

Computer Graphics – MOS 8.5
École Centrale Lyon

Nicolas Bonneel
`nicolas.bonneel@liris.cnrs.fr`
<https://perso.liris.cnrs.fr/nicolas.bonneel/>

Contents

1	Preamble	5
1.1	Preamble of the preamble	5
1.2	Image Representation	6
1.3	Vector Image Representation	6
1.4	A Vector Class	7
1.5	A Triangle Mesh Class	8
2	Rendering	9
2.1	Real-Time Rendering	9
2.1.1	Projection and homogeneous coordinates	9
2.1.2	Rasterization	10
2.1.3	The Z-Buffer	11
2.1.4	Coloring pixels	11
2.1.5	OpenGL and DirectX	11
2.1.6	Advanced effects	12
2.2	Physically-Based Rendering	13
2.2.1	Raytracing / Path-Tracing	13
2.2.2	Photon Mapping	54
2.2.3	Precomputed Radiance Transfer	54
2.2.4	Radiosity	57
2.3	Discussion	59

This class will cover several aspects of computer graphics, and is geared towards rendering. You will need to implement labs, a path tracer, mostly **from scratch** in C++. Little code will be provided: the course will need to be fully understood. I do consider that nothing is fully understood if you cannot implement it from scratch, and conversely, once understood, coding is merely a matter of touch typing. In return, you will get the satisfaction of having implemented your own tools producing beautiful computer graphics results.


Chapter 1

Preamble

This chapter gives an overview of what is considered common knowledge (although you may not have formally learnt it), and prerequisite for the rest of the course. As labs will be implemented in C++, typical C++ prototypes are given.

1.1 Preamble of the preamble


This class will require you to code. It is **largely** advised that you do not write any line of code before you are 100% sure the line is correct. The most time consuming aspect of programming is often debugging, and you should strive to minimize this amount of time.

 In a program, random lines of code have close to 0% chances of working, but near 100% chances of you needing to spend time trying to find them and fix them. You'll be better off not writing them in the first place.

This is particularly true for what we will implement: when implementing a path-tracer, code errors such as mistakes in probability density functions (or even basic vector math operators) can go unnoticed for some time before producing noticeable artifacts and they will thus become hard to track down.

However, bugs happen. Make sure you master a real debugger, with an IDE and ways to quickly step through the code execution (setting break points, stepping inside/over lines of code, inspecting variable values including arrays, structure members, array of structures etc.). I will use Visual Studio for that purpose, but other debuggers exist (and I would not recommend small tools such as *gdb* if used directly in the command line – the goal is to be efficient).

While our code will not result in state-of-the-art performances, we will still try to avoid large performance bottlenecks and maintain good code practices regarding performance when this only results in minor efforts in code writing. For instance, this involves avoiding unneeded square root computations, passing const reference parameters instead of entire objects, or using simple parallelization instructions. I will most often give running times (obtained on a good desktop computer) and code length for your to check if you have done anything stupid in the code (e.g., if you get a 100x slow down or a code 3x as long), to see the impact of design choices on running times and to compare different approaches (e.g., realtime OpenGL vs. slow path tracing). I also believe that code length is a good metric to see if an algorithm is worthwhile. Note that highly tuned code with clever algorithmic tricks would be orders of magnitude faster.

 I occasionally see students compiling without optimization flags and complaining about speed. Do not forget to turn on optimization ! In GCC, use `-O3` ; on Visual Studio, use the Release mode.

Regarding libraries, from an educational perspective I will strive to minimize the number of libraries used in this course. Of course, in a professional setting, you would probably use a library such as *Embree* to compute intersections quickly rather than the code you developed during this course. However, a few functionalities are much less interesting to code, and I will thus recommend libraries or give pieces of codes for a few functionalities. Notably, I will recommend the C++ header only `stb_image` and `stb_image_write` libraries (<https://github.com/nothings/stb>) to read and write images, and I will provide code to read `.obj` mesh files. Unless you want to go further (e.g., adding a GUI), you will not need other codes.

1.2 Image Representation

For this course, we will consider that an image is a 2d array of pixels, each pixel being a triplet of red (R), green (G) and blue (B) values. For implementation purpose, we will consider all rows of the image stored consecutively (row major ordering), interleaving R, G and B values. A typical C/C++ implementation with 0-based array indexing would access coordinate (x, y) in the image using:

```
1 image[y*W*3 + x*3 + 0] = red_value;
2 image[y*W*3 + x*3 + 1] = green_value;
3 image[y*W*3 + x*3 + 2] = blue_value;
```

with `red_value`, `green_value`, `blue_value` between 0 and 255.

Note that other representations are commonly encountered. For instance, a camera sensor stores a file where pixels are interleaved in a Bayer pattern. Certain applications require multispectral images consisting of multiple (>3) sampled wavelengths (e.g., additional infra-red channels) or including transparency (an additional *alpha* channel).

A template code to write an image is provided at <https://pastebin.com/dSCKUD9B>.

1.3 Vector Image Representation

A vector image is an image defined by parametric shapes: lines, circles, squares etc., with parametric rendering types (e.g., gradients). Vector images can support animation. The `.svg` file format is a simple text file format that describes vector graphics.

The idea of the `.svg` file format is to describe shapes using shape commands in an xml-like fashion. A rectangle can be obtained using:

```
<rect width="10" height="10" x="0" y="0" fill="blue" />
```

a line is represented by:

```
<line x1="0" y1="0" x2="1" y2="1" stroke="red" />
```

while a general (closed) polygon is described by pairs of coordinates for each vertex:

```
<polygon points="0,0 10,0 7,10 3,10" />
```

Objects can be grouped using the `<g> ... </g>` pair. All parameters can be animated.

1.4 A Vector Class

While it is a bad practice in software engineering, we will consider everything that has 3 floating point coordinates as a **Vector**. It is considered a bad practice since it violates several software design rules (e.g., allowing cross products between mathematical vectors is ok, but not between colors, points, etc. ; similarly, adding two vectors or a point and a vector is fine but not two points). Still, in practice, it has become widespread in computer graphics to consider a single **Vector** class, to the point that it is the standard for programming languages designed for graphics cards such as GLSL or HLSL. These languages implement classes such as **vec3** or **float3** that contain 3 floating point values that can be accessed either via **.x**, **.y**, and **.z**, or via **.r**, **.g** and **.b** (or even **.s**, **.t** and **.p** when referring to texture coordinates). *In general, this course will take shortcuts to quickly implement prototypes and will not be a reference for software design !*

A typical (partial) example of such a **Vector** class is provided below. You will need to fully implement it, including operations such as dot products, cross products, vector normalization and norm, multiplication by a scalar etc.

```

1 class Vector {
2 public:
3     explicit Vector(double x = 0., double y = 0., double z = 0.){
4         coords[0] = x;
5         coords[1] = y;
6         coords[2] = z;
7     };
8     Vector& operator+=(const Vector& b) {
9         coords[0] += b[0];
10        coords[1] += b[1];
11        coords[2] += b[2];
12        return *this;
13    }
14    const double& operator[](int i) const { return coords[i]; }
15    double& operator[](int i) { return coords[i]; }
16
17 private:
18     double coords[3];
19 };
20 Vector operator+(const Vector& a, const Vector &b) {
21     return Vector(a[0] + b[0], a[1] + b[1], a[2] + b[2]);
22 }
23 double dot(const Vector& a, const Vector& b) {
24     return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
25 }

```

The **explicit** keyword indicates the **Vector**'s constructor cannot be called from implicit conversions. For instance, the code :

```

1 Vector myVector1(1., 2., 3.);
2 Vector result = myVector1 + 1.;

```

would otherwise produce the **Vector** **result** = (2., 2., 3), resulting from the implicit conversion of the real value 1. to a **Vector** by an implicit call to **Vector(1.)**. This is prone to bugs, and **explicit** prevents that from happening.

1.5 A Triangle Mesh Class

This course will mostly manipulate triangle meshes, as they are widely used and efficient (for instance, they are natively supported by your graphics card!). These meshes consist of a set of vertices, and triplets of vertices are connected together to form triangles. The most common structure to store meshes consists in an array of vertices, and an array of triangle faces referencing these vertices. Often, additional information is stored per vertex (e.g., a color, UV coordinates, normals etc. as we shall see later).

The most common implementation of a triangle mesh consists of an array of `Vector`, and an array of triplets of indices referring to the previous array. As in most cases other geometric information is stored as well (typically, at least a normal vector per vertex, but also UV coordinates that we will discuss later), we will consider multiple arrays as in the example below:

```

1 struct TriangleIndices {
2     int vtxindices [3];    // refers to 3 indices in the vertices array of the class ←
                           Mesh
3     int normalindices [3]; // refers to 3 indices in the normal array of the class Mesh
4     int uvindices [3];    // refers to 3 indices in the uv array of the class Mesh
5 };
6 class Mesh {
7 public:
8     // ...
9 private:
10    std::vector<Vector> vertices;
11    std::vector<Vector> normals;
12    std::vector<Vector> uvs;
13    std::vector<TriangleIndices> triangles;
14 };

```

The .obj file format implements this structure. It is a file in text mode. Each line starting with a `v` defines a vertex coordinate (e.g., `v 1.0 3.14 0.00`), and each line starting with an `f` defines a face (most often a triangle, but it also supports more general polygonal faces – e.g., `f 1 2 3` defines a triangle consisting of the first 3 vertices, as indexing starts at 1). Negative indices correspond to offsets relative to the end of the vertex list. Normal vectors start with a `vn`, and uv coordinates with `vt`. The general syntax to define a triangle that has normal and uv coordinates is `f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3`. I uploaded a (poorly coded) obj file reader at <https://pastebin.com/CAgp9r15>.

Chapter 2

Rendering

Two main approaches to rendering have been adopted, focusing either on producing images at fast framerate for realtime applications (video games, simulators, fast previews of complex scenes, visualization, augmented reality etc.) or on producing images that are realistic (mostly for the movies industry) or even physically accurate (lighting simulation for architecture, car paint and light design etc.). We will briefly discuss real-time rendering (Sec. 2.1), and cover physically-based rendering in more depth (Sec. 2.2). From an implementation point of view, realtime rendering systems based on OpenGL or DirectX are often more time-consuming to produce results of similar quality as methods tailored for physically correct results ; however, they allow to framerates that are difficult to achieve with physically correct methods (Fig. 2.1). To remediate implementation issues, most realtime 3d applications are based on complex rendering engines, such as Unity, Unreal Engine, or Amazon Lumberyard (or CryEngine), that make development much faster.

2.1 Real-Time Rendering

Modern real-time rendering systems rely on the rasterization of triangle meshes and the use of Graphics Processing Units (GPUs). These triangles are thus projected on the screen, and shaded according to their materials and the various light sources in the scene. This section briefly describes the process, and will be skimmed over in class.

2.1.1 Projection and homogeneous coordinates

Given 3d coordinates of the vertices of a triangle, these coordinates are first projected on the 2d screen using:

$$p' = PVMp$$

where M is the *model* matrix that represents the 3d transformation applied to the 3d mesh itself, V the view matrix that represents the inverse of the camera transformation (typically, if the camera translates to the right, it is equivalent to translating the point to the left!), and P is a projection matrix that mainly depends on the field of view.

As you notice, these are linear transforms, and you may wonder how this could ever produce a perspective projection or even a translation (which is an affine transform, not purely linear)... In fact, we work with homogeneous coordinates that handle projective geometry ! In this context, our point p typically consists of coordinates $(x, y, z, w = 1)$. The model and view matrices are 4×4 matrices

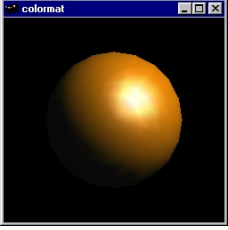
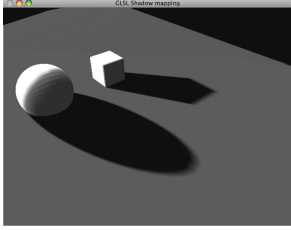

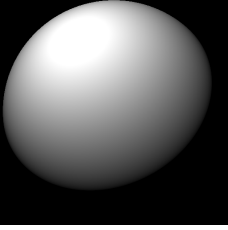
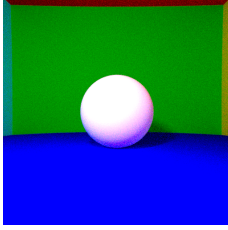

	Basic	Intermediate	Advanced
OpenGL	 <p>~ 60 (deprecated) to 4000 lines of C++ Realtime (< millisecond)</p>	 <p>~ 600 (deprecated) lines Realtime (< millisecond)</p>	 <p>~ 18k lines Realtime (milliseconds)</p>
Path tracing	 <p>~ 120 lines ~ Realtime (~ 4 ms)</p>	 <p>~ 300 lines 2 seconds</p>	 <p>~900 lines 4 min.</p>

Figure 2.1: Basic OpenGL rendering can be obtained in a few lines of deprecated OpenGL (reference implementation in about 60 useful lines of code here: <https://www.opengl.org/archives/resources/code/samples/redbook/colormat.c> using `glutSolidSphere`) or thousands of non-deprecated lines of code (e.g., see here for a typical implementation using shaders: http://www.songho.ca/opengl/gl_sphere.html). Intermediate OpenGL rendering that features soft shadows can be performed in about 600 lines of code with deprecated features <http://fabiansanglard.net/shadowmappingPCF/>. Advanced OpenGL rendering with indirect lighting quickly becomes difficult to implement (example implementation in 18k lines of c++ code: <https://github.com/djbozkosz/Light-Propagation-Volumes>). Note that OpenGL 3.1 (March, 2009) has removed many features that have been deprecated in OpenGL 3.0 (August, 2008), making codes significantly longer and also explaining variations in the above code lengths. The images shown for path-tracing are those obtained from the code developed in this class, and even includes indirect lighting and participating media (*advanced* rendering). Also, sacrificing readability can always shorten code. A small path tracer that produces something along the *Intermediate* result in term of complexity but *Advanced* in term of lighting simulation can be achieved in 99 lines of code here <https://www.kevinbeason.com/smallpt/>.

that now allow for translating the geometry via their fourth column. Similarly, the fourth component w gets transformed to w' via the fourth row of the 4×4 matrix P . The projection on the screen is performed by simply considering the projected point $p'' = [\frac{p'_x}{p'_w}, \frac{p'_y}{p'_w}]$ and its corresponding depth $\frac{p'_z}{p'_w}$ is used to determine which parts are visible or occluded.

2.1.2 Rasterization

Once 2d screen projections are known for the three vertices of a triangle, it remains to fill pixels inside this 2d triangle. This is often performed by computing an axis-aligned bounding box around the triangle, and testing the center of all pixels within this box if they belong to this triangle (we will implement such as test in 3d for raytracing 3d meshes in Sec. 2.2.1 using barycentric coordinates). This also allows to interpolate quantities stored at the vertices of the triangle such as normals, colors or the depth value that will be used next.

2.1.3 The Z-Buffer

When two 2d triangles overlap, their depth should be used to determine which one is in front of the other. An approximate solution is to use what is known as the *painter's algorithm* : 3d triangles are simply sorted by the distance between their center and the camera before being rasterized. However, this technique can be costly for large scenes, and cannot handle all cases (triangle A partly in front of triangle B, which is partly in front of triangle C, which is partly in front of triangle A). For this reason, Wolfgang Straßer, Ivan Sutherland (Turing award) and Ed Catmull (founder of Pixar and former president of Disney Animation Studios) independently described in 1974 a technique called the *Z-Buffer*. This technique simply stores the depth of each pixel being rasterized in a buffer called z-buffer. If the depth of the current pixel being rasterized is further than the depth already stored in the z-buffer for this pixel, the pixel is simply ignored.

2.1.4 Coloring pixels

A relevant color should be assigned to the pixel being rasterized. This color depends on the material of the object, the orientation of the surface (its normal), and the various light sources. The precise illumination model that is routinely used will be described later in the course in the context of raytracing, and consists in the Blinn-Phong BRDF (Sec. 2.2.1).

2.1.5 OpenGL and DirectX

The operations described previously are automatically performed by graphics libraries such as OpenGL or DirectX. These libraries take as input triangle meshes seen as arrays of vertices and polygons as well as transformation matrices (camera transformation, model transformation, projection matrix), and perform rasterization on the GPU.

Typical OpenGL < 3.0 code to display a triangle looks like:

```

1 glBegin(GL_TRIANGLES);
2 glNormal3f(0.f, 0.f, -1.f);
3 glColor4f(1.f, 0.f, 0.f, 1.f);
4 glVertex3f(0.f, 1.f, 0.f);
5 glColor4f(0.f, 1.f, 0.f, 1.f);
6 glVertex3f(-1.f, 0.f, 0.f);
7 glColor4f(0.f, 0.f, 1.f, 1.f);
8 glVertex3f(1.f, 0.f, 0.f);
9 glEnd();

```

while in OpenGL ≥ 3.0 , a similar triangle would look like:

```

1 // define geometry
2 GLfloat positions[3*3] = {0.f, 1.f, 0.f, -1.f, 0.f, 0.f, 1.f, 0.f, 0.f};
3 GLfloat normals[3*3] = {0.f, 0.f, -1.f, 0.f, 0.f, -1.f, 0.f, 0.f, -1.f};
4 GLfloat colors[4*3] = {1.f, 0.f, 0.f, 1.f, 0.f, 1.f, 0.f, 1.f, 0.f, 0.f, 1.f, 1.f};
5 GLuint indices[3] = {0, 1, 2};
6
7 // generates a vertex array with 3 buffers for position, normals and colors
8 GLuint vertexArray, vertexBufferObject[3];
9 glGenVertexArrays(1, &vertexArray);
10 glBindVertexArray(vertexArray);
11 glGenBuffers(3, vertexBufferObject);
12
13 // associate the arrays on the CPU to attributes stored on the GPU

```

```

14 // glBindBufferData uploads the array to the GPU
15 glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject[0]);
16 glBindBufferData(GL_ARRAY_BUFFER, 9*sizeof(GLfloat), positions, GL_STATIC_DRAW);
17 glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, 0);
18 glEnableVertexAttribArray(0);
19
20 glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject[1]);
21 glBindBufferData(GL_ARRAY_BUFFER, 9*sizeof(GLfloat), normals, GL_STATIC_DRAW);
22 glVertexAttribPointer((GLuint)1, 3, GL_FLOAT, GL_FALSE, 0, 0);
23 glEnableVertexAttribArray(1);
24
25 glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject[2]);
26 glBindBufferData(GL_ARRAY_BUFFER, 12*sizeof(GLfloat), colors, GL_STATIC_DRAW);
27 glVertexAttribPointer((GLuint)2, 4, GL_FLOAT, GL_FALSE, 0, 0);
28 glEnableVertexAttribArray(2);
29
30 // define a triangle as 3 vertex indices
31 GLuint elementbuffer;
32 glGenBuffers(1, &elementbuffer);
33 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
34 glBindBufferData(GL_ELEMENT_ARRAY_BUFFER, 3 * sizeof(unsigned int), &indices[0], ←
    GL_STATIC_DRAW);
35
36 // draw this triangle
37 glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, (void*)0);

```

Note that an entire mesh would be handled by creating a larger array of triangles (and not by repeating this routine for all triangles!). In both examples, large parts would be missing to see something, including definition of lights, cameras and materials. However, an important missing routine in the second example is the definition of *shaders* that actually perform the computations (projecting the vertices on screen in a *vertex shader* or *vertex program*, and computing the pixel color based on the lighting in the scene in a *pixel shader* or *fragment program*) – by default, OpenGL < 3.0 has these things already handled.

These functions remain extremely simple and limited: they merely allow you to draw triangles on screen, very efficiently. But most of the job remains to be done: making these triangles look realistic. This is most often accomplished via a number of tricks, which we will give a few examples below.

2.1.6 Advanced effects

Hard shadows from a spot light. The first thing that is obviously missing when displaying triangles on screen is cast shadows. The most common way to cast shadows in a rasterizer is to compute a *shadow map*. A shadow map is a view of the scene from the light point of view, and only storing depth information (you do not need to keep or even compute color values). This depth image is precomputed for each light source (Fig. 2.2). Then, when computing the color value of a given pixel, it becomes easy to reproject this pixel on the shadow map via transformation matrices, and check whether the light arriving from the light source is obstructed by comparing the distance between the shaded point and the light source, and the value stored in the shadow map.

Indirect lighting. A simple solution to obtain indirect lighting¹ consists of the *Instant Radiosity* technique. This technique, despite its name, has nothing to do with the usual radiosity algorithm (see Sec. 2.2.4): it consists in rendering (both colors and z-buffer) the scene from the light sources, and then placing new secondary light sources (often called *Virtual Point Lights*) in the scene whose color match the rendered image. Clustering can be performed in realtime to group all nearby pixels of the same color in the image rendered from the light source to construct a single secondary light source for each cluster of pixels.

¹see Sec. 2.2.1 for more details on indirect lighting in the context of raytracing

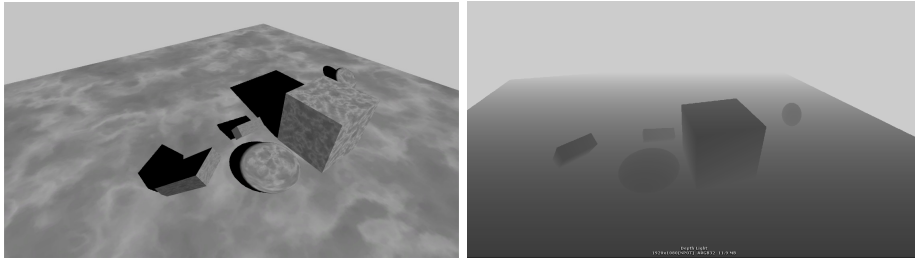


Figure 2.2: Realtime rendering with shadows (left) obtained by a shadowmap (right) that corresponds to the z-buffer of the scene as seen from the light source.

2.2 Physically-Based Rendering

This section covers basics of physically-based rendering to the point that you should be able to implement a path tracer (while it will not work at the speed of production engines, it would give close to production level quality), and have minimal knowledge of other techniques.

2.2.1 Raytracing / Path-Tracing

Path-tracing works by launching rays of light from a virtual camera throughout the scene, computing ray/scene intersections, evaluating light contributions from light sources and making these light rays bounce off the objects. While this approach works counter-intuitively to real-world physics (in which light rays are emitted from light sources rather than the camera!), it can be shown to be strictly equivalent due to Helmholtz reciprocity principle: what only counts is the set of light paths joining the camera sensor and light sources. In fact, an approach called bidirectional path tracing benefits both from rays emitted from the camera and rays emitted from light sources to construct these light paths.

Rendering basic spheres

We will first write a small program that renders and shade a few spheres with direct lighting. First, “launching rays” from the camera to the scene corresponds to generating half-lines (rays) which originate at the camera location and towards each pixel of the camera sensor, and computing the point of intersection of these half-lines with the scene (i.e., the spheres).

The ingredients thus are:

1. Defining classes and operators for handling geometric computations
2. Defining a scene
3. Computing the direction of rays
4. Computing the intersection between a ray and a sphere
5. Computing the intersection between a ray and the scene
6. Computing the color

1 Classes Regarding operators, we will define classes for **Vector** (see Sec. 1.4), **Sphere** (a center **Vector** C and a **double** radius R ; we will also add a color, called **albedo**, stored in a **Vector** as an RGB triple $\in [0, 1]^3$), **Ray** (an origin **Vector** O and a unit direction **Vector** u), and **Scene** (an array/std::vector of **Spheres**). A **Sphere** will further possess a function **intersect** that computes

the point of intersection between a **Ray** and the sphere, if any (at this stage, we can either return a **bool** indicating whether an intersection occurred and pass the relevant intersection information as parameters passed by reference ; or we can return an **Intersection** structure that contains all the relevant information including a **bool** flag). A **Scene** will also possess a similar function.

2 Scene For reproducibility purpose, we can define a standard scene as in Fig. 2.3, that we will use throughout this course. To simplify the introduction, we will first focus on the center sphere. Also for simplicity, we consider the camera if standing upright and looking at the $-z$ direction.

