Computer Graphics – CSE 306
École Polytechnique

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

Last update: March 27th, 2024.

Contents

This class will cover several aspects of computer graphics, including overviews of rendering, geometry, simulation and image processing. A selected few aspects will be discussed in more details as they

will contribute to labs, where you will need to implement them **from scratch** in C++. Little code will be provided: these aspects will need to be fully understood. I do consider that nothing is fully understood until you can implement from scratch, and conversely, once understood, coding is merely a matter of careful touch typing. In return, you will get the satisfaction of having implemented your own tools producing beautiful computer graphics results.

These labs will include a path-tracer (Sec. 2.1.2, 4 labs), an image retargeting algorithm (Sec. 3.3, 1 lab) or sliced optimal transport color matching (Sec. 3.2.2), a fluid simulator that uses Voronoï diagrams (Sec. 4.3 and Sec. 5.4, 3 labs), and a mesh parameterization algorithm from Tutte (Sec. 4.6.1).

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 more 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, sometimes just resulting in significant slow down or spurious noise, and they will thus become hard to track down ; when implementing a fluid solver, errors such as off-by-one indices typically result in completely wrong simulations but are also hard to track down due to the number of indices in the code.

However, bugs happen. Make sure you master a real debugger, preferably with an IDE, and know 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).

Regarding languages, I highly encourage C++ in our class, because it is fast, and I will provide code snippets and support for this langage. If some students have a strong expertise in another langage and are not comfortable with C++, these projects could also be implemented in other langages. However, I **strongly** advise against scripted languages such as Matlab or Python, particularly if you are not expert in them, as they are extremely slow. A naive path tracer implemented in C++ would take a few seconds to run when its Python equivalent would take several hours: you will not be able to debug and experiment with your codes.

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 relatively good desktop computer, though from 2018) 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. I also believe that code length is a good indicator 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 ! With 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, a libLBFGS library for nonlinear optimization (`https://github.com/chokkan/liblbfgs`), and I will provide code to read .obj mesh files, write .svg files or rasterize polygons. Unless you want to go further (e.g., adding a GUI or using fast nearest neighbor search), you will not need other codes.

To get started and be able to quickly jump into the first lab without wasting time, I ask students to come prepared with a properly configured compiler and GUI, and to compile the following code that will be the starting point of our first project: `https://pastebin.com/Qpbw0Q9t`, relying on stb_image and stb_image_write (see above). This code, once compiled, should produce an entirely red image, and it also contains the (very) basic and preliminary structure of our project 1.

1.2 Bitmap Image Representation

A bitmap image considers that an image is a 2d array of pixels. In our case, notably for project 1, each pixel is a triplet of red (R), green (G) and blue (B) values. For implementation purposes, 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:

```
image[y*W*3 + x*3 + 0] = red_value;
image[y*W*3 + x*3 + 1] = green_value;
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 (see Fig. 2.18). Certain applications require multispectral images consisting of multiple (>3) sampled wavelengths (e.g., additional infra-red channels) or including transparency (an additional *alpha* channel), or may store floating point values.

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 ways to fill them (e.g., gradients). Vector images may 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. For instance, 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 by placing each of them in a single `<g> ... </g>` pair. All parameters in an svg file can be animated.

I uploaded an svg writer in C++ that saves polygon soups and supports animations (just call the `save_svg_animated` repeatedly, once for each frame of your animation) at `https://pastebin.com/bEYVtqYy`. To describe frame-by-frame animations, this code superimposes all polygons of all frames in the same image, grouping all polygons that belong to the same frame, and animating the visibility parameter of each group (visibility is the "display" attribute of a shape).

⚠ SVG animations created this way are extremely slow in most browsers and you will not be able to load svg files containing even a few hundreds frames of a few thousands of polygons. For your fluid simulation project, I uploaded here a code that "rasterizes" polygons (i.e., convert them from vector representation to bitmaps) and saves a bitmap (it will only work for convex polygons, like ours): `https://pastebin.com/jVcNAE5Q` . That will also allow you to combines these animation frames into videos using lightweight easy-to-use dedicated tools (e.g., VirtualDub or ffmpeg).

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 langages designed for graphics cards, such as GLSL or HLSL. Theses langages 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 refering 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.

```
1 class Vector {
```

```
public:
  explicit Vector(double x = 0., double y = 0., double z = 0.){
   coords[0] = x;
   coords[1] = y;
   coords[2] = z;
  };
  Vector& operator+=(const Vector& b) {
   coords[0] += b[0];
   coords[1] += b[1];
   coords[2] += b[2];
   return *this;
  }
  const double& operator[](int i) const { return coords[i]; }
  double& operator[](int i) { return coords[i]; }

private:
  double coords[3];
};
Vector operator+(const Vector& a, const Vector &b) {
   return Vector(a[0] + b[0], a[1] + b[1], a[2] + b[2]);
}
double dot(const Vector& a, const Vector& b) {
   return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

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

```
Vector myVector1(1., 2., 3.);
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.)` (which, given the default parameter list, would translate to adding `Vector(1., 0., 0.)`, and would result in `result = Vector(2., 2., 3.)`). 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 for form triangles. The most common structure to store meshes consists in an array of vertices, and an array of triangular faces referencing these vertices. Often, additional informations are 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 refering to the previous array. As in most cases other geometric informations are 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:

```
struct TriangleIndices {
   int vtxindices[3];    // refers to 3 indices in the vertices array of the class ↩
       Mesh
   int normalindices[3]; // refers to 3 indices in the normal array of the class Mesh
   int uvindices[3];     // refers to 3 indices in the uv array of the class Mesh
};
class Mesh {
```

