it gets reflected/refracted into view-frustum of the camera. To eliminate this issue bi-directional path tracing casts rays both from the camera and from the light, and then estimates the light flux that could irradiate the virtual sensor upon connecting the path from the camera and the one from the light with a new, unobstructed ray.

> "*The basic idea is that particles are shot at the same time from a selected light source and from the viewing point in much the same way. All hit points on the respective particle paths are then connected using shadow rays and the appropriate contributions are added to the flux of the pixel in question.*"[3]

Computing the flowing light flux is accomplished using similar probability distribution functions (pdfs) as the infamous bidirectional reflectance distribution function (BRDF).

Physically accurate simulation of irradiated light flux:
Computing how much light is reflected away from a surface in some direction is one of the biggest challenges in designing a usable ray tracing system, as most accurate models are very computationally expensive.
The computationally simplest model is the Blinn-Phong where a diffuse light component is computed using the angle between the surface normal and the incoming

[1] See Reif, J. H. "Computability and Complexity of Ray Tracing"
[2] See Turing, A. M. "On Computable Numbers, with an Application to the Entscheidungsproblem"
[3] Lafortune, E. P. and Willems Y. D. "BI-DIRECTIONAL PATH TRACING" p. 2

light vector and a second specular component adds direct reflections of the light. From these the outgoing light flux is obtained as its product with the material light absorption, the inverse square distance to the light source, if the illuminating rays are not assumed to be parallel, as for simulating the sun and the intensity and color of the light source.[4]
Thus $L_r = \max(0, \vec{N}_s \cdot ||\vec{L}_i||) * \frac{1}{|\vec{L}_i|^2} * E_0$
This model is far from ideal for many physically based rendering applications, because it does not necessarily conserve energy and does not correctly handle highly reflective or metallic surfaces.
Another popular method of ascertaining the radiated light flux is the Oren-Nayar reflectance model. In their paper Oren and Nayar derive and experimentally verify their model of diffuse reflection, which also considers the azimuth angle of incidence reflectance making it more accurate when observed orthogonally to the light and the normal. "*It [a previous work] assumes that the radiance is symmetrical with respect to the surface normal. It will be shown in this paper that this assumption is incorrect*"[5]
The simplified model presented by Oren and Nayar in section 4.4 is more physically accurate than the Blinn-Phong model, however the trigonometric functions it uses are complicated to compute as they can only be approximated using quickly converging infinite series or memory intensive lookup tables and most of them can not be computed through the vector dot product, as most of them do not arise from simple angles between vectors.

AI assisted de-noising:
As with any Monte-Carlo integration problem, convergence is not instantaneous and to save computing time the number of samples is usually reduced to a point where per-pixel noise is noticeable, especially in computationally difficult scenes like those involving caustics or highly indirect light. The image resolution also determines computational effort, as the ray tracing occurs per-pixel and thus methods of increasing the resolution without needing to really compute the associated rays have been of interest. Recently AI technologies have reached the point where they can viably solve both problems whilst being temporarily stable.

> "*Conventional denoisers suffer from temporal lag or introduce undesirable temporal discontinuities. [...] Our network has learned to predict spatial filter weights that blur*

[4]See Phong, B. T. "Illumination for Computer Generated Pictures"
[5] Oren, M. and Nayar, S. K. "Generalization of Lambert's Reflectance Model" p.2

> *these regions providing better reconstruction when the history samples are unavailable.*“[6]
>
> “*Neural reconstruction techniques are becoming more common in real-time rendering due to their ability to produce high-quality images from undersampled inputs through an end-to-end learning process.*”[7]

Such methods provide a fast and increasingly reliable way to improve image quality, despite the comparable immaturity of the field of AI in comparison to traditional computer graphics.

Handling polygons for modeling surfaces:

The basis of most 3D modeling software packages used by artists like Blender or Maya is the n-sided polygon. However handling intersections with any kind of polygon is nigh impossible due to the infinite number of them and all polygons with more than three vertices carry extra challenges, as they may not be convex, do not necessarily have a definable area and may intersect with themselves. To simplify the rendering process the more complex polygons are usually converted to a number of triangles.

Proof: Every polygon in R^2 can be reduced to a set of triangles defining the same shape.