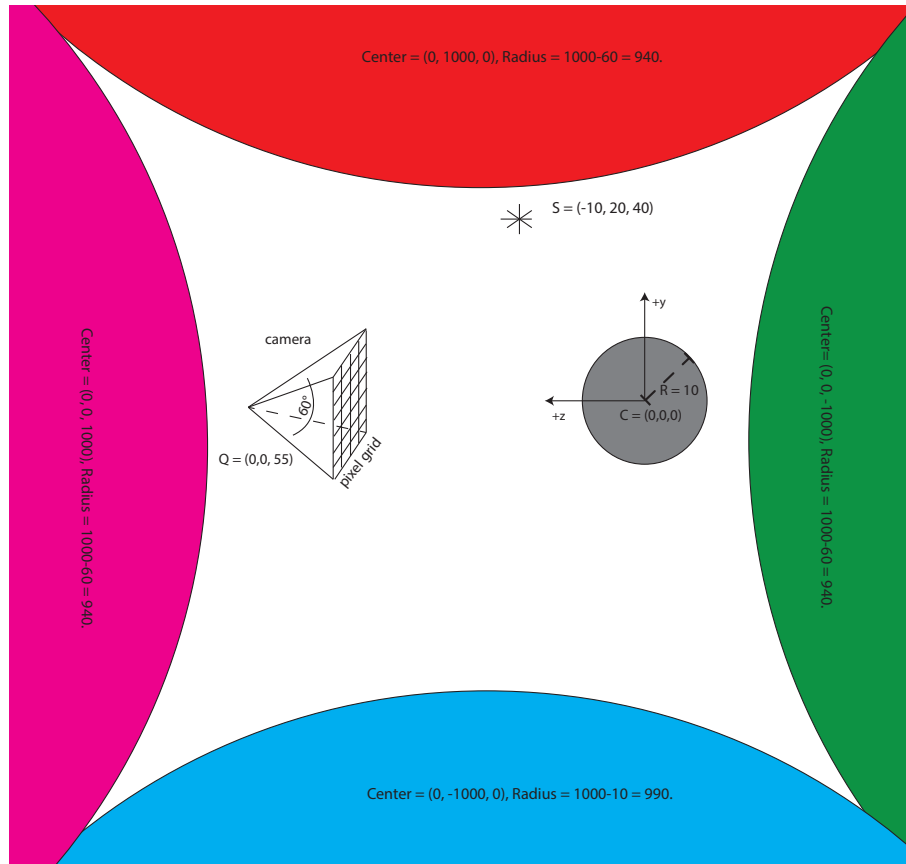


Figure 2.3: We define a standard scene that consists in walls, a ground and a ceiling, all consisting of gigantic spheres approximating planes. We also add a center sphere, which we will focus on as a first step.

3 Computing the direction of rays Our camera consists of a center Q and a virtual plane that makes the screen (or similarly the sensor, if you see our camera as a pinhole, see Sec 2.2.1), see Fig. 2.4. Assuming the screen is at a distance f from the camera center $Q = (Q_x, Q_y, Q_z)$, we will consider that, in our configuration, pixel (x, y) is located at coordinate $(Q_x + x + 0.5 - W/2, Q_y + y + 0.5 - H/2, Q_z - f)$. However, one usually only knows α , the visual angle covering the W pixels in width (called horizontal field of view, or fov), not f . Simple calculus shows that $\tan(\alpha/2) = (W/2)/f$ such that pixels are located at coordinates $(Q_x + x + 0.5 - W/2, Q_y + y + 0.5 - H/2, Q_z - W/(2 \tan(\alpha/2)))$. Note that in our pixel grid, we will index pixels by their row and column number (i, j) . Since image rows are most often stored from top to bottom, this corresponds to using $(x, y) = (j, H - i - 1)$, with $i \in \{0..H - 1\}$ and $j \in \{0..W - 1\}$. From the coordinate of each pixel and the camera center, we can simply compute a normalized ray direction.

4 Ray-Sphere intersection A parametric equation of a ray of origin O and direction u is $X(t) = O + tu$, with $t > 0$. A implicit equation of a sphere centered at C and radius R is $\|X - C\|^2 = R^2$. A point of intersection P , if any, would satisfy both equations. Plugging the first equation into the second yields $\|O + tu - C\|^2 = R^2$. Expanding the squared norm and using scalar product bilinearity yields $t^2\|u\|^2 + 2t\langle u, O - C \rangle + \|O - C\|^2 = R^2$. Assuming unit norm for u leads to the simple quadratic

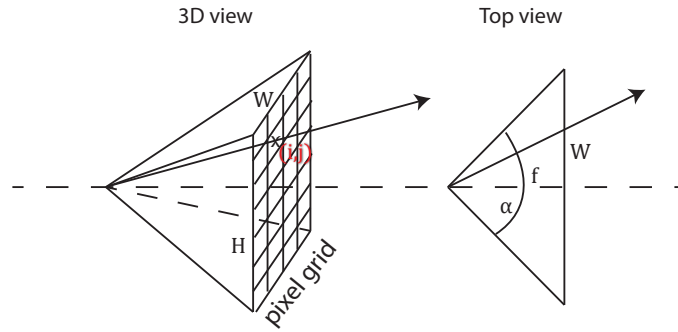


Figure 2.4: Notations for a virtual camera.

equation:

$$t^2 + 2t \langle u, O - C \rangle + \|O - C\|^2 - R^2 = 0$$

A quadratic equation has 0, 1 or 2 real solutions depending on the discriminant, which has geometric interpretations here (see Fig. 2.5). Denoting $\Delta = \langle u, O - C \rangle^2 - (\|O - C\|^2 - R^2)$ the reduced discriminant, no intersection between the line (not the ray) is found if $\Delta < 0$, one (double) intersection is found if $\Delta = 0$ and two are found if $\Delta > 0$. However, one needs to further check that the solution parameter t is non-negative, since otherwise the intersection would occur *behind* the ray origin. Further, in the context of ray-tracing, only the *first* non-negative intersection is of interest, i.e., the (positive) intersection closest to the ray origin. If $\Delta \geq 0$, the two possible intersection parameters are $t_1 = \langle u, C - O \rangle - \sqrt{\Delta}$ and $t_2 = \langle u, C - O \rangle + \sqrt{\Delta}$. If $t_2 < 0$, the ray does not intersect the sphere. Otherwise, if $t_1 \geq 0$, $t = t_1$ else $t = t_2$. The intersection point P is located at $P = O + t u$. For further lighting computation, we will also need to retrieve the unit normal N at P . It can be simply obtained using $N = \frac{P - C}{\|P - C\|}$. We are now ready to produce a first image, by scanning all pixels in the pixel grid, throwing rays, and testing if there is any intersection. If any intersection is found, just setting the pixel white results in Fig. 2.6 (considering only the central sphere of our standard scene in Fig. 2.3).

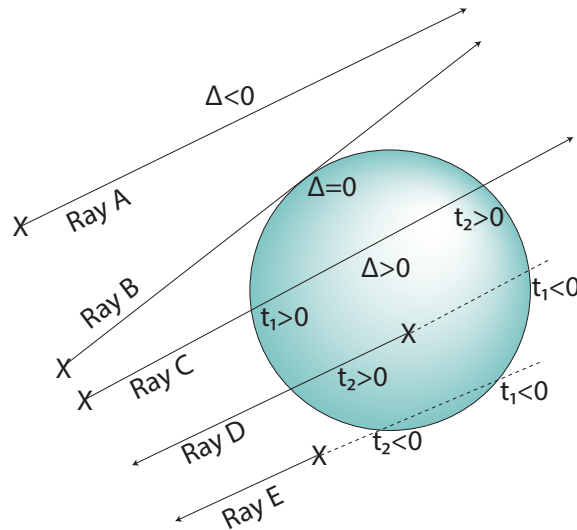


Figure 2.5: Ray-Sphere intersections lead to solving a quadratic equation. Depending on the sign of the discriminant, this leads to either 0, 1 or 2 points of intersection. Here, ray B leads to one (double) intersection, ray C produces a first intersection of interest at t_1 , and ray D produces the intersection of interest at t_2 (the other intersection being behind).

5 Ray-Scene intersection Our scene is composed of multiple spheres (for now). The intersection we are interested in, between a ray and the scene, is the ray-sphere intersection that is closest to the ray origin among all (if any). It can also be useful to return the specific sphere or object ID that has been hit to retrieve object-specific properties, such as material parameters.

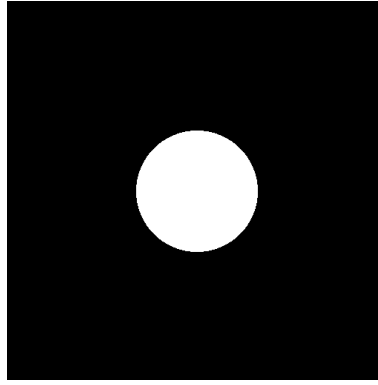


Figure 2.6: Computing the ray-sphere intersection at each pixel leads to our first image. Ok, that’s just a plain white disk, don’t be *too* excited.

6 Shading and shadows computation For now, we will use a simple material model: the Lambertian model. This model assumes that materials scatter light equally in all directions, regardless of incoming light direction. This well represents diffuse materials such as plaster, but will not handle shiny materials such as metals or plastics. Under this model, the intensity reflected off a surface at point P with albedo ρ and normal N , illuminated by an omnidirectional light source of intensity I at position S is given by

$$L = \frac{I}{4\pi d^2} \frac{\rho}{\pi} V_P(S) \langle N, \omega_i \rangle$$

with $\omega_i = \frac{S-P}{\|S-P\|}$, $d = \|S - P\|$. The *visibility* term $V_P(S)$ is such that $V_P(S) = 1$ if S is “visible” from P and 0 otherwise. “Visible” means that launching a ray from P with direction ω_i (towards S) will either encounter no intersection, or that an intersection exists but *further* than the light source², that is, $t > d$. The term in $\frac{I}{4\pi d^2}$ merely says that a light intensity of I Watt will be spread over a sphere surface of $4\pi d^2$, and the amount reaching point P is thus $\frac{I}{4\pi d^2}$ Watt. $sr^{-1}.m^{-1}$ (sr stands for steradian, a unit of solid angle). The term in $\frac{\rho}{\pi}$ is essentially a convention: with albedo values ρ ranging in $[0..1]$, the material respects energy conservation (see Sec. 2.2.1) if $\int_{S^+} c.\langle N, \omega_i \rangle d\omega_i \leq 1$ for some normalization constant c , where S^+ is the hemisphere above the surface. Since $\int_{S^+} \langle N, \omega_i \rangle d\omega_i = 4\pi$, $c = 1/(4\pi)$.

⚠ Due to numerical precision issues, you will certainly observe extreme noise levels (see Fig. 2.7). This is due to the fact that when launching a ray from point P towards the light source S , the first point of intersection that may be found is P itself since precision is limited. The solution to this issue is to launch the ray not from P , but from a point slightly above the surface, $P + \varepsilon N$. Since we are launching rays from a slightly elevated position, it could be that $\langle N, \omega_i \rangle < 0$ at grazing angles. For safety, we will use instead $\max(\langle N, \omega_i \rangle, 0)$.

Gamma correction. Computer screens do not react linearly with the pixel intensities they are fed with. For instance, a linear ramp from pure black to pure white results in a midpoint that seems too dark (Fig. 2.8). To compensate for this effect, we apply *gamma correction* to the images produced by our path tracer. This consists in elevating RGB values (in a normalized range $[0, 1]$) at the power $1/\gamma$, with typically $\gamma = 2.2$. One reason for the need to gamma-correct images is a more perceptually uniform image encoding. Indeed, noise, compression artifacts or quantization artifacts are often more visible on dark pixels than on bright pixels. To allow for more accuracy on darker values, the quantization is made non-uniform by storing gamma-corrected images. Additionally, (integer) pixels values should be clamped in the range $\{0..255\}$ to avoid overflowing `unsigned char` that would

²These *visibility* or *shadow* rays often benefit from faster intersection routines as the exact point of intersection is not required but merely the presence of an intersection within an interval ; feel free to do that.

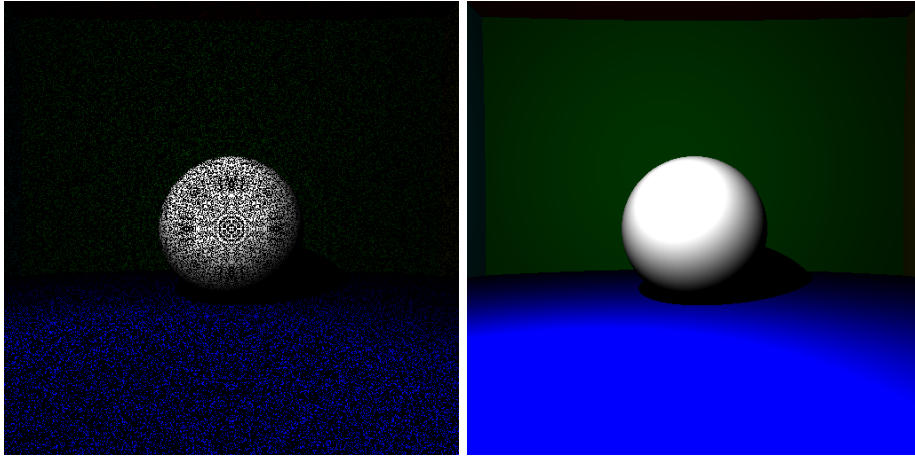


Figure 2.7: Due to numerical precision issues in shadow computations the image appears noisy (left). Launching rays from an offset origin solves this issue (right).

result in *wraparound*. You can see the result of gamma correction on our test scene in Fig. 2.9.

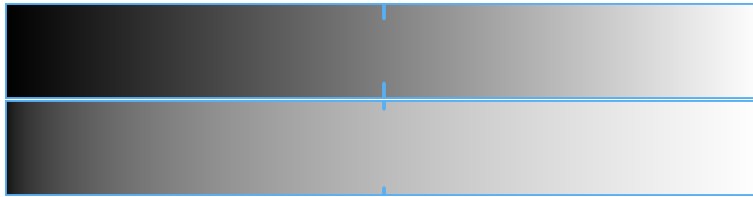


Figure 2.8: A linear ramp (top) and gamma-corrected linear ramp (bottom, with $\gamma = 2.2$). The linear ramp's midpoint appears too dark. Note that perceived results may vary depending on specific screen settings.

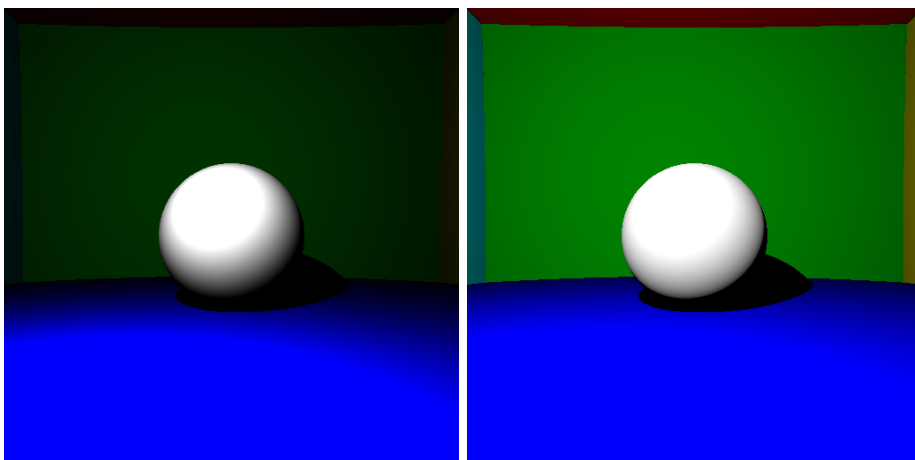


Figure 2.9: Without gamma correction, the scene appears too contrasted (here, $I = 2.10^7$). With gamma correction (and $I = 2.10^{10}$), the scene appears more natural. At this stage, we have roughly 170 lines of (verbose) code which runs in 50ms without parallelization (see end of Sec. 2.2.1) for a 512x512 image.

⚠ A common bug is to gamma-correct or clamp intensity values for all bounces of the light, which is not correct. This typically results in lack of contrasts. These operations compensate for specific image formats. For instance, High Dynamic Range (HDR) formats such as .exr, .pfm or .hdr do not need gamma correction, as this step is usually performed by the image viewer. As such, these should be the very last steps to be performed only once, right before saving the image to disk, and should not be involved in the light simulation process.

Adding reflections and refractions

Reflections. Contrary to Lambertian surfaces that scatter light in all directions, (purely) reflective/specular surfaces only reflect light in a single direction. It is easy to see that the direction ω_r reflected from an incident direction ω_i off a surface with normal N is $\omega_r = \omega_i - 2\langle\omega_i, N\rangle N$ (see Fig. 2.10). A perfect mirror thus only transfers light energy from the incident direction to the reflected direction.

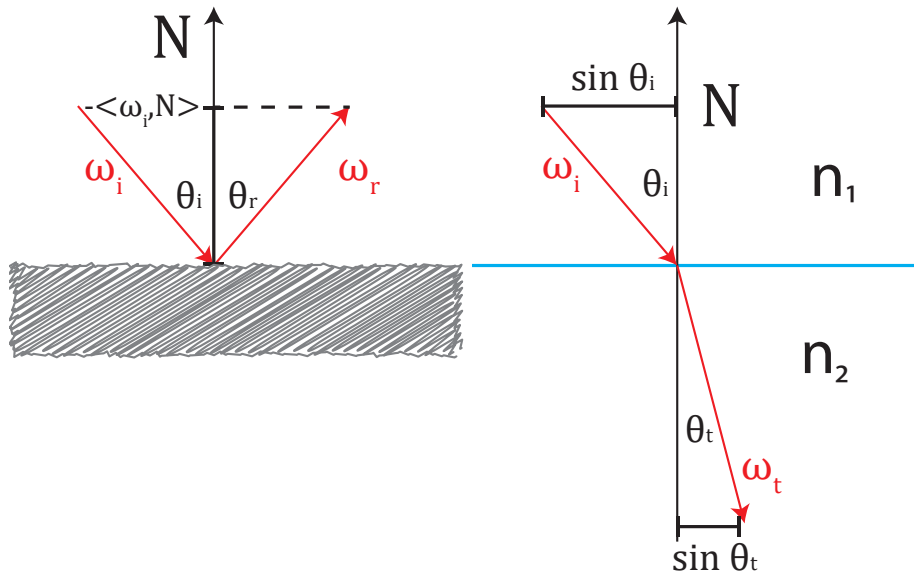


Figure 2.10: The reflected direction is $\omega_r = \omega_i - 2\langle\omega_i, N\rangle N$ (left) and refracted direction (right).

⚠ A common bug is to compute a visibility term (or shadows) on top of the reflected light. You should not do it. Visibility is a shadowing term that refers to specific light sources. A reflective surface will not see our point light sources (they are infinitesimally small) and light sources will not play any role here. Mirrors reflect light coming from all directions, not just that of our point light sources.

In term of implementation, handling reflections will add one of the most important brick of our path tracer. Reflective surfaces lead to recursive code: to compute the light arriving towards the camera sensor, you need to know the amount of light arriving at P from the reflected direction ω_r . But the light coming from this reflected direction could be the result of another mirror reflecting light from elsewhere (and so on). As such, you will now build your first *path* throughout the scene. A typical recursive implementation/pseudo-code would look like:

```

1
2 Vector Scene::getColor(const Ray& ray, int ray_depth) {
3     if (ray_depth < 0) return Vector(0., 0., 0.); // terminates recursion at some ←
        point

```

```

4
5  if (intersect(ray, P, N, sphere_id)) {
6      if (spheres[sphere_id].mirror) {
7          Ray reflected_ray = ...;
8          return getColor(reflected_ray, ray_depth-1);
9      } else {
10         // handle diffuse surfaces
11     }
12 }
13
14 }
15
16 int main() {
17 // first define the scene, variables, ...
18 // then scan all pixels
19 for (int i=0; i<H; i++) {
20     for (int j=0; j<W; j++) {
21         Ray ray(...); // cast a ray from the camera center to pixel i, j
22         Vector color = scene.getColor(ray, max_path_length);
23         pixel[i*W*3+j*3 + 0] = std::min(255, std::pow(color[0], 1./2.2)); // stores R ←
24             channel
25         // same for green and blue
26     }
27 }
28 // save image and return 0
29 }

```

Note that similarly to cast shadows, you need to offset the starting point of the reflected ray off the surface to avoid numerical issues. This will also be the case later for transparent surfaces, indirect lighting etc. and will not be repeated any further.

Refractions. The case of transparent surfaces is very similar to that of mirrors. For transparent objects, rays also continue their lives by bouncing off the surface, but this time, passing through it. The computation of the refracted direction is however slightly more involved. For that, we assume the Snell-Descartes law, written here as $n_1 \sin \theta_i = n_2 \sin \theta_t$. This law essentially says that the tangential component of the transmitted ray ($\sin \theta_t$) is stretched from that of the incoming ray ($\sin \theta_i$) by a factor n_1/n_2 . Decomposing the transmitted direction ω_t in tangential and normal components $\omega_t = \omega_t^T + \omega_t^N$, it is easy to deduce that

$$\omega_t^T = \frac{n_1}{n_2}(\omega_i - \langle \omega_i, N \rangle N)$$

where we have used the fact that the tangential component of ω_i is ω_i minus its normal component (its projection on N).

Regarding the normal component, we have $\omega_t^N = -N \cos \theta_t$ (considering the normal N is pointing towards the incoming ray). This amounts to $\omega_t^N = -N \sqrt{1 - \sin^2 \theta_t}$. And since we have the Snell-Descartes law, this equals: $\omega_t^N = -N \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 \sin^2 \theta_i} = -N \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 (1 - \cos^2 \theta_i)}$. The cosine can be computed by projecting on the normal N , so:

$$\omega_t^N = -N \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 (1 - \langle \omega_i, N \rangle^2)}$$

From this equation, one can see that if $1 - \left(\frac{n_1}{n_2}\right)^2 (1 - \langle \omega_i, N \rangle^2)$ becomes negative, the square root would lead to imaginary results... This can only occur if $n_1 > n_2$. This corresponds to a total internal reflection, and occurs if $\sin \theta_i > \frac{n_2}{n_1}$.

⚠ During the computations, we made sure the normal N was pointing towards the incoming ray. This is typically the case when the ray enters a sphere. However, when the ray exits the sphere, the geometric normal returned by our intersection test has the wrong sign. Make sure to use the correct refraction indices and normal sign in this case! You can detect the case of a ray exiting the transparent sphere when $\langle \omega_i, N \rangle > 0$. Also, make sure to offset the starting point of your refracted ray... on the correct side! In general, for refraction, beware of signs.

A trick to simulate hollow spheres is to make two spheres of the same center and slightly different radii, and then inverting the normals of the inside sphere. A result showing reflection and refraction on a full and hollow sphere is shown in Fig. 2.11. Also, ideally, the index of refraction should depend on the wavelength. To achieve dispersion, we would throw rays of a single wavelength, combining them on the sensor ; we will not do that here.

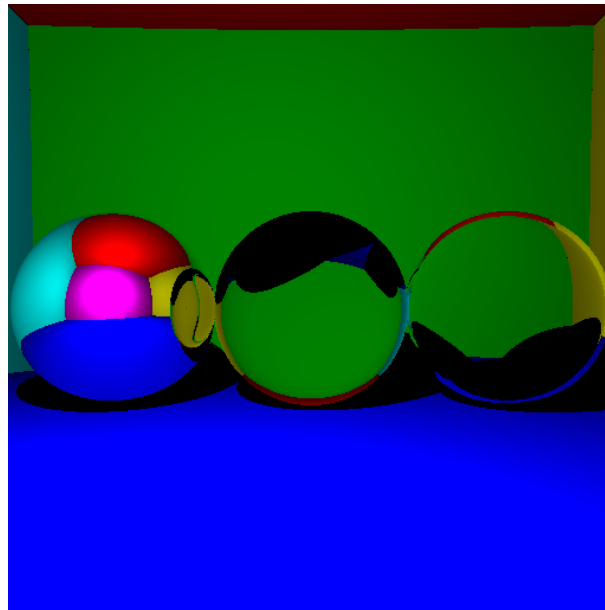


Figure 2.11: A sphere with reflection, a full sphere with refraction, and an hollow sphere with refraction. Notice how the full sphere inverts the scene behind as it acts as a lens. The refraction index used is 1.5, corresponding to glass. The image is computed in 75ms (without parallelization) with about 230 lines of code.

