Path tracing: méthodes de Monte Carlo pour la synthèse d'images

Theoretical introduction

Aurélien Delval

Projet de Programmation Numérique

October 14, 2024

UVSQ université PARIS-SACLAY | ISTY
Institut des Sciences et Techniques des Yvelines
CAMPUS DE MANTES EN YVELINES
CAMPUS DE SAINT-QUENTIN-EN-YVELINES

Table of contents

General introduction

General introduction

Nowadays, 3D computer graphic techniques consist of two main families:

- **Rasterization** is a fast technique that consists of two main steps:
  - A 3D scene is converted into a 2D vector image using a **3D projection**.
  - This vector image is then converted into pixels (the *rasterization* process itself).
  - The color of each pixel is finally computed by *shaders*, subprograms tasked with simulating diverse lighting effects.

  This is the cheapest and most common technique. It is famously implemented by APIs such as **DirectX** and **OpenGL**.

General introduction

- **Ray tracing** (and similar) techniques are usually more expensive but have a higher fidelity. The general idea is to simulate the trajectory of photons (or "*light rays*"). This is usually done backwards (from the camera to a light source), as most rays never hit the camera.
  - We throw a ray for each pixel of the image we want to generate.
  - If the ray encounters an object, it will bounce at a certain angle.
  - After enough bounces or distance, we compute the ray's resulting color based on the objects met.

$\rightarrow$ In this project, we are interested in developing a **path tracing** renderer, which combines standard **ray tracing** techniques and **Monte Carlo** methods.

Basic ray casting

Principle of ray casting

Ray casting is a simplified version of ray tracing, where rays stop when they encounter an object. We can generate a complete render of our scene by casting one ray per pixel of the image.

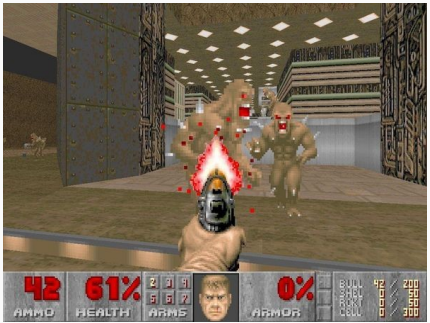


Figure 1: The original Doom game used the technique of ray casting (**Source**: Computer History Museum, © id Software)

- Even if we do not compute bounces in path casting, all its concepts are reused by the more advanced ray tracing techniques.
- We will first implement a ray casting renderer before generalizing it to path tracing.

Definition of the camera

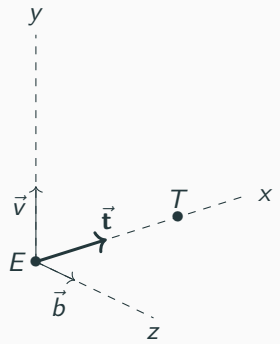


Figure 2: Model of camera

First, we define our "**camera**" by:

- an "*eye*" position E
- a direction $\vec{\mathbf{t}}$ pointing to a target T
  Additionally, we define two other vectors to form an orthonormal basis:
  - a vertical $\vec{v}$
  - a third direction $\vec{b} = \vec{\mathbf{t}} \times \vec{v}$

By default, we consider the initial values:

$$\begin{cases} \vec{t_0} = (1, 0, 0) \\ \vec{v_0} = (0, 1, 0) \\ \vec{b_0} = (0, 0, 1) \end{cases} \tag{1}$$

Rotating the camera

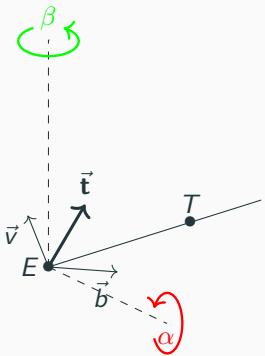


Figure 3: Camera rotation with pitch and yaw for $\alpha = \beta = \frac{\pi}{6}$

We will control the camera orientation with two rotations:

