Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Dániel Arató

GLOBAL ILLUMINATION WITH CONTINUOUS RAY PACKETS

ADVISOR

Dr. László Szirmay-Kalos

BUDAPEST, 2016

Table of Contents

HALLGATÓI NYILATKOZAT

Alulírott **Arató Dániel**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2016. 12. 09.

.....
Arató Dániel

Abstract

Accurate and realistic lighting is a key requirement for all 3D graphics applications aspiring for photorealistic rendering quality, whether games, architectural applications or offline rendering engines. In fact a rigorous physically based approach to light transport is what differentiates graphics systems that can produce correct and unbiased images from those that cannot.

However the perfectly faithful simulation of all quanta emitted by a single lightbulb even in the narrow spectrum of visible light — in the neighbourhood of 10^{20} per second or more — is known to be prohibitively expensive. For more than three decades global light transport, especially indirect illumination has thus been considered a hard problem that inspired many creative approximations to cut down on rendering time.

Physically based techniques have nevertheless remained the undisputed best way to attack the problem. The algorithm that best embodies this approach is path tracing, a highly parallel method for finding physically plausible paths connecting camera and light source in the virtual scene.

The chief obstacle to making path tracing practical in online applications is the inevitable presence of random noise in the output. This is usually dealt with by brute force: by sampling each surface point visible from the camera thousands of times and using numerical integration until the image has converged, although more sophisticated noise reduction techniques have been developed. A different way to fight noise is to make use of the spatial coherence of the scene geometry and perspective parallel rays.

This thesis suggests a natural and intuitive addition to path tracing engines aimed at removing noise and/or speeding up the rendering process by exploiting spatial coherence. A basic demonstrative implementation is provided in C++ along with a reference path tracer program that shares some of the same code. The two solutions are compared in terms of code complexity, generality and practical performance in simple test cases.

Összefoglaló

A megvilágítás pontos és valósághű utánzása a fotorealisztikus képminőségre törekvő háromdimenziós grafikai alkalmazások egyik legfontosabb kelléke, akár játékokról, akár építészeti alkalmazásokról vagy offline renderelő motorokról legyen szó. Sőt, a fényterjedés szigorú, fizikailag megalapozott megközelítése választja el a helyes és hibátlan képszintézisre képes grafikai rendszereket azoktól, amelyek nem képesek erre.

Az összes foton hű nyomon követése viszont még egyetlen izzólámpa által a látható tartományban kibocsátott — másodpercenként 10^{20} vagy még több — fénykvantum esetén is közismerten kivitelezhetetlen. A fényterjedés, különösen a közvetett megvilágítás tehát bő három évtizede nehéz problémának számít, és számos szellemes közelítő eljárást ihletett, amik sokat lefaragtak a képszintézis időigényéből.

A fizikailag megalapozott technikák ennek ellenére máig a feladat megoldásának legjobb módjai maradtak. Ezt a megközelítést a legjobban a rekurzív sugárkövetés testesíti meg: nagymértékben párhuzamosítható eljárás, ami fizikailag helyes útvonalakat keres a kamera és a fényforrások között a szimulált térben.

A rekurzív sugárkövetés interaktív alkalmazásokban való használatának fő akadálya a kimenetben elkerülhetetlenül fellépő véletlen zaj. Ezzel általában úgy birkóznak meg, hogy a kamerából látszó összes felületi pontot több ezerszer mintavételezik, és numerikus integrálással közelítik a megoldást, amíg a kép össze nem áll, bár kifinomultabb zajszűrési technikákat is kifejlesztettek már. A zaj kiküszöbölésére meg lehet próbálni felhasználni a színtér geometriájából és az egy pontból eredő sugarakból adódó koherenciát is.

A jelen dolgozat javaslatot tesz rekurzív sugárkövető motorok olyan természetes és intuitív kiegészítésére, amely igyekszik a térbeli koherencia kiaknázásával megszüntetni a zajt és gyorsítani a képszámítási folyamatot. A dolgozathoz tartozik egy egyszerű C++ nyelvű bemutató alkalmazás, valamint egy összehasonlítási alapul szolgáló rekurzív sugárkövető program, ami részben ugyanarra a kódra épül. Összehasonlítjuk a megoldások komplexitását, általánosságát és néhány egyszerű tesztesetben gyakorlati teljesítményüket is.

Introduction

Global illumination — the physically accurate simulation of light transport in a virtual world — is an infamously computationally intensive problem. Ever since the need arose to render realistic 3D virtual scenes and since the publication of the rendering equation[3] in the 1980's people have been looking for efficient ways to cope with the problem.

Path tracing[1] emerged as the most successful attempt and became a pivotal algorithm in global illumination and all of computer graphics. Its simplicity and uncompromising physically based nature give it an almost unrivaled degree of reliability and accuracy while still being relatively easy to implement. In fact path tracing is still the gold standard by which other unbiased[1] rendering algorithms are judged today.

Yet even after significant theoretical progress and a thousandfold increase in the processing power of a typical personal computer since Immel et al. and Kajiya's original papers were released[2][3], some problems have still not been overcome. Noise[2] and speed — two sides of the same coin — remain essentially unsolved issues in the field. Serious effort has gone into developing GPU accelerated near real-time path tracer engines with impressive results[35] as well as highly effective noise reduction strategies (see 1.7). Still the only kind of project utilizing real-time path tracing is little proof-of-concept games[37][38][39].

In the present thesis a method of speeding up path-based rendering by grouping rays together according to spatial coherence is outlined. The algorithm is at an early stage of development but some of the results are promising.

The structure of the thesis is organized as follows:

---

[1]Rendering procedures converge to a certain limit point in the space of possible images. The limit depends on both the input and the algorithm's own properties. If the limit is equal to the physically expected solution for any input, the algorithm is called unbiased; if not, it is biased.

[2]Noise means the visible "graininess" of an output image rendered with path tracing. It is a result of random sampling every time a ray hits a diffuse surface.

1. Chapter One is dedicated to a concise discussion of conventional path tracing both as the standard method for global illumination and as a reference for the new algorithm.
2. In Chapter Two we look at what communities and industries could benefit from quicker rendering algorithms with less noise.
3. Chapter Three introduces the concept of "zones" and what they are used for in the new algorithm. A detailed discussion of both phases (or passes) of the algorithm is given.
4. In Chapter Four the results, the performance and other characteristics of the new method are compared to a basic reference implementation of traditional path tracing.
5. In Chapter Five the results and findings learned from the project are assessed and reviewed.
6. Chapter Six outlines possible future improvements to the algorithm.
7. An acknowledgement paragraph follows and then the list of references.
8. An appendix of example output images is found at the end.

The complete source code of both demonstrative applications is made available on GitHub. The path tracing reference program is called Retra[44] (for "reference tracer"), the original rendering program is called Silence[45].

1 A Brief Overview of Conventional Path Tracing

A quick review of the classic path tracing[3] algorithm follows after establishing a basic theoretical framework. The same framework shall be used in Chapter Two. No prior knowledge of the algorithm is assumed. Readers well versed in the theory of physically based rendering may feel free to skip this chapter.

1.1 The Lighting Model

This section serves as an introduction to the basic properties of visible light as simulated in path tracing. We are going to take some liberties to arrive at a compact, intuitive and almost physically plausible model of light rooted in geometrical optics.

In path tracing light is treated as a very large number of discrete, infinitesimally small particles traveling through space and occasionally interacting with material surfaces or being registered by a virtual camera. It is a well known fact that physical light is *not* literally a stream of particles. In some cases it does behave as such, in others not at all. In fact physicists have yet to come up with an appropriate metaphor that fully and accurately captures the nature of electromagnetic radiation. Remarkably though the particle model is good enough to fool the human eye, as proven by path tracing.

Each particle has a wavelength associated with it and a finite amount of energy which is a function of that wavelength. Exactly what values nature defines does not matter to us for reasons we will see shortly.

Light particles are created by light sources. The physical reasons of light emission do not concern us. We can just imagine that a limitless stream of particles is emanating from all surface points of all light sources all the time.

All light sources in the model are either point lights or diffuse area light sources. Point lights emit light uniformly in all directions. All surface points of area light sources emit light uniformly in a hemisphere whose plane is tangential to the surface at the surface point.

[3]There is some confusion over the name of the algorithm: some sources call it "(recursive) ray tracing" or "Whitted ray tracing", others "path tracing", yet others just "the Monte Carlo algorithm" even though that term is somewhat ambiguous. We shall use the name "path tracing" throughout the thesis.

In vacuum and in homogeneous media light particles always travel in a straight line. Again, that's not true but true enough for any errors to be imperceptible under the conditions humans are used to. The trajectory of a single undisturbed particle over time is a geometrical ray.

We shall use the word "ray" to mean a set of n particles traveling along such a trajectory where n is a suitably large number: if we need to split a ray into k rays of equal energy we shall always assume that $\frac{n \bmod k}{n}$ is a negligible number. In effect we treat the total energy of a ray as a continuous rather than discrete quantity, so the amount of energy in a single notional particle is irrelevant. Such rays are the central tool (and corresponding data structure) the path tracing algorithm is built upon.

How fast do light particles travel? A reasonable first approximation is that they travel from the light source to their final destination instantly, at infinite speed. Light is so much faster than ordinary positive mass objects around us that this exaggeration compromises our model in absolutely no way.[4]

We imagine that all wavelengths of light travel together in a single ray no matter what happens to the ray. This is another physically inaccurate simplification. Light particles are independent of each other. When refracted on a barrier between different material media each wavelength of light reacts differently and they are scattered away from each other. Modern rendering engines use spectral path tracing[5][23] to account for this phenomenon.

Finally, in order to represent a ray's intensity[5] across the visible spectrum we need to use a finite number of representative wavelengths. We assign each ray a tuple (r,g,b) where $r,g,b \in \mathbb{R}_{\geq 0}$ to represent its color, i.e. we consider the ray to carry r, g and b units of energy in the form of red, green and blue[6] light particles respectively. We need not be concerned with what the words "red", "green" and "blue" mean because

[4]The curious reader might be interested in exotic raytracing projects that deliberately *reduce* the speed of light[36]. They show off fantastic, dreamlike relativistic effects.

[5]Intensity basically means the amount of power carried by the ray. The definition of intensity requires division by the surface area struck and we have said that rays have no finite thickness. "Energy" should be more proper but the term "intensity" is nevertheless more commonly used.

[6]Red, green and blue are traditionally chosen for biological reasons. There is nothing special about these wavelengths in a physical sense.

we are going to leave it up to an image viewing application to interpret them. The three dimensions of the tuple are taken to be perfectly independent.

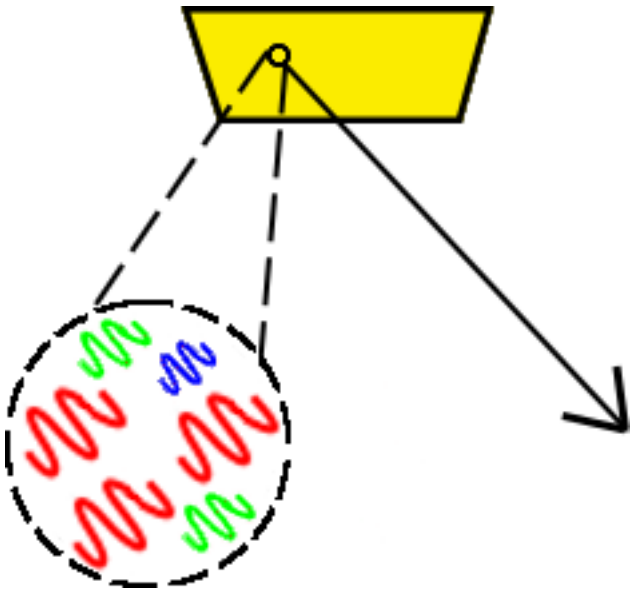


Figure 1.1: A single ray of notional light particles

We have established that when undisturbed a ray will travel along a straight line. What happens when there is something in the way?

1.2 Light-Material Interaction

When a particle of light collides with a physical surface, a number of things can happen:

- **Absorption.** The energy of the particle is absorbed by the physical object.
- **Specular reflection.** The particle bounces off at more or less the same angle it arrived at, such as off a mirror.
- **Diffuse reflection.** The particle bounces off at a more or less random angle.
- **Refraction.** The particle penetrates the object and continues its path inside it at an angle determined by refractive indices.

It is important to note that light-material interaction is a stochastic process. What a concrete particle will do after hitting a concrete surface any given time cannot be known in advance. The best we can do is to assign probabilities to each outcome. The probabilities will depend on the characteristics of the exact material in question.