Fresnel law. Both the coefficient of reflection and transmission are fully determined by the refraction indices n_1 and n_2 , via Fresnel equations. In practice, these equations are relatively costly to evaluate, and one often rely on Schlick's approximation of Fresnel coefficients. For dielectrics, this reads:

$$k_0 = (n_1 - n_2)^2 / (n_1 + n_2)^2$$

$$R = k_0 + (1 - k_0)(1 - |\langle N, \omega_i \rangle|)^5$$

$$T = 1 - R$$

where k_0 is the reflection coefficient at normal incidence, R is the reflection coefficient for incidence ω_i , and T the transmission coefficient. An option *could* be to call our function `Scene::getColor` twice, once for the reflected ray and once for the refracted ray, and modulate the two resulting colors with the reflection and transmission coefficients, and summing them. However, this would double the number of rays in the scene for *each* light bounce. Instead, we will randomly launch either a reflection ray, or a refraction ray. For that, we find a (uniform) random number u between 0 and 1, and launch a reflection ray if $u < R$ and a refraction ray otherwise. We then do not need to rescale the resulting value. Of course, this would result in an extremely noise image since adjacent pixels will get assigned different random numbers. As such, we will launch multiple rays for each pixel, resulting in multiple

paths, and average the resulting color. This scheme will be further discussed along with Monte Carlo integration next, in Sec. 2.2.1.

⚠ To avoid noisy images, you need to average the result of multiple paths. It is extremely important that for each light bounce in the scene, a *single* call to `Scene::getColor` is performed. To make it clearer: you launch K rays from the camera center C to the same pixel (i, j) , then for each light bounce of these rays you send (at most) one secondary ray (for reflection, transmission, or indirect lighting as we will see next). This results in K paths throughout the scene, resulting in K different colors. You then average these K colors to obtain the pixel value. Never recursively call `Scene::getColor` more than once: this would result in impractically too many secondary rays.

Note: you can similarly handle multiple point light sources by adding the contribution of just one randomly chosen light source and averaging different realizations, rather than adding all contributions. This becomes interesting when one can weigh this randomness by the intensity or distance of light sources. We will see a similar approach next, to handle indirect lighting.

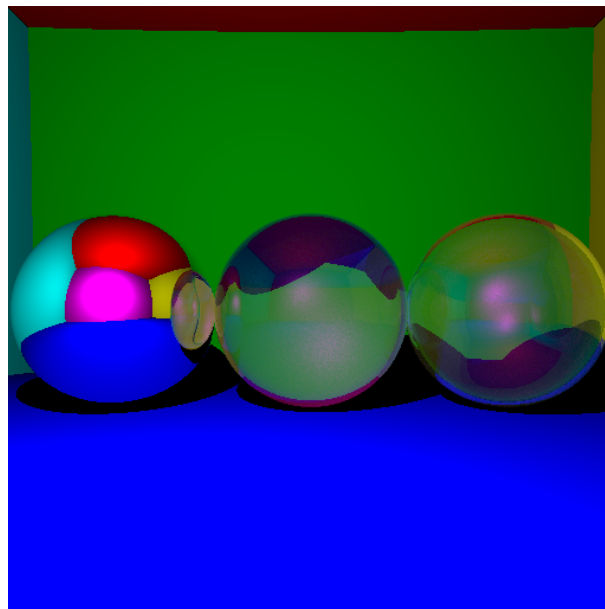


Figure 2.12: Same as Fig. 2.11 but with Fresnel reflection taken into account on transparent surfaces. For this image, I took 1000 rays per pixel, which resulted in a rendering that took about 1 minute (without parallelization, and about 260 lines of code).

Adding indirect lighting

Indirect lighting is an extremely important factor to realism. To my knowledge, it has first been introduced in a physically correct manner (at least via Virtual Point Lights, as opposed to artists manually tuning light sources) in *Pirates of the Caribbean 2* (2006) with the Renderman renderer. Indirect lighting is the reason why the ceiling in your classroom does not appear black, although no (direct) light sources are illuminating it (Fig. 2.13). Simulating indirect lighting is probably one of the most difficult aspect of rendering, and will require several ingredients: understanding the *rendering equation*, understanding *Monte Carlo integration*, and implementing good *importance sampling* strategies.

The Rendering Equation. The equation that gives the outgoing *spectral radiance* (i.e., the



Figure 2.13: Classroom illuminated only via direct lighting (left), and direct+indirect lighting (right). Notice the overly dark ceiling on the left. Model from <https://www.blendswap.com/blend/15639>.

result of `Scene::getColor`) is:

$$L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_{\Omega} f(x, \omega_i, \omega_o, \lambda, t) L_i(x, \omega_i, \lambda, t) \langle \omega_i, N \rangle d\omega_i \quad (2.1)$$

This equation simply says that your `Scene::getColor` function depends on the point x in the scene (in our case, it is evaluated at intersection points P), the (opposite of the) ray direction $-\omega_o$, the light wavelength λ (in our case, we merely render R , G and B channels) and a time parameter t . It results in the sum of the emitted light L_e at x in the direction ω_o (and wavelength λ and time t) and the contribution of all light reflected at point x . The light reflected at x is simply the sum of all *incoming* light contributions L_i from the hemisphere Ω falling on x , modulated by a function f that is called *Bidirectional Reflectance Distribution Function* or *BRDF*, which describes the appearance or shininess of materials, and a dot product/cosine function that accounts for light sources projected area (a small area light at grazing angle will see its contribution smeared over a large area). Notations can be seen in Fig. 2.14.

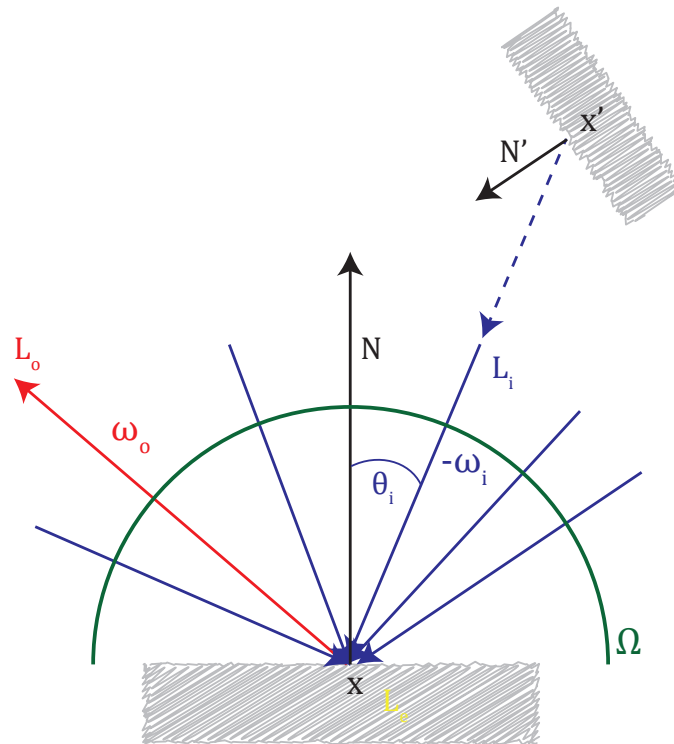


Figure 2.14: Notations for the Rendering Equation. Note that from now on, we denote by convention ω_i a vector that points **outwards** the surface, like ω_o . Since this mostly influence dot product signs, this is usually understood from context.

It is interesting to see that the incoming light at point x from direction ω_i is exactly the outgoing light at a point x' from direction $-\omega_i$, assuming a vacuum medium (we will see in Sec. 2.2.1 how to handle *participating media*). As such, using the rendering equation at point x' (and ignoring spectral and temporal variables for conciseness ; we will also occasionally ignore position variables when the context is clear enough in the future), we could rewrite Eq. 2.1 at point x as

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega} f(x, \omega_i, \omega_o) \left(L_e(x, \omega_o) + \int_{\Omega'} f(x', \omega'_i, -\omega_i) L_i(x', \omega'_i) \langle \omega'_i, N' \rangle d\omega'_i \right) \langle \omega_i, N \rangle d\omega_i$$

and recursively, the lighting reaching point x' comes from other locations in the scene and so on. This type of recursive integral equation is called a **Fredholm integral of the second kind**, as, in fact, there is a single unknown radiance function L to be determined, that is both outside and inside the integral.

This results in an integration over an infinite dimensional domain, called *Path Space* that represents a sum of light paths with $0, 1, 2, \dots, \infty$ bounces, that needs to be performed numerically.

Bidirection Reflectance Distribution Functions (BRDFs). An important function in the rendering equation above is the term f , the BRDF. This term describes the amount of light being reflected off a surface towards a direction ω_o if it arrives from a direction ω_i (Fig. 2.15). Conditions for their physical meaningfulness are that they are positive ($f \geq 0$), they respect Helmholtz reciprocity principle, that is, they are symmetric ($f(\omega_i, \omega_o) = f(\omega_o, \omega_i)$)³ and preserve energy, that is $\int_{\omega} f(\omega_i, \omega_o) \langle \omega_i, N \rangle d\omega_i \leq 1, \forall \omega_o$ ⁴

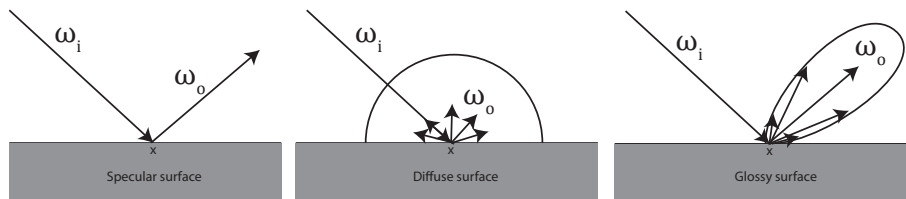


Figure 2.15: Typical BRDFs.

These BRDFs can be provided as tabulated functions, for instance coming from *gonioreflectometers* that are physical devices that measure reflected light off surfaces at different angular values. Notable databases of BRDFs include MERL 100 isotropic BRDF dataset⁵ (see Fig. 2.16 ; note that isotropic BRDFs can be reparameterized using only 3 dimensions, $\theta_i, \theta_r, \phi_d$ instead of 4 angular values $\theta_i, \phi_i, \theta_r, \phi_r$ – a parameterization called Rusinkiewicz parameterization), Ngan’s 4 anisotropic BRDFs⁶, and UTIA 150 anisotropic BRDFs⁷. These tabulated values can be heavy to store and manipulate, and can further be compressed, for instance by projecting them on *spherical harmonics*. Applications of these spherical harmonic projected BRDFs will be discussed in the context of Precomputed Radiance Transfer in Sec. 2.2.3.

BRDFs can also be described via closed-form expressions, that can either be ad-hoc (also coined as “phenomenological” for political correctness, but they are all more or less Gaussian lobes around the purely specular direction – we will see the Blinn-Phong BRDF model in Sec. 2.2.1) – or derived from microgeometry analysis assuming microfacet models (e.g., Cook-Torrance, Oren-Nayar, Torrance-Sparrow, Ashikhmin-Shirley, He et al., ...).

For now, we have seen and will focus on three particular cases: $f_r(\omega_i, \omega_o) = \delta_{\omega_r}(\omega_o)$ with ω_r the reflection of ω_i around the normal N as we have seen in Sec. 2.2.1, $f_t(\omega_i, \omega_o) = \delta_{\omega_t}(\omega_o)$ with ω_t the

³This is not *always* the case, though most often. Notably, for transparent surface, $f(\omega_i, \omega_o) = \left(\frac{n_2}{n_1}\right)^2 f(\omega_o, \omega_i)$

⁴This can be derived from the fact that $\int_{\omega} \int_{\omega'} L_i(\omega_i) f(\omega_i, \omega_o) \langle \omega_i, N \rangle d\omega_i d\omega_o \leq 1, \forall L_i$.

⁵<https://www.merl.com/brdf/>

⁶<https://people.csail.mit.edu/addy/research/brdf/>

⁷http://btf.utia.cas.cz/?brdf_dat_dwn

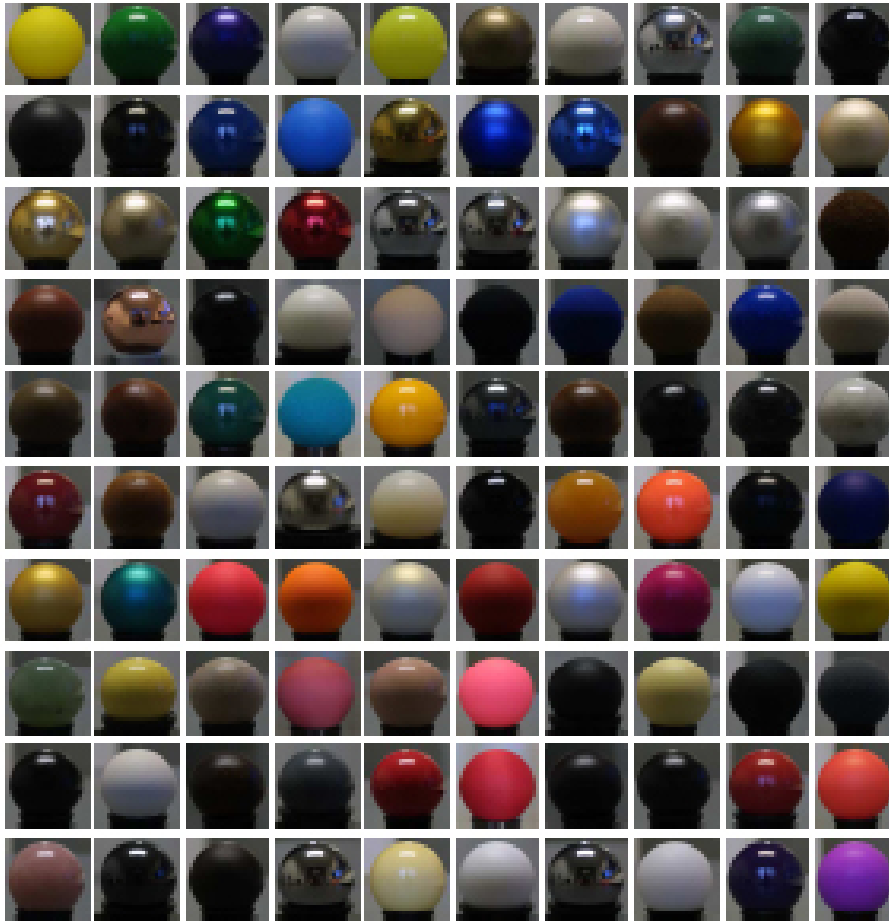


Figure 2.16: BRDFs from the MERL dataset.

transmission of ω_i inside the surface of normal N as we have seen in Sec. 2.2.1, and $f_d(\omega_i, \omega_o) = \frac{c}{\pi}$ the diffuse BRDF as in Sec. 2.2.1. Note that f_r and f_t involve Dirac distributions, and Eq. 2.1 should thus be (re-)interpreted in the sense of distributions. We will see later in Sec. 2.2.1 how to implement the Blinn-Phong BRDF.

Monte Carlo integration. We need to perform numerical integration to evaluate Eq. 2.1. You have probably seen during your curriculum various ways to numerically integrate functions, such as the rectangle method (midpoint rule), trapezoidal rule, or even higher order methods such as Newton Cotes. These methods divide the integration domain in regular intervals, and consider the function is piecewise-something within these intervals. The major drawback is that regularly dividing an integration domain of dimension d (let alone an infinite dimensional space!) produces exponentially many intervals, such that even dividing in 10 intervals each dimension of a 4-d domain would result in 10^4 intervals (remember that this integration needs to be performed for possibly millions of pixels, that in practice, we often need more than 4 dimensions, and that 10 intervals per dimension would likely miss important high frequency features).

To alleviate this issue, Monte Carlo integration has been proposed as a way to stochastically evaluate integrals. This technique has been historically developed in the context of the Manhattan project for nuclear simulation and is now widely used in computer graphics, but also mainly in economics, nuclear physics and medical imaging. It is simply expressed in general term as:

$$\int_{\Omega} f(x)dx \approx \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x_i)}$$

where x_i are random samples following the probability density function p . This converges to the true integral assuming $p > 0$ wherever $f \neq 0$. The intuition is that if you can give a sample half the

probability of occurring, but then you need to compensate and count it twice. However, this process converges slowly: the integration error decreases in $\mathcal{O}(1/N^{0.5})$ ⁸.

A major tool to improve the integration error is *importance sampling*. Importance sampling will try to find a probability density function p that is near proportional to f . In fact, if p is exactly proportional to f , that is, $p = \alpha f$, then

$$\int_{\Omega} f(x) dx \approx \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{\alpha p(x_i)} \quad (2.2)$$

$$= \frac{1}{N} \sum_{i=1}^N \frac{1}{\alpha} \quad (2.3)$$

$$= \frac{1}{\alpha} \quad (2.4)$$

that is, the estimator would converge without any sample, in $\mathcal{O}(1)$! This is due to the definition of probability distributions: they should integrate to 1, so if they integrate to 1 and are proportional to f , then the constant of proportionality is the (inverse of the) integral. In short, if you are *able* to build an exactly proportional probability density function (pdf), then you do not need numerical integration in the first place ! However, this method is interesting if you know that your p is a good approximation of f , up to a constant (unknown) scaling factor.

Exercise. To test your understanding of Monte Carlo integration, please write a program that estimates

$$F = \int_{[-\pi/2, \pi/2]^3} \cos(x y z) dx dy dz$$

using an isotropic Gaussian probability density function f of standard deviation $\sigma = 1$ (f does not really look like a Gaussian, but gives at least more priority on values near $(0, 0, 0)$ and is sufficient for the sake of exercise – a better proxy would give higher values around each axis).

For that, we will use the `<random>` header from the STL which provides reasonably good random numbers (at least, as opposed to the `rand()` function), and we will consider the Box-Muller transform, that produces 2 Gaussian samples given 2 uniform random values:

```

1 #include <random>
2 static std::default_random_engine engine(10); // random seed = 10
3 static std::uniform_real_distribution<double> uniform(0, 1);
4
5 void boxMuller(double stdev, double &x, double &y) {
6     double r1 = uniform(engine);
7     double r2 = uniform(engine);
8     x = sqrt(-2 * log(r1)) * cos(2 * M_PI * r2) * stdev;
9     y = sqrt(-2 * log(r1)) * sin(2 * M_PI * r2) * stdev;
10 }
```

Note that this 3-dimensional Gaussian has a pdf given by $p(x, y, z) = \left(\frac{1}{\sigma\sqrt{2\pi}}\right)^3 \exp(-(x^2 + y^2 + z^2)/(2\sigma^2))$, as a joint density of 3 independent 1-dimensional Gaussian functions. The exact value is close to 24.3367. With 10000 samples, you should at least get the 24 part correct...

⁸This can be somewhat improved to $\mathcal{O}(\frac{(\log N)^d}{N})$ by using well-chosen deterministic samples that *uniformly* cover the integration domain such as the commonly used Sobol sequence or other *low-discrepancy* sequences – a technique called quasi-Monte Carlo. See *Variance Analysis for Monte Carlo Integration* <https://dl.acm.org/doi/pdf/10.1145/2766930>

⚠ The resulting code should only have **1 for loop**, and **not 3** nested loops, looping over x , y and z (like for the midpoint rule for example) ! This would otherwise entirely miss the point of Monte Carlo integration: having a code whose complexity does not depend on the dimensionality of the integrand. This remark is akin to that of Fresnel refraction: in fact, when we randomly chose between reflecting or refracting rays, we actually did Monte Carlo integration, with p being a discrete probability distribution!

Implementing indirect lighting. We are now ready to add indirect lighting to our path tracer. Realizing that we actually did implement indirect lighting already for mirror and transparent surfaces, we will consider for now that our surfaces are either purely diffuse of albedo ρ (and $L_e = 0$), or emissive (with $f = 0$). We aim a building a path sampling the path space where the light contribution is reasonably high, and at each light bounce over a diffuse surface at point x we locally evaluate the interaction and use it recursively :

$$L_o(x, \omega_o) = \frac{\rho}{\pi} \int_{\Omega} L_i(x, \omega_i) \langle \omega_i, N \rangle d\omega_i \quad (2.5)$$

To importance sample a diffuse surface, we would ideally sample the integrand $L_i(x, \omega_i) \langle \omega_i, N \rangle$. But as noted before, it is simply impossible (otherwise the problem would be already solved). A simple option is to only sample according to the second term $\langle \omega_i, N \rangle$. Assuming $N = (0., 0., 1.)$, this can be achieved by using a formula similar to Box-Muller formula:

$$r_1, r_2 \sim \mathcal{U}(0, 1) \quad (2.6)$$

$$x = \cos(2\pi r_1) \sqrt{1 - r_2} \quad (2.7)$$

$$y = \sin(2\pi r_1) \sqrt{1 - r_2} \quad (2.8)$$

$$z = \sqrt{r_2} \quad (2.9)$$

$$(2.10)$$

It is easy to see that this formula directly gives a vector of unit norm, and the pdf of these samples is $p((x, y, z)) = z/\pi$. Using a frame change formula, one can easily bring it to a frame such that the z coordinate above is aligned with our actual normal vector N . Producing a local frame around N can be achieved by first generating two orthogonal tangent vectors T_1 and T_2 . To generate T_1 , we could directly use a normalized version of the vector $(N_z, 0, -N_x)$ for example, since it is easy to see that $\langle N, T_1 \rangle = 0$ by construction. This would *often* work, until numerical issues arise near the normal vector $N = (0, 1, 0)$, which would produce a tangent vector near $T_1 = (0., 0., 0.)$. To avoid that, we detect the smallest component of N (in absolute value!), force it to be zero, swap the two other components and negate one of them to produce T_1 , which we normalize. Then T_2 is obtained by taking the cross product between N and T_1 . And given N, T_1 and T_2 , we obtain the random **Vector** in the correct frame by using $V = xT_1 + yT_2 + zN$, where (x, y, z) were generated by the formula above. We will call this function `random_cos(const Vector &N)`.

With the method above to generate random vectors, and the known pdf p , it becomes easy to perform Monte Carlo integration. You will realize that cosine terms cancel out, as well as the factor π (the term π in $\frac{\rho}{\pi}$ is cancelled by π from the pdf: $p = \frac{\langle N, \omega_i \rangle}{\pi}$ when dividing by the pdf).

Other importance sampling formulas can be found in the Global Illumination Compendium by Philip Dutré⁹.

Now, you may realize that working only with point light sources (for now) will result in strictly no rays arriving by chance on these infinitesimally small lights. To address this issue, we directly sample our point light source using the formulas we used until now resulting in the *direct lighting* contribution, and *add it* to the random contribution we are generating (call the *indirect lighting* contribution).

⁹<https://people.cs.kuleuven.be/philip.dutre/GI/TotalCompendium.pdf>

Similarly to Fresnel, if you sample one ray per pixel the resulting image will be extremely noisy due to all that randomness, but shooting many rays per pixel will make it converge to a nice and smooth image. If you have already implemented this strategy for Fresnel materials, you do not need to change anything.

Also, realize that the code you just wrote for handling indirect lighting on diffuse surfaces just looks like the code for mirror surfaces – just the reflected ray goes in a random direction instead of a deterministic mirror direction. The code should look like:

```

1 Vector Scene::getColor(const Ray& ray, int ray_depth) {
2     if (ray_depth < 0) return Vector(0., 0., 0.); // terminates recursion at some ↵
3         point
4     if (intersect(ray, P, N, sphere_id)) {
5         if (spheres[sphere_id].mirror) {
6             // handle mirror surfaces ...
7         } else {
8             // handle diffuse surfaces
9             Vector Lo(0., 0., 0.);
10            // add direct lighting
11            double visibility = ... ; // computes the visibility term by launching a ray ↵
12                towards the light source
13            Lo = light_intensity/(4*M_PI*squared_distance_light) * albedo/M_PI * ↵
14                visibility * std::max(dot(N, light_direction), 0.);
15
16            // add indirect lighting
17            Ray randomRay = ...; // randomly sample ray using random_cos
18            Lo += albedo * getColor(randomRay, ray_depth-1);
19
20            return Lo;
21        }
22    }
23 }

```

and should produce results similar to those of Fig. 2.17.