- a **pitch** of angle α (rotation around the z axis):

$$pitch_\alpha(x, y, z) = \begin{pmatrix} cos\,\alpha & -sin\,\alpha & 0 \\ sin\alpha & cos\,\alpha & \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{2.1}$$

$$= \begin{pmatrix} x\,cos\,\alpha - y\,sin\,\alpha \\ x\,sin\,\alpha + y\,cos\,\alpha \\ z \end{pmatrix} \tag{2.2}$$

Rotating the camera

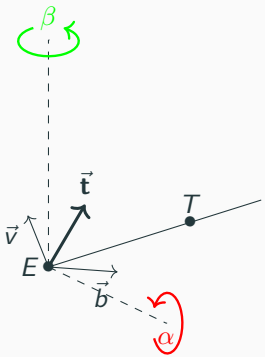


Figure 3: Camera rotation with pitch and yaw for $\alpha = \beta = \frac{\pi}{6}$

We will control the camera orientation with two rotations:

- a **yaw** of angle β (rotation around the y axis):

$$yaw_\beta(x, y, z) = \begin{pmatrix} cos\,\beta & 0 & sin\,\beta \\ 0 & 1 & 0 \\ -sin\,\beta & 0 & cos\,\beta \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3.1}$$

$$= \begin{pmatrix} x\,cos\,\beta + z\,sin\,\beta \\ y \\ -x\,sin\,\beta + z\,cos\,\beta \end{pmatrix} \tag{3.2}$$

Rotating the camera

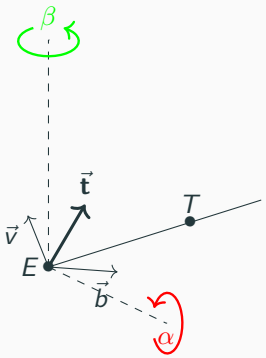


Figure 3: Camera rotation with pitch and yaw for $\alpha = \beta = \frac{\pi}{6}$

From there, we can compute $\vec{t}$, $\vec{v}$ and $\vec{b}$ from $\vec{t_0}$, $\vec{v_0}$, $\vec{b_0}$, α and β:

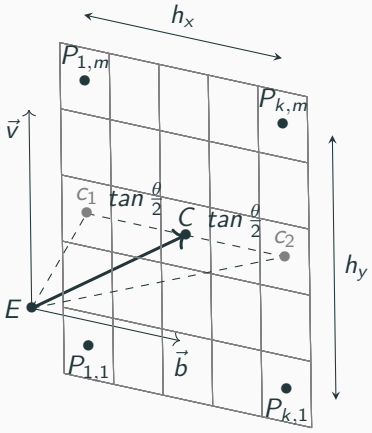
$$\vec{t} = yaw_\beta \circ pitch_\alpha(\vec{t_0}) = \begin{pmatrix} cos\,\alpha\ cos\,\beta \\ sin\,\alpha \\ -cos\,\alpha\ sin\,\beta \end{pmatrix} \tag{4}$$

$$\vec{v} = yaw_\beta \circ pitch_\alpha(\vec{v_0}) = \begin{pmatrix} -sin\,\alpha\ cos\,\beta \\ cos\,\alpha \\ sin\,\alpha\ sin\,\beta \end{pmatrix} \tag{5}$$

$$\vec{b} = yaw_\beta \circ pitch_\alpha(\vec{b_0}) = \begin{pmatrix} sin\,\beta \\ 0 \\ cos\,\beta \end{pmatrix} \tag{6}$$

□

Computation of the ray vectors



To generate a 2D rasterized image, we will project it as a "**viewport**" in our 3D scene.

It is defined by:

- a viewing angle θ (or **F**ield **O**f **V**iew)
- a width k and a height m (corresponding to the dimensions in pixels)

The viewport has a center $C = E + \vec{t}$ (we assume they are sperated by a distance $d = 1$).
Each pixel (i, j) of the viewport has center $P_{i,j}$.