In the previous section we made the assumption that a ray contains so many particles that it does not matter there is a finite number of them. Relying on the exact same assumption we can say that a surface actually absorbs, reflects and transmits continuous fractions of a ray's energy. Then, conveniently, the factor by which each kind of interaction reduces the color intensity of a ray will be equal to the probability of that interaction (ignoring the effect of the color of the surface).

The portion of light that is absorbed by the surface has no further effect other than heating up the receiving object by some immeasurable amount. The other portions continue flying through the virtual world[7] and may or may not change the picture the camera is generating in the end.

In the model we only allow smooth surfaces (i.e. differentiable an arbitrary number of times). This guarantees that a particle hitting a surface will always behave as though hitting a plane perpendicular to the surface normal at the contact point.

Note that real physical materials typically do not behave as ideal specular reflectors or ideal (Lambertian[24]) diffuse reflectors. Consider the flat surface of a polished wooden table and the camera and a light source at mirrored angles to the surface. The camera should see a blurred image of the light source that looks slightly larger than the original. This is because instead of reflecting ideally such a surface will scatter the light slightly resulting in a fuzzy image.

Ideally white surfaces will reflect all incoming radiation, ideally black surfaces will absorb it all. Most real materials will both absorb and reflect some amount of the incoming light, and many exhibit refraction as well. Also, subsurface scattering[6] is a complicated combination of refraction and reflection that we are not going to go into here.

If a given material has non-grey color (white and black being special cases of grey) then a different amount is absorbed of an incoming ray's total energy at each primary color wavelength. We interpret this effect as changing the color of the ray in our model: we define a componentwise operation to "incorporate" the effect of the surface into the color of the ray.

$$\text{paint} : RGB \to RGB \to RGB$$
$$\text{paint}(\text{ color, color' }) = (\text{color.r} * \text{color'.r, color.g} * \text{color'.g, color.b} * \text{color'.b})$$

[7]The terms "scene" and "world" are used interchangeably to mean the full formal description of the simulated environment the renderer displays.

If for example a ray of color (r, g, b) encounters a surface that reflects all of red, half of green and none of blue particles the reflected ray shall have color $(r, 0.5g, 0)$.

Exactly what percentage of the energy per unit surface area is reflected in a certain direction when light meets a certain kind of material is governed by the bidirectional reflectance distribution function (BRDF) of that material[7]. A BRDF is a function of the *contact point*, the *incoming direction*, the *outgoing direction*, the *wavelength* of light in question and, rarely, the amount of *irradiance* (certain physical substances will experience "saturation" when strongly lit[26]). The first and last terms are almost universally ignored. The first one because most simulated objects are, or can be broken up into, homogeneous entities. The last one because it is considered to have an inconsequential effect on radiance[8].

An object's BRDF is assumed not to change over time or when in motion. These are very reasonable, if not one hundred percent correct assumptions for the purposes of practical rendering.

The same kind of function for refractive transmission rather than reflection is called a bidirectional transmittance distribution function. A BTDF has the same type as a BRDF.

A BRDF and a BTDF extended with each other is sometimes called a bidirectional scattering distribution function. Although there is a wide variety of sensible and even physically plausible BSDF's we shall restrict the model to a subset of them for the sake of simplicity. First, we shall ignore the wavelength variable and say that all wavelengths scatter the same as suggested in Section 1.1. Second, all simplified BSDF's shall be a linear combination of four ideal functions representing Lambertian reflection, specular reflection, metallic reflection (a special case of the former), and refraction.

$$\mathrm{BSDF}_{simple}(a, b, c, d) = a \cdot \mathrm{BRDF}_{diffuse} + b \cdot \mathrm{BRDF}_{specular} + c \cdot \mathrm{BRDF}_{metallic} + d \cdot \mathrm{BTDF}_{refractive}$$

where $1 = a + b + c + d$. (Absorption shall be represented in the color of the material.)

Most light bouncing around the virtual scene never makes it to the camera so its effect on the image seen by the camera is zero. Time and effort spent on tracing these

[8] *Irradiance* is the amount of *incoming* power being recieved by the surface per unit area. *Radiance* is the amount of *outgoing* power leaving the surface per unit solid angle per unit area.

rays of light is wasted. To counter this problem, path tracing follows rays starting from the camera instead of the other way around. (The opposite is usually called "light path tracing". The terms "forward path tracing" and "backward path tracing" are not recommended because of their ambiguity. These two options do not cover the whole range of possible approaches though.)

Taking this approach without compromising correctness is made possible by a fundamental symmetry in all physically plausible BSDF's and in the end by an elegant property of light as a physical phenomenon. A ray of light that is unabsorbed when it meets a surface will take the exact same path as a ray of light traveling in the reverse direction provided they suffer the same kind of material interaction. Or, in terms of BSDF's:

$$\mathrm{BSDF}(\omega_i, \omega_o) = \mathrm{BSDF}(\omega_o, \omega_i)$$

for any ω_i and ω_o[25]. So if a path has been established from the camera to a light source, it is a physically valid assumption to say that light is flowing from the light source to the camera along the same path. Now if the operation used to incorporate the color of a surface into the color of the ray is commutative, we should get the same end result as if we traced light rays from the light source. Indeed it is very easy to see that

$$\mathrm{paint}(\mathrm{paint}(c,(r,g,b)),(r',g',b')) = \mathrm{paint}(\mathrm{paint}(c,(r',g',b')),(r,g,b))$$

for any c, r, g, b, r', g' and b'.

It is worth noting that because of their stochastic nature paths are sometimes called "random walks".

1.3 The Camera Model

There is no reason to complicate the camera model especially since the simulation of the optical properties of actual cameras is not part of our discussion. We shall rely on the simplest possible option: a pinhole camera model.

We imagine that there is an ideal invisible camera in the world space defined by its *viewpoint* (or "aperture" in optics parlance), a rectangular *screen* in front of the viewpoint and the *resolution* of the screen. It's almost as if we are seeing the virtual world through a window. (Technically, a pinhole camera produces the image on a screen *behind* the viewpoint. The only difference is the image being point-reflected and natural near-plane clipping, that is, the invisibility of virtual objects nearer than the screen.)

The screen is divided up according to the resolution into rectangular areas called pixels. The so-called viewing frustum emerges naturally: it is the far side of the infinite pyramid whose apex is the viewpoint and its edges are the four rays connecting the viewpoint and the corners of the screen, cut by the screen plane.

A camera of this kind sees every object in its viewing frustum sharply or "in focus" (although the latter term is misleading as the model involves no lenses). When a light ray happens to pierce the imaginary screen and exactly strike the viewpoint its color tuple is registered for the screen pixel it just traveled through. As discussed in the previous section, we reverse this process to trace eye paths to all light sources. The intensities of all rays through the same pixel are added up to yield the full intensity experienced by the pixel[27]. Although rays of arbitrarily large energy are allowed, a screen pixel becomes saturated at $(1,1,1)$, which appears as the color white. In directions unblocked by any object, the camera sees the color of the sky (part of the world description).

1.4 The Core Algorithm

With that we are ready to review how classic path tracing works on a technical level.

1.4.1 A Bird's Eye View

The following pseudocode procedure describes the basic path tracing algorithm on a very abstract level.

$$\text{pathtrace} : World \to Camera \to \mathbb{N}_+ \to \mathbb{N}_+ \to RGB^p$$

```
pathtrace( world, camera, s, d )
    for each pixel in camera.screen:
        for [0,s):
            shoot a ray from camera.viewpoint through the pixel
            while 0 < ray.intensity and ray.depth++ <= d:
                find nearest intersected surface
                if no surface hit:
                    ray.paint( color of sky )
                    break
                if surface is a light:
                    ray.paint( color of light )
                    break
                ray.paint( color of intersected surface )
                if intersected surface...
                    reflects diffusely:
                        dampen ray to amount of diffuse reflectance
                        continue in random direction
                    reflects specularly:
                        dampen ray to amount of specular reflectance
                        continue in mirror direction
                    refracts:
                        dampen ray to amount of transmission
                        continue in refracted direction
            ray.color = (0, 0, 0)
        pixel.color = average of resulting ray colors
    return pixels
```

(The variables *world*, *camera*, *s* and *d* are chosen by the user. *s* denotes the number of samples per pixel, *d* is the maximum recursion or iteration depth allowed.)

This high-level overview already reveals a few interesting properties. The operations in the innermost loop need to be performed *O(psd)* times where *p* is the number of pixels in the chosen screen resolution, *s* is the number of samples per pixel and *d* is the maximum path depth allowed. Intersection detection is the only task in the loop that requires any kind of serious work so we expect it to be a performance bottleneck in the algorithm. Practical experiences bear this out[8].

The algorithm belongs to the category of so-called embarrassingly parallel problems[31] because all pixels and all paths through them can be computed perfectly independently of all others. In other words both outermost loops can be parallelized at will.

The characteristic noise comes exclusively from the only nondeterministic operation in the function: "continue in random direction". If we could somehow *know* the average color of rays from that point on (essentially the integral on the right of the rendering equation), all the noise would disappear instantly.

Also note that absorption is accounted for implicitly by incorporating the color of each surface hit. A surface of color (r,g,b) will have the same basic color as $(2r,2g,2b)$ but reflect half as much of an incoming ray's intensity.

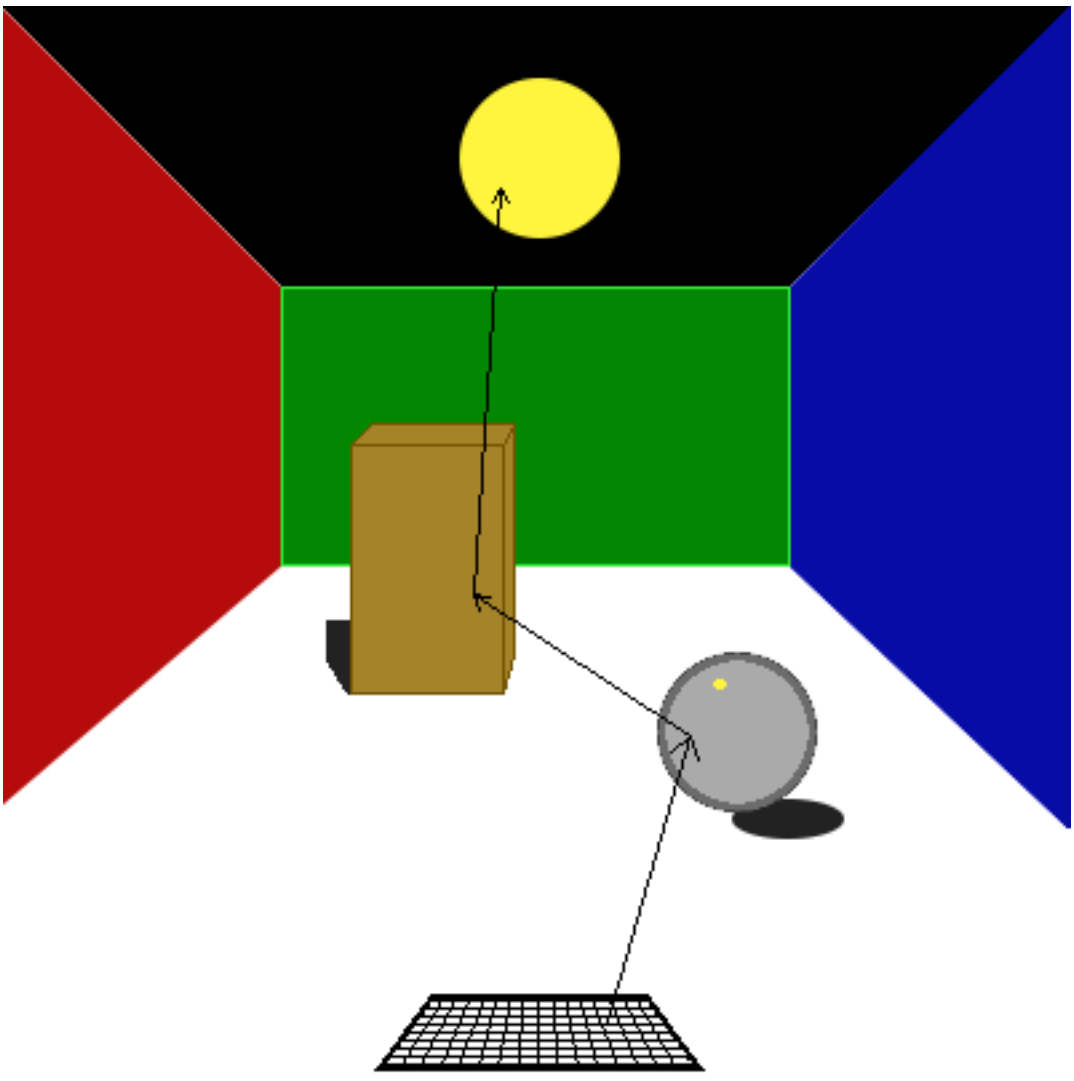


Figure 1.2: The process of path tracing. The eye ray hits the specular ball first, is reflected over to the cardboard box, then after random diffuse reflection the ray hits the light by chance

1.4.2 Technical Details

Intersection detection is a nontrivial task. Fortunately, by the late 1990's fast and easy to implement algorithms have been found for all common shapes[9][33].