Russian Roulette. Until now, we have truncated light paths to a maximum number of bounces controlled by the initial value of `ray_depth`. This leads to a biased rendering: one can construct a scene that requires an arbitrarily high number of light bounces (for instance, take an arbitrary number of mirrors redirecting one light source to a room). We thus did not integrate over the entire infinite dimensional space of light paths, but over a truncated version of it. It is however possible to integrate over this infinite-dimensional space. Instead of killing rays after a certain number of bounces, you only kill them with some probability, and divide the light contribution by this probability. You can fine tune this probability to be proportional to the current path intensity (if the first 5 encountered albedos are very dark, it is unlikely that any future light source will be sufficiently bright to compensate light absorption, so we make a 6th bounce unlikely – but if it occurs, then we compensate this low probability by putting a large weight), but in any case, this results in an unbiased rendering. Unfortunately, this also tends to introduce significant noise (there is always a tradeoff between bias and noise), so we will not implement it here.

Parallelization. Our code starts to be relatively slow, due to the number of paths that need to be generated. An easy parallelization instruction is:

```

1 #pragma omp parallel for
2 for (int i=0; i<H; i++) {
3     // ...
4 }

```

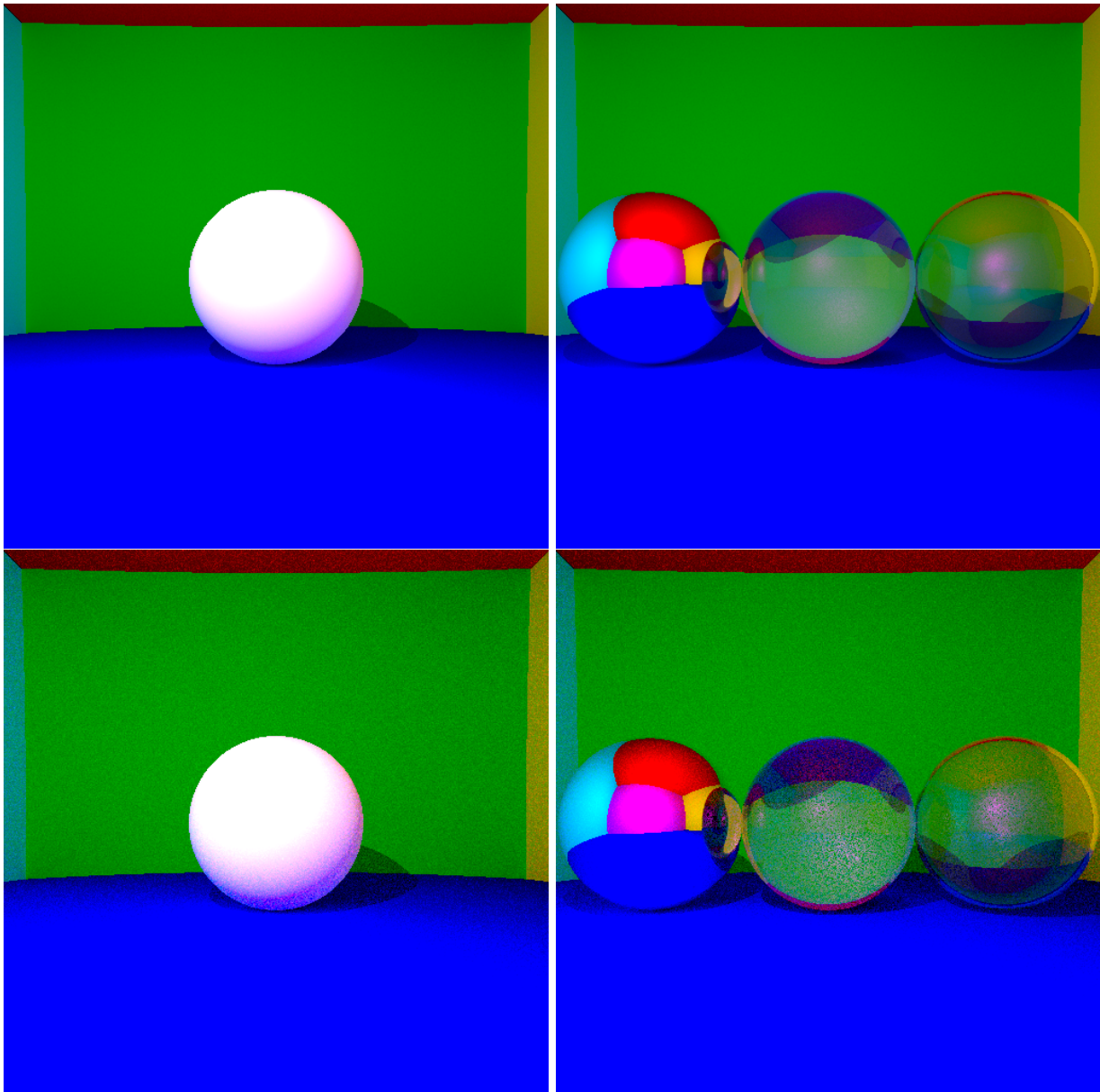


Figure 2.17: Rendering with indirect lighting (290 lines of code). First row, the renderings with either a diffuse or transparent central spheres take about 35 seconds in parallel (or 7 minutes without parallelization) using 1000 paths per pixel, and a maximum ray depth (`max_path_length` in the code below) of 5. Second row, the rendering takes 1.2 seconds (in parallel) for 32 paths per pixel.

This instructs the compiler to perform the for loop in parallel. Make sure to enable OpenMP, using Project properties -> Configurations Properties -> C/C++ -> Language -> Open MP Support with Visual Studio, or `-fopenmp` on recent GCC or `-openmp` on old GCC. Old Clang do not support OpenMP. On MacOS, you may need to link with OpenMP using `-L/usr/local/opt/libomp/lib -I/usr/local/opt/libomp/include -fopenmp -lomp`. Parallelization instructions should in general go on the outermost loop, since starting threads has an inherent non-negligible system cost. By default, the above instruction would evenly split the H lines of pixels in `OMP_NUM_THREADS` blocks (or as many as the number of cores you have), and run these blocks in parallel. This is equivalent to `#pragma omp parallel for schedule(static, ceil(H/(double)omp_get_num_threads()))` and is ideal when all rows of pixels have the same computational time. However, when this is not the case (which often occurs), threads end up waiting for other threads to finish, doing nothing. A dynamic schedule can then be used, as in `#pragma omp parallel for schedule(dynamic, 1)` which instructs OpenMP to feed threads one row as soon as it is available. Dynamic scheduling is generally more costly than static scheduling, though the scheduling cost is here negligible with respect to computation times.

⚠ The `std::default_random_engine` is not thread safe. Also, the `thread_local` directive is not compatible with OpenMP threads. You may need to instantiate one random number generator per thread.

Antialiasing

As we are always sampling rays in the middle of each pixels, there is a discontinuity between adjacent pixels: a ray may hit the sphere for a pixel and miss it in the next pixel. This results in a phenomenon called aliasing. In fact, camera sensor cells have an area, they are not points. More precisely, actual camera sensor cells are arranged in a pattern called *Bayer pattern* (Fig. 2.18). Each sensor cell is sensitive to either red, green and blue through a colored filter array, and since the eye is more sensitive to green light than red or blue, there are twice as many “green cells” (or rather grayscale cells covered with a green filter) than red or blue cells. Once a photograph is taken, the resulting raw image is then converted to an RGB pixel grid using demosaicing (or debayering) algorithms. We will not simulate Bayer patterns as we can directly emulate an RGB-sensitive pixel array.

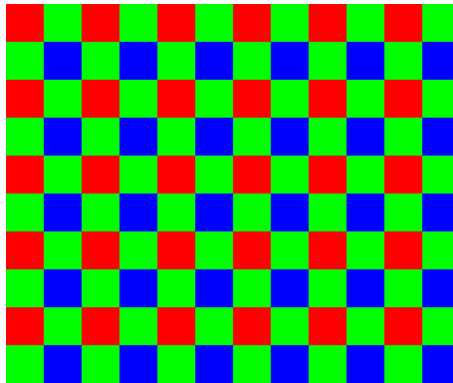


Figure 2.18: Camera sensor cells are arranged in a Bayer pattern, interleaving red, green and blue filtered sensors.

The idea here is to integrate the radiance that reaches the camera sensor over the surface of each pixel. For that, we are actually integrating:

$$L^{i,j} = \int_{A_{i,j}} L_i(x, \omega_i(x)) dx$$

where $\{i, j\}$ are the pixel indices, $A_{i,j}$ represents the surface of pixel (i, j) , and $L_i(x, \omega_i(x))$ represents the light reaching the camera sensor at point x from a direction that is fully determined by x and the camera center ($\omega_i(x) = \frac{x-C}{\|x-C\|}$). In practice, this would amount to box filtering the input radiance, which is not spectrally ideal and could still result in some amount of aliasing (notably for high frequency textures or geometries).

Instead, we would rather filter the signal more smoothly, by integrating:

$$L^{i,j} = \int_{A_{i,j}} L_i(x, \omega_i(x)) h_{i,j}(x) dx$$

where h is some nice smooth kernel. While interesting choices include Mitchell-Netravali’s filtering or windowed Sinc filters, we will simply use a Gaussian filter centered in the middle of pixel i, j for our function h . We have now seen Monte Carlo integration, and it is becoming clear that the above computation is well suited to it: we can efficiently design an importance sampling approach that produces samples more often in the middle of each pixels according to a Gaussian probability! In fact, we have already implemented Box-Muller’s technique earlier as an exercise. And while evaluating the

Monte Carlo estimate, one realize that again, the Gaussian kernel h and the pdf p exactly cancel out since we have importance sampled the integrand according to h .

Our main function now looks like:

```

1 int main() {
2 // first define the scene, variables, ...
3 // then scan all pixels
4 #pragma omp parallel for schedule(dynamic, 1)
5 for (int i=0; i<H; i++) {
6   for (int j=0; j<W; j++) {
7     Vector pixelColor(0., 0., 0.);
8     for (int k=0; k<NB_PATHS; k++) {
9       Vector rand_dir = ...; // as before but targeting pixel (i,j)+boxMuller()*spread
10      Ray ray(C, rand_dir); // cast a ray from the camera center C with rand_dir ←
          direction
11      pixelColor += scene.getColor(ray, max_path_length);
12    }
13    pixel[i*W*3+j*3 + 0] = std::min(255, std::pow(pixelColor[0]/NB_PATHS, 1./2.2)); //↔
          stores R channel
14    // same for green and blue
15  }
16 }
17 // save image and return 0
18 }

```

and produces the image in Fig. 2.19.

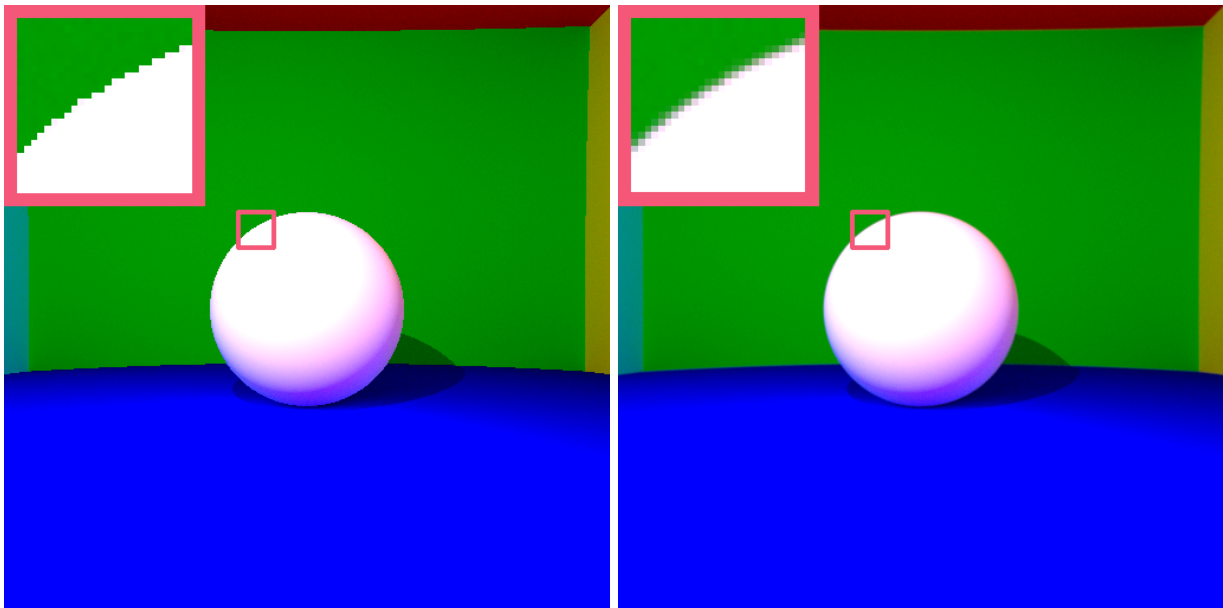


Figure 2.19: Image without (left) and with (right) antialiasing.

Remark. It is now clear that, using a Gaussian importance sampling strategy, samples for pixel (i, j) have some probability to fall *outside* of pixel (i, j) (in fact, as soon as the Box-Muller function will return one value larger than 0.5). Given the cost of retrieving $L_i(x, \omega_i(x))$, it would be a waste to only use it for pixel (i, j) and not for all the neighboring pixels (i', j') for which $h_{i', j'}$ is sufficiently large. It is indeed interesting to *splat* $L_i(x, \omega_i(x))$ over a small pixel neighborhood. However, care must be taken to avoid concurrency issues while parallelizing code. To simplify the implementation, we will not implement this technique which correlates samples received by neighboring pixel.

Spherical / area light sources

Another important factor to realism is the presence of soft shadows (Fig. 2.20). Soft shadows are the result of light sources having an area and not being points, hence resulting in penumbras. For simplicity, we will support spherical light sources (since we have primitives for them), but the method extends to other shapes.



Figure 2.20: Classroom image without (left) and with (right) soft shadows. Notice the shadow of the blackboard on the wall and tables on the ground.

A naive solution would simply to set a positive value for the emission L_e of all spherical light sources, and wait for our random rays to reach these light sources (and remove our point light source). This would theoretically work, but also produce very noisy images. In fact, the smaller the light source, the less likely light paths will randomly reach them, and the noisier the images (Fig. 2.21).

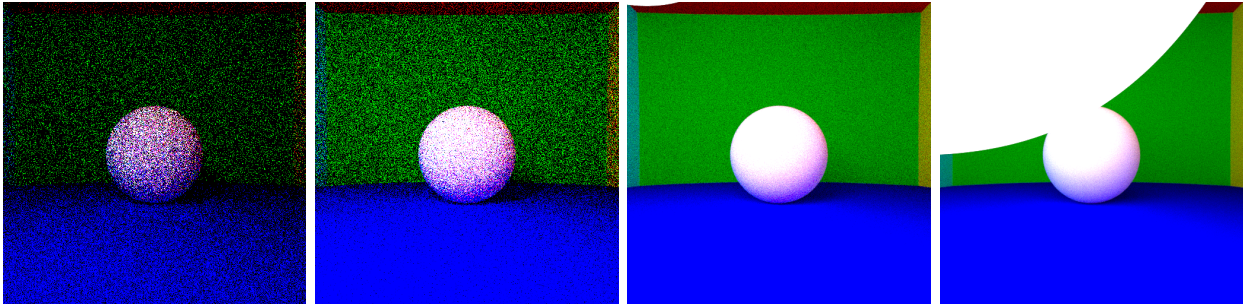


Figure 2.21: Naively handling soft shadows using spherical light sources of radius 1, 2, 10, and 20. As the radius increases, light paths have more chances to randomly reach light sources, which reduces noise. Also notice the soft shadows appearing. These renderings still have 1000 (uncorrelated) samples per pixel, which is very large for typical scenes. The rendering takes about 25 seconds (in parallel) for 280 lines of code.

Recall that for diffuse surfaces, we are looking to numerically evaluate an expression of the form:

$$L_o(x, \omega_o) = \frac{\rho}{\pi} \int_{\Omega} L_i(x, \omega_i) \langle \omega_i, N \rangle d\omega_i$$

Similarly to point light sources, we will separate direct and indirect contributions. The formalism will be made clearer here: we split the integration domain Ω in two parts: the part Ω_d (d for direct) that consists in the area of the hemisphere where spherical light sources project and the rest of the hemisphere, Ω_i (i for indirect). Ω_d is such that launching rays in a direction $\omega_i \in \Omega_d$ from x would reach a spherical light source, unless blocked by some geometry. This is akin to point light sources, where Ω_d what an infinitesimally small domain.

We hence keep our process in which we add indirect and direct lighting together. For indirect lighting, we will only make a small change to our existing code (since these rays do not directly reach light sources, they can be importance sampled according to the cosine term as we did before): if we

launch a random ray for indirect lighting contribution but it still hits a light, then we should count its contribution as zero (otherwise this value would be counted twice, once in the direct lighting computation, and once in the indirect lighting computation). We are left with implementing importance sampling for direct lighting, that is, light rays directed towards light sources.

We could use a formula for importance sampling directions within the spherical cap Ω_d . But it is easier and more general to re-parameterize the rendering equation via a change of variable for which instead of integrating over (part of) an hemisphere, we would integrate over (part of) the scene directly. This means that we would sum over small area patches in the scene rather than small solid angles (see Fig. 2.22).

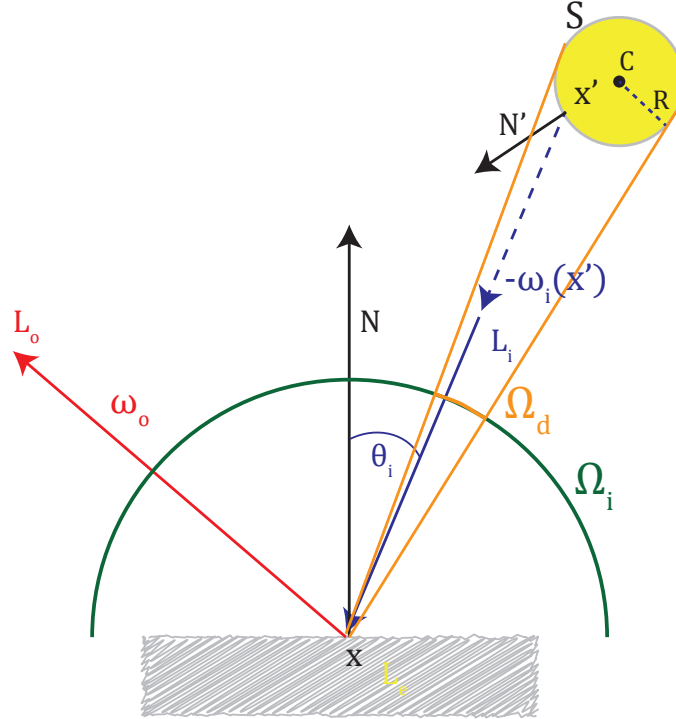


Figure 2.22: Notations for integrating over elements in the scene.

As always, when making a change of variable within an integral, one needs to account for the determinant of the Jacobian of this change of variable. It appears that this determinant is $D = \frac{\langle N', -\omega_i \rangle V_x(x')}{\|x - x'\|^2}$ where V_x is still the visibility function, and N' the normal of the area patch around point x' . The rendering equation for purely diffuse surfaces now looks like:

$$L_o(x, \omega_o) = \frac{\rho}{\pi} \int_S L_i(x, \omega_i(x')) \langle \omega_i(x'), N \rangle \frac{\langle N', -\omega_i(x') \rangle V_x(x')}{\|x - x'\|^2} dx'$$

with $\omega_i(x') = \frac{x' - x}{\|x' - x\|}$, and S the surface of our light source.

In fact, the coefficient $G(x, x') = \langle \omega_i(x'), N \rangle \frac{\langle N', -\omega_i(x') \rangle V_x(x')}{\|x - x'\|^2}$ is often called the *form factor* between x and x' . We will also use it later in the context of Radiosity (Sec. 2.2.4).

We will now seek to stochastically sample our spherical light sources in the scene (instead of directly sampling directions towards them). Given the term in $\langle N', -\omega_i(x') \rangle$, it is obvious that we should avoid sampling points on the “edge” of the spherical light, as this dot product will be close to zero, and that we would prefer sampling values for which $\langle N', -\omega_i(x') \rangle$ is large. Also, the *visibility* term V_x is such that half our spherical light sources will be occluded by the other half... so we would like to sample points only on the visible side. Fortunately, we have already written some code, `random_cos(const`

`Vector &N`), that takes a `Vector N` (that used to be our normal vector, but could be anything) and returns a random `Vector` which has more chances of being sampled around `N` than orthogonally to it. It samples them according to a probability density function $p(V) = \frac{\langle V, N \rangle}{\pi}$.

To generate a point x' on our spherical light source S of center C and radius R from a point x , we first build the vector $D = \frac{x-C}{|x-C|}$ that defines the visible hemisphere of S , we call $V = \text{random_cos}(D)$ to obtain a unit direction that has more chance of facing D , and finally obtain x' using $x' = RV + C$. The probability density function at x' is $p(x') = \frac{\langle V, D \rangle}{\pi} \cdot \frac{1}{R^2}$, where $1/R^2$ is due to the samples being *stretched* in two dimensions by a factor R .

Regarding $L_i(x, \omega_i(x'))$, we now need to spread our I Watts of light power over the surface of a sphere of radius R , with each of these point radiating in all directions of the hemisphere with a cosine factor. The number of $\text{Watts.m}^{-2}.\text{sr}^{-1}$ is thus $\frac{I}{4\pi^2 R^2}$.

The code now looks like:

```

1 Vector Scene::getColor(const Ray& ray, int ray_depth, bool last_bounce_diffuse) {
2     if (ray_depth < 0) return Vector(0., 0., 0.); // terminates recursion at some ←
        point
3
4     if (intersect(ray, P, N, sphere_id)) {
5         if (spheres[sphere_id].is_light) {
6             if (last_bounce_diffuse) { // if this is an indirect diffuse bounce
7                 // if we hit a light source by chance via an indirect diffuse bounce, return ←
                    0 to avoid counting it twice
8                 return Vector(0., 0., 0.);
9             } else {
10                return Vector(1., 1., 1.) * light_intensity / (4 * M_PI * M_PI * R * R); // R is the ←
                    spherical light radius
11            }
12        }
13        if (spheres[sphere_id].is_diffuse) {
14            // handle diffuse surfaces
15            Vector Lo(0., 0., 0.);
16            // add direct lighting
17            Vector xprime = random_point_on_light_sphere();
18            Vector Nprime = (xprime - centerLight) / (xprime - centerLight).norm();
19            Vector omega_i = (xprime - P) / (xprime - P).norm();
20            double visibility = ...; // computes the visibility term by launching a ray ←
                    of direction omega_i
21            double pdf = dot(Nprime, (x - centerLight) / (x - centerLight).norm()) / (M_PI * R * R);
22            Lo = light_intensity / (4 * M_PI * M_PI * R * R) * albedo / M_PI * visibility * std::max(←
                    dot(N, omega_i), 0.) * std::max(dot(Nprime, -omega_i), 0.) / ((xprime - P).←
                    squared_norm() * pdf);
23
24            // add indirect lighting
25            Ray randomRay = ...; // randomly sample ray using random_cos
26            Lo += albedo * getColor(randomRay, ray_depth - 1);
27
28            return Lo;
29        }
30    }
31 }

```

Note the similarity of this approach to an approach that would consider the scene to have a single point light whose position is not deterministic but stochastically sampled on the surface of a sphere. This code can simulate simple caustics (Fig. 2.24).

⚠ Always replace in your code $\langle x, y \rangle$ by $\max(\langle x, y \rangle, 0)$. After millions of rays being launched in all directions, you will be sure to find numerically small but negative values that could mess with your simulation. Also, you now test the visibility by launching a ray towards a point sampled on the light source and testing for intersections. However, your light source is a sphere that is part of the scene. It is thus possible for our visibility query to return a point on the light source that is numerically *almost* the same as the point that has been sampled on the light source (if there is no shadow, the resulting intersection point and the point sampled on the sphere should be mathematically the same, but numerical errors will arise). An epsilon should be added in the visibility test to avoid self shadowing, in a similar way that rays were launched by a slightly offsetted point above the surface.