Figure 4: Viewport with $k = m = 5$

Computation of the ray vectors

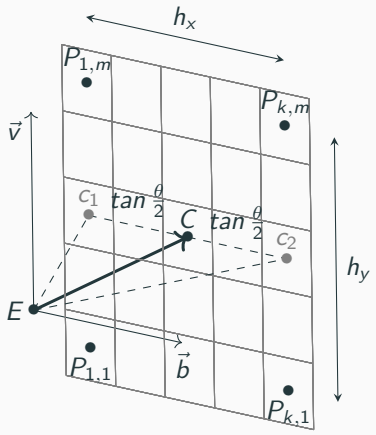


Figure 4: Viewport with $k = m = 5$

Problem: We want to calculate the **rays** $\vec{r_{i,j}}$ which are the normalized vectors giving the direction $E \to P_{i,j}$.

- First, notice that the viewport width is:

$$h_x = 2 \tan\frac{\theta}{2} \tag{7}$$

(consider separately the triangles ECC_1 and ECC_2)

- The "aspect ratio" of the viewport is $\frac{m-1}{k-1}$, hence the height:

$$h_y = h_x \frac{m-1}{k-1} \tag{8}$$

Remark: Notice how the vectors $\vec{v}$ and $\vec{b}$ are on the plane of the viewport. This will be useful in the next step.

Computation of the ray vectors

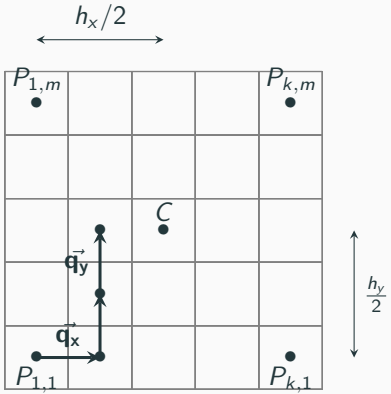


Figure 5: Pixel-shifting vectors on the 2D viewport

- We define the "**pixel-shifting**" vectors, which will allow us to "jump" from one pixel to another:

$$\begin{cases} \vec{q_x} = \frac{h_x}{k-1}\vec{b} \\ \vec{q_y} = \frac{h_y}{m-1}\vec{v} \end{cases} \tag{9}$$

- We use the bottom-left pixel $P_{1,1}$ as the reference:

$$P_{1,1} = C - \frac{h_x}{2}\vec{b} - \frac{h_y}{2}\vec{v} \tag{10}$$

- This allows us to compute the position of any pixel:

$$P_{i,j} = P_{1,1} + (i-1)\vec{q_x} + (j-1)\vec{q_y} \tag{11}$$

Computation of the ray vectors

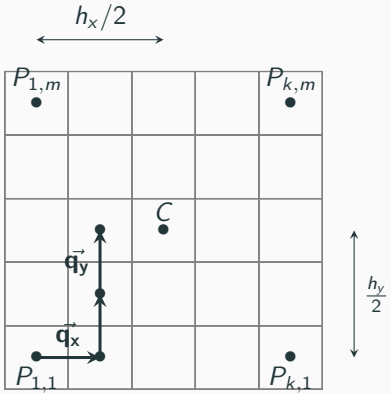


Figure 5: Pixel-shifting vectors on the 2D viewport

- In the example of figure 5, we use this method to calculate the position of $P_{2,3}$.
- We define $\vec{p_{i,j}} = P_{i,j} - E$, and finally we get the rays:

$$\vec{r_{i,j}} = \frac{\vec{p_{i,j}}}{||\vec{p_{i,j}}||} \tag{12}$$

□

→ Using these vectors, we can trace the ray until it encounters an object or travels a maximum distance.

Ray path across a scene

We define the **scene** as the set of objects represented in our image. When we extend the path of a ray, we need to find with which object of the scene (if any) it first collides.

In all cases, a ray $\vec{r}$ originating from eye e traveling a distance T will reach a point x such that:

$$x = e + T\vec{r} \tag{13}$$

For the sake of simplicity, we will only consider **sphere** and **box** objects. We have to treat them separately to find their intersection(s) with the prolonged ray $\vec{r}$.