Each instance of light-material interaction is decided by taking a random number in a range divided up into segments proportionate to the probability (i.e. the weight) of each interaction in the material's simplified BSDF.

Lambertian reflection is quite easy to implement. Ordinary Gaussian distributions along all three dimensions can be used to obtain a random direction, and all that remains is to mirror it if it falls below the surface. Another way is to pick uniformly random points in the half unit cube around the hemisphere and reject them until one falls inside or on the hemisphere[34]. Note that the cosine law[28] stating that radiance depends on the angle the light is arriving at must also be observed in the intensity of the ray.

Ideal reflection is trivial. Ideal reflectance and metallic (Fresnel) reflectance are two different subtypes of the same phenomenon, they must be treated separately in an actual implementation even though no distinction is made in the high-level pseudocode. Refraction direction is governed by Snell's law[29].

Instead of storing the final colors of all individual rays sent through a pixel in practice the colors are added up in a single tuple and then the sum value is divided by s at the end; a simple and memory-efficient way of obtaining the average.

The trivial Portable PixMap format[32] is a convenient one to output results in and is understood by most image viewer programs.

1.5 Variants of the Core Algorithm

Some obvious deficiencies of basic path tracing were soon noted and there was enough low-hanging fruit for the original algorithm to start evolving rapidly.

The perceptive reader will have noticed that most of the rays are actually wasted in the above routine because they never reach a light source. The easiest way to rectify that is by **direct light sampling**, now universally used and considered an integral part of path tracing. For each light source a number of *shadow rays* are sent from the contact point with the diffuse surface in the direction of random surface points on the light source. The percentage of the light source's total irradiance actually arriving at the

contact point is taken to be the percentage of shadow rays that are not interrupted by any object in the scene. This also produces correct soft shadows.

One can cheaply improve the sampling strategy even further by picking outward directions intelligently every time a diffuse surface is hit. This is called **importance sampling**, and the most common and natural way to do it is to use the BRDF as a random distribution function: the probability of a certain direction being picked is proportionate to the BRDF value of that direction. The more irradiance is expected from a given direction the more likely the ray is to travel in that direction. Integration strategies with such controlled sampling are called quasi-Monte Carlo methods because their samples are not truly and completely random. It should be mentioned though that any of these methods are susceptible to pathological cases unless combined in an advanced approach called multiple importance sampling[10].

A nondeterministic strategy to terminate weak rays early, called **Russian roulette**, is also very common. After each bounce if a ray's total intensity is below a predetermined threshold, it is terminated with probability $1-\frac{1}{m}$ where m is chosen beforehand so that no excessive computing effort is spent on deep paths that are relatively unimportant. The expected amount of intensity lost in this operation is in effect redistributed among the rays that make it through by multiplying their intensities by m[11].

Another apparent problem was the fact that sending rays through the center of each screen pixel is prone to aliasing artifacts. Some researchers came up with ideas to use multiple different rays per each pixel, which spawned **distributed ray tracing**[4], while most others tried replacing the infinitesimally thin rays with shapes with positive volume. This gave rise to **beam tracing**, **cone tracing**, **pencil tracing** and more.

Yet others realized that carefully perturbing the intersection points slightly once a valid path has been found can be used to construct more valid paths and **Metropolis light transport** was born[12].

Section 1.2 hinted at eye ray tracing and light ray tracing not being the only available options. One of the most fashionable unbiased rendering algorithms today is **bidirectional path tracing**[13]. It starts by constructing paths both from the light sources and the camera a few bounces deep and then looks for ways to join light paths

and eye ray paths together. Bidirectional path tracing performance is usually superior to vanilla path tracing. Light rays help with indirect illumination and caustics.

1.6 Spatial Acceleration Structures

Finding the nearest intersection of a ray with any surface in the virtual scene is one of the most common tasks in any ray tracing engine. Naive path tracing was soon improved dramatically by including so-called acceleration structures that help reduce the number of intersection tests to be performed by analyzing the spatial relations of the scene in a preprocessing step. By revealing what object are present in a given sector of the world not only do they speed up finding intersecions, they also help solve the visibility problem[30].

This section is kept short because neither application presented in this thesis actually uses any acceleration structures (unless you count the zone trees Silence uses but that's a very different kind of thing). Acceleration structures are only relevant to our discussion inasmuch as they can be used to speed up any related rendering algorithm (they are *somewhat* orthogonal to one's choice of core algorithm). For now it's enough that we are aware of them.

The most popular acceleration structures are the following:

- **Binary space partitioning** (BSP) **trees** recursively split the world space in two along arbitrary (normally axis-aligned) planes. Each node in the tree has zero or two children.
- **k-d** (most commonly 3-d) **trees** are BSP trees where each subsequent level has divisions along the next world axis. Level zero is the root node, then level one splits the space along $x=x_0$ (for some suitable x_0), then level two splits each half separately along some $y=y_0$ and $y=y_1$ into four leaves total, then level three splits each leaf again by the z coordinate and so on.
- **Octrees** recursively divide the world space into eight equal parts along the planes $x=(x_0+x_1)/2$, $y=(y_0+y_1)/2$ and $z=(z_0+z_1)/2$ simultaneously where $[x_0,x_1]\times[y_0,y_1]\times[z_0,z_1]$ is the set of points in the current node. Each node has either zero or eight children.

- **Bounding volume hierarchies** take a bottom-up approach, first they wrap each individual object in a bounding box (or other bounding volume), then nearby objects are grouped together into bigger bounding boxes and so on until all of the scene geometry is enclosed in a single bounding box.

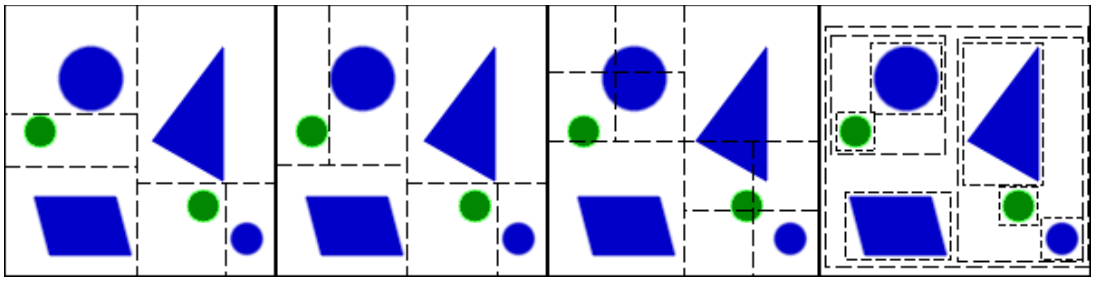


Figure 1.3: Acceleration structures in 2D. BSP tree, 2-d tree, quadtree, BVH

All of the above are essentially space partitioning schemes that serve to organize parts of the world geometry into easy-to-manage, fast data structures. As mentioned above the demo applications use no such data structures in order to keep them small and simple.

1.7 Strategies to Reduce Noise

As mentioned before efficient noise reduction is one of the main subjects of current global illumination research. Smart noise filtering algorithms in path traced images have almost grown into a field of their own in recent years. Successful filtering techniques deserving of mention include the following:

- greedy error minimization in two steps[14],
- random parameter filtering[15],
- and neural networks[16] trained on pairs of noisy and clean renders of the same scene.

1.8 Other Algorithms

Path tracing is of course not without competition on the realistic image synthesis market. Other successful approaches to global illumination exist. Techniques that deserve mention include:

- photon mapping[17], a two-pass algorithm that shoots light rays first and then samples their density around points visible from the camera;

- radiosity[18], an old and proven approach that uses the finite element method but works for diffuse materials only;
- and voxel cone tracing[19], which divides the virtual space up into small cubes and uses a variant of ray tracing to sample it.

These methods are very far removed from our immediate topic though and we shall discuss them no further.

2 Applicability

Path tracing and related methods have traditionally been important in a number of fields. Novel algorithms are often incorporated into the tools used in these fields making life easier for both professionals and hobbyists.

This chapter explores some possible use cases for the zone based algorithm if and when it reaches maturity.

2.1 Acoustic Modeling

Beam tracing solutions have been used to model the behaviour of sound waves in closed spaces for years. A variant of "zone tracing" could provide an attractive, low-cost alternative.

2.2 Architecture

The new method could prove extremely useful for architects, lighting and interior designers. With near real-time noiseless path tracing professionals would be able to demonstrate architectural environment models to costumers in a more interactive and lifelike experience.

Low polygon count indirectly-lit interiors like the one below (render by H. W. Jensen, model by Ch. Valtin) are good examples of where the new algorithm could shine.

2.3 Art

As with most image synthesis algorithms it only takes a few simple changes to the algorithm to get exciting freestyle artistic tools. By virtue of its simplicity the model lends itself well to creative experimentation to achieve surreal rather than realistic effects. A few ideas:

- Scrap occlusion detection to create an eerie, dreamlike effect.
- Twist or bend zones in space after building the tree for an Alice in Wonderland feel.
- Use distributions to shift the color of zones instead of determining intensity during rasterization.

2.4 Cinema

The movie industry is another area where where realistic 3D rendering is a must. At the same time iteration time is also crucial and artists usually benefit from faster algorithms. Algorithms like zone tracing can help reduce iteration time from amateur animation up to high-budget movies.

2.5 The Demoscene

The demoscene is a worldwide community of computer graphics enthusiasts dedicated to making "demos" that push the boundaries of CG algorithms. The challenge is to create an executable of tightly constrained size that generates and plays high quality video and audio real-time. A heavily space-optimized variant of zone tracing could serve as a valuable tool in creating impressive demos.

2.6 Games

Present day hardware technology is still at least an order of magnitude of speed away from making traditional path tracing practical in video games[43]. The new algorithm (or a convenient variant thereof) could help game delevopers achieve real-time realistic global illumination on strong hardware.

2.7 Mobile Platforms

3D graphics engines for tablet computers and smartphones already exist. In the coming years as mobile hardware improves realistic rendering on mobile devices may become an exciting new research area. Zones may be a worthwhile addition to rendering systems running on such limited hardware.

3 Introducing the New Algorithm

A significant part of Chapter One was devoted to discussing noise and different ways to cope with it. Rather than developing complicated methods to isolate and reduce noise, we can also try and avoid incurring noise in the first place. The algorithm in this thesis attempts to take the latter approach.

Describing the low-level implementation details of the rendering program would make for a dull read. Such technicalities are relegated to Section 3.5 near the end of this chapter. Instead the author shall focus on providing a high-level overview of the principles of the algorithm and the intuition behind them. By the time we are done, the reader should be able to start implementing her own application based on the same principles. In fact the concepts are sufficiently simple and natural for even non-technical readers to follow along.

The title of the thesis is something of a misnomer in that zones are only loosely related to ray packets but the method did not have a set name at the time nor did the author want to use a confusing nonsense title. Zones are a variant of ray packets in the sense that they both make use of ray coherence, all (hypothetical) rays originate at a common point and they both attempt to reduce the number of intersection tests needed for a high-quality render. See Section 3.2 for details.

3.1 A Banal Observation

The motivation for a new data structure stems from the realization that in most input scenes with simple geometry direct irradiance is at least partially "trivial" or "obvious".

When illuminated directly by the (nearly) white light, the red wall on the left is going to appear in the shades of red according to some well-behaved, smooth function $L_i(\vec{x})$ (the shadow cast by the cardboard box notwithstanding). Any person intuitively knows this and not because we are deeply attuned to the nature of light; the solution really is trivial in some sense so there must be a straightforward way to compute the illumination without expensive random sampling.