```
public:
 // ...
private:
  std::vector<Vector> vertices;
  std::vector<Vector> normals;
  std::vector<Vector> uvs;
  std::vector<TriangleIndices> triangles;
};
```

The .obj file format encodes this structure as an ASCII file. 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 so far, 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 not discuss real-time rendering as teaching this kind of materials either requires an extensive dedicated course involving complex low-level libraries (e.g., either OpenGL/Vulkan or DirectX) or reimplementing them for educational purposes (e.g., implementing a rasterizer with a *z-buffer*), or involves manipulating rendering engines that already implement the interesting part (e.g., Unreal Engine, Unity, Amazon Lumberyard). Nowadays, physically-based rendering and real-time rendering techniques tend to converge, with a few games implementing path-tracing (e.g., Cyberpunk 2077) or with physically-based rendering engines being accelerated on the GPU (graphics card).

However, in addition to physically-based rendering, we will briefly cover recent advances in image-based rendering, with techniques like Neural Radiance Fields (NeRF) and Gaussian splatting, that produce realistic renderings based on photographs of a scene.

2.1 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.

2.1.1 Realism

Realism can be important in computer graphics, for instance, to avoid falling in the (debated) *Uncanny Valley*, i.e., the state where a digital or robot human is quite realistic but not photorealistic enough, which makes it look zombie-like and creepy (Fig. 2.1). However, there are multiple notions of realism[1]. There is **functional realism**, where the scene is depicted such that a task can be performed well. For instance, to build a piece of IKEA furniture, it is much easier to deal with simplified drawings than actual photos of the furniture. There is **physical realism**, where care is taken to faithfully simulate reality. This is where real-life applications come in, such as architecture or lighting design. Note that many lighting simulation techniques for computer graphics can be adapted for the physical

[1] *Three Varieties of Realism in Computer Graphics*, by Ferwerda

simulation of sound propagation, or to simulate invisible wavelength radiations. And finally, there is **photo-realism**, where the goal is to look as close as possible to a photograph, possibly taking into account perception (e.g, rendering flares, or ignoring invisible details, or cheating because the brain is mostly incapable of detecting inaccurate from accurate physics). We will mostly address physical realism here.

Figure 2.1: Uncanny valley for renderings in movies (Tintin and Polar Express) and robots.

2.1.2 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 Helmoltz 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 posess 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 occured 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 posess a similar function.

2 Scene For reproducibility purpose, we can define a standard scene as in Fig. 2.2, 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 is standing upright and looking at the $-z$ direction.

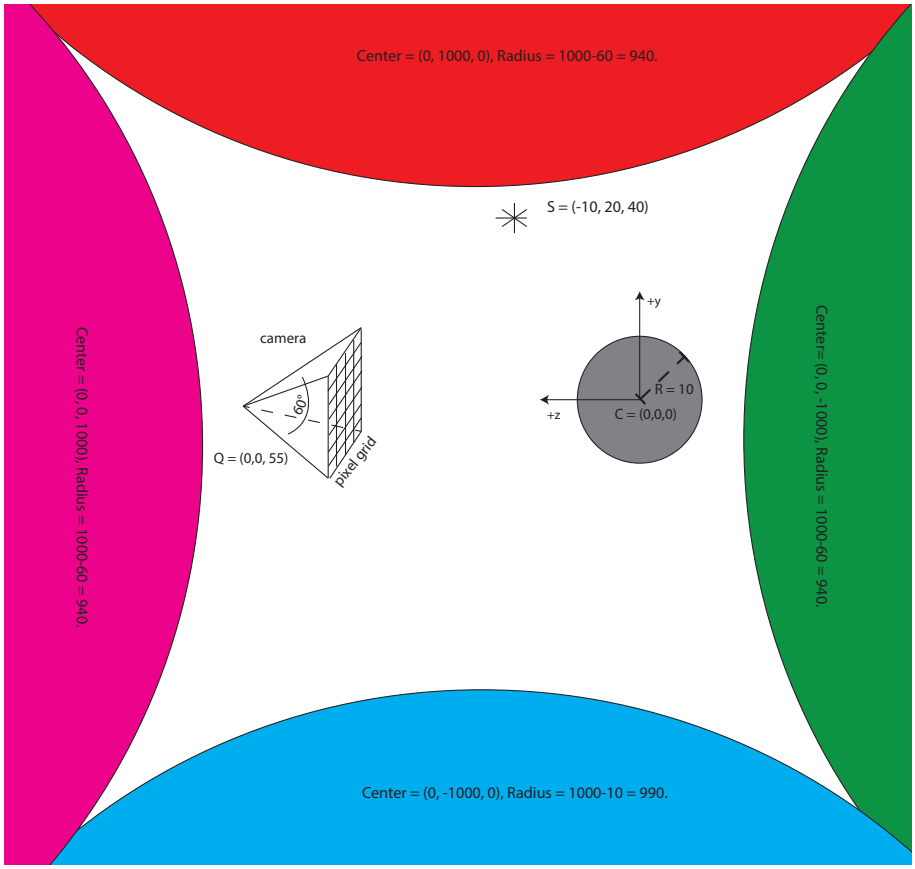


Figure 2.2: We define a standard scene that consists in walls, a ground and a ceiling, all consisting of gigantic spheres approximating planes. We also add a center sphere, which we will focus on as a first step.

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 2.1.2), see Fig. 2.3. 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 $\tan(\alpha/2) = (W/2)/f$ in such a way 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 $X(t) = O + t\,u$, 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 $\|O + t\,u - C\|^2 = R^2$. Expanding the squared norm and using scalar product bilinearity yields $t^2\|u\|^2 + 2\,t\,\langle u, O - C\rangle + \|O - C\|^2 = R^2$. Assuming unit norm for u leads to the simple quadratic

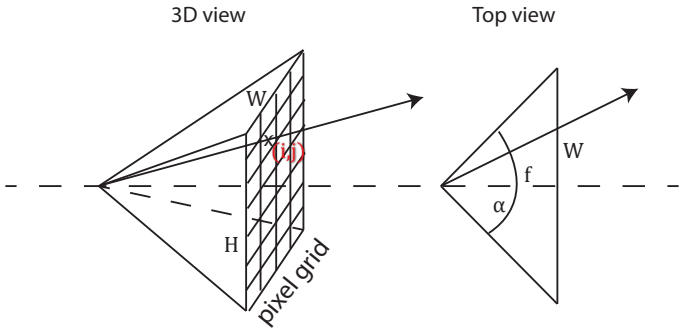


Figure 2.3: Notations for a virtual camera.

equation:

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