Ray-sphere intersection

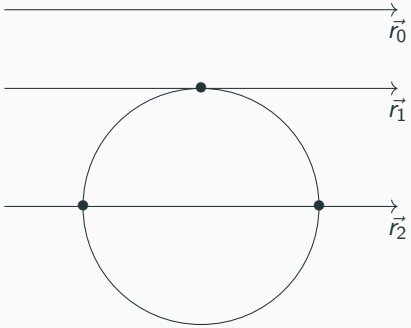


Figure 6: Possible ray-sphere intersections (0, 1 or 2 points)

First, a sphere equation is:

$$||x - c||^2 = R^2 \tag{14}$$

...with x the set of points belonging to it, c its center and R its radius.
By replacing x with the ray equation (13) we get:

$$||e + T\vec{r} - c||^2 = R^2 \tag{15}$$

Ray-sphere intersection

We define $\vec{v} = e - c$ the distance between the eye and sphere center and rewrite (15) as:

$$||\vec{v} + T\vec{r}||^2 = R^2 \tag{16.1}$$

$$\Leftrightarrow \vec{v}^2 + T^2\vec{r}^2 + 2\vec{v}.T\vec{r} = R^2 \tag{16.2}$$

$$\Leftrightarrow \vec{r}^2\,\mathbf{T}^2 + 2\vec{v}.\vec{r}\,\mathbf{T} + \vec{v}^2 - R^2 = 0 \qquad \text{(rewriting as a quadratic equation of } T\text{)} \tag{16.3}$$

$$\Leftrightarrow \mathbf{T}^2 + 2\vec{v}.\vec{r}\,\mathbf{T} + \vec{v}^2 - R^2 = 0 \qquad \text{(removing } \vec{r}^2 \text{as it is a unit vector)} \tag{16.4}$$

This equation has for discriminant:

$$\Delta = (2\vec{v}.\vec{r})^2 - 4(\vec{v}^2 - R^2) \tag{17}$$

Thus the solution(s), if any will be given by:

$$T = \frac{-2\vec{v}.\vec{r} \pm \sqrt{\Delta}}{2} \tag{18}$$

□

And the corresponding intersection point(s) is/are $p = e + T\vec{r}$, the closest on being the one we'll use.

Ray-box intersection using the slab method

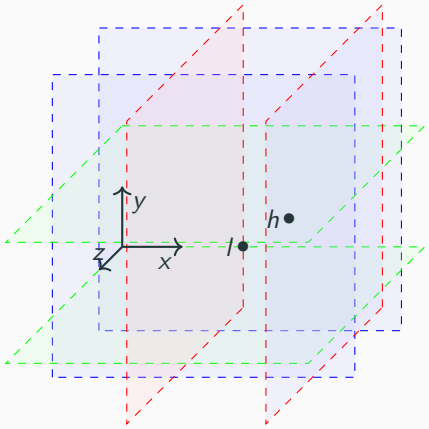


Figure 7: Axis-aligned box represented as a three-slab intersection

For ray-boxes intersections, we can use the efficient **slab method**.

An axis-aligned box (AAB) can be defined by two points l and h delimiting its low and high bounds respectively.

We can go further by defining the notion of **slabs**, ie. the space contained between two parallel planes. Using this definition, an AAB can be seen as the intersection formed between the slabs of its three pairs of parallel planes.

Ray-box intersection using the slab method

We have:

$$\begin{cases} l = (l_0, l_1, l_2) \\ h = (h_0, h_1, h_2) \end{cases} \tag{19}$$

...where:

- l_0 and h_0 are the coordinates of the x-aligned planes
- etc.

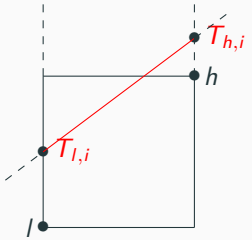


Figure 8: Visualisation of the segment intersecting a single dimension slab

Assuming the intersection(s) exist, the distances that the ray must travel on each coordinate to encounter the low and high boundary planes are, $\forall i$:

$$\begin{cases} T_{l,i} = \frac{l_i - e_i}{r_i} \\ T_{h,i} = \frac{h_i - e_i}{r_i} \end{cases} \tag{20}$$

Ray-box intersection using the slab method

We have:

$$\begin{cases} l = (l_0, l_1, l_2) \\ h = (h_0, h_1, h_2) \end{cases} \tag{19}$$

...where:

- l_0 and h_0 are the coordinates of the x-aligned planes
- etc.