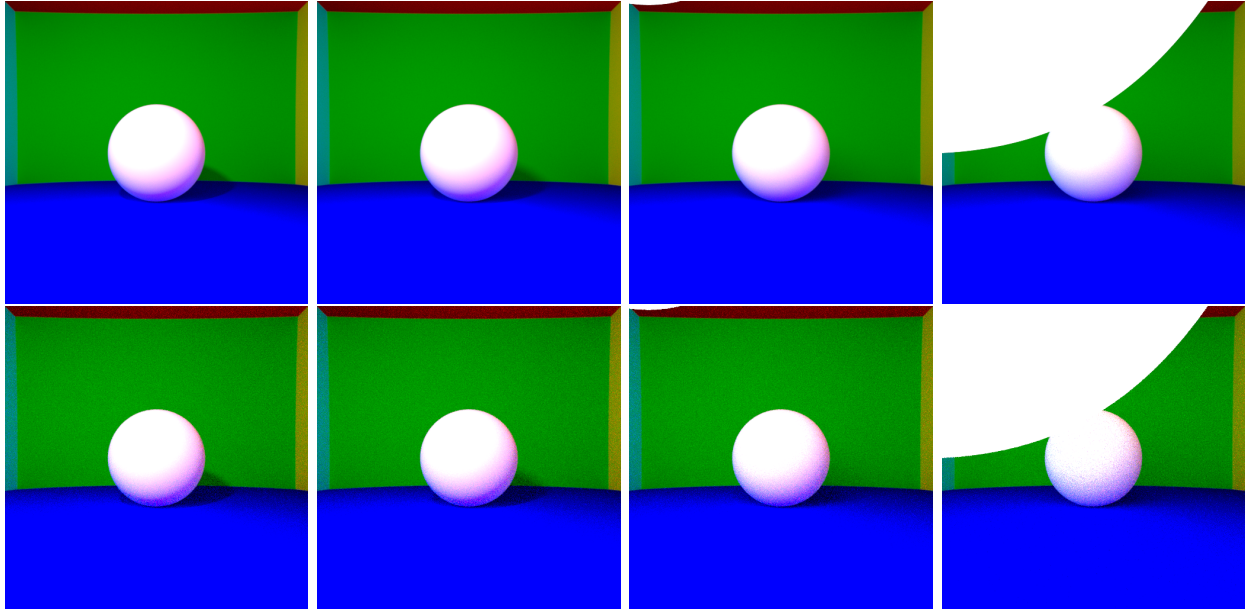


Figure 2.23: Soft shadows by directly sampling the spherical light source (the code is now about 320 lines) of radius 1, 2, 10 and 20. Using 1000 samples per pixel and 5 light bounces (top row), it takes about 1 minute per image. Using 32 samples per pixel (bottom row), about 2 seconds. Note that noise could be decreased by taking into account correlations between pixels (see text).

Depth of Field, motion blur and camera models

Our generated images are sharp at all distances. However, photographs tend to be sharp only around a certain distance, called the focus distance. In fact, our camera model corresponds to what is known as a *pinhole* camera (Fig. 2.26): just a dark box of length f (called *focal length*) pierced with a tiny hole (in practice, the optimal hole size is $d = 2\sqrt{f\lambda}$). This kind of setup has been known for a long time. In fact, it is suspected that it was known since paleolithic times¹⁰. In more recent times, pinholes were used to paint realistic scenes by projecting landscapes on a canvas, a setup called camera obscura, *locus obscurus* or *camera clausa* – for instance this led to early realistic depictions of Venice sceneries (Fig. 2.25)¹¹.

To implement depth of field (DoF), we will assume a circular aperture. The idea is to realize that all points at the focus distance describe a plane where points project to points on the sensor and remain sharp (Fig. 2.27), and that light passes through the aperture before reaching the lens. The result is exactly as if we made infinitely many renderings from pinhole cameras, where the tiny hole location

¹⁰see <http://paleo-camera.com/> for discussions on suspected paleolithic and neolithic setups.

¹¹In fact, the Hockney-Falco thesis says that the drastic increase in realism in the 17th century is due to such technological advances ; other famous artists may have used such devices such as Vermeer (1632-1675) https://en.wikipedia.org/wiki/Hockney%E2%80%93Falco_thesis.

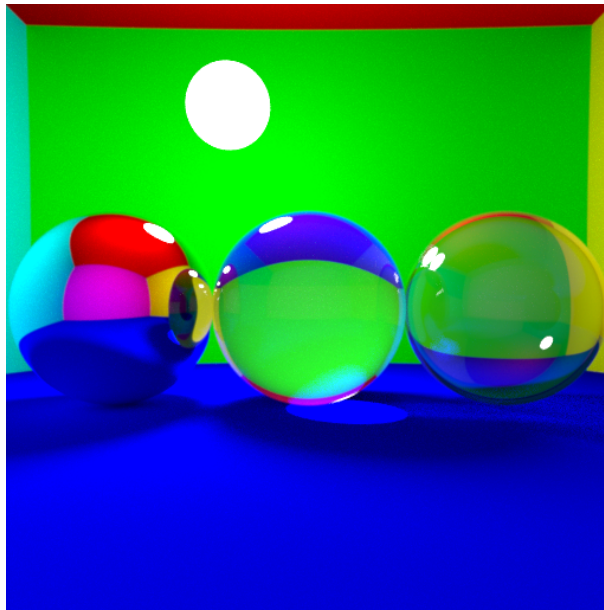


Figure 2.24: Moving the light a little bit reveals caustics in the transparent scene. Here the light sphere is at position $(-10, 25, -10)$ and of radius 5. These indirect specular bounces are hard to capture and thus produce much higher levels of noise (here, 5000 samples per pixel were used). Other techniques such as bidirectional path tracing or photon mapping better capture caustics.



Figure 2.25: The camera obscura was used for precisely painting scenes. This was used by a number of artists such as Canaletto (1697-1768, left), or Luca Carlevarijs (1663-1730, right: Venicians arriving in London in 1707)

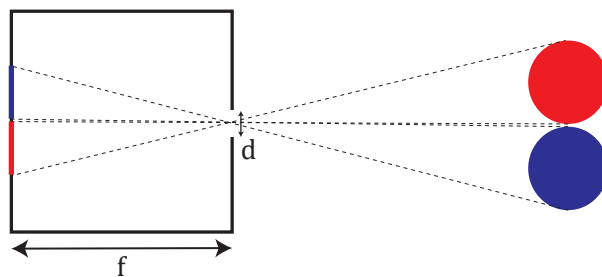


Figure 2.26: A pinhole camera is just a small hole in a dark chamber that lets light come in and displays a sharp view of the outside world on the screen. The image is flipped: in our path tracer, we have just put our sensor at a virtual location at a distance f outside the box for a more intuitive implementation and non-flipped renderings (in our setup, the *camera location* C is the hole, and the pixel grid is outside).

varies inside a small disk of the size of the aperture, and then average results. For implementation purpose, similarly to the pinhole case, we will keep the camera sensor and lens locations swapped. As

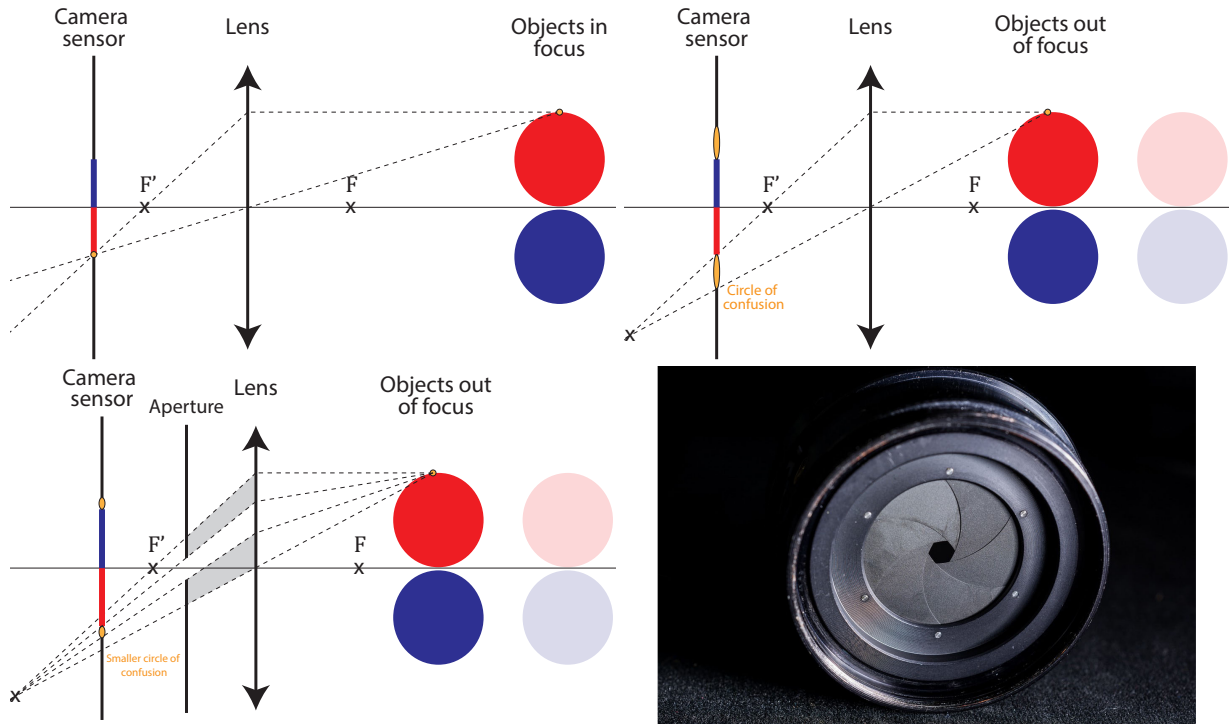


Figure 2.27: **Top row.** Using a camera lens, an object if placed at the focus distance will appear sharp (left) as the image of a point of the object is a point on the sensor. However, moving the distance closer to (or away from) the camera makes the object appear blurry as the image of a point is a small disk called the circle of confusion. **Bottom row.** By adding a shutter aperture (setup on the left, photo on the right), the circle of confusion can be made much smaller resulting in sharper images away from the focus distance by blocking light (and hence resulting in darker images). If the circle of confusion is smaller than a pixel, the image appears sharp. Cameras allow for varying the position of F' , varying the distance of the lens to the camera sensor, and the size of the aperture (the first two vary together in parfocal lenses to remain in focus while zooming).

such, we will simply find new starting points for our rays that are slightly tangentially offsetted from the camera location Q , and recompute their directions such that all rays targeting a given pixel cross at the plane that remains in focus (up to antialiasing).

To achieve that, we first generate a ray from the camera center Q (the pinhole center) as before (red ray in Fig. 2.27). Then we find the point P that would be in focus. This point is given in our case by $P = Q + \frac{D}{|u_z|}u$ where D is the distance at which objects appear in focus, u is the (original) ray direction, and u_z its z coordinate (since objects appear sharp on a plane at a distance D from Q in the optical axis – up to Petzval field curvature)¹². Once P is found, you can generate a point inside the aperture shape (here, a disk, but you can simulate bokeh of various shapes) which will serve as your new origin Q' and compute the ray direction as the unit vector u' towards P (Fig. 2.28). Generating a point on a disk can be performed in polar coordinate by choosing the square root of a uniform random number as the radius r , and a uniform random angle θ in $[0, 2\pi]$. Results can be see in Fig. 2.29.

Similarly, while the shutter of the camera is open, objects may have moved. This produces another kind of blur called *motion blur*. This is easily simulated in our path-tracer: now, rays have an additional time parameter, and objects have a way of describing their motion (in my simple implementation, they merely have a single speed vector defaulting to zero, but more complex motion is possible). To simulate motion blur, we randomly select the time parameter of the generated ray within the time the shutter is open, and compute the intersection with scene that includes object motion. In my simple implementation, I merely translate the sphere origin by the sphere's speed multiplied by the ray time

¹²You may also simulate tilt-shift photography by changing the orientation of this plane.

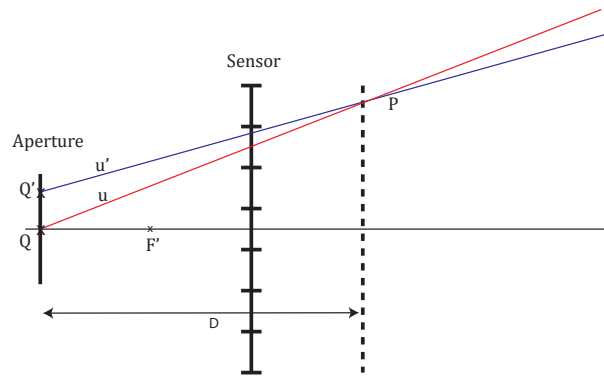


Figure 2.28: Depth of field can be obtained in our path tracer by starting rays from a point on the aperture shape (instead of the pinhole center) such that rays cross at the focusing distance D .

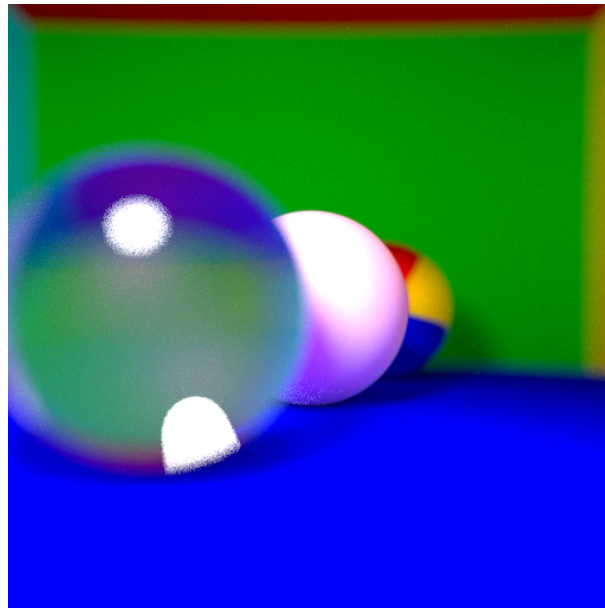


Figure 2.29: Depth of field result in our path tracer, adding less than 10 lines of code, bringing it to 330 lines. Here, 2000 samples per pixel were used because of specular paths, though depth of field often necessitate more samples.

parameter in the Ray-Sphere intersection test. By essentially adding two lines of code and modifying a couple of others, we obtain the result shown in Fig. 2.30.

Meshes

The next big thing in our path tracer is the support of triangle meshes. It is highly uninteresting for me to make you implement a loader for mesh files, so I provide an ugly one that can be downloaded at: <https://pastebin.com/CAgp9r15>

Sure, that adds 200 lines to our path tracer, but let's start simpler.

Ray-Plane intersection. A plane is defined by a normal vector N and a point A that belongs to the plane. All points P from the plane thus have the equation $\langle P - A, N \rangle = 0$. Substituting P by the equation of a ray starting at O , of direction u , leads to $\langle O + t u - A, N \rangle = 0$, and hence, the unique solution, if it exists, is:

$$t = \frac{\langle A - O, N \rangle}{\langle u, N \rangle}$$

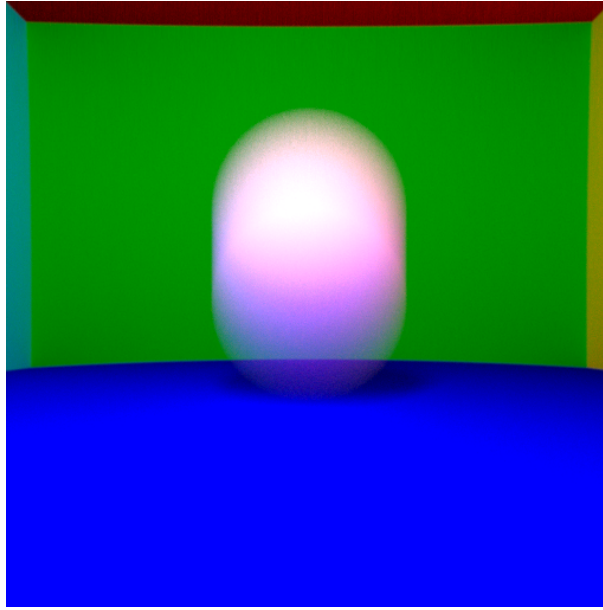


Figure 2.30: Motion blur is obtained by adding a time parameter to the rays. The time value is selected randomly within the interval of time the camera shutter is kept open. The ray-sphere intersection here considers a linear motion of the sphere. This merely adds 2 lines of code, and modifies a couple of others.

We are still only interested in positive solution.

Ray-Triangle intersection. A point P is within a triangle defined by vertices A , B and C if $P = \alpha A + \beta B + \gamma C$, $\alpha, \beta, \gamma \in [0, 1]$ and $\alpha + \beta + \gamma = 1$. α , β and γ are called the *barycentric coordinates* of P , and when P is inside ABC , they represent ratios of areas, e.g., $\alpha = \frac{\text{area}(PBC)}{\text{area}(ABC)}$. It is often impractical to have 3 barycentric coordinates for something intrinsically 2-dimensional, so we often reparameterize it by saying that $P = A + \beta e_1 + \gamma e_2$ where $e_1 = B - A$ and $e_2 = C - A$ (also use the fact that $\alpha + \beta + \gamma = 1$). Using the ray equation, we obtain a linear equation for the point of intersection of the form $\beta e_1 + \gamma e_2 - t u = O - A$. In matrix form:

$$\left(\begin{array}{c|c|c} e_1 & e_2 & -u \end{array} \right) \begin{pmatrix} \beta \\ \gamma \\ t \end{pmatrix} = \begin{pmatrix} O - A \end{pmatrix}$$

We note that for a 3x3 matrix

$$\det \left(\begin{array}{c|c|c} A & B & C \end{array} \right) = \langle A, B \times C \rangle$$

where \times denotes the cross product, and that swapping columns change the sign of the determinant while circular permutation does not. We also note N the non-normalized normal, using $N = e_1 \times e_2$. Using Cramer's formula, we obtain the solution of this linear system by ratios of determinants:

$$\beta = \frac{\langle O - A, e_2 \times -u \rangle}{\langle e_1, e_2 \times -u \rangle} = \frac{\langle e_2, (A - O) \times u \rangle}{\langle u, N \rangle} \quad (2.11)$$

$$\gamma = \frac{\langle e_1, (O - A) \times -u \rangle}{\langle e_1, e_2 \times -u \rangle} = -\frac{\langle e_1, (A - O) \times u \rangle}{\langle u, N \rangle} \quad (2.12)$$

$$\alpha = 1 - \beta - \gamma \quad (2.13)$$

$$t = \frac{\langle e_1, e_2 \times (O - A) \rangle}{\langle e_1, e_2 \times -u \rangle} = \frac{\langle A - O, N \rangle}{\langle u, N \rangle} \quad (2.14)$$

$$(2.15)$$

We obtain the *Möller–Trumbore intersection algorithm*.

Ray-Mesh intersection. A mesh will be considered as a set of triangles, so, for now, we will merely traverse all triangles and return the intersection closest to the camera, in exactly the same way we traverse all objects of the scene to return the closest intersection to the camera. This will be considerably slow, but we will improve next.

To obtain our first mesh renderings, we will need now to inherit the class `Sphere` from a more general `Geometry` abstract class. An abstract class is a class that has some pure virtual functions (functions that are not implemented at all, they are tagged `virtual` and their prototype ends with `= 0` to indicate no implementation is provided), and so, these class cannot be instantiated. Here, our pure virtual function is the `intersect()` routine. We will now use the `TriangleMesh` provided class, and make it inherit as well from `Geometry`. Our scene will now consists of an array of pointers to `Geometry` rather than (pointers to) `Sphere`.

⚠ A common bug is to duplicate properties such as materials/albedo/transparency... in the parent (`Geometry`) and children (`Sphere` and `TriangleMesh`) classes, which results in the wrong variables being used. Be sure to have all common properties **only** in the parent class. You may want to debug your code using a mesh consisting of a single manually constructed triangle.

For a simple demo object, we will be using a low poly cat mesh, available at <http://www.cadnav.com/3d-models/model-47556.html> (Edit: as of 2021, cadnav is down. I have put this model at : <https://perso.liris.cnrs.fr/nbonneel/cat.rar>). It has 3954 polygons to test.

⚠ Unless you made a fancy GUI, normalized your models upon loading, or know or made your 3d model, it is a good habit to check the obj file as a text file or display the bounding box to make sure sizes are reasonable and the orientation looks correct. 3D modelers can use different units so you could end up with a kilometer-sized cat or millimeter-sized cat that will not be visible, and the orientation is not standardized either so that the up vector can be arbitrarily the $+Y$ or $+Z$ coordinate (most often). Here, our cat model is roughly in the range $(-35..30, 0.45, -8..8)$ which means our cat is a pretty big boi (given our spheres are of radius 10), and given our ground is at a Y coordinate of -10 , our cat is floating in the air. I will first scale it by a factor 0.6 and translate it by $(0, -10, 0)$ to obtain Fig. 2.31.

⚠ For visual studio users, it is unfortunate that temporary files for compiling your project have an `.obj` extension. Concretely, this means that if you place your `.obj` mesh in your binary folder and perform a project “Clean up”, it will remove all temporary `.obj` files **and** your mesh. Either put your meshes in a subfolder, or save it somewhere else just in case you mistakenly clean your project.

Acceleration structures – Bounding Box. Right now, the rendering is pretty slow due to the linear time spent in checking all triangles of the mesh – more than 6 minutes for 32 samples per pixel and 5 light bounces – though only adding about 40 lines of code (excluding the 200 lines obj loader). A simple optimization is to test whether the axis aligned bounding box of the object is intersected by the ray, and then only checking all triangles if the ray intersects the bounding box.

We have seen the equation for a ray-plane intersection. A bounding box is defined by the intersection between the volumes enclosed by pairs of planes. As such, as simple algorithm consists considering the pairs of intersections between the ray and pairs of planes. Theses pairs of intersections define 3

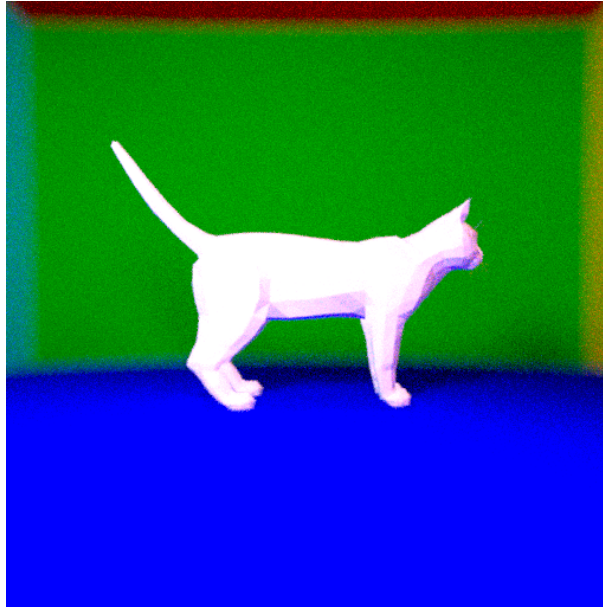


Figure 2.31: Our cat model, just scaled by a factor 0.6 and translated by the vector $(0, -10, 0)$. At 32 samples per pixel (spp) and 5 light bounces, it took 6 min and 20 seconds (in parallel) by naively testing all triangles using the *Möller–Trumbore intersection algorithm*. By adding a simple ray-bounding box test (and 30 lines of code) this falls to 1 min and 10 seconds. Using a simple BVH (and about 50 additional lines), the rendering time even falls down to less than 3 seconds, with a close to 150x speedup compared to the naive approach !

intervals, one for the two planes of constant X , one for the two planes of constant Y and one for the two planes of constant Z . If these intervals have a non-null intersection, this means a ray-bounding box intersection exists, and the ray-triangles intersection can be performed. An interval intersection test hence corresponds to testing whether $\min(t_1^x, t_1^y, t_1^z) > \max(t_0^x, t_0^y, t_0^z)$ (where if this is true, the actual intersection is $\max(t_0^x, t_0^y, t_0^z)$), denoting t_0^x the intersection along the ray with the first plane of constant x (similarly for subscript 1 and superscripts y and z – see Fig. 2.32 for notations in 2-D). It is also interesting to see that a ray-plane intersection with axis-aligned planes take a particularly simpler form.