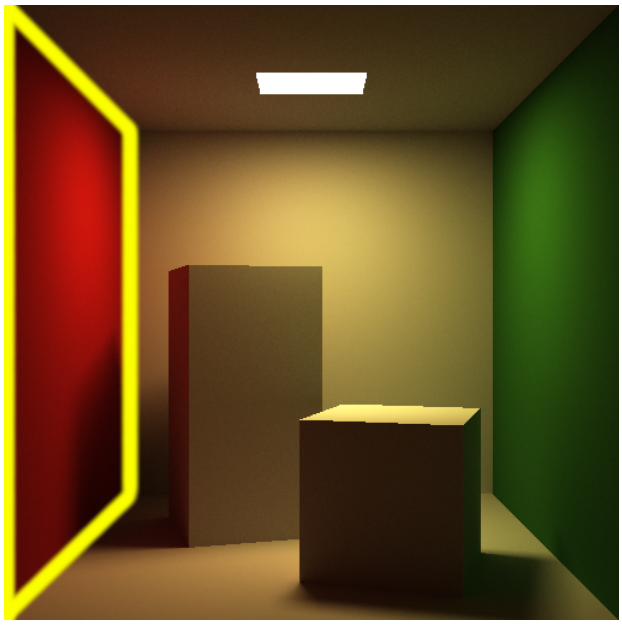


Figure 2.1: A typical scenario

The reason we don't see stochastic noise in actual photographs (or indeed in physical reality) is that most objects are delimited by smooth surfaces and made up of homogeneous or smoothly varying materials (see Section 1.2 about ignoring the BSDF's dependence on the contact point). This means that radiant intensity, and therefore the apparent color, transitions seamlessly across the surface. In other words if we take

$$L_\Sigma(\vec{x}) = \int_\Omega \mathrm{BRDF}(\vec{x}, \vec{\omega}_o, \vec{\omega}_i) L_i(\vec{x}, \vec{\omega}_i)(\mathrm{normal}(\vec{x}) \cdot \vec{\omega}_i)$$

then

$$\lim_{\vec{x} \to \vec{x}_0} L_\Sigma(\vec{x}) = L_\Sigma(\vec{x_0})$$

for any x_0. (There are some exceptions, materials made up of many small parts with different characteristics such as sand viewed at a distance on the order of meters.)

We see smoothly varying hues of red on the left wall for the same reason. All terms of the integral change smoothly, so L_Σ must also change smoothly. We shall try and find an abstraction to capture this intuition in a straightforward and elegant way.

3.2 Zones

Zones are the algorithm's core data structure. They are meant to capture the most crucial characteristics of a full continuous bundle of light emanating from an emitter or reflector in all directions within a hemisphere or a pyramid.

A zone implementation must convey the following information:

- the basic **color** of the light,
- a function to tell how the **intensity** is **distributed** inside it,
- the one **surface** it is radiated from,
- a finite number of **rays** it is **delimited** by (if any) and
- all **shadows** inside it, where a shadow is
    - the surface blocking the zone,
    - a finite number of rays the umbra is delimited by and
    - a finite number of rays the penumbra is delimited by.

Conceptually a zone represents an infinite number of distinct light rays traveling from a given surface to other nearby surfaces or to a camera. The purposes of zones are twofold:

- First, to determine in one sweep the direct illumination of linear surfaces, e.g. the shading of the flat wall lit by a nearby light discussed in the previous section;
- and second, to establish the set of paths that eye rays can potentially take, and crucially, those they cannot. E.g. if two back-culled triangles are facing away from each other, any path that would travel directly between those two triangles must be invalid.

Both of the above can be used to make the actual process of rendering the scene easier.

In the first phase of the algorithm each light source begets a separate "zone tree". All light rays coming directly from a light source together make up the root node of the tree. Then for every other surface in the scene whose unculled side[9] is (at least partially) directly visible to the light source a child zone representing the reflected or refracted rays is created. The process is repeated for the current set of leaves in the tree as many times as needed.

In the Cornell box scene used in figure 2.1 the root zone would represent the continuous set of all light rays leaving the light on the ceiling in all directions of the

[9]The unculled face of a surface is often referred to as "the front" in systems such as OpenGL.

hemisphere. The next level of the tree would consist of similar zones coming from each wall, the floor and the faces of the cardboard boxes that are directly lit and so on.

In order to keep track of the color of the light rays, in each zone a single representative ray called a "pivot" is chosen. The "color of the zone" is actually more precisely the color of the pivot. Just as in regular raytracing, the color of a surface the zone as a whole interacts with is incorporated into the color of the pivot. In practice it is useful to make the pivot the ray – or one of the rays – with the most intensity inside the zone. The pivot doubles as the baseline to which the intensities of other rays inside the zone are compared. The function that specifies the intensity distribution in the zone takes the pivot as an implicit argument. A common and natural example of such a distribution function would be the following:

$$Intensity\left(\vec{x}\right)=\frac{1}{\left|\vec{x}-\text{pivot}.\text{start}\right|^2}\cdot\text{pivot}.\text{intensity}$$

(In this common case the actual direction of the pivot ray is irrelevant.)

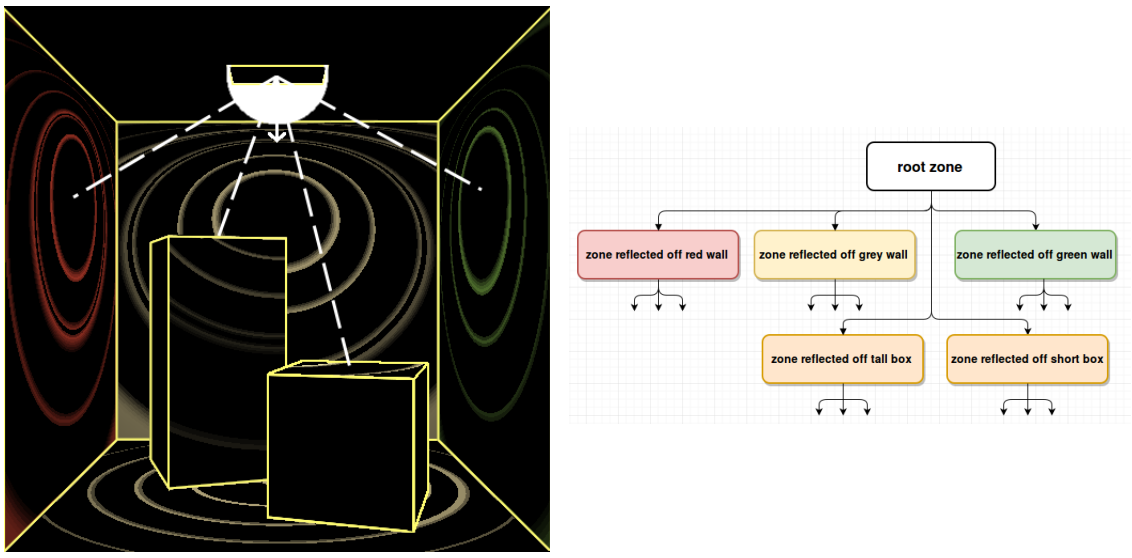


Figure 2.2: The concept of a zone tree. The dashed lines represent some suitable pivot rays

Once the desired depth has been reached for every light source we end up with a "zone forest". This structure encompasses all possible paths a light ray can take in the virtual scene up to *d* bounces if the forest is *d* deep. No physically plausible path shorter or equal in length can run outside the branches of the forest or across branches, although the forest may contain some implausible paths as well. Most of the implausible paths are filtered out by keeping track of the shadows: every zone notes all surface primitives that occlude it. If we want to see if a specific point is illuminated by a given zone we

only need to construct a path back up the branch of the tree it sits in and see if any of the intersection points are in the dark in higher-up nodes (more on this in 3.4).

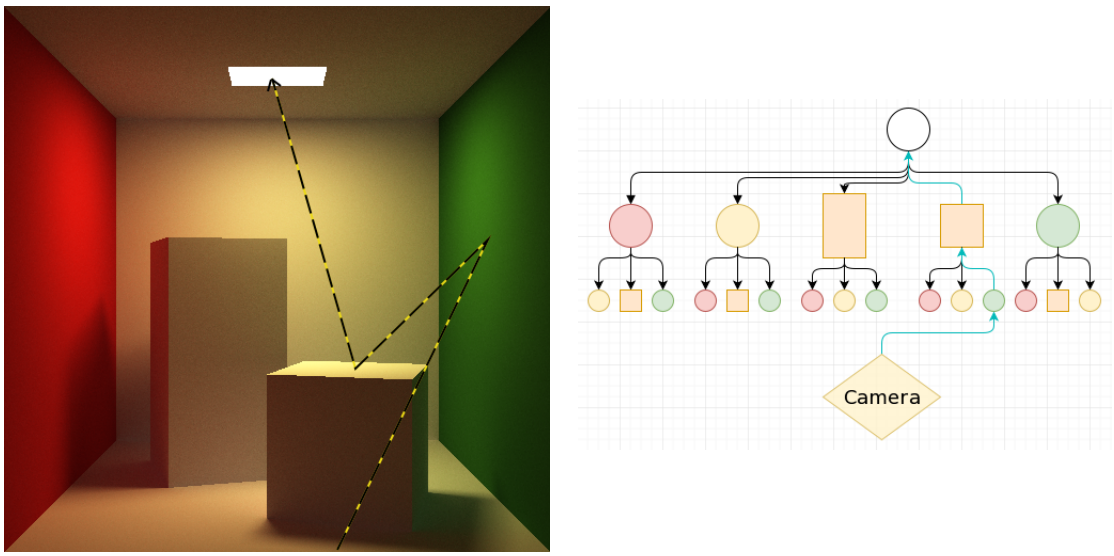


Figure 2.3: All plausible paths run inside one of the branches of the zone tree

As seen in 1.2 light rays can undergo four different kinds of interactions when encountering a material object, so one surface can spawn 0 to 4 separate child zones according to its simplified BSDF. This means that a single node in the zone forest may have up to $4n$ children where n is the number of surface primitives in the scene. Since the size of each zone tree explodes exponentially in n as it deepens it is practical to specify a tight limit on tree depth. In most scenes the changes from depth d to $d+1$ typically stop being noticeable beyond 3 or 4 levels. In outdoor scenes zone tree branches will sometimes terminate by themselves because they fail to hit anything. A minimum pivot intensity for all zones can also be specified to further limit the (average) depth of the tree.

Note that a child zone being reflected or refracted in a certain way is a mere label: for now we don't worry about the geometrical implications of these concepts for the rays inside the zone. That is left to the second phase of the algorithm to deal with (see 3.4).

The rendering routine should walk the entire forest, find the zones that hit the camera viewpoint through at least one pixel of the virtual screen, determine their contributions to the image one by one and then add them up to yield the final output. Treating not only every light source but each contributor zone separately and summing their effects made on the pixels is correct because of the linearity of radiant intensity,

see 1.3. For example the root zone that belongs to the rectangular light (or two triangular lights) in the scene used above would contribute a single white trapezoid on the ceiling. Its children would contribute the direct illumination layer of the walls and objects, making them visible to the camera. Their children in turn would contribute the first bounce of indirect illumination and so on. Generating an output image as seen by the camera using zones alone would be awkward however. In the second phase of the algorithm a sort of guided path tracing will be used to render the final image (see 3.4.2).

Incidentally the tree data structure also makes it easy to render and view only certain parts of all the light illuminating the scene, such as turning lights on and off or looking at specific levels of the tree separately.

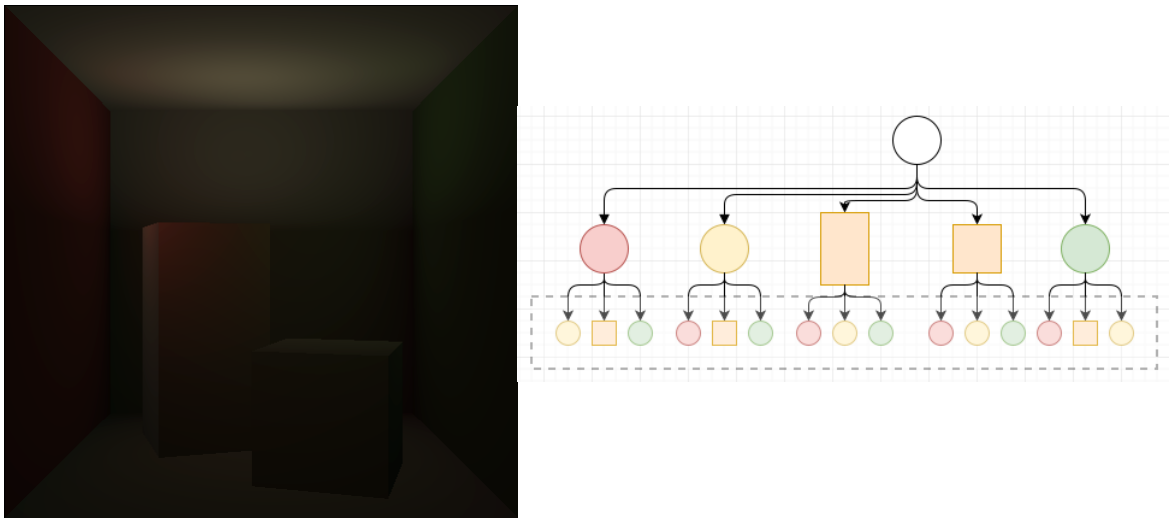


Figure 2.4: Displaying level 2 of a zone tree alone

Linear reflector of refractor surfaces – such as planes and polygons – preserve the relative homogeneity of rays in a common zone. Nonlinear shapes complicate matters considerably but we shall try to incorporate spheres as well because they are ubiquitous in global illumination rendering tests.