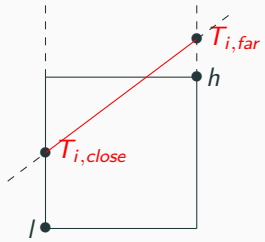


Figure 8: Visualisation of the segment intersecting a single dimension slab

The segments that intersect wich each slab i are defined by, $\forall i$:

$$\begin{cases} T_i^{close} = min\{T_{l,i}, T_{h,i}\} \\ T_i^{far} = max\{T_{l,i}, T_{h,i}\} \end{cases} \tag{20}$$

Ray-box intersection using the slab method

The intersection points with the box is the intersection between those segments at distances:

$$\begin{cases} T^{close} = max\{ T_0^{close}, T_1^{close}, T_2^{close} \} \\ T^{far} = min\{ T_0^{far}, T_1^{far}, T_2^{far} \} \end{cases} \tag{21}$$

Once again, the intersection point that interests us is the closest one:

$$p = e + T_{close}\vec{r} \tag{22}$$

□

Remark: Note that the computed intersection exists iff $T_{close} < T_{far}$.

Computing the first collision in the scene

Now that we can get the intersection of a ray with any object from the scene, the following algorithm returns the first collision:

Algorithm 1: Computation of the first collision in the scene

Input: Eye e, ray $\vec{r}$, scene $S = \{O_i\}_{i=1}^{n}$ (array of objects)
Result: hit object O, intersection point p

```
1  O ← ∅
2  p ← ∅
3  T_min ← +∞
4  for i ← 1 to n do
5  |   p_i ← getIntersection(e, r, O_i)
6  |   if p_i ≠ ∅ then
7  |   |   T_i ← ||p_i − e||^2
8  |   |   if T_i < T_min then
9  |   |   |   O ← O_i
10 |   |   |   p ← p_i
11 |   |   end
12 |   end
13 end
```

Ray casting: putting it all together

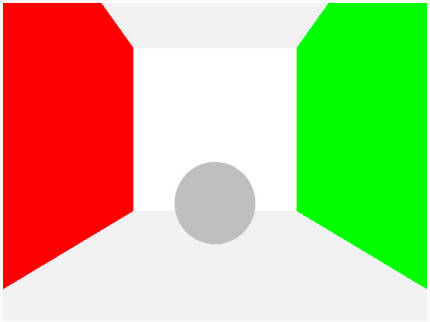


Figure 9: Render of a Cornell box with the described path-casting method

You should now be able to generate renders similar to the one on the left.

- Note that lighting is simply computed by affecting the object's colors to the pixels.
- There exist more advanced path-casting variants to compute shadows, but this goes outside the scope of this project.

$\rightarrow$ In the next section, we will see how the light gets reflected when hitting an object to simulate the ray bounces and perform "full" ray tracing.

A bit of physics

Bidirectional reflectance distribution function

Formally, the reflectance is defined as the ratio of reflected/received light at certain angles by an opaque material, and is given by the **bidirectional reflectance distribution function** (BRDF):

$$f_r(\omega_i, \omega_r) = \frac{dL_r(\omega_r)}{dE_i(\omega_i)} \tag{23}$$

where:

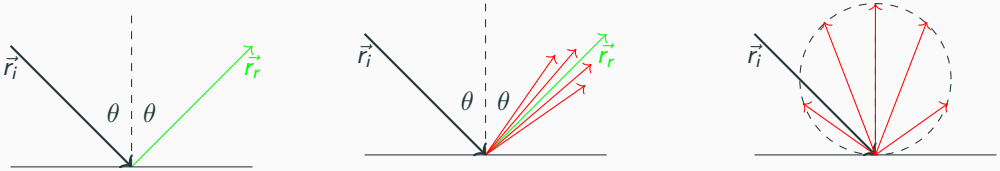
- ω_i is the direction of the incoming light and ω_r is the direction of the emitted light.
- E_i is the quantity of incident light, and L_r the quantity of reflected light.