A quadratic equation has 0, 1 or 2 real solutions depending on the discriminant, which has geometric interpretations here (see Fig. 2.4). 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. 2.5 (considering only the central sphere of our standard scene in Fig. 2.2).

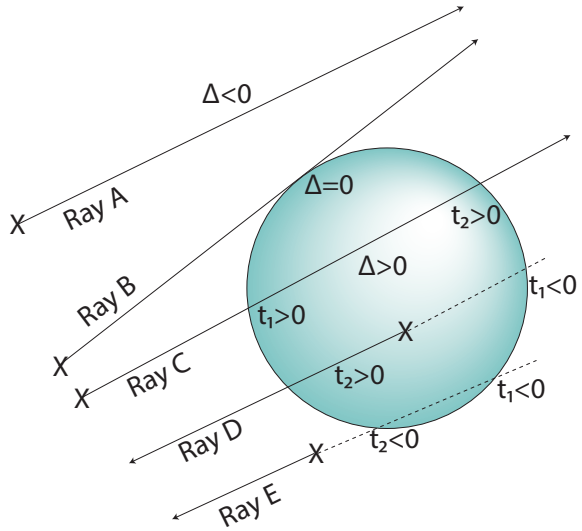


Figure 2.4: Ray-Sphere intersections lead to solving a quadratic equation. Depending on the sign of the discriminant, this leads to either 0, 1 or 2 points of intersection. Here, ray B leads to one (double) intersection, ray C produces a first intersection of interest at t_1, and ray D produces the intersection 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.

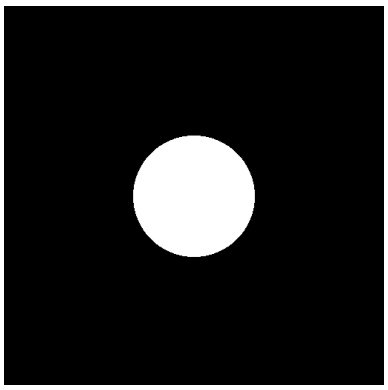


Figure 2.5: Computing the ray-sphere intersection at each pixel leads to our first image. Ok, that's just a plain white disk, don't be *too* excited.

Remark. An easy way to debug is to re-order your spheres when creating the scene. The rendering should remain exactly the same after re-ordering. If this is not the case, then you have a bug necessarily in your ray-scene intersection. If the scene looks odd but is stable after reordering, then your bug is more likely in your ray-sphere intersection test.

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 source[2], 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. 2.1.2) if $\int_{S^+} c.\langle N, \omega_i \rangle \mathrm{d}\omega_i \leq 1$ for some normalization constant c, where S^+ is the hemisphere above the surface. Since $\int_{S^+} \langle N, \omega_i \rangle \mathrm{d}\omega_i = 4\pi$, $c = 1/(4\pi)$.

⚠ Due to numerical precision issues, you will certainly observe extreme noise levels (see Fig. 2.6). 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 (i.e., P may well be a tiny epsilon *below* the surface, and launching a way from P will result in the surface being intersected again). The solution to this issue is to lauch 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

[2]These *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 to speed up your code.

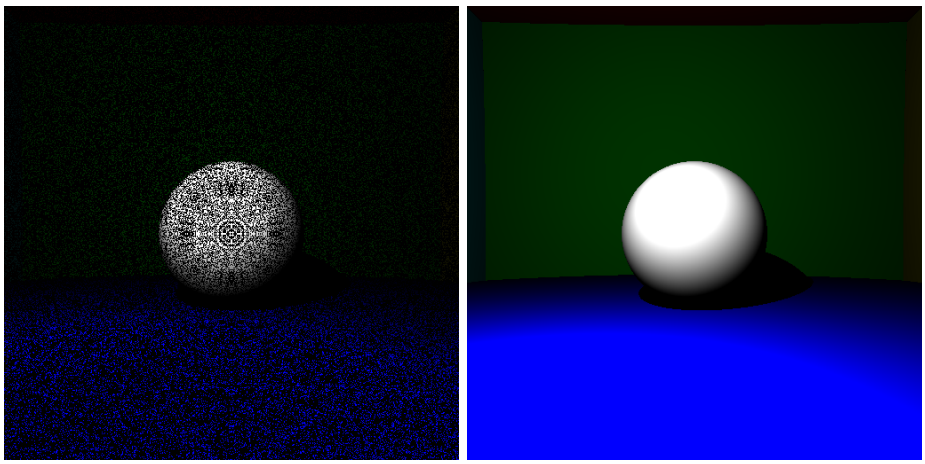


Figure 2.6: Due to numerical precision issues in shadow computations the image appears noisy (left). Launching rays from an offseted origin solves this issue (right).

too dark (Fig. 2.7). 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 higher accuracy for 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 result in *wraparound*. You can see the result of gamma correction on our test scene in Fig. 2.8. We call the color space where light physics happen *linear color space* while after gamma correction, colors end up in the *gamma color space*.

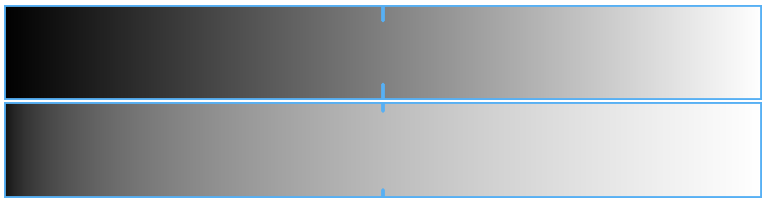


Figure 2.7: A linear ramp (top) and gamma-corrected linear ramp (bottom, with $\gamma = 2.2$). The linear ramp's midpoint appears too dark. Note that perceived results may vary depending on specific screen settings.

⚠ A common bug is to gamma-correct or clamp intensity values each time a ray of light bounces (see next), or, when averaging intensities, gamma-correcting before taking the average, which is not correct. This typically results in lack of contrasts. Gamma-correction or clamping compensates 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 within the light simulation process.

Adding reflections and refractions

Reflections. Contrary to Lambertian diffuse 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

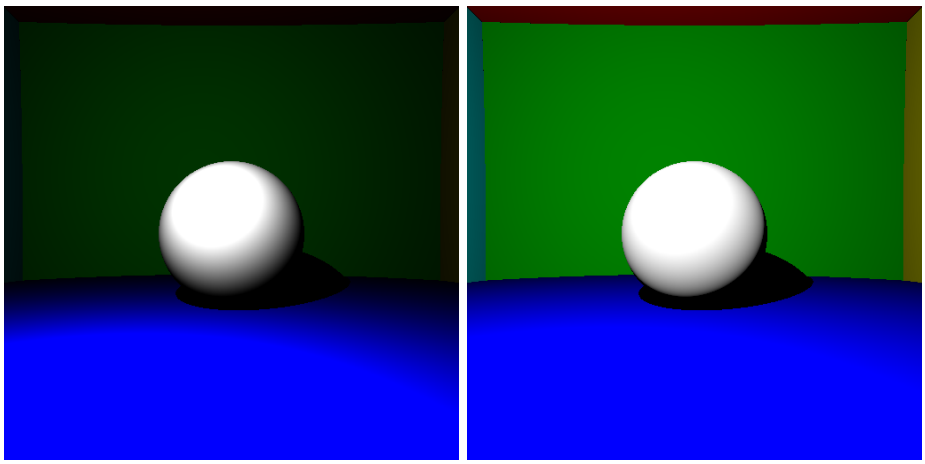


Figure 2.8: Without gamma correction, the scene appears too contrasted (here, $I = 2.10^7$). With gamma correction (and $I = 2.10^{10}$), the scene appears more natural. At this stage, we have roughly 170 lines of (verbose) code which runs in 50ms without parallelization (see end of Sec. 2.1.2) for a 512x512 image.

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. 2.9). A perfect mirror thus only transfers light energy from the incident direction to the reflected direction.

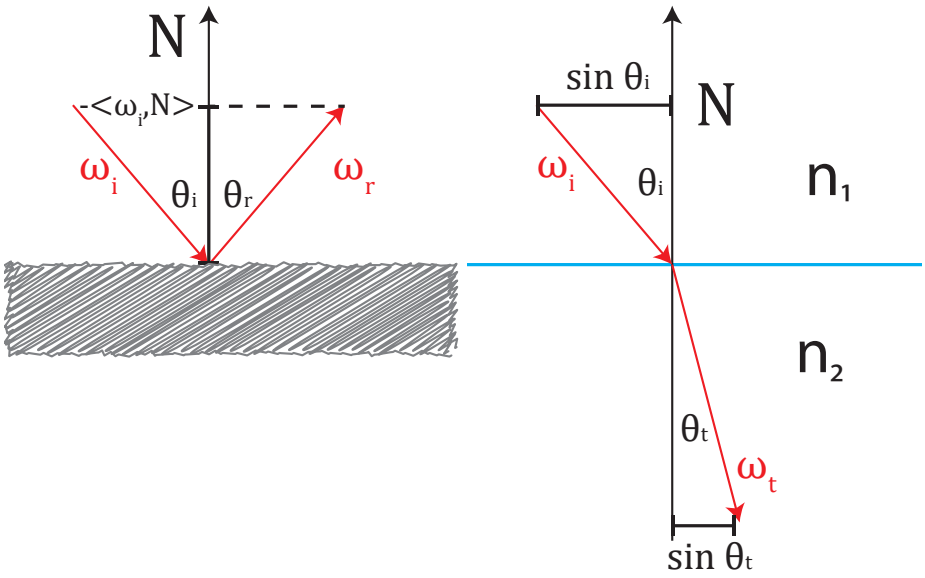


Figure 2.9: The reflected direction is $\omega_r = \omega_i - 2\langle \omega_i, N \rangle N$ (left) and refracted direction (right).

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 ↩
          channel
24    // same for green and blue
25  }
26 }
27 // save image and return 0
28 }
```