3.3 Relationship to Alternatives

Zones are not entirely unlike traditional 3D rendering data structures such as beams or ray packets[21] in that they group rays together according to their coherence. The main difference is that zones represent sets of *light* rays and the zone forest represents *light* paths while beams and ray packets are normally used to group *eye* rays

together. Zones are also less rigorous and more permissive towards invalid paths (see 2.4).

The model also bears a resemblance to illumination networks[20] although it tries to reduce the number of ray-surface intersection tests even further.

The idea of constructing potential light paths from the light sources and then producing an image with eye ray tracing also recalls bidirectional path tracing[13].

Quite frankly in comparison with the aforementioned approaches the idea of zones may be labeled utterly trivial. However there are significant performance gains waiting to be reaped from this simple and intuitive model.

3.4 Outline of the Algorithm

The algorithm operates in two phases, both of which are going to be discussed in more detail in the following couple of sections.

- **Phase One**: building the zone forest.
- **Phase Two**: rasterizing zones to the screen by ray tracing.

In Phase One the algorithm extracts the zone forest, a container of all potential future eye ray paths. The rendering algorithm in Phase Two is informed by the intelligence acquired about the scene in Phase One.

An overview of the algorithm on the same abstraction level as 1.4.1 is presented in pseudocode on the next page.

3.4.1 Phase One

Phase One is designed to exploit spatial coherence in scenes with simple geometry and mostly diffuse surfaces. Discussed at length in 3.2, a zone tree is set up for each light source by creating a root zone and adding another zone under it for each surface that is reached by at least one ray of the original zone. The same step is iterated a predefined number of times. The resulting zone forest captures virtually all light transport happening in the scene up to a given number of bounces.

$$\text{render} : World \rightarrow Camera \rightarrow \mathbb{N}_+ \rightarrow RGB^p$$

```
render( world, camera, d )
    // Phase One
    for each light in the world:
        leaves = [ light.rootZone ]
        for [0,d):
            for each leaf in leaves:
                for each surface in the world:
                    if leaf does not hit surface:
                        continue[10]
                    if surface reflects diffusely:
                        add reflected child zone under leaf
                    if surface reflects specularly:
                        add reflected child zone under leaf
                    if surface transmits:
                        add refracted child zone under leaf
    // Phase Two
    for each pixel in camera.screen:
        shoot a ray from camera.viewpoint through the pixel
        find nearest intersected surface
        if no surface hit:
            pixel.color = color of sky
            continue
        zones = every zone coming from the surface
        for each zone in zones:
            if zone does not hit camera:
                continue
            ray.paint( color of zone )
            while zone is not a root zone:
                occlusion = how much of zone's intensity is blocked
                            from intersection point
                if 1 == occlusion:
                    ray.color = (0, 0, 0)
                    break
                dampen ray by occlusion
                if zone was born from...
                    diffuse reflection:
                        ray.intensity *= parent zone's intensity
                                         at intersection point
                        send ray toward brightest point
                                         of parent zone's surface
                    specular reflection:
                        send ray in mirror direction
                    refraction:
                        send ray in refracted direction
                zone = parent zone
                intersect zone's surface
            pixel.color += ray.color
    return pixels
```

[10]The *continue* instruction will immediately skip to the next iteration of the loop. It is also known as *next* in some languages.

As mentioned before zones, and consequently, the whole zone tree contains some useless, invalid paths as well. The point is that no valid path gets *left out*: if a path from the camera to a light source is valid then it *must* travel inside one of the branches of the zone forest.

Phase One traces infinite sets of *light* rays as opposed to *eye* rays, i.e. it traces light forward. This rare approach means we are willing to do some extra CPU work (exactly how much strongly depends on the scene being rendered) to get a complete picture of all the light transport happening in the scene. If done right forward light tracing nets us the following advantages compared to the opposite approach:

- **Decoupling of light transport and camera position**: We should be able to render faster to a dynamic camera than with ordinary eye path tracing because the zone forest is independent of camera position and does not need to be rebuilt as the camera moves around. Multiple cameras will be cheaper.
- **Easy shadows**: No shadow rays ("shadow zones") should be necessary.
- **Cheaper caustics**: Surface points whence shadow rays are blocked by see-through objects are hard to deal with in path tracing. The zone tree "guides" the sampling rays through the refractive object instead of shooting rays blindly in all directions.

Another way to think about the zone tree is as the simplest and most straightforward of all acceleration structures (stretching the meaning of the term a bit): all it does is describe the sets of paths that end up being valuable in the raytracing/rasterization phase. Tracing is then limited to only those paths that actually connect the camera with a light source or are blocked on their way to a light source, basically leading to a particularly effective form of importance sampling.

Consider a dark room (B in figure 2.5) illuminated only by a single extremely bright beam of light shone through a keyhole. The beam hits a wall or a few objects and the rays in it are scattered all over the room. The rays might potentially end up reaching every last surface primitive effectively lighting up the whole room.

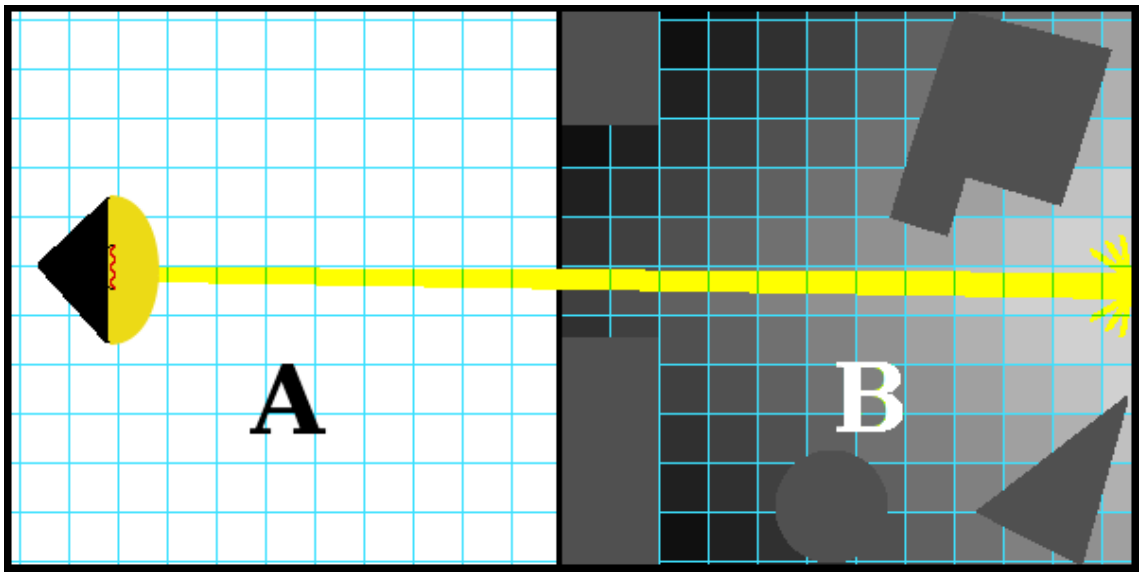


Figure 2.5: The keyhole scenario, a notoriously difficult type of scene

The bidirectional path tracing algorithm[13] and especially the venerable Metropolis path mutation algorithm[12] excel at such difficult test cases. Once the Metropolis algorithm finds the "bottleneck" in the scene, path generation becomes faster and more efficient.

Our zone based approach attacks the problem the other way around: it first detects all possible light paths by building the zone tree. Then in the second phase it determines that most of the zones need not be rendered because they stay in chamber A where they are invisible to the camera (these queries are cheap). Actual path tracing is restricted to those branches of the zone tree that reach across the keyhole.

3.4.2 Phase Two

The bulk of the processing effort is made in Phase Two. Phase Two typically takes several orders of magnitude longer than Phase One.

Phase Two is about rasterizing all visible zones to the screen. It basically uses a more controlled variant of path tracing where every ray follows a given zone tree branch upward. For every bounce the loop checks whether the latest intersection point is completely covered from the light of the zone by an occluder, and if it is, the path is terminated with a null result. At each diffuse bounce the zone's intensity at the intersection point is queried and incorporated into the color of the ray. While not entirely correct, notice that Phase Two in the pseudocode is deterministic, it does not

use randomness. If full correctness can be achieved without reintroducing nondeterminism, we will have reached our main goal.

The author has tried two different rasterization strategies:

1. **Zone by Zone**: For each zone see if it hits the camera. If it does shoot one ray through each pixel of the 2D bounding box of its source surface to determine the zone's contribution to the color of the pixel. The problem with this strategy is, of course, the exponentially increasing number of zones as one goes deeper into the zone trees.
2. **Pixel by Pixel**: Shoot one ray through each pixel of the screen to see what surface it hits. Look up all the zones emanating from that surface (2D bounding boxes help). Add up their contributions to the color of the pixel. (There can be any number of zones coming from a given surface due to indirect illumination.)

By and large the latter strategy has won in terms of performance. It is even easier to parallelize and requires less intersections to be calculated. "Zone by zone" will use exponentially more operations as the scene gets larger (has more objects in it) whereas "pixel by pixel" may turn out to be sublinear if the bulk of the surfaces are hidden from the camera (such as in the keyhole scenario discussed in the previous section). Only the pixel by pixel strategy is presented in the pseudocode above. Both strategies will be available in the codebase for experimentation.

The challenges of geometry arising from reflection and refraction are dealt with exactly the same way as in ordinary path tracing: since we are using rays of infinitesimal thickness again we can just bounce them off surfaces according to the same laws as in Chapter One without complications.

A major benefit to the zone forest in Phase Two is that it unambiguously "guides" the eye rays back to the corresponding light source. We don't need to work out what surface the ray hits next in each iteration because we already *know* where the light is coming from. This effectively translates to less intersection tests overall, which in turn means better performance.

Instead of shooting shadow rays to test the visibility of light sources from diffuse surfaces Phase Two relies on the intelligence acquired about shadows in Phase One. We expect a zone to be able to tell us what fraction of its intensity is blocked by

occluding surfaces. We use this information to adjust the color of the ray or to terminate the ray altogether if it turns out to go through a shadow.

3.5 Technical Details and a Few Tricks

Both algorithms are implemented purely on the CPU. Silence is only a proof of concept: a demonstration of the algorithm's viability. In fact both the clarity of the source code and the merits of the whole zone tracing concept would have suffered from a graphics processor implementation. As Professor Szirmay-Kalos reckoned if it's not worth doing on the CPU, it's not worth doing on the GPU either[11]. The algorithm is easy to parallelize anyway so we can expect fair results from an eventual GPU "zone tracer" as well.

C++ was chosen as the language of implementation because it has long been the lingua franca of the computer graphics community as well as a language almost universally understood among working software developers. It also compiles to relatively efficient executables.

The featuresets of Retra and Silence are kept extremely small on purpose to highlight their differences while also keeping them directly comparable. Silence is still in alpha status. The author apologizes for all bugs that inevitably ended up in the codebase and is counting on the computer graphics community to point out his most glaring mistakes and perhaps offer advice on how to build up the implementation more skillfully the next time around.

3.5.1 The Implementation of Zones

On the implementation level a zone is an object made up of the following data members:

- A reference[12] to the **scene** it is in.
- A reference to its **node** in the zone tree.
- A **light beam**, which in turn includes
    - The **source**: the originating surface.

[11]In the sense that prototyping, testing and comparing an experimental algorithm on the CPU against CPU implementations of established algorithms in the same domain is a useful, cheap and fairly reliable way to estimate its potential.

[12]The word *reference* is used loosely here to mean either a pointer or an actual C++ reference.

- The **apex**: the point behind the source the light seems to flow from. If an area light were replaced by a point light this is where the point light should be placed.
- The **pivot**: a strong ray used as a reference for other notional rays in the same beam.
- The **edge list**: the delimiters of the beam. An empty edge list is taken to mean "radiates indiscriminately in all directions of the hemisphere".

- Zero or more **shadow beams**.
- A reference to the **object** (or **vacuum**) it's traveling inside.

3.5.2 Intensity Distribution Functions Used

In this preliminary proof-of-concept version of the algorithm we shall abandon complete fidelity to physical reality. Although a physically correct and unbiased algorithm may well emerge once all the rough edges have been rounded off, for now issues such as conservation of energy are left for another time to be solved.

The current implementation uses more or less ad-hoc intensity distribution functions derived from physically plausible ones. A few examples of such functions are presented here. In a serious implementation a more rigorous scientific approach is needed.