$\rightarrow$ Computation of the BRDF depends on the material's physical properties, which can be complex. We will use idealized material models to do so.

Specular and diffuse reflection

Incoming light on an opaque surface can be reflected in two ways:

- **Specular** reflection is the light reflected at the same angle to the surface normal as the incident light.
- **Diffuse** reflection is the light scattered in all directions.



(a) Purely specular reflection (mirror surface)

(b) Diffuse reflection (glossy surface)

(c) Purely diffuse / Lambertian reflection (matte surface)

Figure 10: Examples of specular or diffuse reflection

Lambertian reflectance

At least for the beginning of this project, we will use the **Lambertian reflectance** model, which corresponds to a perfectly matte material (as in figure 10.c).

- First, we affect a color $c_S = (r, g, b)$ to the surface, corresponding to the quantity of transmitted (ie. not absorbed) light by the material. We have $r, g, b \in [0, 1]$.
- Moreover, the incoming light ray is defined by its incidence angle θ and intensity $I_i \in \mathbb{R}^3$.
- The **total** reflected brightness on each color component will be:

$$I_r = I_i \odot c_S . cos\,\theta \tag{24}$$

... where $\odot$ is the element-wise (or Hadamard) product, and θ the angle between the normal and the *new* ray $\vec{r_i}$ (the closest θ is to 0, the more light per surface area the object receives).

Generalization to the rendering equation

So far, we have seen how the BRDF allows us to estimate the intensity of the light traveling a given path.
However, in the BRDF equation (23), $dL_r(\omega_r)/dE_i(\omega_i)$ is the contribution of the light coming from ω_i to the light emitted at ω_r (note the use of differentials).

$\rightarrow$ We need to take into account all the possible incoming ω_i, as well as the light that might be emitted by the object, to get the total light emitted at ω_r.

Generalization to the rendering equation

All this is described by the **rendering equation**, which is typically written as:

$$L_r(x, \omega_r, \lambda, t) = \overbrace{L_e(x, \omega_r, \lambda, t)}^{\text{Emitted light}} + \underbrace{\int_\Omega f_r(x, \omega_i, \omega_r, \lambda, t) L_i(x, \omega_i, \lambda, t) cos\,\theta_i \; d\omega_i}_{\text{Light reflected from all incoming } \omega_i} \tag{25}$$

... where:

- L_o is the outgoing light, L_e the light emitted by the surface, f_r the BRDF and L_i the incoming light.
- x is the light impact point, with Ω the hemisphere centered around x containing all possible ω_i.
- λ is the wavelength (ie. color) and t the time.

Generalization to the rendering equation

All you really need to know about the rendering equation is that it is very difficult to solve in practice:

- Indeed, any object A emits light, which contributes to the light received and re-emitted by any object B, which itself contributes to other objects (including A!), etc...
- Mathematically, this makes the rendering equation an integral equation of the *second kind* (the function we're looking for is itself in the integral)...

We will give up on solving it analytically in the general case, and will instead make use of **Monte Carlo** methods...

Introduction to Monte Carlo methods

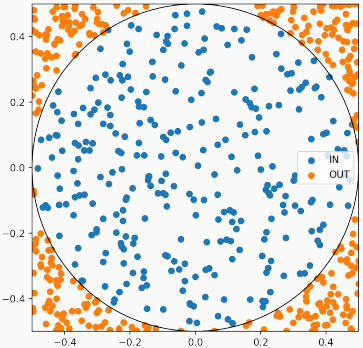
Definition

Monte Carlo methods are a very broad class of techniques:

- The general idea is to exploit randomness to solve otherwise deterministic problems.
- These methods typically use the following reasoning:
  1. Define the problem's input domain.
  2. Draw random samples from this domain.
  3. Perform deterministic calculations on those samples to obtain an approximate solution.
- By the law of large numbers, and if our algorithm is unbiased, the more samples we draw, the closer we get to the exact solution! This is a really strong property of Monte Carlo methods.