We can now write a `BoundingBox` class containing the two extremas of our bounding box (B_{\min} and B_{\max}), compute the bounding box of the mesh, and write a function for ray-bounding box intersection. This makes the routine 6 times faster.



Beware of computing the bounding box **after** having translated and scaled your model !

Acceleration structures – Bounding Volume Hierarchies (BVH). The previous idea can be implemented recursively: if the ray hits the bounding box of the mesh, we can further test if it hits the two bounding boxes containing each just half of the mesh (and so on with a quarter of the mesh etc.). The idea is to build a binary tree, with the root being the entire mesh’s bounding box. We then take the longest axis of the bounding box. Then for each triangle, we determine if its barycenter is within the first half or the second half of this axis. This determines two sets of triangles, for which we can compute their bounding boxes and that can be set as the two children of the root node. This process is recursively performed for these two children nodes, until some criteria is met (for instance, until the number of triangles in a leaf node is smaller than some threshold). Beware that this procedure does not produce a space partition: bounding boxes can overlap, since the decision to put a triangle on one side or the other is only based on its barycenter, while **bounding boxes are computed using the triangle’s 3 vertices** (see Fig. 2.33).

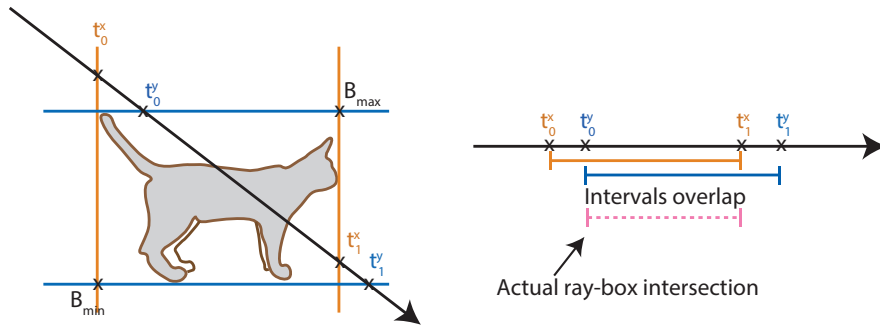


Figure 2.32: A ray-bounding box intersection can be performed by testing the overlap between intervals defined by pairs of ray-planes intersections.

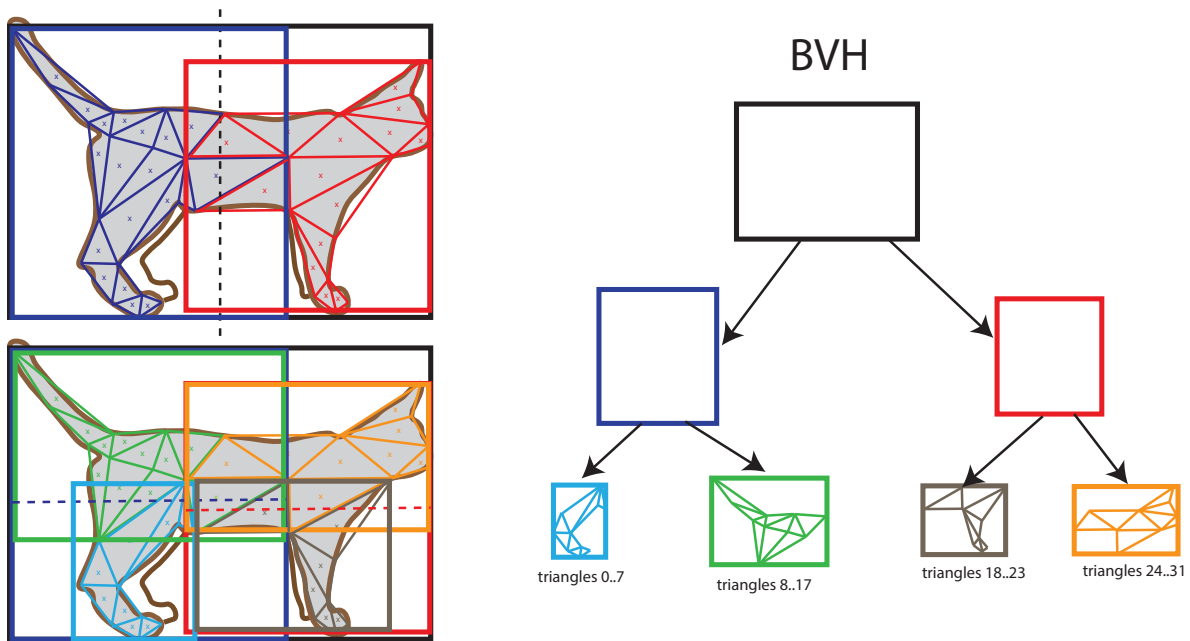


Figure 2.33: A BVH recursively computes bounding boxes. The overall bounding box (black) is split into 2 categories (blue and red) based a vertical split in the middle of the black box. Triangles are assigned to either the blue or red categories based on their centroid. The bounding boxes of these two sets of triangles are computed (and they may overlap), and then each subdivided into 2 new categories (cyan and green, and orange and grey). The process can go further. Here, the 4 leaves of the tree contain consecutive indices of triangles referring to a permutation of the original set of triangles.

In practice, building this tree can be performed using a method akin to *Quick Sort*: triangles are simply reordered in a way that consecutive triangles are in the same bounding box. This can be done by keeping track of a pivot and performing swaps such that elements before the pivot are smaller, while elements after it are always larger. This looks like:

```

1 node->bbox = compute_bbox(starting_triangle, ending_triangle); //BBox from ←
   starting_triangle included to ending_triangle excluded
2 node->starting_triangle = starting_triangle;
3 node->ending_triangle = ending_triangle;
4 Vector diag = compute_diag(node->bbox);
5 Vector middle_diag = node->bbox.Bmin + diag*0.5;
6 int longest_axis = get_longest(diag);
7 int pivot_index = starting_triangle;
8 for (int i=starting_triangle; i<ending_triangle; i++) {
9   Vector barycenter = compute_barycenter(indices[i]);
10  // the swap below guarantees triangles whose barycenter are smaller than ←

```

```

    middle_diag are before "pivot_index"
11  if (barycenter[longest_axis] < middle_diag[longest_axis]) {
12      std::swap(indices[i], indices[pivot_index]);
13      pivot_index++;
14  }
15  }
16  // stopping criterion
17  if (pivot_index <= starting_triangle || pivot_index >= ending_triangle-1 || ←
    ending_triangle-starting_triangle < 5 ) return;
18  recursive_call(node->child_left, starting_triangle, pivot_index);
19  recursive_call(node->child_right, pivot_index, ending_triangle);

```

Remark: We used the middle of the axis as a criterion for separating triangles. In unbalanced scenes (with many more triangles on one side than the other) this may not be optimal. A heuristic consists in minimizing the *Surface Area Heuristic* (SAH)¹³ to find a better place to cut.

Once the tree is built, the ray-BVH intersection can be performed by recursively visiting boxes that are intersected. An interesting option is to perform a depth-first traversal until a triangle is intersected (if any), and to avoid visiting bounding boxes that are further than the best triangle found so far¹⁴:

```

1  if (!root.bbox.intersect(ray)) return false;
2  std::list<Node*> nodes_to_visit;
3  nodes_to_visit.push_front(root);
4  double best_inter_distance = std::numeric_limits<double>::max();
5  while(!nodes_to_visit.empty()) {
6      Node* curNode = nodes_to_visit.back();
7      nodes_to_visit.pop_back();
8      // if there is one child, then it is not a leaf, so test the bounding box
9      if (curNode->child_left) {
10         if (curNode->child_left->bbox.intersect(ray, inter_distance)) {
11             if (inter_distance < best_inter_distance) {
12                 nodes_to_visit.push_back(curNode->child_left);
13             }
14         }
15         if (curNode->child_right->bbox.intersect(ray, inter_distance)) {
16             if (inter_distance < best_inter_distance) {
17                 nodes_to_visit.push_back(curNode->child_right);
18             }
19         }
20     } else {
21         // test all triangles between curNode->starting_triangle
22         // and curNode->ending_triangle as before.
23         // if an intersection is found, update best_inter_distance if needed
24     }
25 }

```

Doing so drastically improves the render time: now less than 3 seconds for 32 spp! The traversal order can also be optimized: since we perform a depth first traversal, it can be useful to first traverse boxes that are closer to the ray origin. Feel free to add this to your pathtracer !

¹⁴A similar remark holds between objects of the scene: it is not useful testing the triangles of a mesh whose bounding box is further than the best triangle found so far.

Normals and Textures

Now that we have computed geometric intersections with triangles, we can use barycentric coordinates to interpolate values on the mesh. The first thing we will do is interpolating normals. In fact, 3d models are often provided with per-vertex normals (or even per-vertex-per-triangle: one vertex can have different normals depending on which triangle it is considered to belong to). These artist-defined normals control the perceived smoothness of the shape, without changing the geometry itself, by allowing each shaded point to receive a normal that is interpolated from the normals of the vertices of the intersected triangle. Specifically, we can compute the *shading normal* as $\hat{N}(P) = \frac{\alpha(P)N_A + \beta(P)N_B + \gamma(P)N_C}{\|\alpha(P)N_A + \beta(P)N_B + \gamma(P)N_C\|}$ where $\alpha(P)$, $\beta(P)$ and $\gamma(P)$ are the barycentric coordinates of P within the triangle ABC whose artist defined normals at A , B and C are respectively N_A , N_B , and N_C . This shading normal can be used in all lighting computations¹⁵. This process is called *Phong interpolation* (and has nothing to do with the Phong BRDF except this is the same inventor...). The result can be seen in Fig. 2.34.

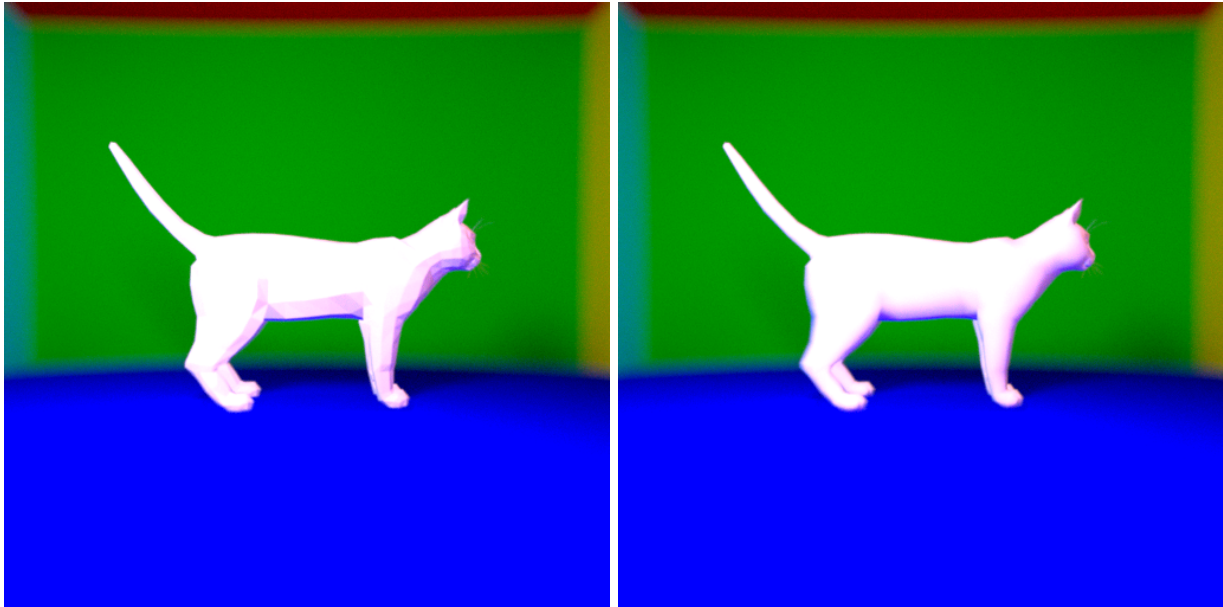


Figure 2.34: Cat model without (left) and with (right) interpolation of normals.

Similarly, vertices are associated with “per-vertex-per-triangle” UV coordinates. These coordinates correspond to a parameterization of the mesh, which is non-trivial to obtain in the general case. UV coordinates associate to each vertex of each triangle a 2D point within a texture map. The texture domain is normalized in the range $[0, 1]^2$. Interpolated UV coordinates are often interpreted *modulo 1*, that is, only the fractional part of the texture coordinates are used (if values are positive – consider the texture is a flat torus), which can be useful for tiling textures (a wall made of bricks can be geometrically modeled by a single quad, with UV coordinates $(0, 0)$ and (N, N) at its extremities: a texture of a single brick can be then used, and will produce a tiling of $N \times N$ bricks). UV coordinates interpolation is similarly performed: $\hat{UV}(P) = \alpha(P)UV_A + \beta(P)UV_B + \gamma(P)UV_C$. The interpolated UV coordinates are then scaled by the width and height of the texture, and the texture color is then queried at the corresponding pixel (Fig. 2.35). This color can serve as the albedo, for example.

We are now ready to implement textures. We will be using `stb_image` (see Sec. 1.1) to load the image and the `stbi_load` function, and retrieve its width and height. Each triangle is associated with a `group` that corresponds to the material index within the associated `cat.mtl` file. This material file, in this case, contains a single material, so all `group` values are set to 0 for all triangles – this may not be the case for more complex objects, where different textures can be used for different parts of the mesh. You can add a function to load one (or several) textures upon loading the mesh, and your intersection routine should now return an albedo computed locally. The result can be seen in Fig. 2.36.

¹⁵One can however notice that tweaking the integration hemisphere may break BRDF energy conservation...

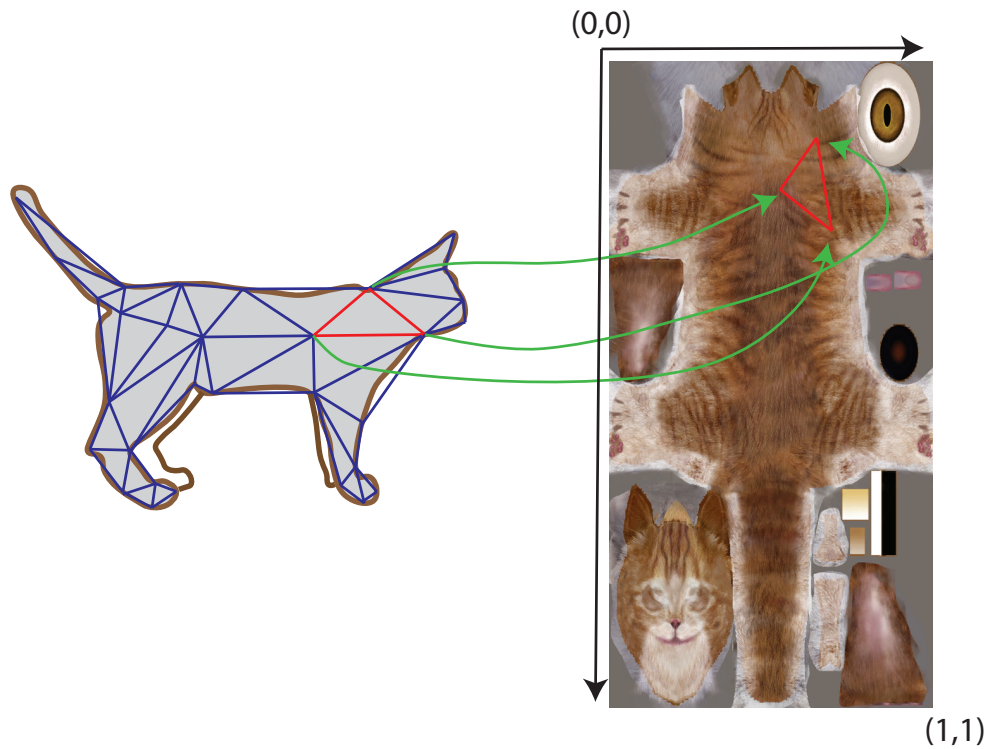


Figure 2.35: UV coordinates associate for each 3D vertex a 2D coordinate in the texture map, that can be interpolated using barycentric coordinates.

⚠ Albedo values are in the range $[0, 1]$ while textures are integers stored in unsigned chars. Do not forget to divide by 255 ! But at this stage, you may realize that your texture was saved in a gamma-corrected color space, so you would also need to apply a gamma function of $color^{2.2}$ to the queried colors. Also, make sure to convert your pixel coordinates (x, y) to integers **before** accessing textures with formulas such as `texture[y*W*3+x*3+c]`. If these coordinates contain fractional parts, the wrong pixel will be accessed ! Finally, beware that the origin $(0,0)$ of UV coordinates is conventionally the top left of the texture, while most often textures are loaded from bottom to top.

Blinn-Phong BRDF

Our materials were until now “perfect”: perfectly diffuse, perfectly specular or perfectly transparent. However, most real-world materials are some combinations of these materials, or have some aspects of these materials. A simple model was initially presented by Phong, called the Phong BRDF, and is formulated as $f(\omega_i, \omega_o) = \frac{\rho_d}{\pi} + \rho_s \langle \omega_i, R_N(\omega_o) \rangle^\alpha$, with $R_N(\omega_o)$ the reflection of ω_o around the normal N , and α the *Phong exponent* that controls the frequency of the reflection (high α produces smaller highlights, giving the impression of a more shiny material, see Fig. 2.38). However, this model does not model well highlight distortions at grazing angles (Fig. 2.37). A modified Phong BRDF model is given by the Blinn-Phong model :

$$f(\omega_i, \omega_o) = \frac{\rho_d}{\pi} + \rho_s \frac{\alpha + 8}{8\pi} \langle N, H \rangle^\alpha$$

which better handles grazing incidences (the correct normalization factor is also slightly more complex). The term $H = \frac{\omega_i + \omega_o}{\|\omega_i + \omega_o\|}$ is called the *half-vector*, a vector halfway between the incident and outgoing directions (considering both vectors go *away* from the surface). We will implement this model.

To implement the Blinn-Phong BRDF, you could simply replace the diffuse BRDF we used by this BRDF. That would work – up to large noise levels for specular materials. Our importance sampling

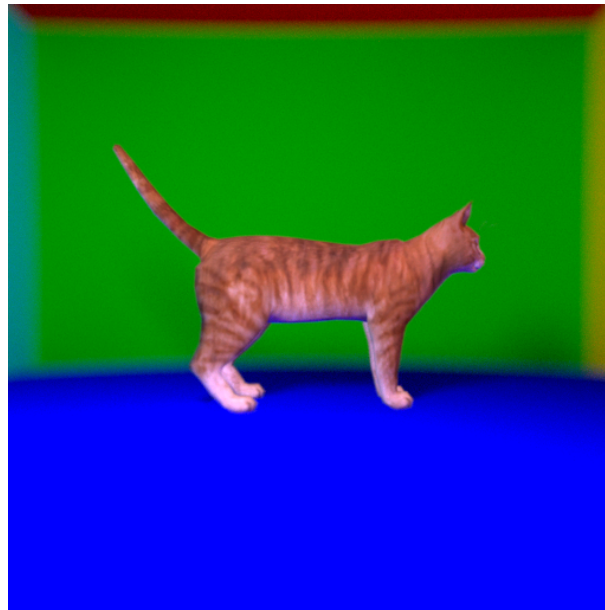


Figure 2.36: Cat model with textures, with a gamma function applied. The code is now about 700 lines long, including the 200 lines obj file reader.

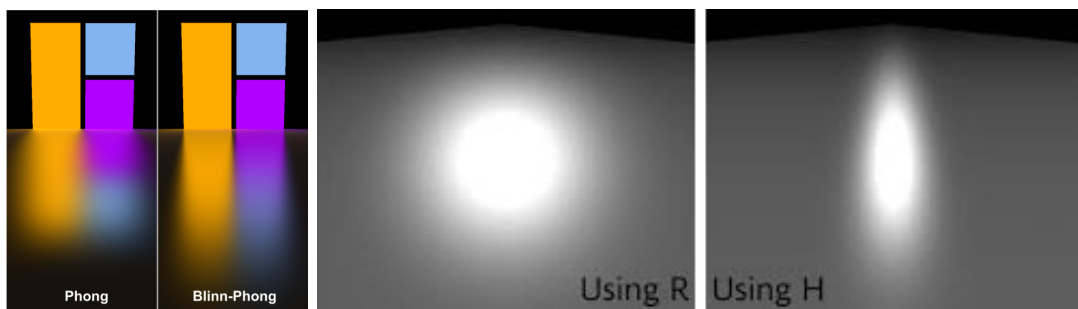


Figure 2.37: The original Phong model does not appropriately model the distortion of highlights at grazing angles (left of each pair) while this is solved by the Blinn-Phong model (right of each pair). *Left image pair by an unknown author. Right image pair by Lecocq et al. 2017.*

strategy consisted in sampling the hemisphere according to a simple cosine function, which produces more often directions near the surface normal and few directions at grazing angles. However, if the BRDF is highly specular, we expect the integrand to be very large near the reflected direction, and very low far from it. This does not coincide with where we importance sampled our directions.

The goal will be to produce an importance sampling strategy for the Blinn-Phong model.

Importance sampling the specular lobe. We will first focus on the specular component and assume $\rho_d = 0$ and $\rho_s = 1$. We have seen how to importance sample a direction that follows a cosine law around the normal of the surface – we called this function `random_cos(const Vector &N)`. There is a generalization of this importance sampling strategy that allows to sample according to some power of a cosine law¹⁶.

¹⁶See again Philip Dutré’s Global Illumination Compendium <https://people.cs.kuleuven.be/~philip.dutre/GI/TotalCompendium.pdf>

$$\begin{aligned}
 r_1, r_2 &\sim \mathcal{U}(0, 1) \\
 x &= \cos(2\pi r_1) \sqrt{1 - r_2^{\frac{2}{\alpha+1}}} \\
 y &= \sin(2\pi r_1) \sqrt{1 - r_2^{\frac{2}{\alpha+1}}} \\
 z &= r_2^{\frac{1}{\alpha+1}}
 \end{aligned}$$

where the pdf is given by $p(X) = \frac{\alpha+1}{2\pi} \cos^\alpha \theta$, where here, θ is the angle with the $+z$ axis (or any other vector, up to a frame change as we did earlier).

We can use this formula to sample a half-vector H (which is the direction that follows some lobe-shaped law around the normal), and bring it to our local frame with the same change of variables as before. We finally need to mirror ω_o by H to obtain the desired sampled direction ω_i . This last step introduces a transformation that needs to be taken care of in the pdf: we now have $p(\omega_i) = \frac{1}{4\langle \omega_o, H \rangle} \frac{\alpha+1}{2\pi} \langle H, N \rangle^\alpha$. Let us call this entire sampling procedure `random_pow(const Vector &N, double alpha)`.