The simplest distribution is theoretically almost plausible but we never encounter it in practice:

$$Intensity_{constant}(\vec{x}) = 1 \cdot \text{pivot}.\text{intensity}$$

An infinite plane emitter would behave this way. Infinite planes don't exist in nature so we have little use for this distribution.

The next simplest distribution, already mentioned in 3.2, is actually completely grounded in physics:

$$Intensity_{sphere}(\vec{x}) = \frac{1}{|\vec{x} - \text{pivot}.\text{start}|^2} \cdot \text{pivot}.\text{intensity}$$

This kind of radiator emits light uniformly in all directions and its intensity falls off regularly as one backs away from it according to the inverse square law. Point lights work like this, as do spherical lights.

When viewed from a large distance an emitter of finite size works like a point light for all intents and purposes. Therefore we can allow the following assumptions:

a) If the receiving surface point is up close to the radiating surface, we can substitute an infinite plane light for the radiator and assume the point receives nearly its full intensity.

b) If the receiving surface point is far away from the radiating surface, we can substitute a point light for the radiator and calculate the resulting intensity according to the inverse square law.

In order to transition smoothly from area light to point light behaviour we construct the following function:

$$distance = |\vec{x} - \text{pivot}.\text{start}|$$

$$Intensity_{polygon}(\vec{x}) = \frac{1}{e^{distance}} \cdot Intensity_{constant}(\vec{x}) + (1 - \frac{1}{e^{distance}}) \cdot Intensity_{sphere}(\vec{x})$$

Note that pivot direction does not factor into any of these distribution functions.

3.5.3 Shadows

As of the submission of the thesis shadows do not work yet. A preliminary suggestion to handle polygonal shadows is presented nonetheless.

For the umbra we pair up the vertices of the occluder polygon with the "most aligned" vertices of the light source, i.e. pointing in about the same direction from the center of mass of their respective polygons. The umbra is taken to be the volume delimited by the occluder and the rays connecting each pair. For the penumbra we do the same but with the "least aligned" vertices (pointing in opposite directions). This approach is uncomplicated and works reasonably well.

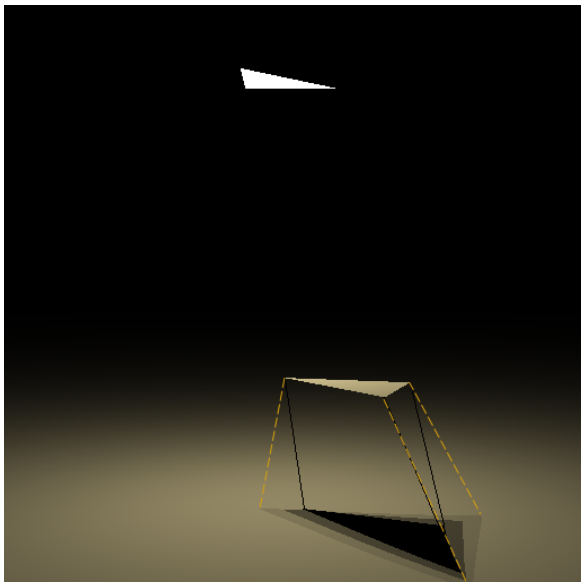


Figure 2.6: umbra, penumbra and the first split in the interpolation

The degree of occlusion in the umbra is 1 by definition. For shading the penumbra an easy interpolation scheme can be used: let us divide the penumbra into 2^n ranges by repeatedly averaging first the edges of the umbra and the edges of the penumbra, then the two nearest resulting edges, n times. Given a point x in the kth range indexed from the umbra the degree of occlusion at x will be $\frac{1-k}{n}$.

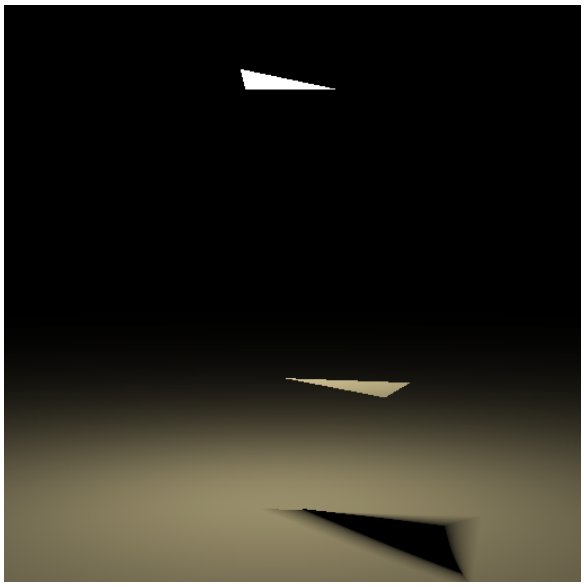


Figure 2.7: full interpolation of the penumbra

Depending on the complexity of the scene this approximation may run faster or slower than brute force light sampling by shadow rays. $n=4 \Rightarrow 2^n=16$ is a reasonable practical choice for n.

Note that even the simplest non-polygonal shape – a sphere – should cause considerable difficulties as an occluder.

3.5.4 Input-Specific Specializations

Some implementation nuances were tailored a little to the particular kind of scenes used in testing. This was done in order to show the viability of the algorithm, and obviously compromises generality. Consider the Cornell box scene: the side walls are at a right angle to the light source. Using this fact we can write the direct illumination of a wall as

$$L_i(\alpha) = \mathrm{I} \cdot \frac{\sin(\alpha)\cos^3(\alpha)}{d^2}$$

where α is the angle between the ceiling and the light ray, d is the distance between the light source and the wall, L_i is the light intensity arriving to the point on the wall at angle α and I is the initial intensity. This function peaks at $\alpha = \frac{\pi}{6}$, so that is the angle the best lit point is going to be found at.

Another example of molding the program into what works best in practice is "reflect-to-diffuse truncation". The effect of light reflected off a mirror or metallic object to a diffuse surface is usually barely visible to the naked eye. During Phase One if a diffuse surface is hit after a reflecting surface, Silence takes to liberty to stop expanding the tree after that node altogether. The rare cases where a reflective surface does change the illumination of a diffuse surface in a visible way are thus handled incorrectly, or more precisely, not handled at all. Metal rings placed on a flat diffuse surface are a famous example of this effect.

Note that no truncation is allowed after refractive surfaces because they often produce very apparent caustics.

The complications arising from refractive objects being inside one another (such as a glass ball in a tank of water) are sidestepped entirely in Silence by disallowing overlapping transparent objects.

Whether or not minor hacks like these are appropriate for the scene being processed could be decided algorithmically in a smarter, more advanced implementation.

3.5.5 Classes and Their Responsibilities

No serious software engineering thesis can be complete without a mundane discussion of the concrete class hierarchy used in the application and some UML. The application analyzed is Silence because it is the main focus of the thesis.

Since it adds little value to the rest of the thesis the reader may safely skip this part unless they are here exactly for some rudimentary Object-Oriented software design.

The interrelations of the application's core classes can be seen in the following diagram:

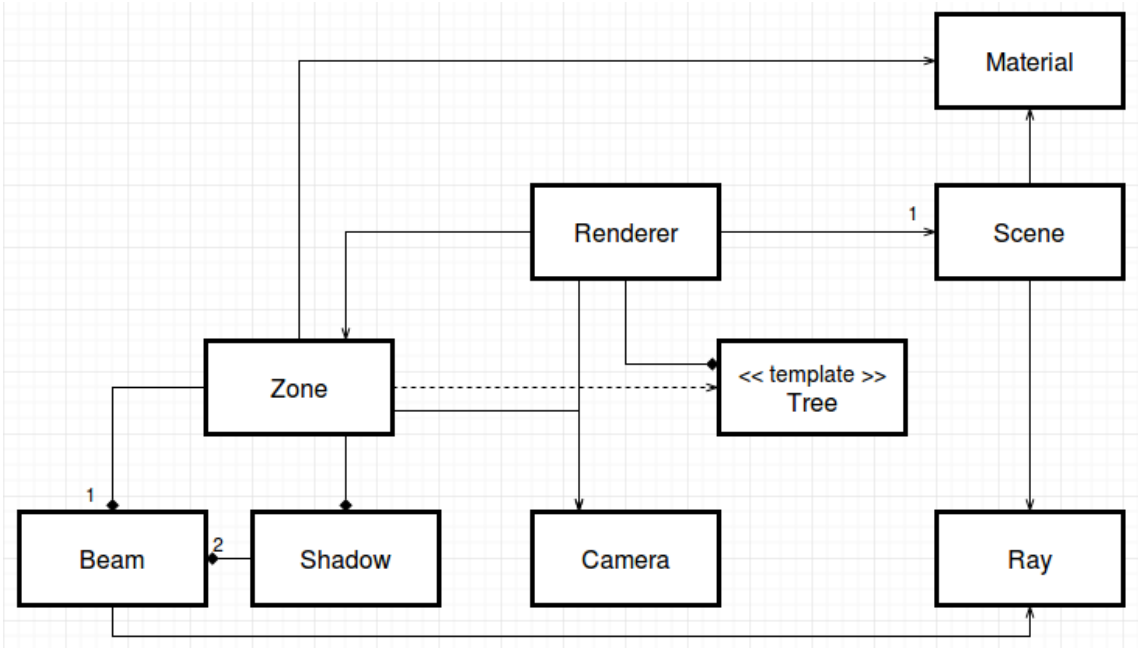


Figure 2.8: UML class diagram of the core classes

A manager class named **Renderer** is what conducts the whole rendering process. The *main* function issues requests exclusively to a member of this class, it does not interact with **Zone**s and **Camera**s directly.

A **Renderer** is attached to a **Scene**. Several Cameras can be assigned to a **Renderer**. This architecture supports an arbitrary number of **Camera**s in the same **Scene** and is easily extensible.

Most classes are designed to allow free inspection from the outside but little to no mutation. Widespread inspection of each other is quite natural in a problem domain where everything revolves around the same purpose: tracing light and rendering an image.

Zones and **Shadow**s *contain* **Beam**s instead of inheriting from them. This is due only to the author's conceptual preference. Inheritance would have been just as appropriate, if not more.

Rays are a very lightweight immutable class with no intensity info, only pure geometry.

The contents of a **Scene** are represented using an extensive class hierarchy with the abstract classes **Object** and **Surface** on top. **Object** captures the idea of a single homogeneous entity with constant properties across its arbitrary number of **Surface**s, whether a **Light** or a passive **Thing**. A **Surface** is a general geometric primitive.

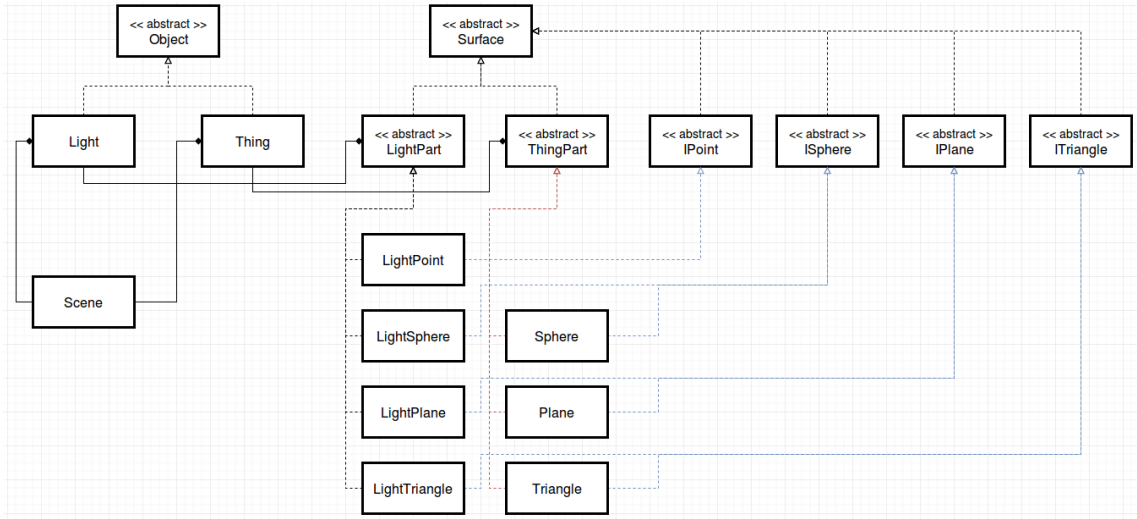


Figure 2.9: UML class diagram of the Scene hierarchy

All concrete surface primitives are derived from either **LightPart** or **ThingPart**, both abstract classes. **LightPart**s and **ThingPart**s are contained by **Light** and **Thing**, respectively, and they are in turn contained by the **Scene** itself as shown in figure 2.9.