Implementing it iteratively rather than recursively is also easy to do, and the exercise is left to the reader.

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

⚠ A common bug is to compute a visibility term that produces shadows over the reflected color of the mirror. You should not do it. Visibility is a shadowing term that refers to specific light sources, and casts shadows for direct lighting. Mirrors merely reflect light coming from any direction, not just that of our point light sources, and whether a light source is visible from the point of view of the mirror surface is not relevant.

Refractions. The case of transparent surfaces is very similar to that of mirrors. For transparent objects, rays also continue their lives by bouncing off surfaces, but this time, passing through it. The computation of the transmitted 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 deduct 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 $1 - \left(\frac{n_1}{n_2}\right)^2 \left(1 - \langle \omega_i, N \rangle^2\right)$ 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 towards the correct side! More generally, when handling 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. Doing so, when the ray will enter the inside sphere, the test for the sign of $\langle \omega_i, N \rangle$ will consider that we are actually leaving the sphere (although we are entering it), and will thus consider that the "outside" is made of a transparent refractive material and the "inside" is air (doing so, we have actually produced a negative sphere inside the normal sphere: a sphere that has air inside, and some refractive material outside). A result showing reflection and refraction on a full and hollow sphere is shown in Fig. 2.10. Also, ideally, the index of refraction should depend on the wavelength (we usually take $n(\lambda) = A + \frac{B}{\lambda^2}$ for some A and B). To achieve this effect of dispersion, we would throw rays of a single wavelength, combining them on the sensor ; we will not do that here.

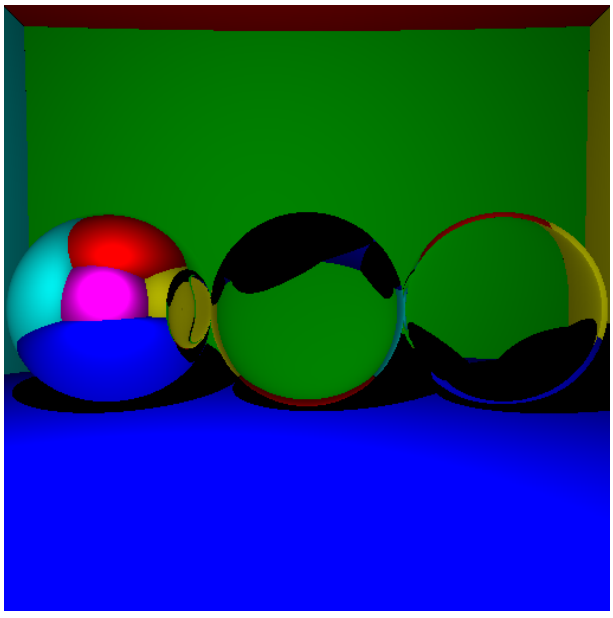


Figure 2.10: A sphere with reflection, a full sphere with refraction, and an hollow sphere with refraction. Notice how the full sphere inverts the scene behind as it acts as a lens. The refraction index used is 1.5, corresponding to glass. The image is computed in 75ms (without parallelization) with 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 depend on light polarization, and one often instead relies 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. In practice, this means that a ray is both partially reflected and partially refracted. 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, we just consider that `Scene::getColor(incoming_ray)` exactly returns `Scene::getColor(refracted_ray)` or `Scene::getColor(reflected_ray)` depending on the random 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 light paths, and average the resulting colors. This scheme will be further discussed along with Monte Carlo integration next, in Sec. 2.1.2.

⚠ 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 of all light sources at the same time. It becomes interesting for the random value to account for the intensity or distance to light sources, and reweight light contributions accordingly – this will become clearer when we will implement Monte Carlo integration and indirect lighting.

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. 2.12). 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 result of `Scene::getColor`) is:

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

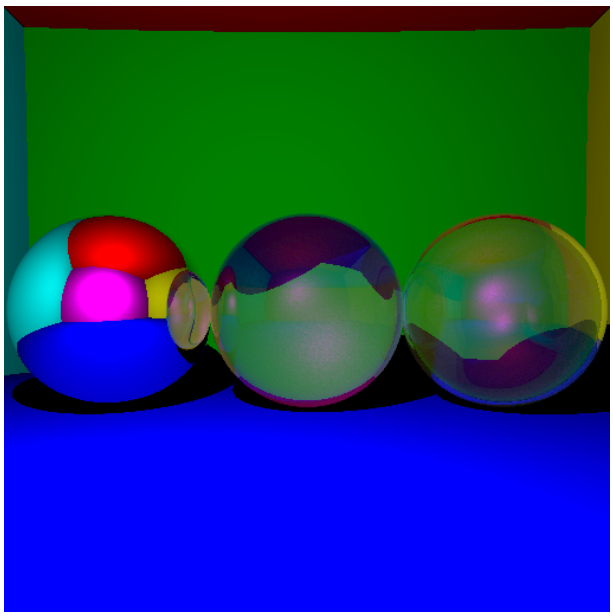


Figure 2.11: Same as Fig. 2.10 but with Fresnel reflection taken into account on transparent surfaces. For this image, I took 1000 rays per pixel, which resulted in a rendering that took about 1 minute (without parallelization, and about 260 lines of code).