Introductory example: estimation of π

The classic problem to introduce Monte Carlo methods is the approximation of π:



- Let a square with sides of length $l = 1$, and its inscribed circle of radius $r = \frac{1}{2}$ and of center $(0, 0)$
- We will randomly scatter n points $\{(x_i, y_i)\}_{i=1}^{n}$
  - (x_i, y_i) is in the circle if $x_i^2 + y_i^2 \leq r^2$
- Let n_{IN} the number of points inside the circle from our sample distribution.

Figure 11: First 200 Monte Carlo samples for the estimation of π

Introductory example: estimation of π

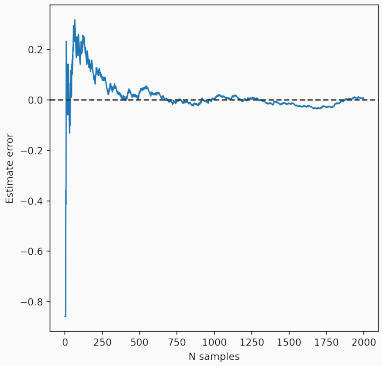


Figure 12: Evolution of estimation error for increasing number of samples: our solution converges to the exact value of π

- The intuition is that the ratio between the areas should be equal to the ratio between the numbers of IN and total samples.
- For a large number of samples:

$$\lim_{n\to\infty} \frac{n_{IN}}{n} = \frac{\pi r^2}{l^2} \tag{26}$$

$$\Leftrightarrow \lim_{n\to\infty} \frac{4n_{IN}}{n} = \pi \tag{27}$$

- On the left figure, we plot our estimation's error versus the number of drawn samples, which does converge.

Path tracing: combining ray tracing and Monte Carlo sampling

Principle of path tracing

The idea of **path tracing** is to use Monte Carlo sampling to cast rays from each pixel of the camera and to generate the bounces randomly:

- For individual rays, it is easy to compute the BRDF.
- The more rays (or samples) we compute for each pixel, the closer we get to the exact solution of the rendering equation.

Because this algorithm is unbiased, it converges to the rendering equation's solution and can generate extremely realistic images.

- It also requires very little physics to do so and naturally generates effects such as diffuse lighting and ambient occlusion. This is not the case with classic ray tracing methods.

(However, it is computationally intensive because many samples may be required.)

Random rays generation for Lambertian BRDF

When an incoming ray $\vec{r_i}$ hits a Lambertian material, it will be reflected in the hemisphere defined by the surface's normal $\vec{n}$ at the intersection point p (see left figure).

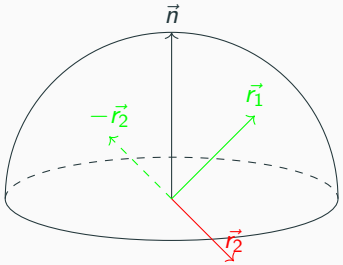


Figure 13: Generation of random rays in a hemisphere defined by its normal vector $\vec{n}$: $\vec{r_1}$ is directly accepted, $\vec{r_2}$ will need a direction change.

- The generation of random reflected rays $\vec{r_i}$ is fairly straightforward. We start by generating a fully random ray:

$$\vec{r_r} = \begin{pmatrix} rand() \\ rand() \\ rand() \end{pmatrix} \qquad (28)$$

Random rays generation for Lambertian BRDF

When an incoming ray $\vec{r_i}$ hits a Lambertian material, it will be reflected in the hemisphere defined by the surface's normal $\vec{n}$ at the intersection point p (see left figure).

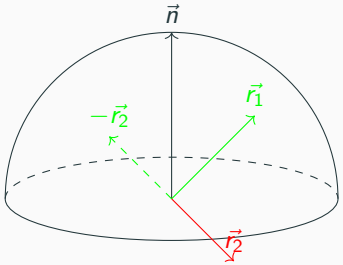


Figure 13: Generation of random rays in a hemisphere defined by its normal vector $\vec{n}$: $\vec{r_1}$ is directly accepted, $\vec{r_2}$ will need a direction change.