Importance sampling a mixture model. We would like to sample a distribution of the form $p(x) = \sum_i \alpha_i p_i(x)$, with $\sum_i \alpha_i = 1$. This can be achieved by using a uniform random number between 0 and 1 to determine which of the p_i to sample, with probability α_i . But then, multiple choices are possible to numerically evaluate the integral $I = \int f(x)dx = \int \sum w_i f_i(x)dx$. The first, most immediate, option is to ignore the particular form of the integrand, and compute the estimate as $I \approx \sum_k \frac{f(x_k)}{\sum_i \alpha_i p_i(x_k)}$. However, this requires to evaluate $p_i(x_k)$ for all p_i . In our context, we have two p_i 's: one for the diffuse part that is cheap to compute, and one for the specular part that is expensive to compute. Having to evaluate the pdf for the specular part although we sampled the diffuse part is not optimal. There is another option that also works, by realizing that you actually evaluate a sum of integrals. In that case, the uniform random number that you initially chose actually corresponds to selecting which of the f_i you want to evaluate. The estimator becomes $I \approx \sum_k \frac{f_{i(k)}(x_k)}{\alpha_{i(k)} p_{i(k)}(x_k)}$ where $i(k)$ is the index of the k 's randomly sampled pdf p_i , and x_k the corresponding sample¹⁷. Doing so allows to first determine which term is sampled, and then *only* evaluate this part. This implies that if the diffuse component is chosen, there is no other complex function to evaluate compared to our implementation for diffuse materials¹⁸. We end up with a code similar to:

```

1 Vector Scene::getColor(const Ray& ray, int ray_depth) {
2     if (ray_depth < 0) return Vector(0., 0., 0.); // terminates recursion at some ←
3         point
4     Vector Lo(0., 0., 0.);
5     if (intersect(ray, P, N, sphere_id)) {
6         if (spheres[sphere_id].mirror) {
7             // handle mirror surfaces ...
8         } else {
9             // handle Phong materials
10            // add direct lighting
11            Vector xprime = random_point_on_light_sphere();
12            Vector Nprime = (xprime-centerLight)/(xprime-centerLight).norm();
13            Vector omega_i = (xprime-P)/(xprime-P).norm();

```

¹⁷Similar weighting strategies exist for more general classes of integrand, which is called **Multiple Importance Sampling**

¹⁸There is a third option, but it works less well in practice – see *Variance reduction for Russian-roulette* <http://cg.iit.bme.hu/~szirmay/c29.pdf> for details.

```

14 double visibility = ... ; // computes the visibility term by launching a ray ←
    of direction omega_i
15 double pdf = dot(Nprime, (x-centerLight)/(x-centerLight).norm())/(M_PI*R*R);
16 Vector brdf_direct = PhongBRDF(...); // the entire Blinn-Phong model
17 Lo = light_intensity/(4*M_PI*M_PI*R*R) * brdf_direct * visibility * std::max(←
    dot(N, omega_i), 0.)*std::max(dot(Nprime, -omega_i), 0.)/((xprime-P).←
    squared_norm() * pdf);
18
19
20 // add indirect lighting
21 double diffuse_probability = rho_d/(rho_d+rho_s); // we should use some color←
    average of rho_d and rho_s
22 if (uniform(engine) < diffuse_probability) { // we sample the diffuse lobe
23     Ray randomRay = ...; // randomly sample ray using random_cos
24     Lo += albedo/diffuse_probability * getColor(randomRay, ray_depth-1);
25 } else {
26     Ray randomRay = ...; // randomly sample ray using random_pow and mirroring ←
    of ray.direction
27     if (dot(randomRay.direction, N) < 0) return Vector(0., 0., 0.); // make sure←
    we sampled the upper hemisphere
28     Vector brdf_indirect = rho_s * (alpha+8)/(8*M_PI) * PhongSpecularLobe(...); ←
    // just the specular part of the Blinn-Phong model
29     double pdf_pow = ...; // the pdf associated with our function random_pow ←
    with the reflection
30     Lo += brdf_indirect * std::max(dot(N, randomRay.direction), 0.)/((1-←
    diffuse_probability)*pdf_pow) * getColor(randomRay, ray_depth-1) ;
31 }
32 }
33 }
34 return Lo;
35 }

```

Regarding the choice of ρ_s , it is usually taken as white for dielectrics (e.g., plastics), but can be colored for metals. Results can be seen in Fig. 2.38.

Camera and object motion

We can move an object by a transformation T by instead transforming the rays via the inverse T^{-1} of T . Specifically, considering a 4x4 affine transform T , you need to transform the ray origin $(O_x, O_y, O_z, 1.0)$ and direction $(u_x, u_y, u_z, 0.0)$ by T^{-1} . The point of intersection found should then be transformed by T and its normal should be transformed by the inverse transpose matrix $(T^{-1})^T = T^{-T}$. Doing so has several advantages over directly transforming the vertices of each object upon loading them. First, a BVH can be appropriate for a mesh but not for a rotated version of it. But second, and more importantly, this allows for instantiating objects by merely storing several transforms of the same geometry. And finally, it allows for animating objects by merely playing with the transformations, rather than building a BVH at each frame of the animation.

Moving the camera is more straightforward: just transform the origin and direction of the ray when initially generating rays.

Recall that the inverse of a rotation is its transpose, the inverse of a diagonal scaling matrix is a diagonal matrix with the inverse of the scaling factors, and the inverse of a translation is a translation in the opposite direction. Our affine transforms usually are compositions of these elementary transforms. So, if a matrix encodes the transformation $y = sRx + t$ with s a scaling factor, R a rotation matrix and t a translation, then $x = R^T(y - t)/s$. As such, when there is no scaling factor and when dealing with vectors such as the normal vector, the inverse transposed transformation is the original transformation.

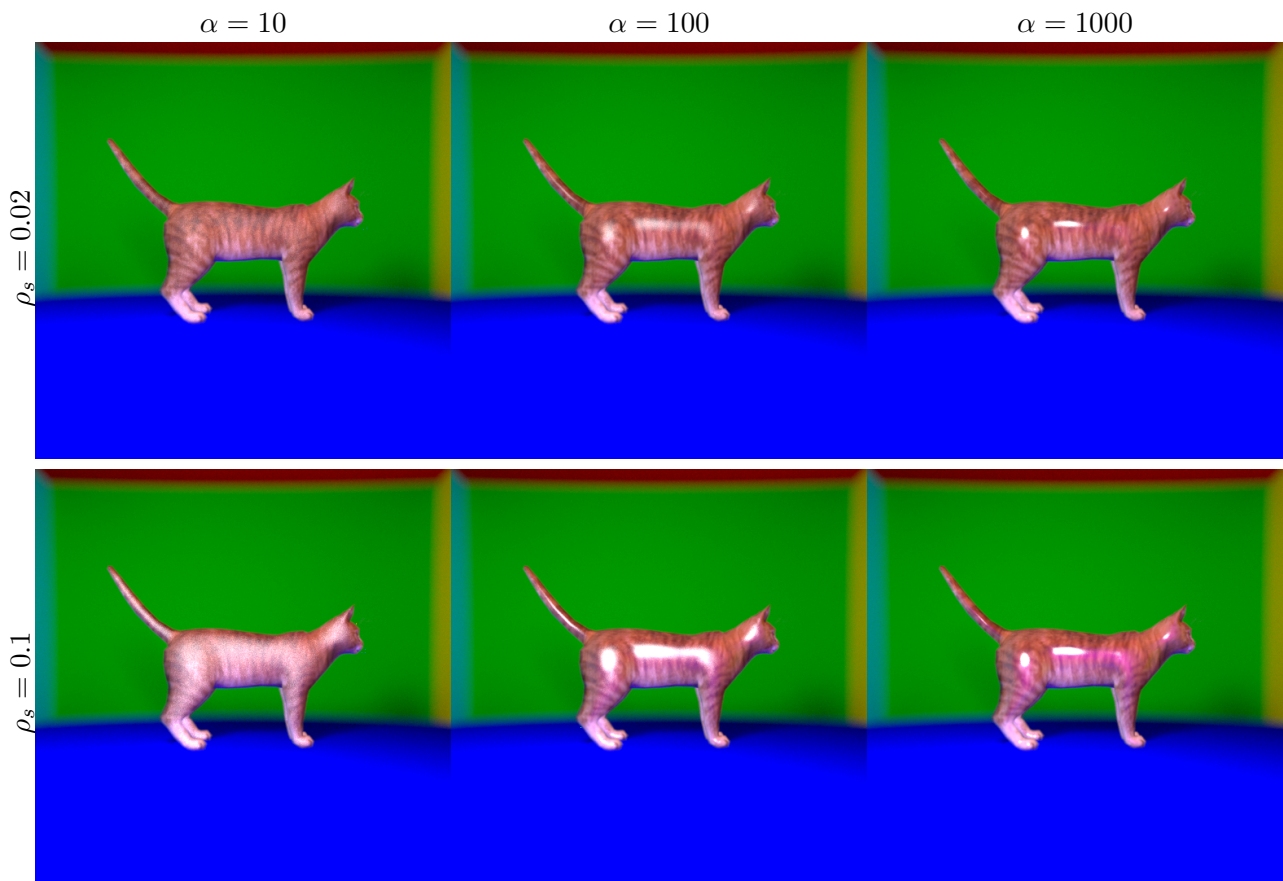


Figure 2.38: Cat model with a Blinn-Phong BRDF with varying α and ρ_s (here, ρ_s is not colored). α controls the roughness of the material (i.e., the size of highlights) while ρ_s controls the intensity of highlights. Note that since ρ_d is guided by a texture between 0 and 1 and ρ_s is a constant, this particular rendering may not preserve energy. The rendering takes 1min20 for 1000spp – the code is about 740 lines long.

⚠ Beware: you may have used the coordinates of the light source in your code, outside of the ray-object intersection test (e.g., during the shading computation). Do not forget to *also* transform these coordinates if you want to move the light source !

Normal Mapping

A common way to fake small details without increasing the geometric complexity of the mesh is to use normal maps – a second way to tweak the *shading normal*, the fake normal used during the shading computation in place of the geometric normal. A normal map is simply a texture that stores the shading normal in some local frame. The two coordinates UV within the normal maps are mapped to tangent and bi-tangent vectors (i.e., two vectors orthogonal to the geometric normal that form an orthogonal basis, as we did when we first implemented indirect lighting), and the RGB value within each pixel encodes the shading normal vector in this local frame. As such, most normal maps are blueish: the blue component represents the normal component of the shading normal, and the shading normal is most often close to the geometric normal that would be encoded as pure blue: $(0, 0, 1)$. However, to handle negative values, RGB pixel values are transformed using a $RGB * 2 - 1$ transformation, so in fact, a shading normal that would be identical to the geometric normal would be encoded $(0.5, 0.5, 1)$ (or $(127, 127, 255)$ in `unsigned char`).

To obtain the tangent and bitangent vectors, we will not proceed as before. In fact, our tangent

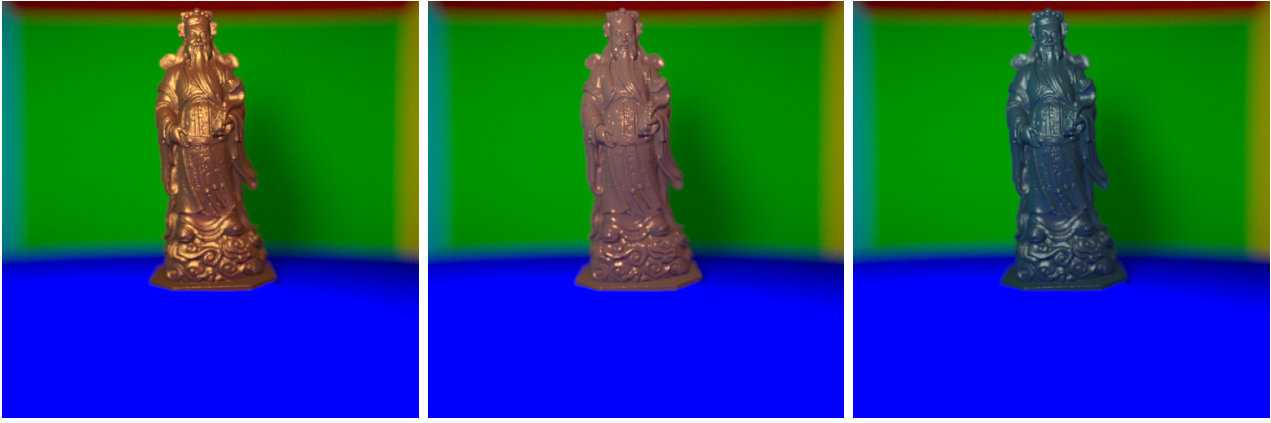


Figure 2.39: Another model (see <http://www.cadnav.com/3d-models/model-45798.html>) with Blinn-Phong BRDFs. The BRDF parameters can be found in the supplemental materials of “Experimental Analysis of BRDF Models” (<https://people.csail.mit.edu/wojciech/BRDFAnalysis/BRDFFits.pdf>), a document that contains fits of several analytical BRDF models on 100 measured materials. Here, they correspond to *metallic-gold*, *alum-bronze*, and *green-metallic-paint*. Note that this 3d mesh has both few very large triangles and many small triangles. This results in a highly unbalanced BVH, and the rendering time suffers: about 25min for 1000 spp and 5 bounces, for (only) 143k triangles – something that could be fixed with the Surface Area Heuristic for better balancing. The mesh has first been scaled by a factor 0.1, then translated by `Vector(0, 21, 45)`; the focus distance is 44 instead of 55.



Figure 2.40: I rotated the cat around the vertical axis by 45 degrees using matrix transforms (along with a hardcoded translation), and rotated the camera by -10 degrees around the x axis.

vectors did not matter before since our reflectance model was isotropic. Conventionally, these vectors T and B (for Tangent and Bi-tangent) are aligned with the UV parameterization: a vector $V(P)$ in 3D space at point P , can be expressed as a linear combination of $T(P)$ and $B(P)$ at P : $V(P) = V_u(P)T(P) + V_v(P)B(P)$.

As such, in a triangle DEF with UV coordinates D_u and D_v (similarly for E and F), and space coordinates D_x, D_y, D_z (similarly for E and F), we have

$$\begin{aligned} E - D &= (E_u - D_u)T + (E_v - D_v)B \\ F - D &= (F_u - D_u)T + (F_v - D_v)B \end{aligned}$$

In matrix form, this reads:

$$\begin{pmatrix} T_x & B_x \\ T_y & B_y \\ T_z & B_z \end{pmatrix} \begin{pmatrix} E_u - D_u & E_v - D_v \\ F_u - D_u & F_v - D_v \end{pmatrix} = \begin{pmatrix} E_x - D_x & F_x - D_x \\ E_y - D_y & F_y - D_y \\ E_z - D_z & F_z - D_z \end{pmatrix}$$

It becomes easy to invert the system, as:

$$\begin{pmatrix} T_x & B_x \\ T_y & B_y \\ T_z & B_z \end{pmatrix} = \begin{pmatrix} E_x - D_x & F_x - D_x \\ E_y - D_y & F_y - D_y \\ E_z - D_z & F_z - D_z \end{pmatrix} \begin{pmatrix} E_u - D_u & E_v - D_v \\ F_u - D_u & F_v - D_v \end{pmatrix}^{-1}$$

where the inverse of a 2x2 matrix is easily computed using $A^{-1} = \frac{1}{\det(A)} \text{Cof}(A)^T$ with Cof the cofactor matrix:

$$\begin{pmatrix} E_u - D_u & E_v - D_v \\ F_u - D_u & F_v - D_v \end{pmatrix}^{-1} = \frac{1}{(E_u - D_u)(F_v - D_v) - (F_u - D_u)(E_v - D_v)} \begin{pmatrix} F_v - D_v & -(E_v - D_v) \\ -(F_u - D_u) & E_u - D_u \end{pmatrix}$$

Written differently, we have:

$$T = \frac{1}{\det} ((E - D)(F_v - D_v) - (F - D)(F_u - D_u)) \quad (2.16)$$

$$B = \frac{1}{\det} (-(E - D)(E_v - D_v) + (F - D)(E_u - D_u)) \quad (2.17)$$

$$\det = (E_u - D_u)(F_v - D_v) - (F_u - D_u)(E_v - D_v) \quad (2.18)$$

Now, we can easily compute normalized T and B at each vertex of each triangle of the mesh¹⁹, and interpolate these vectors at the desired intersection point P using barycentric coordinates. The resulting shading normal becomes $\hat{N} = r(P)T(P) + g(P)B(P) + b(P)N(P)$ where r, g, b represent the red, green and blue components of the normal map (with the affine transform to bring them in $[-1, 1]$), and $T(P), B(P), N(P)$ represent the tangent, bitangent and (geometric) normal vectors at point P ²⁰. See Fig. 2.41 for the result.

Participating Media

Until now we have considered the medium in which light travels is just vacuum. It is however quite common for the medium to scatter light – for instance, fog, clouds, the atmosphere, dust... These media are called “participating media”. We will simulate that.

The first things to observe is that light is absorbed and scattered away as it travels through the medium. Light is absorbed exponentially with the distance traveled, as the Beer-Lambert law. But there is another phenomenon: light reaching neighboring particles is also in-scattered, *adding* its contribution to the light ray being considered. This is illustrated in Fig. 2.42.

These phenomena transform the rendering equation by modifying the intensity of the light reaching a point P_2 if it came from P_1 in a direction ω_i , while up to now, the light emitted from P_1 in direction ω_i was the same as the light received by P_2 from that direction. The absorption of light can be described by a multiplicative factor $T(t)$ that depends on the distance parameter t the light has traveled through the medium. The in-scattered light will be denoted L_v . We have:

$$L_i(P_2, \omega_i) = T(\|P_1 - P_2\|)L_o(P_1, \omega_i) + L_v(P_1, \omega_i)$$

¹⁹You may need to fiddle a little bit with the code: you may or may not have per vertex normals, and you may want to obtain per vertex (and not per vertex per triangle) tangents and bitangents. Here, we will consider that we have obtained one tangent T per vertex of the mesh by averaging the T computed for all triangles containing this vertex, orthogonalize it w.r.t. the per-vertex normal by removing its component along the normal, and then compute the bitangent B as the crossproduct between N and T .

²⁰Similarly to the smooth shading normals we have implemented in Sec. 2.2.1, having a shading normal that is not exactly the geometric normals can lead to issues in energy conservation.



Figure 2.41: Horse model without (left) and with (middle) normal mapping ; the normal map of the body is illustrated on the right. The code is about 850 lines and runs in 1min 12s (left) or 1min 15sec (right) using 1000spp and 5 bounces. The mesh has only 5333 polygons, but normal mapping makes it look more complex. The mesh can be downloaded here: <http://www.cadnav.com/3d-models/model-46223.html>. It has been rotated like the cat, scaled by 0.15 and translated by (10, -10, 0). The order of the textures to be loaded (since there is no .mtl file) is: `body2.d.tga`, `gear.d.tga`, `body2.d.tga`, `gear.d.tga`, `body2.d.tga`

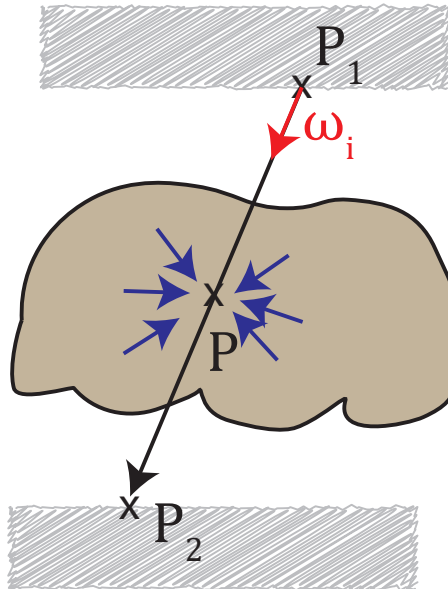


Figure 2.42: The light coming from direction ω_i is absorbed by the medium, but the medium also contributes positively (arrows in blue) to the light reaching point P_2 .

The factor $T(t)$ is called the transmittance function, and equals:

$$T(t) = \exp\left(-\int_0^t \sigma_t(P(r))dr\right)$$

where $P(r) = P_1 + r\omega_i$ and σ_t is the extinction coefficient of the medium, that can be seen as the gas density of the medium, with $\sigma_t = \sigma_a + \sigma_s$ the sum of the absorption coefficient and the scattering coefficient. In a few cases of interest, this integral can be computed in closed form. This is the case of homogenous media, where σ_t is a constant and thus $T(t) = \exp(-\sigma_t t)$. This is also the case for exponentially decaying fog (such as in the atmosphere) where $\sigma_t(y) = \alpha \exp(-\beta(y - y_0))$ with y the altitude over some ground level y_0 , in which case $T(t) = \exp\left(\frac{\alpha}{\beta\omega_{i,y}} (\exp(-\beta(P_y - y_0)) - \exp(-\beta(P_{1,y} - y_0)))\right)$ with $\omega_{i,y}$ the y component of the ω_i direction, and similarly for P_y and $P_{2,y}$. Here P_1 is the ray origin while P_2 is the first ray-scene intersection along the ray direction ω_i . An illustration of the effect of

absorption can be seen in Fig. 2.43.



Figure 2.43: The absorption term T , using a uniform extinction coefficient (left, $\sigma_t = 0.03$) and exponentially decreasing model (right, $\sigma_t = \exp(-0.3(y + 10))$).

Regarding L_v the in-scattered radiance, it corresponds to all light reaching points along the ray that scatter light in the direction ω_i . It can also simply be expressed as:

$$L_v(P_1, \omega_i) = \int_0^t \sigma_s(P(r))T(r) \int_{\mathbb{S}^2} f(\omega_i, v)L_i(P(r), v)dvdr$$

Here f is called the *phase function* and acts similarly to a BRDF. This function tells how much a light is reflected off a particle (e.g., of dust) or a molecule (e.g., of gas), similarly to the way a BRDF describes how much light is reflected off a surface. For simplicity, we will implement a uniform phase function (i.e., $f = 1/(4\pi)$) though you can google Mie scattering formula for large particles, and Rayleigh scattering for particles smaller than the light wavelength (giving its color to the sky).

At first sight, it seems that adding this integration in our path-tracer would be extremely costly. In fact, recall that what we are doing is Monte-Carlo that essentially does not care about the dimensionality of the integrand ! We are here merely adding a couple of dimensions to an integral equation that already had many. What we need is to merely be able to evaluate the integrand only with random parameters for r and v , and the way we average over all light path will take care of evaluating the integral. Our code should just look like:

```

1 Vector Scene::getColor(const Ray& ray, int ray_depth) {
2   if (ray_depth < 0) return Vector(0., 0., 0.); // terminates recursion at some ←
   point
3
4   Vector Lo(0., 0., 0.);
5   if (intersect(ray, P, N, sphere_id)) {
6     if (spheres[sphere_id].mirror) {
7       // handle mirror surfaces ...
8     } else {
9       // handle Phong materials
10    }
11  }
12  // return Lo; // previous code without participating media
13
14  double T = ....; // transmittance function (use closed form expression)

```

```


15   Vector Lv = sigma_s_r*T_r*phase_func*getColor(random_ray, ray_depth-1); // ←
      evaluate the integrand with a random "r" and random "v"
16   double pdf = ...; // pdf for the choice of "r" and "v"
17   return T*Lo + Lv/pdf; // return the radiance modified by the participating medium
18
19 }

```

The problem is that the above code contains 2 calls to `getColor`, one (hidden) to compute the indirect lighting contribution for Phong materials, and another (shown) for the participating medium computation. This will makes the number of rays in the scene explode. While one option is to use a smaller `ray_depth` for the participating medium, a simpler solution lies within *Single Scattering*. In the (direct) single scattering approximation, only the *direct* component is sampled instead of the entire sphere for the in-scattered contribution (while the light source will contribute a lot to the in-scattered radiance, the indirect lighting from objects and from nearby particles is often a much smaller contribution). We will thus not call `getColor` but send rays toward the light source for which the intensity is either that of the light source, or zero if it is occluded.

Regarding the random distance r , we could use a uniform random number in $(0, t)$ (with t the distance between the origin of the ray and the nearest intersection). But the exponentially decaying nature of the absorption makes it less relevant to sample a point that is very far away (since the light that will reach P_2 will be highly absorbed). We could instead use an importance sampling strategy that maximizes the contribution of light sources²¹. Instead, we will adopt a slightly simpler strategy: using an exponential distribution. To sample r with an exponential distribution of parameter λ , we can use $r = -\log(u)/\lambda$ with u a uniform random number in $(0, 1)$ ²² and the corresponding pdf is $p(x) = \lambda \exp(-\lambda x)$.

Regarding the random choice of v , we will sample a point on the light source, use the change of variable formula (which includes the visibility term, squared distance..), and throw a ray in this direction v . We will use the same pdf as we computed earlier for sampling spherical area light sources. The resulting images can be seen in Fig. 2.44.

 You may see very few bright pixels that do not seem to make sense. These are called *fireflies* and correspond to events of very low probability that would require many many more rays to be compensated... You may want to discard paths where the pdf is smaller than an epsilon. Beware however that it biases the rendering, but again, variance vs. bias is a tradeoff.