First of all only strictly convex polygons need to be considered, as every polytope can be reduced to a strictly convex set using binary space partitioning. The BPS-tree can be ignored, as the spatial relationship of the convex polygons is not important. The vertices can then be numbered by closest neighbors. Unless the first three vertices are all collinear, in which case the middle vertex is redundant, the first three vertices can be separated into one triangle leaving another strictly convex shape. This does not require proof, as any non-convex polygon potentially formed could be reduced to convex polygons, again using binary space partitioning. A recursive application of this algorithm guarantees that the polygon will be perfectly reduced to triangles, as every iteration preserves shape and reduces the main polygon by one vertex, and the algorithm must finish after at most n-3 iteration given any n-sided polygon, making any optimized algorithm for reducing convex shapes to triangles finish in at most O(n-3).

[6] Thomas, M. M. et al. “Temporally Stable Real-Time Joint Neural Denoising and Supersampling” p. 14

[7] Thomas, M. M. et al. “Temporally Stable Real-Time Joint Neural Denoising and Supersampling” p. 2

I conjecture that an analogous reduction can be performed in every dimension given any polytope.

Derivation of intersection algorithms using parametric functions:
To compute the path light rays travel, one must be able to compute where rays intersect the scene geometry. This can be done by moving along the ray in small steps and iteratively checking if the new point is on the other side of the geometry as the last point, or by solving for the exact intersection point.

Ray-Sphere Intersection:
Let R be a ray in $\mathbb{R}^3$
Let $\vec{R_o}$ be the origin of R
Let $\vec{R_d}$ be the direction vector of R
Thus $\forall P \in R(P = \vec{R_o} + \vec{R_d} * t)$, $t \in \mathbb{R}_{\geq 0}$

Let S be a sphere in $\mathbb{R}^3$
Let $\vec{S_o}$ be the origin of S
Let S_r be the radius of S
Thus $\forall P \in S((P - \vec{S_o})^2 = S_r^2)$

Let P_i be the intersection point of R and S if one exists, or the set of intersection points if two exist. (A sphere is strictly convex and a convex surface and a line segment can intersect at at most two points, where both intersection points lie on the boundary of the convex surface)
$\forall P_i(P_i \in R \wedge P_i \in S)$

Combining the first two equations to maintain the equality above we get
$((\vec{R_o} + \vec{R_d} * t) - \vec{S_o})^2 = S_r^2$, $t \in \mathbb{R}_{\geq 0}$ or
$((\vec{R_o} + \vec{R_d} * t) - \vec{S_o})^2 - S_r^2 = 0$, $t \in \mathbb{R}_{\geq 0}$
If we first transform the space such that $||\vec{S_o}|| = 0$ (i.e. centered on the origin)
and then apply the reverse transformation to the result, the equation above simplifies to
$(\vec{R_o} + \vec{R_d} * t)^2 - S_r^2 = 0, t \in \mathbb{R}_{\geq 0}$
Using binomial expansion we get the following quadratic polynomial of t

$\vec{R_o}^2 + 2 * \vec{R_o} * \vec{R_d} * t + \vec{R_d}^2 * t^2 - S_r^2 = 0, t \in \mathbb{R}_{\geq 0}$

Where $\vec{v_1} * \vec{v_2}$ represents the vector dot product and $\vec{v}^2$ the vector dot product with itself.

The roots of a quadratic polynomial can easily be solved using the quadratic formula, and if this equation has no real solutions, or the only solutions are negative,
then R does not intersect S in $\mathbb{R}^3$.

If $\exists P_i$ it can now be calculated thusly

Let $p = \frac{2*(\vec{R_o} - \vec{S_o})*\vec{R_d}}{\vec{R_d}^2}$

Let $q = \frac{(\vec{R_o} - \vec{S_o})^2 - S_r^2}{\vec{R_d}^2}$

$t_{1,2} = -\frac{p}{2} \pm \sqrt{(\frac{p}{2})^2 - q}$

$\vec{P_i} = \vec{R_o} + \vec{R_d} * t$

Ray-Plane intersection:

Let R be a ray in $\mathbb{R}^3$

Let $\vec{R_o}$ be the origin of R

Let $\vec{R_d}$ be the direction vector of R

Thus $\forall P \in R(P = \vec{R_o} + \vec{R_d} * t)$, $t \in \mathbb{R}_{\geq 0}$