Figure 2.12: 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`.

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* spectral radiances L_i from all directions of the hemisphere Ω at point 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 see in Fig. 2.13.

It is interesting to see that the incoming light L_i at point x from direction ω_i is exactly the outgoing light L_o at a point x' from direction $-\omega_i$, assuming vacuum in-between (we will see in Sec. 2.1.2 how to handle *participating media*, where this light can get attenuated or scattered when it goes from x' to x). 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 by replacing L_i by its expression calculated at point

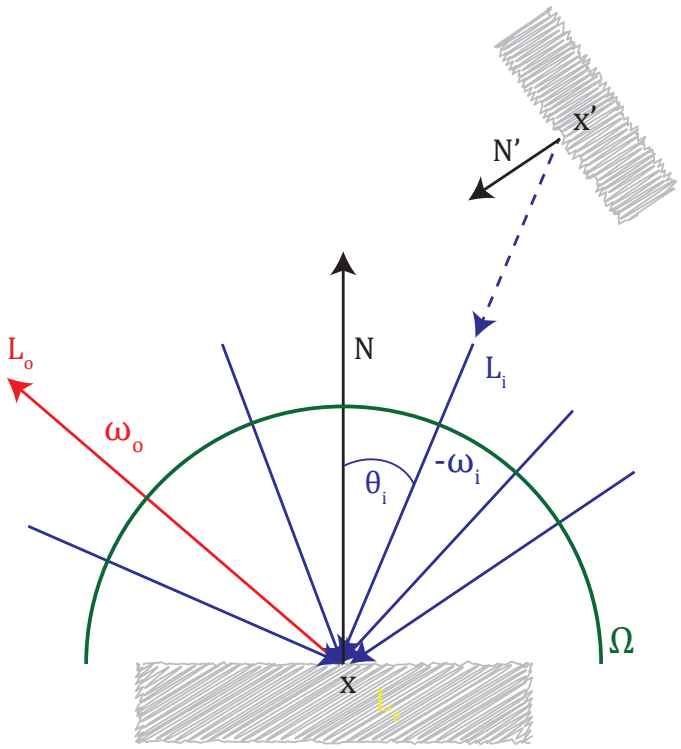


Figure 2.13: Notations for the Rendering Equation. Note that from now on, we denote by convention ω_i a vector that points **outwards** the surface, similarly to ω_o. Since this mostly influences dot product signs, this is usually understood from context.

x' as

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

and recursively, the light $L_i(x',\omega_i')$ reaching point x' comes from other locations x'' 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..∞ bounces, that needs to be performed numerically. This corresponds to a sum of integrations problems over Ω, $\Omega \times \Omega$, $\Omega \times \Omega \times \Omega$ etc.

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. 2.14). Conditions for their physical meaningfulness are that they are positive ($f \geq 0$), they respect Helmoltz reciprocity principle, that is, they are symmetric ($f(\omega_i,\omega_o) = f(\omega_o,\omega_i)$)[3] and preserve energy, that is $\int_\omega f(\omega_i,\omega_o)\langle\omega_i,N\rangle \mathrm{d}\omega_i \leq 1, \forall\omega_o$[4]

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 dataset[5] (see Fig. 2.15 ; note that isotropic BRDFs can be reparameterized using only 3 dimensions, θ_i, θ_r, ϕ_d instead of 4 angular values θ_i, ϕ_i,

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

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

[5]`https://www.merl.com/brdf/`

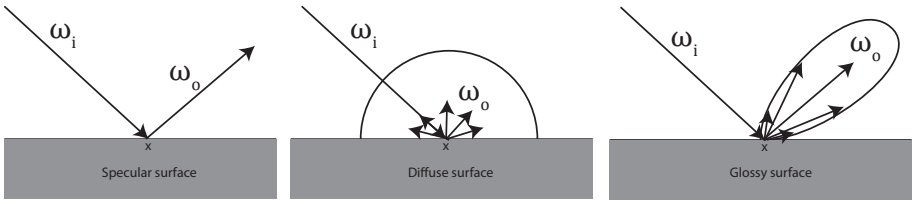


Figure 2.14: Typical BRDFs.

θ_r, ϕ_r – a parameterization called Rusinkiewicz parameterization), Ngan's 4 anisotropic BRDFs[6], and UTIA 150 anisotropic BRDFs[7]. 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. 2.1.4.