The **Tree** template class is entirely agnostic of its contents, it has no knowledge of either **Beam**s or **Zone**s. It is a well-known rule of thumb in software design that containers and functors should be unaware of their contents when possible[13]. That said, two sensible reasons could be given for fusing **Tree** with **Zone** into a concrete type in this particular case:

1. Less indirections in the source code would make it more readable. The following quote from the source code shows the clumsiness of the orthodox approach.

   ```
   const Triplet& color = (*leaf)->getValue()->getLight().getColor();
   ```

[13] In fact functors are outright *prohibited* from depending on the underlying type.

2. A smart **Tree** could actively participate in the rendering algorithm. If the programmer is allowed to, say, invoke a method to "walk back up the tree from node *n* to the root, apply *f* to each node and return the cumulative result" a lot of boilerplate is saved and the core rendering loop becomes cleaner.

3.6 Summary

In general one can expect the following gains and penalties for choosing a zone-based approach over naive path tracing:

+ Noise-free rendering

+ Partial decoupling of light transport and camera position

+ Multiple cameras are cheaper

+ Degrades gracefully (under tight time constraints one can always fall back to rendering just the zones near the top of each tree)

+ Clean and natural separation of light tree levels or "bounces"; easy to test and debug

+ Versatile "acceleration structure", can be used with different core algorithms

– Extra development effort

– Some optimizations will definitely compromise unbiased results (see 3.5.4)

4 Comparison

4.1 Empirical Results

This chapter includes some practical performance measurements done with both Retra and Silence side by side. Test cases that took a very long time were run once while easier cases were repeated three times each and the results averaged. All measurements were done on a laptop with the following specs:

- **Processor**: Intel Core i5-5200U running at 2.2 GHz (hyperthreading enabled)
- **Memory**: 4 GB of DDR3 RAM
- **Display**: 15.6" LED monitor, 1366x768 pixels with 466:1 contrast
- **Operating System**: Debian GNU/Linux 8 "Jessie"

The reason for including display specs is the fact that stochastic noise in path traced output makes the two algorithms non-trivial to compare. The author used his best judgement to determine when an image has converged enough to be very hard to tell from the actual solution at no zoom.

The performance tests are admittedly rather limited. Silence is still in alpha status so test cases had to be picked not to expose its defects.

The test cases are small, closed scenes made up of spheres, planes and triangles described in human-readable JSON format[40]. A built-in scene file parser is included with both rendering programs.

In terms of performance the new method appears to fall strictly between direct rasterization and vanilla path tracing with no acceleration structures. "Zone accelerated path tracing" is about one or two orders of magnitude faster at rendering global illumination than plain path tracing depending on the exact scene being rendered.

Since even Retra can produce direct illumination (more precisely an image with a single bounce) in a matter of seconds we expect direct illumination to be very fast in Silence.

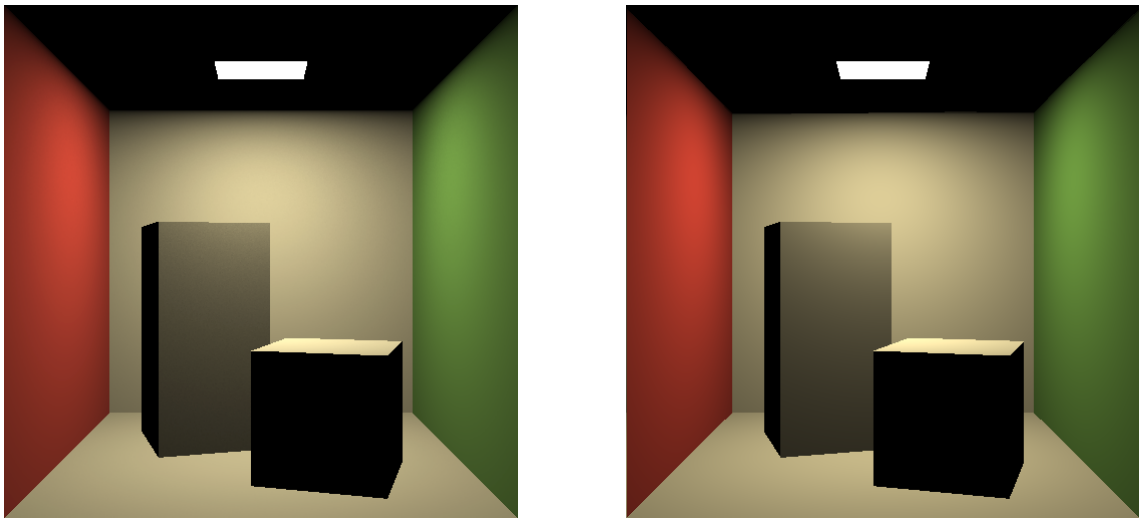


Figure 3.1: Cornell box, direct illumination. Left: Retra, 103 secs (256 SPP); right: Silence, 2 secs

The difference in efficiency is more dramatic when we allow each program only a limited number of paths to work with ("samples per pixel" is not really applicable to Silence). The following images were produced using a maximum of 2^{20} (a little over one million) individual paths each. The difference in visual quality is striking.

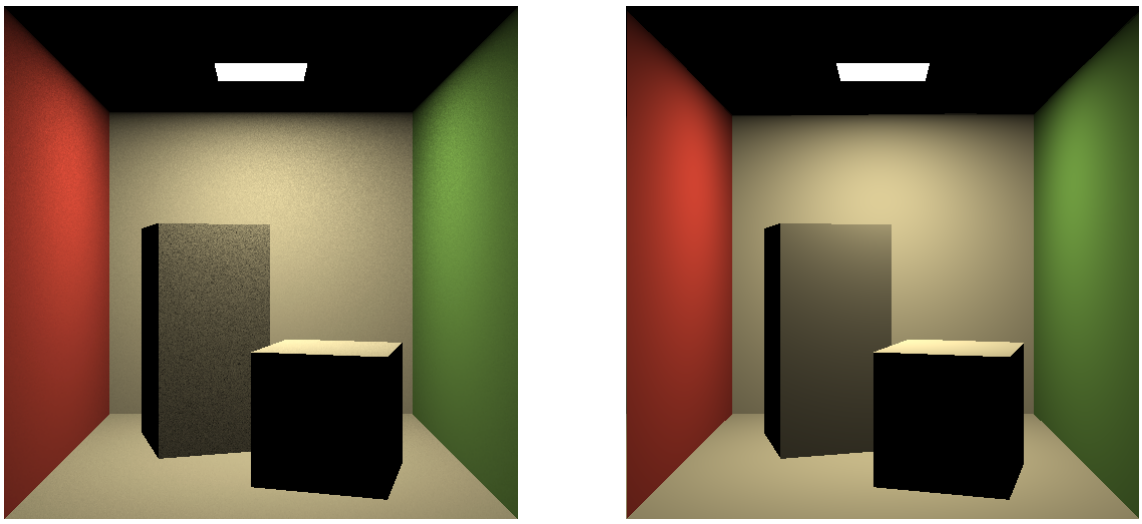


Figure 3.2: direct illumination, 2^{20} paths max. Left: Retra, 2 secs (4 SPP); right: Silence, 2 secs

In this case both programs took about the same time to render the scene.

The following minimal scene demonstrates a typical global illumination effect known as color bleeding.

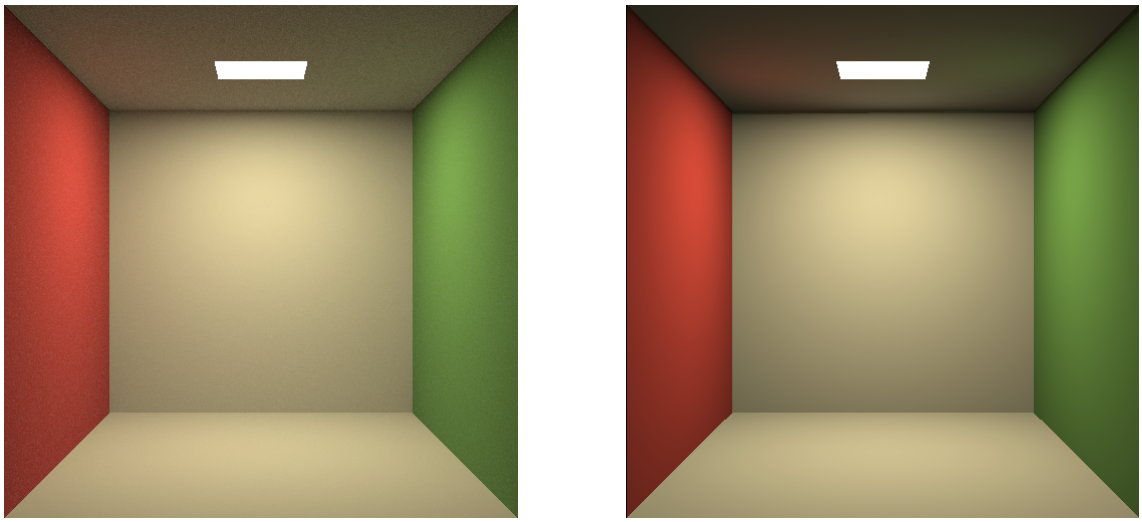


Figure 3.3: empty box, global illumination. Left: Retra, 41 mins (256 SPP); right: Silence, 11 secs

Note that Retra spends most of the rendering time on shadow rays even though the scene is completely empty. Silence is able to make better use of the limited resources because it eschews random sampling for more direct, analytically based shading. The zone tree guides the rays towards the areas where most of the light is actually coming from and minimizes the overhead caused by producing useless paths of no interest.

The extra information about the intensity distribution of zones speeds up flat surface shading considerably.

4.2 General Features

The following table summarizes the key differences between the two algorithms both in terms of theoretical capabilities and practical performance as measured.

	Naive Path Tracing	**Path Tracing with Zones**
Correctness (as implemented)	unbiased	biased
Correctness (theoretical)	unbiased	unbiased
Generality (as implemented)	perfectly general	simple geometry, constant colored surfaces
Generality (theoretical)	perfectly general	*unknown*
Time complexity (as implemented)	$O(psdn)$	$O(n^d + pdn)$
Time complexity (theoretical)	$O(psd(\log n))$[22]	*unknown*
Storage complexity[14]	$O(n)$	$O(n^d)$
Lines of Code	1273	1897
Cyclomatic Complexity[15]	298	481
Performance (direct illumination)	103 seconds	2 seconds
Performance (2^{20} paths max)	2 seconds	2 seconds
Performance (full)	2460 seconds	11 seconds

[14]Today computer memory is cheap and abundant, so storage complexity is of little practical consequence.

[15]Lines of code and cyclomatic complexity were measured by the codebase metrics tool CCCC[42].

5 Assessment of Work and Results

5.1 Points of Design

The author made a decision early on during the development of Retra to stick to the old C++98 standard, a choice he now regrets. For instance the awkwardness of the old "time.h" library in a multithreaded environment definitely outweighs any portability concerns. The far superior and more reliable "chrono" C++11 library is recommended instead. All in all we have to conclude that choosing to use old standards and libraries for increased compiler compatibility and portability is a bad idea.

With this in mind Silence was developed as a C++11 application from the beginning. The author experienced no issues whatsoever with the C++11 support of the popular g++ compiler (part of the GNU Compiler Collection).

It initially seemed that a compact custom file format for scene descriptions was going to be easier to use and more flexible than a third party importer library. Indeed it proved adequate for prototyping and demonstrating the algorithm but will no longer be viable in a more serious implementation.

Adhering to Object-Oriented principles was of relative rather than absolute importance to the project, but not at all due to lack of conviction that a clean and rigorously organized codebase is just as important as results. The author needed to get Silence working quickly to meet the deadline.

The author is fairly satisfied with the design and overall code quality of Retra, but much less so with Silence. This is entirely the author's own fault: rushed work as the deadline drew closer did not help. The clearest, most valuable lesson from working on this thesis is that delaying work on your thesis until the last possible minute is inadvisable.

5.2 Points of Demonstration

Some would say that using simplistic test cases fails to demonstrate the potential of an algorithm. The author respectfully disagrees. The sparse, simple and easy-to-edit

scenes used in this thesis proved quite enough to highlight the differences between the traditional and the modified algorithm.

A miniature animation subsystem is also included for demonstration purposes. Basic trajectories or "motions" for in-worlds objects are defined in a JSON file similar to scene descriptions.

The thesis mixes set theory and type theory notation to convey the types of data and functions in a really informal way. This is done to give an immediate, intuitive and clear idea of each type and function in a language agnostic way. A check from the supervisor concluded that this was acceptable.

The author wishes he knew more GIMP or some other image editor application. The illustrations in the thesis are quite lacking.

The author also realizes that textual formats like TeX are superior to binary document formats but he chose not to learn TeX for fear of using up too much time. No praise can be given to the bug-ridden and unreliable LibreOffice though.