Let P be a plane in $\mathbb{R}^3$

Let $\vec{P_o}$ be the origin of P

Let $\vec{P_n}$ be the normal vector of P

Thus $\forall x \in P((x - \vec{P_o}) * \vec{P_n} = 0)$ where "$*$" denotes the vector dot product

Substituting x for the parametric equation for R we get

$(\vec{R_o} + \vec{R_d} * t - \vec{P_o}) * \vec{P_n} = 0, t \in \mathbb{R}_{\geq 0}$

t can now be solved for thusly:

$(\vec{R_o} + \vec{R_d} * t - \vec{P_o}) * \vec{P_n} = 0, t \in \mathbb{R}_{\geq 0}$

$\models \vec{R_o} * \vec{P_n} + (\vec{R_d} * t) * \vec{P_n} - \vec{P_o} * \vec{P_n} = 0, t \in \mathbb{R}_{\geq 0}$

$\models (\vec{R_d} * t) * \vec{P_n} = \vec{P_o} * \vec{P_n} - \vec{R_o} * \vec{P_n}, t \in \mathbb{R}_{\geq 0}$

$\models (\vec{R_d} * t) * \vec{P_n} = (\vec{P_o} - \vec{R_o}) * \vec{P_n}, t \in \mathbb{R}_{\geq 0}$

$\models t * (\vec{R_d} * \vec{P_n}) = (\vec{P_o} - \vec{R_o}) * \vec{P_n}, t \in \mathbb{R}_{\geq 0}$

$\models t = \frac{(\vec{P_o}-\vec{R_o})*\vec{P_n}}{\vec{R_d}*\vec{P_n}}, t \in \mathbb{R}_{\geq 0}$

If t is computed to be negative, then R never intersects P

The intersection point can again be computed as

$\vec{P_i} = \vec{R_o} + \vec{R_d} * t$

Ray-Disk Intersection:

Ray intersections with a planar disc manifold in $\mathbb{R}^3$ are also simple, as given any disc D

$\forall \vec{P_i} \in D(\vec{P_i} \in (P \cap S))$

where P is a plane and S is a sphere whose intersection uniquely defines the disc manifold.

As such the intersection can be found by first computing the Ray-Plane intersection against P as described above and then checking if this intersection point lies within S.

Implementation:

My basic "work-in-progress" implementation of the algorithms discussed in this paper, without AI de-noising can be found at https://github.com/1110011/SoftwareRayTracing

References:

[3] Lafortune, Eric P. and Willems, Yves D.: "BI-DIRECTIONAL PATH TRACING" at the Department of Computer Science, Katholieke Universiteit Leuven, Belgium https://graphics.cs.kuleuven.be/publications/BDPT/BDPT_paper.pdf

[5] Oren, Michael and Nayar, Shree K. "Generalization of Lambert's Reflectance Model" in "Proceeding of the 21st annual conference on Computer graphics and interactive techniques" p. 239-246 doi:10.1145/192161.192213

At the Department of Computer Science, Columbia University

[4] Phong, Bui Tuong "Illumination for Computer generated Pictures" in "Communications of the ACM", Volume 18, Issue 6, June 1975, p. 311-317 at the University of Utah https://doi.org/10.1145/360825.360839

[1] Reif, J.H., Tygar, J.D. and Yoshida, A.: "Computability and Complexity of Ray Tracing" in "Discrete & Computational Geometry" p. 265-287 (1994)

[6,7] Thomas et al.: "Temporally Stable Real-Time Joint Neural Denoising and Supersampling" in "Proceedings of the ACM on Computer graphics and interactive Techniques", Volume 5, Issue 3, July 2022, Article No. 21 p. 1-22

At Intel Corporation, USA https://doi.org/10.1145/3543870

[2] Turing, Alan M.: “On Computable Numbers, with an Application to the Entscheidungsproblem” in “Proceeding of the London Mathematical Society” Volume s2-42, Issue 1, 1938, p. 230-265 (1938) https://doi.org/10.1112/plms/s2-42.1.230

License and legal information:

Hiermit erkläre ich, dass ich die vorliegende Facharbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Quellen als solche kenntlich gemacht habe.

S. Bork