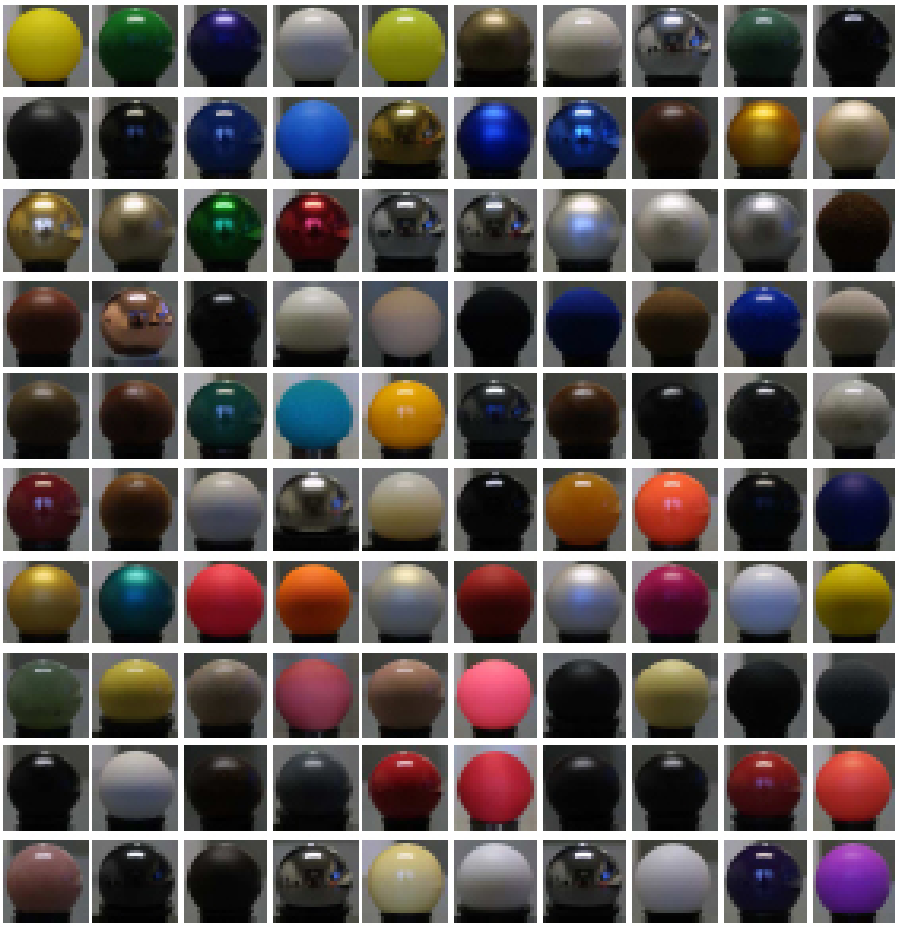


Figure 2.15: BRDFs from the MERL dataset.

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. 2.1.2) – or derived from microgeometry analysis assuming microfacet models (e.g., GGX, Cook-Torrance, Oren-Nayar, Torrance-Sparrow, Ashikhmin-Shirley, He et al., ...). BRDFs can be generalized to different settings. For instance, BSDF (Bidirectional Scattering Distribution Functions) do not consider a distribution over the hemisphere of directions defined by opaque surfaces, but the sphere of directions in the context of transparent surfaces. Scattering or phase functions similarly describe how light behave

[6] `https://people.csail.mit.edu/addy/research/brdf/`
[7] `http://btf.utia.cas.cz/?brdf_dat_dwn`

when it reaches a small particle (of dust, of water etc.) as a function of the incoming and outgoing light directions on the sphere (see Sec. 2.1.2). The SVBRDF, or spatially varying BRDF, is akin to a texture that stores a full (often very compressed) BRDF at each pixel. The BSSRDF, or Bidirectional Subsurface Scattering Distribution Function adds as a parameter where (or how far) an incoming ray of light will exit off the surface. This now assumes that a ray of light arriving at a point x will be scattered inside the surface and leaves the surface at another point $\tilde{x}$ on the surface. This is typically used to render skin, milk, marble or leaves, where light tends to somewhat penetrate the surface.

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. 2.1.2, $f_t(\omega_i, \omega_o) = \delta_{\omega_t}(\omega_o)$ with ω_t the transmission of ω_i inside the surface of normal N as we have seen in Sec. 2.1.2, and $f_d(\omega_i, \omega_o) = \frac{c}{\pi}$ the diffuse BRDF as in Sec. 2.1.2. 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. 2.1.2 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 polynomial. 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 is coarse and would likely miss important high frequency features).

To alleviate this issue, Monte Carlo integration has been proposed as a way to stochastically estimate integrals. This technique has been historically developped 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 terms as:

$$\int_\Omega f(x)\mathrm{d}x \approx \frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)}{p(x_i)}$$

where the x_i are random samples following the probability density function p. This converges to the true integral assuming $p > 0$ wherever $f \neq 0$ as the number of samples n tends to infinity. The intuition is that you can give a given sample half the probability of occurring, but then you need to compensate and count it twice to remain consistent. However, this process converges slowly: the integration error decreases in $\mathcal{O}(1/n^{0.5})$.

Importance sampling. 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

$$\int_\Omega f(x)\mathrm{d}x \approx \frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)}{\alpha p(x_i)} \tag{2.2}$$

$$= \frac{1}{n}\sum_{i=1}^{n}\frac{1}{\alpha} \tag{2.3}$$

$$= \frac{1}{\alpha} \tag{2.4}$$

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 because you already know the value of the integral! 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

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

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 gives 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:

```
#include <random>
static std::default_random_engine engine(10); // random seed = 10
static std::uniform_real_distribution<double> uniform(0, 1);

void boxMuller(double stdev, double &x, double &y) {
      double r1 = uniform(engine);
      double r2 = uniform(engine);
      x = sqrt(-2 * log(r1))*cos(2 * M_PI*r2)*stdev;
      y = sqrt(-2 * log(r1))*sin(2 * M_PI*r2)*stdev;
}
```

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 integration domain is compactly supported although our Gaussian density can produce samples at any point in space: a solution is to consider that the function we are integrating extends infinitely by that its value is 0 outisde of the chosen integration domain. The exact value is close to 24.3367. With 10000 samples, you should at least get the 24 part correct...

⚠ The resulting code should only have **1 `for` loop**, and **not** 3 nested loops, looping over x, y and z (which would be the case 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 directly proportional to the reflection and refraction coefficients!

Quasi-Monte Carlo. One reason why Monte Carlo converges slowly is because indendently drawn random points are *too random*, i.e., they may agglutinate or leave large gaps, even though the distribution is uniform when drawing sufficiently many random points. Instead, it is possible to use sets of deterministic points that are *more uniformly distributed* by construction. To measure the uniformity of a point set X of n points in d dimensions, the (extreme) discrepancy of that point set

can be computed as

$$D_n(X) = \sup_{B \in \mathcal{B}} \left| \frac{A(B,X)}{n} - \text{area}(B) \right|$$

where $\mathcal{B}$ is the set of all d-dimensional axis-aligned boxes, and $A(B,X)$ counts the number of points from X falling in the box B. It gives the worst possible integration error when using a Monte Carlo estimator using these samples for integrating any indicator function of an axis aligned box. Or intuitively, it detects if there are rectangular regions that are too coarsely or too densely sampled. From this value (that can be algorithmically computed), one can estimate the convergente rate of the numerical integration process via the Koksma-Hlavka's inequality:

$$\left| \int_{[0,1]^d} f(x) \mathrm{d}x - \frac{1}{n} \sum_{i=1}^{n} f(x_i) \right| \leq V_{HK}(f) D_n(X)$$

where $V_{HK}(f)$ is the Hardy and Krause variation, which somewhat characterizes the smoothness of f. Sequences of points whose discrepancy D_n behaves in $\mathcal{O}(\frac{(\log n)^d}{n})$ exist and are qualified of *low-discrepancy sequences*, such as Sobol' sequences. Using them allows to improve the convergence rate of Monte Carlo integration from $\mathcal{O}(n^{-0.5})$ to $\mathcal{O}(\frac{(\log n)^d}{n})$. Different point sets whose convergence rate have been studied by Fourier analysis are shown in Fig. 2.16. For our path tracer, we will stick to independent and identically distributed random samples for simplicity.