To conclude this course on path-tracing, I will just show a nicer scene. Because let's face it: the colors I previously used are just ugly. See Fig. 2.45;

²¹See for instance *Importance Sampling of Area Lights in Participating Media* <http://library.imageworks.com/pdfs/imageworks-library-importance-sampling-of-area-lights-in-participating-media.pdf>

²²This can be easily demonstrated using the inverse cumulative distribution function.

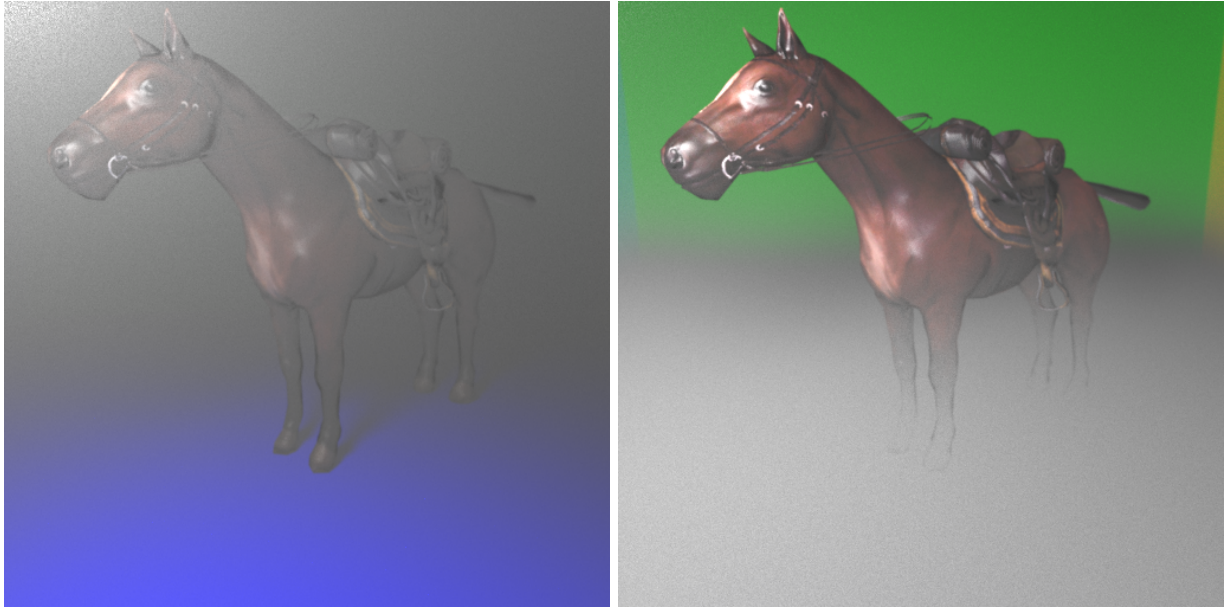


Figure 2.44: Adding the in-scattered radiance to the models presented in Fig. 2.43, with uniform (left, $\sigma_s = 0.004$) and exponential (right, $\sigma_s = 0.5 \exp(-0.3(y + 10))$) fog. I used $\lambda = 0.3t$. The code is 900 lines and renders in 3min 40sec.

2.2.2 Photon Mapping

A completely different approach relies in launching photons from all light sources, making them interact with the scene and storing photons on the 3d geometry at each bounce: this produces a *photon map* that contains millions of photons desposited in the scene (Fig. 2.46). This photon map is stored within an acceleration structure tailored for spatial search (while we could use A BVH as well, kd-trees that produce a space partitionning are often preferred in photon mapping). The scene is finally raytraced from the camera (without making the ray bounce), and at each ray-scene intersection, nearby photons are collected using the acceleration structure, and density estimation is performed to estimate how much energy is reflected towards the camera. Density estimation can be performed by looking for a fixed number of neighbors and looking how far we need to look for these photons, or it can be performed by counting how many photons fall within a fixed search radius. This raytracing step is the *final gathering*.

Similarly to bidirectional path-tracing, launching photons from light sources allows to better capture phenomena like caustics, that are otherwise difficult to capture with (unidirectional) path-tracing.

2.2.3 Precomputed Radiance Transfer

Let's write the rendering equation without emissivity:

$$L_o(\omega_o) = \int_{\Omega} f(\omega_i, \omega_o) L_i(\omega_i) \langle \omega_i, N \rangle d\omega_i$$

We can easily decompose the different quantities on orthogonal basis functions defined on the (hemi-)sphere: $\{\mathcal{F}_k\}_k$. Let's denote the decomposition using hat symbols, and include the cosine term in the BRDF:

$$f(\omega_i, \omega_o) \langle \omega_i, N \rangle = \sum_k \hat{f}_{\omega_o}^k \mathcal{F}_k(\omega_i) \quad (2.19)$$

$$L_i(\omega_i) = \sum_k \hat{L}_i^k \mathcal{F}_k(\omega_i) \quad (2.20)$$



Figure 2.45: A nicer scene that includes an exponential fog, better colors for the walls and the ground, and the Davy Jones model that can be found at <http://www.cadnav.com/3d-models/model-45279.html>. Since there is no .mtl file, the textures (by number) should be loaded in that order: 2, 3, 11, 5, 1, 0, 9, 8, 6, 10, 7, 4. These textures include *alpha maps* (used in this rendering) that tell whether an intersection should be considered as opaque or transparent (it should be tested inside the ray-triangle intersection test), as well as *specular maps* (not used in this rendering) that give the ρ_s coefficient per pixel. Rendering time: 4 min. for 1000 spp.

With this decomposition, one can rewrite the above rendering equation:

$$\begin{aligned} L_o(\omega_o) &= \int_{\Omega} \sum_k \hat{f}_{\omega_o}^k \mathcal{F}_k(\omega_i) \sum_l \hat{L}_i^l \mathcal{F}_l(\omega_i) d\omega_i \\ &= \sum_k \sum_l \hat{f}_{\omega_o}^k \hat{L}_i^l \int_{\Omega} \mathcal{F}_k(\omega_i) \mathcal{F}_l(\omega_i) d\omega_i \end{aligned}$$

If the basis functions are orthogonal with respect to the inner product $\langle \mathcal{F}_k, \mathcal{F}_l \rangle = \int_{\Omega} \mathcal{F}_k(\omega_i) \mathcal{F}_l(\omega_i) d\omega_i$, this means that

$$L_o(\omega_o) = \sum_k \sum_l \hat{f}_{\omega_o}^k \hat{L}_i^l$$

In other words, one can easily compute the integral by just performing a scalar product between vectors of coefficients. It can become easy to use this technique for rendering, by precomputing tabulated values for the decomposition of a BRDF onto some basis functions and the decomposition of some incident lighting (e.g., computed using photon mapping, or modeled using an environment map, see Fig. 2.47 and 2.49), and performing the dot product in realtime.

Spherical Harmonics. Spherical Harmonics (SH) are commonly used orthogonal basis functions on the sphere (Fig. 2.48). They are analogous to the Fourier transform on the plane (eigenfunctions



Figure 2.46: Interior scene: (a) Traditional ray tracing. (b) Photon map. (c) Precomputed radiance estimates at 1/4 of the photon positions. (d) Complete image with direct illumination, specular reflection, and soft indirect illumination. *Fig. 5.2 of the SIGGRAPH 2002 course “A Practical Guide to Global Illumination using Photon Mapping”.*

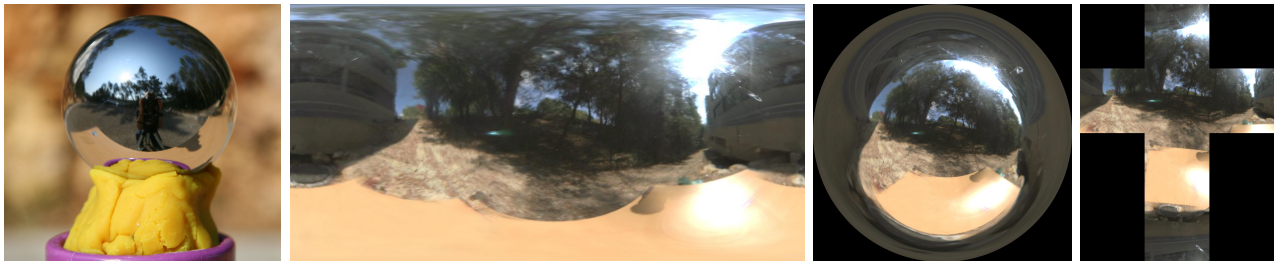


Figure 2.47: An environment map (or envmap) is simply a panoramic image representing the incident radiance at a point. It is often used for outdoor scenes since it well approximates distant illumination, and can be captured by mobile phone apps that stitch photographs into a panoramic image, or by taking photo(s) of a chrome ball (left). Here, the same environment map is shown with 3 different parameterizations: Latitude-Longitude, light probe, and cube map.

of the Laplacian operator are sine and cosines on the plane, and are spherical harmonics on the sphere ; if you are not sure about what is a Fourier transform, see Sec.??). They hence represent a frequency decomposition of the signal. Similarly to the Fourier transform, they possess a discrete version, that can be efficiently evaluated using Fast Fourier Transforms. Additionally, they possess *rotation formulas*: one can obtain the SH decomposition of a rotated version of the signal using a simple (block diagonal) matrix-vector multiplication of the SH coefficients of the original signal. This property can be useful for frame changes and interpolation. The $m = 0$ subset of SH are called *Zonal Harmonics*.

Finally, from the rendering equation expressed in term of dot product between SH coefficients, it becomes clear that a low frequency illumination over a high frequency (e.g., specular) surface will produce the same result than a high-frequency illumination over a low-frequency (e.g., diffuse) material – see Fig. 2.49. This is one reason why photographers use light diffusers: they will make skin more matte, and remove shiny reflections.

Spherical Wavelets. Spherical Harmonics have the same limitations that Fourier basis functions:

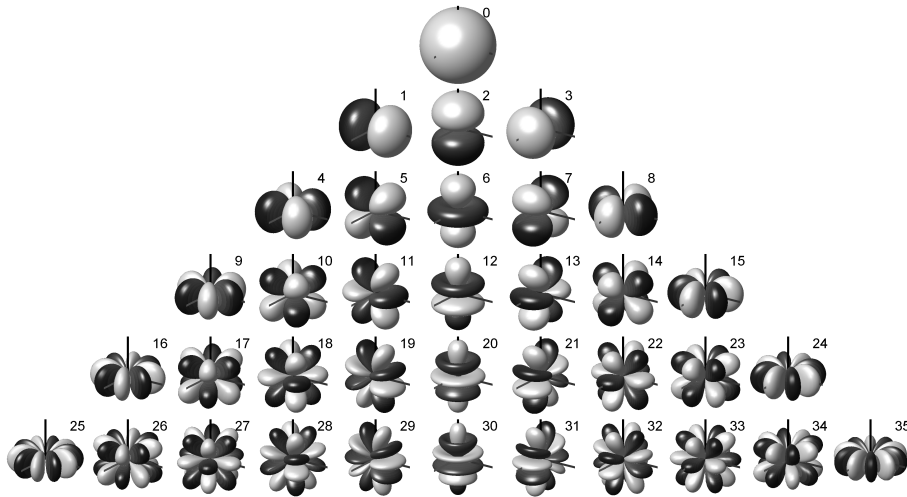


Figure 2.48: Spherical Harmonics up to degree 5 (Source Dr Franz Zotter, Wikimedia Commons).

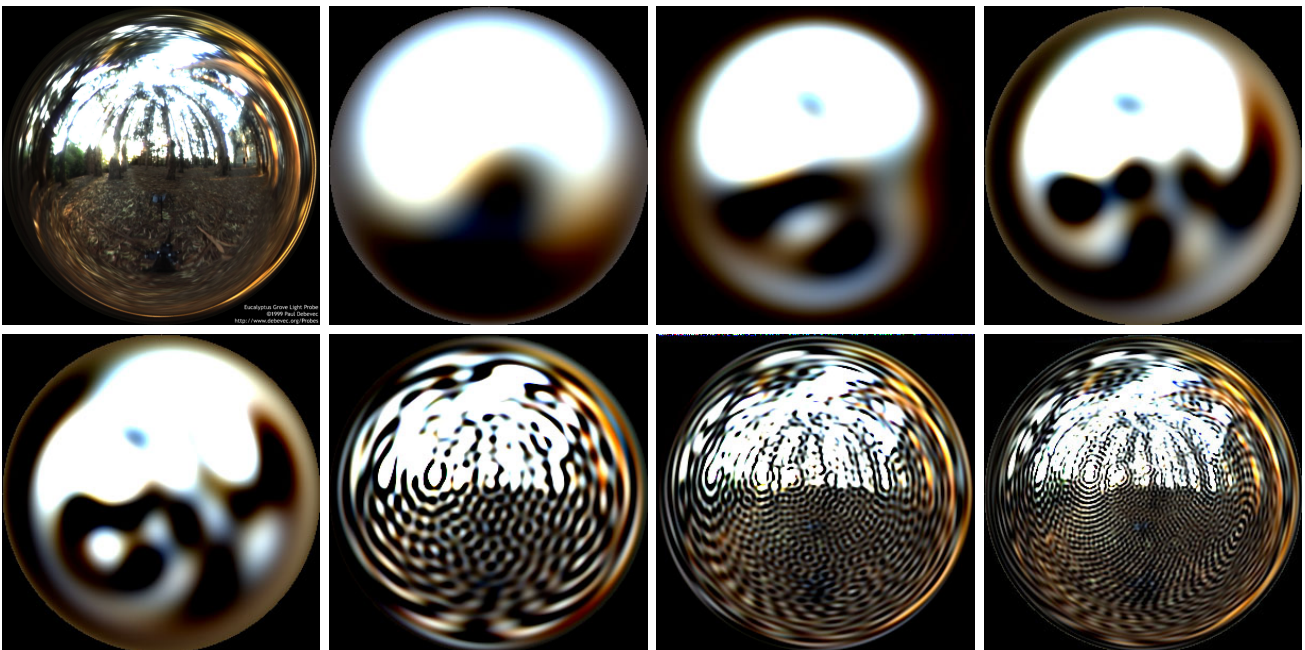


Figure 2.49: First image: input environment map, as a light probe. Other images: Progressively increasing the number of terms in the spherical harmonics decomposition increases accuracy. In that order: 3, 5, 6, 7, 30, 55, 80 spherical harmonic bands (N bands correspond to N^2 coefficients).

they are non-local, and tend to induce ringing artifacts when clamped abruptly. Compressing highly specular BRDFs with SH is thus not very efficient. In this context, wavelets that were introduced for image processing have been extended to work on the sphere. A simple Haar wavelet decomposition on the sphere can be obtained via successive triangulations of the sphere, filtering and differences. A detailed hands on introduction to spherical wavelets in matlab can be found in Gabriel Peyré's Numerical Tours: https://www.numerical-tours.com/matlab/meshwav_4_haar_sphere/.

2.2.4 Radiosity

In the special case of diffuse surfaces, with isotropic omnidirectional emissivity (L_e does not depend on ω_o), and assuming vacuum (then the incident radiance L_i is exactly the outgoing radiance L_o coming from another point, at equilibrium, and is simply our unknown denoted L) the rendering equation can be further simplified:

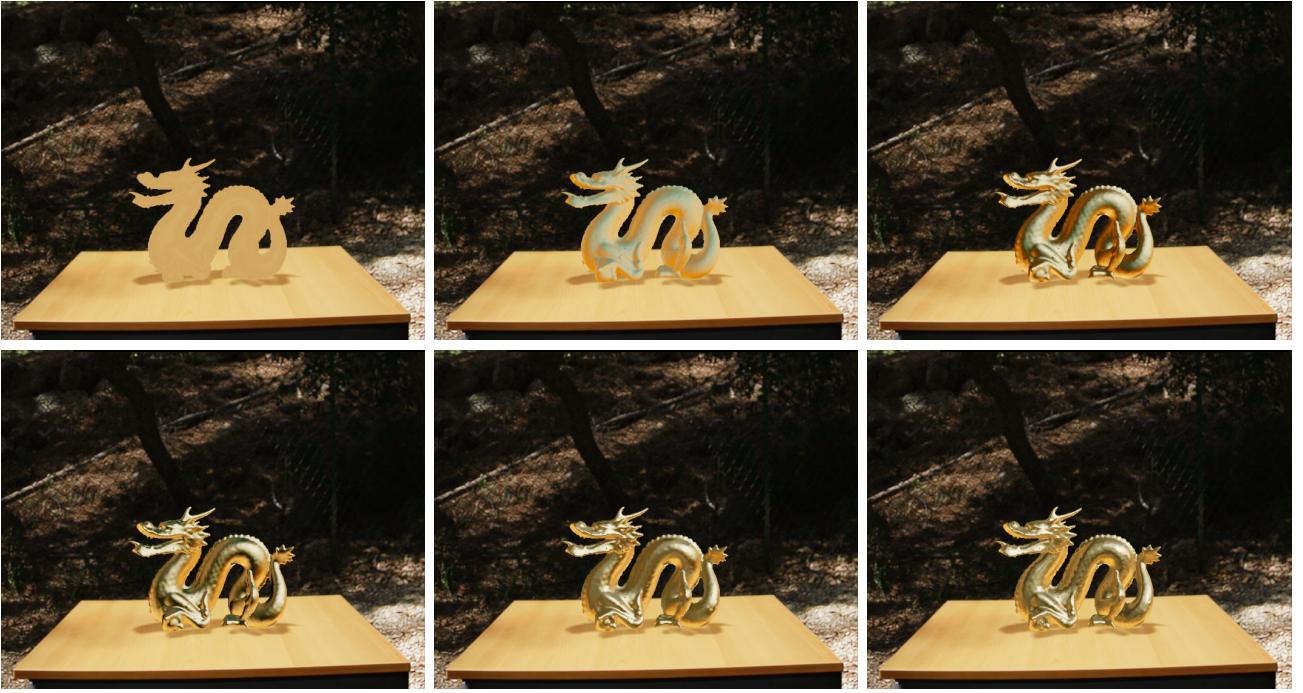


Figure 2.50: Progressively increasing the number of spherical harmonic bands to represent a gold BRDF makes it more shiny. In that order: 1, 2, 3, 4, 6, 12 SH bands (N bands correspond to N^2 coefficients).

$$L(x) = L_e(x) + \frac{\rho(x)}{\pi} \int_{\Omega} L(x, \omega_i) \langle \omega_i, N \rangle d\omega_i$$

Notice how the result does not depend on any direction: one can freely navigate in the scene without needing to recompute anything. We will rewrite the rendering equation so as to integrate over the scene surface elements rather than directions, as we did in Sec. 2.2.1:

$$L(x) = L_e(x) + \frac{\rho(x)}{\pi} \int_S L(x, \omega_i(x')) G(x, x') dx'$$

with $G(x, x') = \langle \omega_i(x'), N \rangle \frac{\langle N', -\omega_i(x') \rangle V_x(x')}{\|x - x'\|^2}$ the *form factor* we talked about earlier in Sec. 2.2.1.

The idea is to decompose again the unknown radiance L onto basis functions. Typically, either constant or piecewise linear functions are used per triangle of the mesh. For instance, using constant basis functions per triangles, and denoting B_k the basis functions which is 1 over triangle k and 0 elsewhere, we can rewrite the above expression in this basis as:

$$L^k = L_e^k + \frac{\rho^k}{\pi} \sum_l L^l G^{k,l}$$

This yields a particularly simple linear system, written in matrix/vector form:

$$L = L_e + \text{diag}\left(\frac{\rho}{\pi}\right)GL$$

and by rearranging terms:

$$L = \left(\text{Id} - \text{diag}\left(\frac{\rho}{\pi}\right)G\right)^{-1} L_e = M^{-1}L_e$$

Solving linear systems in general is out of the scope of this class²³. However, a particularly simple approach is to use Jacobi iterations, that read at iteration $n + 1$:

$$L^{i,n+1} = \frac{1}{M_{i,i}} \left(L_e^i - \sum_{j \neq i} M_{i,j} L^{j,n} \right)$$

where $L^{i,n}$ is the radiosity at triangle i and iteration n , and converges to the true solution L^i as $n \rightarrow \infty$. It happens that each additional Jacobi iteration simulates one new light bounce.

The last detail I did not mention is how to compute the matrix G . This matrix (assuming piecewise constant basis functions) has coefficients $G^{i,j} = \int_{T_i} \int_{T_j} G(x, x') dx dx'$ where the integration is over all pairs of triangles. Since $G(x, x')$ includes a visibility term, there is no real hope to have a closed form expression in the general case: this integral is performed by sampling pairs of points, computing the visibility term by raytracing, and evaluating the integral using Monte Carlo integration. To generate uniformly random points within a triangle (with pdf $p(x)1/\text{area}$), one can again rely on the *Global Illumination Compendium*:

$$r_1, r_2 \sim \mathcal{U}(0, 1) \tag{2.21}$$

$$\alpha = 1 - \sqrt{r_1} \tag{2.22}$$

$$\beta = (1 - r_2)\sqrt{r_1} \tag{2.23}$$

$$\gamma = r_2\sqrt{r_1} \tag{2.24}$$

$$\tag{2.25}$$

with α , β and γ the barycentric coordinates of the sampled point. A radiosity result can be seen in Fig. 2.51. More recent approaches allow for glossy materials²⁴.

2.3 Discussion

Nowadays, most work on radiosity has been abandoned: this approach is most often costly and (almost) limited to diffuse scenes, but mostly, highly dependent on the mesh quality. Rendering a large diffuse flat wall cannot be done with a single quadrilateral (or two triangles) but many triangles that would ideally align with cast shadows (a few approaches try to progressively refine the mesh where needed).

Also, offline, costly path-tracing and real-time GPU rasterization tend to get inspired by each other. A couple of game engines start to integrate path-traced effects on the GPU for rendering specular or transparent objects, mostly by using very very few samples per pixel combined with clever filtering tricks (e.g., using deep learning). Conversely, it can be sometimes useful to rasterize the first bounce of a path-tracer since the first intersection between camera rays and the scene can often be found directly by rasterizing (if no depth of field effect is desired).

²³You can see a couple of slides I wrote at <https://projet.liris.cnrs.fr/origami/math/presentations/matrices.pdf>

²⁴*Implicit visibility and antiradiance for interactive global illumination*, <https://hal.inria.fr/inria-00606794/PDF/ImplicitVisibilityAndAntiradiance.pdf>

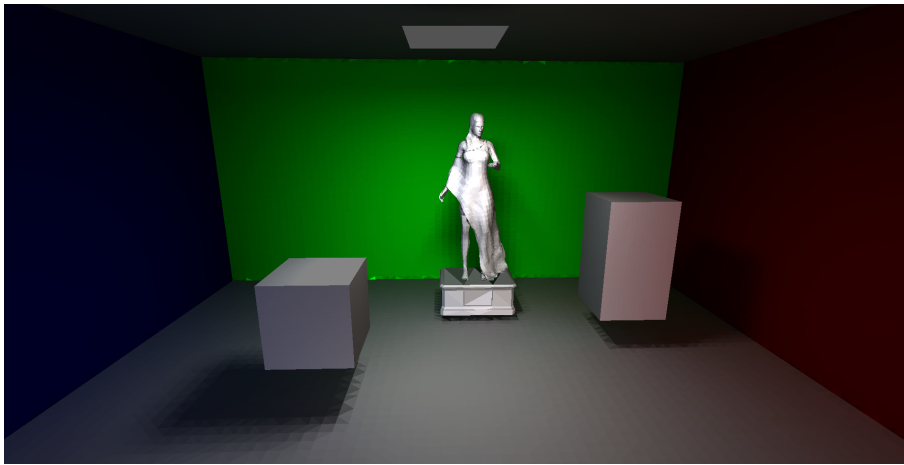


Figure 2.51: Radiosity result – my (very naive) implementation has 100 lines for building and solving the linear system + 450 lines for defining basic classes (`Vector`, `Triangle`, `Mesh`...), reading obj files, constructing the BVH, intersecting. There are 10 light bounces (i.e., Jacobi iterations), 62 892 triangles and piecewise constant basis functions. The entire matrix M is densely stored so it is huge in memory (about 100GB in total for one matrix M per RGB color channel) – much better strategies exist – and the computing time is a few hours. The mesh is available at <https://perso.liris.cnrs.fr/nbonneel/radiositymesh.obj> – triangles with `group==3` are emissive.