Some ubiquitous techniques were surprisingly hard to find solid sources on. One would expect seminal papers to turn up on the first page of a simple Google search for terms like "Whitted ray tracing" or "Russian roulette". One would be wrong.

The discussions of both traditional and modified path tracing were composed with the reader in mind and an honest attention to detail. It is the author's sincerest hope that the thesis is of real value and that readers will feel they gained something from it.

5.3 Findings

The actual performance increase from the new method seems to fall remarkably close to what the author initially expected (a real surprise in an industry where serious estimates routinely fail by orders of magnitude). Silence should be able to beat the naive renderer Retra by one or two orders of magnitude in most test cases depending on what level of output convergence we demand of the latter.

Keeping in mind that Silence is an extremely rudimentary and primitive first attempt at harnessing the model's power it is safe to expect more algorithmic and performance gains from a more sophisticated and well thought-out implementation.

We have seen that circa one million rays is enough for Silence, a basic pure-CPU implementation to produce renders of acceptable, even enjoyable, quality. 30 million rays a second is well within what a modern high-performance GPU-accelerated rendering engine can manage, which means such an engine should be able to render these scenes at 30 FPS in comparable quality without breaking a sweat.

Quite naturally the bulk of the rendering time is still spent on finding ray-surface intersections. This is expected and unlikely to change either in this algorithm or indeed in any other that simulates global illumination.

6 Possible Further Improvements

Silence, a basic toy implementation of the new method comes with a lengthy todo list. The most important task ahead is of course to actually iron out the details of the algorithm and produce an accurate, unbiased instance of it.

There is no exact roadmap for the development of Silence. It has been a one man project and it remains in a state of flux and continuous experimentation. Perhaps the way to a correct and robust zone based renderer is to start with a more rigorous, more disciplined theoretical approach, but frankly, that transcends the scope of this thesis.

The following additions should be considered in the long term:

- Implement correct **occlusion**.
- Use **smart interpolation** across each surface (instead of sending rays pixel by pixel) to drastically reduce number of rays needed for rasterization.
- Gather more **geometric information** during the tree building phase, e.g. detect impossible paths between entire groups of surfaces.
- Handle true **general BRDF's**.
- Handle more complex **geometry**. (Although the algorithm may turn out to be ill-suited for this.)
- Develop smarter ways to **discard zones**. (Weak and unimportant zones that contribute little to the final image soak up much of the processing effort. Static limit cutoff is too primitive to combat this issue effectively.)
- Are **pivots** even necessary? (In retrospect it seems that pivot rays are a rather useless component to zones. We could go ahead and use pivotless zones instead.)
- Use a third party scene loader library like **Assimp**[41].
- **Import/Export** zone trees to file. (Large static scenes may benefit from this.)
- Develop a **GPU accelerated** version. (Expected to be several orders of magnitude faster.)

7 Acknowledgements

The author wishes to express his thanks to supervisor Dr László Szirmay-Kalos for his valuable remarks and encouragement and to his own family and significant other for their patience and understanding in the final stages of development and writing.

The suggestions of Péter Hudák and Kornél Kálmán, graduates of BUTE, also helped improve this thesis and are much appreciated.

References

[1] T. Whitted: *An Improved Illumination Model for Shaded Display*, 1980. (Communications of the ACM 23)

[2] D. S. Immel, M. F. Cohen, D. P. Greenberg: *A Radiosity Method for Non-Diffuse Environments*, 1986. (SIGGRAPH '86)

[3] J. Kajiya: *The Rendering Equation*, 1986. (SIGGRAPH '86)

[4] R. L. Cook: *Stochastic Sampling in Computer Graphics*, 1986. (Transactions on Graphics 5)

[5] M. Radziszewski, K. Boryczko, W. Alda: *An Improved Technique for Full Spectral Rendering*, 2009. (Journal of WSCG 17)

[6] H. W. Jensen, S. R. Marschner, M. Levoy, P. Hanrahan: *A Practical Model for Subsurface Light Transport*, 2001. (SIGGRAPH '01)

[7] F. Nicodemus: *Directional Reflectance and Emissivity of an Opaque Surface*, 1965. (Applied Optics)

[8] B. Walter, P. Shirley: *Cost Analysis of a Ray Tracing Algorithm*, 1997.

[9] T. Möller, B. Trumbore: *Fast, Minimum Storage Ray/Triangle Intersection*, 1997. (Journal of Graphics Tools)

[10] E. Veach, L. J. Guibas: *Optimally Combining Sampling Techniques for Monte Carlo Rendering*, 1995. (SIGGRAPH '95)

[11] M. Ragheb: *Russian Roulette and Particle Splitting*, lecture notes, 2013.

[12] E. Veach, L. J. Guibas: *Metropolis Light Transport*, 1997. (SIGGRAPH '97)

[13] E. P. Lafortune, Y. D. Willems: *Bi-Directional Path Tracing*, 1993. (COMPUGRAPHICS '93)

[14] F. Rousselle, C. Knaus, M. Zwicker: *Adaptive Sampling and Reconstruction using Greedy Error Minimization*, 2011. (Proceedings of the 2011 SIGGRAPH Asia Conference)

[15] P. Sen, S. Darabi: *On Filtering the Noise from the Random Parameters in Monte Carlo Rendering*, 2012. (ACM Transactions on Graphics 31)

[16] N. Kalantari, S. Bako, P. Sen: *A Machine Learning Approach for Filtering Monte Carlo Noise*, 2015. (SIGGRAPH '15)
http://cvc.ucsb.edu/graphics/Papers/SIGGRAPH2015_LBF/

[17] H. W. Jensen: *Global Illumination using Photon Maps*, 1996. (Rendering Techniques '96)

[18] C. M. Goral, K. E. Torrance, D. P. Greenberg, B. Battaile: *Modeling the Interaction of Light Between Diffuse Surfaces*, 1984. (Computer Graphics 18.)

[19] C. Crassin, F. Neyret, M. Sainz, S. Green, E. Eisemann: *Interactive Indirect Illumination Using Voxel Cone Tracing*, 2011. (Proceedings of Pacific Graphics 2011)

[20] Ch. Buckalew, D. Fussell: *Illumination Networks: Fast Realistic Rendering with General Reflectance Functions*, 1989. (SIGGRAPH '89)

[21] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, I. Wald: *Packet-Based Whitted and Distribution Ray Tracing*, 2007. (Proceedings of Graphics Interface 2007)

[22] M. Pellegrini: *Ray-Shooting on Triangles in 3-Dimensional Space*, 1993. (Algorithmica 9)

[23] Wikipedia: *Spectral rendering*, https://en.wikipedia.org/wiki/Spectral_rendering

[24] Wikipedia: *Lambertian reflectance*, https://en.wikipedia.org/wiki/Lambertian_reflectance

[25] Wikipedia: *Helmholtz reciprocity*, https://en.wikipedia.org/wiki/Helmholtz_reciprocity

[26] Wikipedia: *Saturable absorption*, https://en.wikipedia.org/wiki/Saturable_absorption

[27] Wikipedia: *Superposition principle*, https://en.wikipedia.org/wiki/Superposition_principle

[28] Wikipedia: *Lambert's cosine law*, https://en.wikipedia.org/wiki/Lambert%27s_cosine_law

[29] Wikipedia: *Snell's law*, https://en.wikipedia.org/wiki/Snell%27s_law

[30] Wikipedia: *Hidden surface determination*, https://en.wikipedia.org/wiki/Hidden_surface_determination

[31] Wikipedia: *Embarrassingly parallel*, https://en.wikipedia.org/wiki/Embarrassingly_parallel

[32] Plain PPM file format specification, http://netpbm.sourceforge.net/doc/ppm.html#plainppm

[33] Internal math functions of the Ogre 3D engine, https://github.com/ehsan/ogre/blob/master/OgreMain/src/OgreMath.cpp

[34] StackExchange thread on sampling the unit sphere uniformly, http://stats.stackexchange.com/questions/7977/how-to-generate-uniformly-distributed-points-on-the-surface-of-the-3-d-unit-sphere

[35] The Brigade real-time path tracer engine, https://home.otoy.com/render/brigade/

[36] J. Ph. Hakenberg: a special relativistic raytracer, http://www.hakenberg.de/diffgeo/special_relativity.htm

[37] D. Bucciarelli: *Sfera*, https://code.google.com/archive/p/sfera/

[38] L. Dymchenko: *AntiPlanet*, http://www.virtualray.ru/eng/download.html

[39] E. Biddulph: *Quake 2 Realtime GPU Pathtracing*, http://amietia.com/q2pt.html

[40] JavaScript Object Notation examples, http://json.org/example.html

[41] The Open Asset Import Library, http://www.assimp.org/

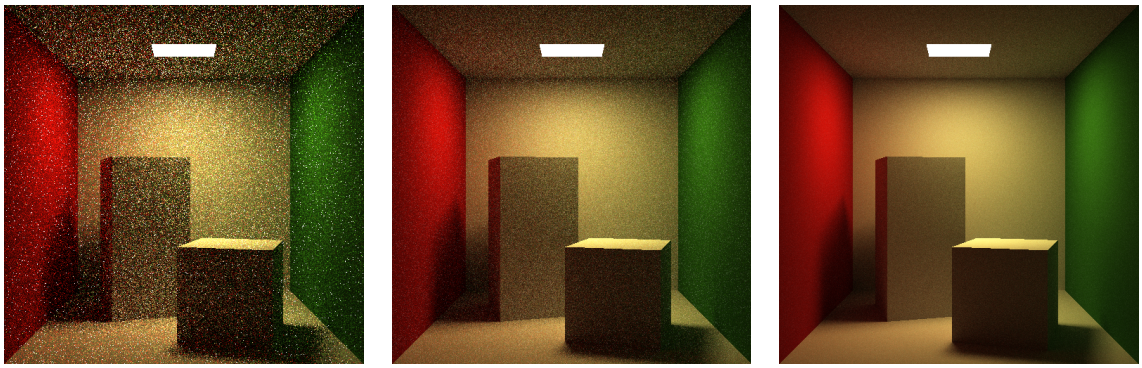
[42] The C and C++ Code Counter, http://cccc.sourceforge.net/

[43] J. D. Carmack: *Principles of Lighting and Rendering*, keynote talk at QuakeCon 2013, https://www.youtube.com/watch?v=IyUgHPs86XM

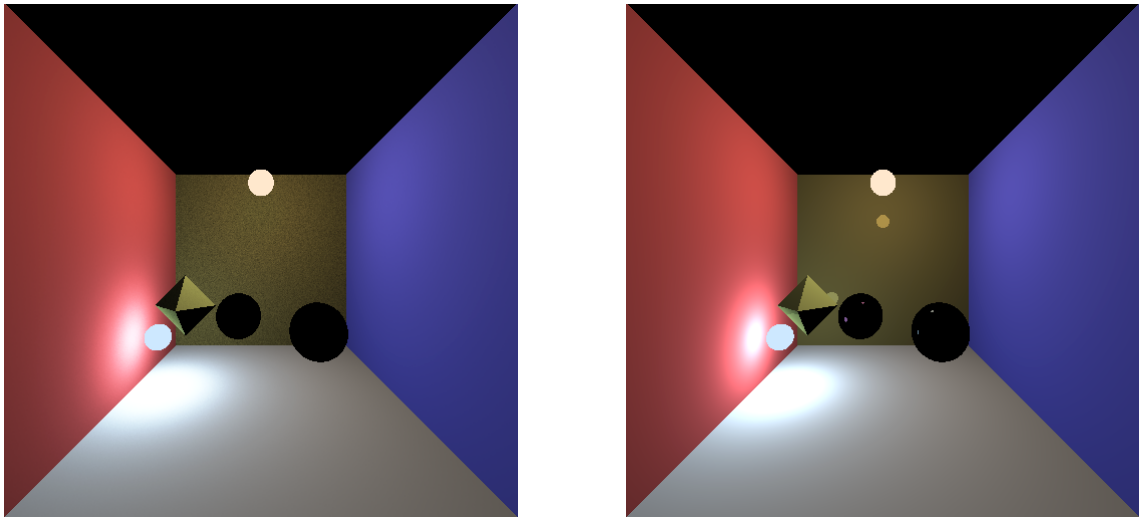
[44] The path tracing demo application, https://github.com/nilthehuman/Retra

[45] The novel demo application, https://github.com/nilthehuman/Silence

Appendix



A demonstration of noise naturally disappearing from a path traced image gradually as the sample size is increased. From left to right: results after 5 seconds, after 1 minute and after 10 minutes.



Direct illumination in a scene involving metallic surfaces similar to gold or copper. Left: Retra, 117 secs (256 SPP); right: Silence, 2 secs