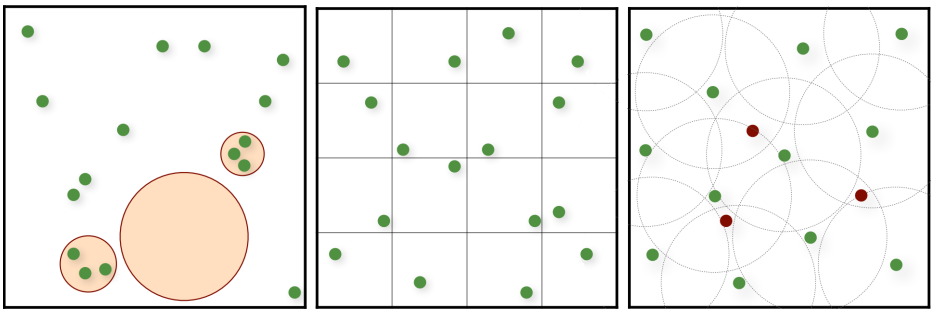


Figure 2.16: Independent random sampling (left) can result in gaps and clusters that make Monte Carlo integration converge in $\mathcal{O}(n^{-0.5})$. Already, with jittered/stratified sampling (middle), convergence is improved to $\mathcal{O}(n^{-0.75})$ by decomposing the domain with a grid and enforcing one random point per grid cell. The point set on the right is a Poisson disk sampling that makes Monte Carlo integration converge in $\mathcal{O}(n^{-0.5})$ asymptotically, but still behaves better than independent random sampling for low sample count. Such point set is obtained by random sampling and then rejecting samples in red that are too close to any other existing sample. We will see other constructions such as based on Lloyd iterations in Fig. 4.12 that also make Monte Carlo integration converge in $\mathcal{O}(n^{-0.75})$ (see *Variance Analysis for Monte Carlo Integration* [Pilleboue et al. 2015], and for other insight, *Fourier Analysis of Correlated Monte Carlo Importance Sampling* [Singh et al. 2019]

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 (i.e., a mirror does not directly receive light from light sources, it redirects indirect light from the environment), we will consider for now that our surfaces are either purely diffuse of albedo ρ (and $L_e = 0$), or purely emissive (with $f = 0$ and $L_e \neq 0$). We want to randomly sample the space of light paths, but more frequently where the light contribution is high, in order to do a Monte Carlo estimation of the integral with importance sampling. We will also truncate the infinite sum of integrals of increasing dimensions to a finite sum, by making light bounce a finite number of times. At each light bounce over a diffuse surface at point x we locally evaluate the integrand, which requires recursively sending another ray of light that needs to be importance sampled. For the diffuse case, the rendering equation simplifies to:

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

To importance sample in this case, 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 already be 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 the Box-Muller formula:

$$r_1, r_2 \sim \mathcal{U}(0, 1) \tag{2.6}$$
$$x = \cos(2\pi r_1)\sqrt{1 - r_2} \tag{2.7}$$
$$y = \sin(2\pi r_1)\sqrt{1 - r_2} \tag{2.8}$$
$$z = \sqrt{r_2} \tag{2.9}$$
$$\tag{2.10}$$

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 change of frame 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 $V = x\,T_1 + y\,T_2 + z\,N$, 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 Dutré[8].

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). 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 the Fresnel component of transparent materials, you do not need to change anything.

[8] https://people.cs.kuleuven.be/ philip.dutre/GI/TotalCompendium.pdf

Remark. Note that the noise that you will observe is related to the variance of the Monte Carlo estimator of the integral. In fact, the average mean squared error in your rendering for an estimator $\tilde{F}(X)$ of $\int f(x)\mathrm{d}x$ can be expressed[9] as $E\left[\left(\int f(x)\mathrm{d}x - \tilde{F}(X)\right)^2\right] = \left(\int f(x)\mathrm{d}x - E\left[\tilde{F}(X)\right]\right)^2 + E\left[(E[F(X)] - F(X))^2\right]$ where E is the expectation. One recognizes $E\left[(E[F(X)] - F(X))^2\right]$ as the variance of $F(X)$ and $\int f(x)\mathrm{d}x - E\left[\tilde{F}(X)\right]$ is called the bias, i.e., how far the estimator is off the true value of the integral on average. For the Monte Carlo estimator, $\tilde{F}(X) = \frac{1}{n}\sum_i f(x_i)$ is an unbiased estimator of $\int f(x)\mathrm{d}x$, so the mean squared error directly corresponds to the variance of $\tilde{F}(X)$. In the opposite direction, you could consider a biased estimator of the integral that always returns zero. In that case, the variance would be zero and the image would be noise free. However, the mean squared error would directly correspond to the bias, i.e., $E[\int f(x)\mathrm{d}x - 0] = \int f(x)\mathrm{d}x$, the entire result of the integral. In Monte Carlo estimation, there is always a tradeoff between variance (i.e., noise) and bias (i.e., average error).

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:

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

    if (intersect(ray, P, N, sphere_id)) {
        if (spheres[sphere_id].mirror) {
          // handle mirror surfaces ...
        } else {
          // handle diffuse surfaces
          Vector Lo(0., 0., 0.);
          // add direct lighting
          double visibility = ... ; // computes the visibility term by launching a ray towards the light source
          Lo = light_intensity/(4*M_PI*squared_distance_light) * albedo/M_PI * visibility * std::max(dot(N, light_direction), 0.);

          // add indirect lighting
          Ray randomRay = ...; // randomly sample ray using random_cos
          Lo += albedo * getColor(randomRay, ray_depth-1);