- To ensure the generated ray $\vec{r_r}$ is indeed in the correct hemisphere, we compute $\vec{r_r}.\vec{n}$:
  - If $\vec{r_r}.\vec{n} \geq 0$, $\vec{r_r}$ is already in the hemisphere.
  - If $\vec{r_r}.\vec{n} < 0$, $\vec{r_r}$ is in the opposite hemisphere. We will simply use $-\vec{r_r}$.
- In either case, we make sure to normalize $\vec{r_r}$.

Recursive ray sampling

The recursive path-tracing algorithm to compute a single sample with Lambertian materials is:

Algorithm 2: Path-tracing algorithm for a single sample

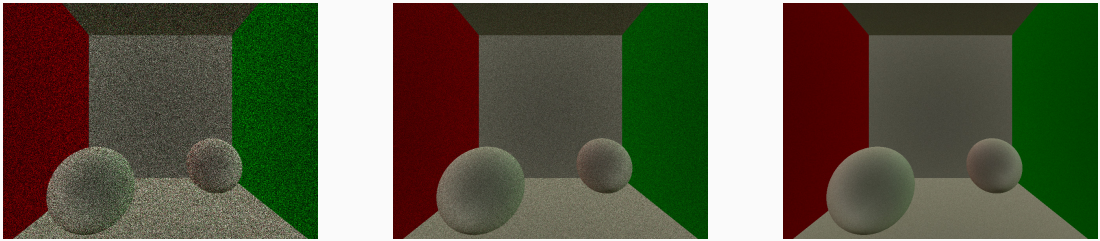
Input: Eye e, ray $\vec{r}$, scene $S = \{O_i\}_{i=1}^{n}$ (array of objects), current depth d, max depth d_{max}
Result: ray color c

if $d = d_{max}$ **then**
  return BLACK
end
$O, i \leftarrow$ getFirstCollision$(e, \vec{r}, S)$
if $i = \varnothing$ **then**
  return BLACK
end
$\vec{r}_{new} \leftarrow$ randomRay(O, i)
$c_l \leftarrow$ traceRay$(i, \vec{r}_{new}, S, d + 1, d_{max})$; `// Recursive call`
$BRDF \leftarrow \frac{O.reflectance}{\pi} cos\,\theta$; `// We divide the BRDF by` π `to get the fraction of the total light reflected to newRay`
$c \leftarrow c_i.BRDF.\frac{1}{2\pi} + O_{emittance}$; `// Total color` c `corresponds to the reflected + emitted light. 1/2`π `corresponds to the ray probability.`
return c

... where we would then average the colors of each sample that is not completely black.

Path tracing: putting it all together

We render the same scene as in figure 9, with increasing numbers of samples per pixel (ppx). Notice how "noise" decreases accordingly:



(a) 10 samples ppx (b) 100 samples ppx (c) 1000 samples ppx

Figure 14: Path tracing renders with increasing numbers of samples

$\rightarrow$ Still far from perfect, but we have successfully implemented a complete path tracing renderer!

Steps for implementation

Steps for implementation

Once you got familiar with the theory, here are the steps to follow for the implementation described in this document:

- Implement a camera system that can compute rays
- Implement ray-spheres and ray-boxes intersection detection, as well as first collision detection
- Validate a first ray casting renderer on simple scenes

From there:

- Implement BRDF with Lambertian reflectance
- Implement the recursive path tracing algorithm
- Generalize the ray casting renderer to path tracing

Steps for implementation

At first, you might struggle to obtain good-looking results. **To easily improve your renderer**, you can have a look at stuff like:

- Sky lighting
- Gamma correction
- Other shapes (infinite floors, ...)

During the second semester, we will have multiple improvement opportunities to explore:

- Ray tracing methods are **embarassingly parallel**!
- To improve convergence, there are methods such as:
  - bidirectional path tracing
  - Metropolis light transport
- Alternative material types
  - Glossy and specular reflections
  - BSDF (for transparent material)