          return Lo;
        }
    }
}
```

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 progressively redirecting the light from one light source towards a small hole illuminating a room). We 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

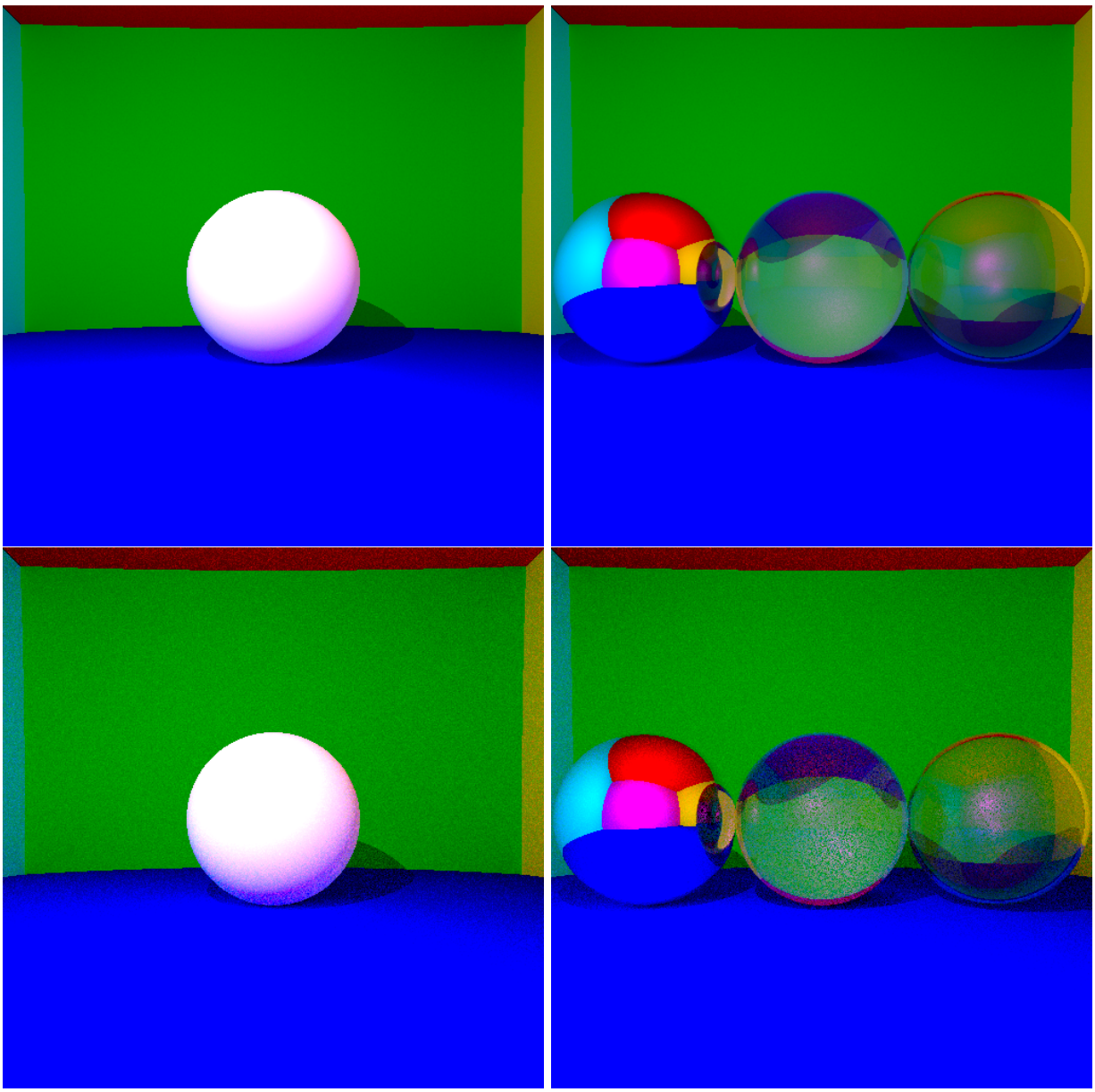


Figure 2.17: Rendering with indirect lighting (290 lines of code). First row, the renderings with either a diffuse or transparent central spheres take about 35 seconds in parallel (or 7 minutes without parallelization) using 1000 paths per pixel, and a maximum ray depth (`max_path_length` in the code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

any case, this results in an unbias 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.

Control variates. There are other ways to reduce noise in Monte Carlo renderings. A first solution is to use *control variates*. A control variate is a function h that resembles the integrand f, but for which the exact integral H is known. The estimator of the integral becomes

$$I = \int f(x)\mathrm{d}x \approx \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - \alpha h(x_i)) + \alpha H$$

which depends on a scalar value α. In that case, the variance of the estimator is $Var(I(X)) = Var(f(X)) + \alpha^2 Var(h(X)) - 2\alpha Cov(f(X), h(X))$. To find the best value for α, one can differentiate $Var(I(X))$ with respect to α and solve $\frac{dVar(I(X))}{d\alpha} = 0$. This amounts to $2\alpha Var(h(X)) = 2Cov(f(X), h(X))$, or simply $\alpha = \frac{Cov(f(X),h(X))}{Var(h(X))}$. Said otherwise, knowing the integral of a function

that correlates well with the integrand improves the variance (i.e., the square of the integration error, for an unbiased estimator) as the correlation improves. The approach can be extended to the case where one knows several control variates.

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

```
#pragma omp parallel for
for (int i=0; i<H; i++) {
 // ...
}
```

This instructs the compiler to perform the for loop in parallel. Make sure to enable OpenMP, using Projet 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 do support OpenMP. On MacOS, you may need to link with OpenMP using `-L/usr/local/opt/libomp/lib -I/usr/local/opt/libomp/include -Xpreprocessor -fopenmp -lomp -Ofast`. 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 a thread 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 just 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.

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:

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

where $\{i, j\}$ are the pixel 2-d indices (where hopefuly, it is clear from the context when i merely stands for "incoming", like for L_i or ω_i, and when i is a vertical or horizontal pixel coordinate like here),

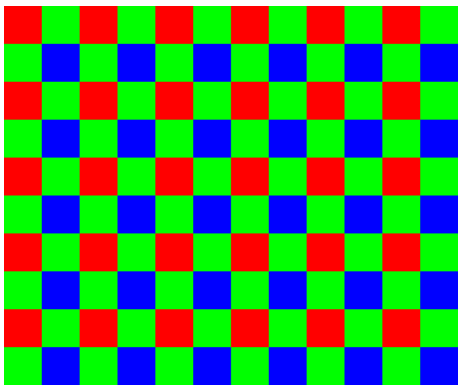


Figure 2.18: Camera sensor cells are arranged in a Bayer pattern, interleaving red, green and blue filtered sensors.

$A_{i,j}$ 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 an incoming 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 will rather filter the signal more smoothly, by integrating:

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

where h is some nice smooth kernel (see Sec. 3.1.1 for more details on filtering and convolutions). While interesting choices include Mitchell-Netravali's filtering or windowed Sinc filters, we will simply use a Gaussian filter centered at the middle of pixel (i, j) as 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:

```
int main() {
// first define the scene, variables, ...
// then scan all pixels
#pragma omp parallel for schedule(dynamic, 1)
for (int i=0; i<H; i++) {
 for (int j=0; j<W; j++) {
  Vector pixelColor(0., 0., 0.);
  for (int k=0; k<NB_PATHS; k++) {
     Vector rand_dir = ...; // as before but targeting pixel (i,j)+boxMuller()*spread
     Ray ray(C, rand_dir); // cast a ray from the camera center C with rand_dir ↩
         direction
     pixelColor += scene.getColor(ray, max_path_length);
   }
   pixel[i*W*3+j*3 + 0] = std::min(255, std::pow(pixelColor[0]/NB_PATHS, 1./2.2)); //↩
       stores R channel
   // same for green and blue
 }
}
// save image and return 0
}
```

and produces the image in Fig. 2.19.

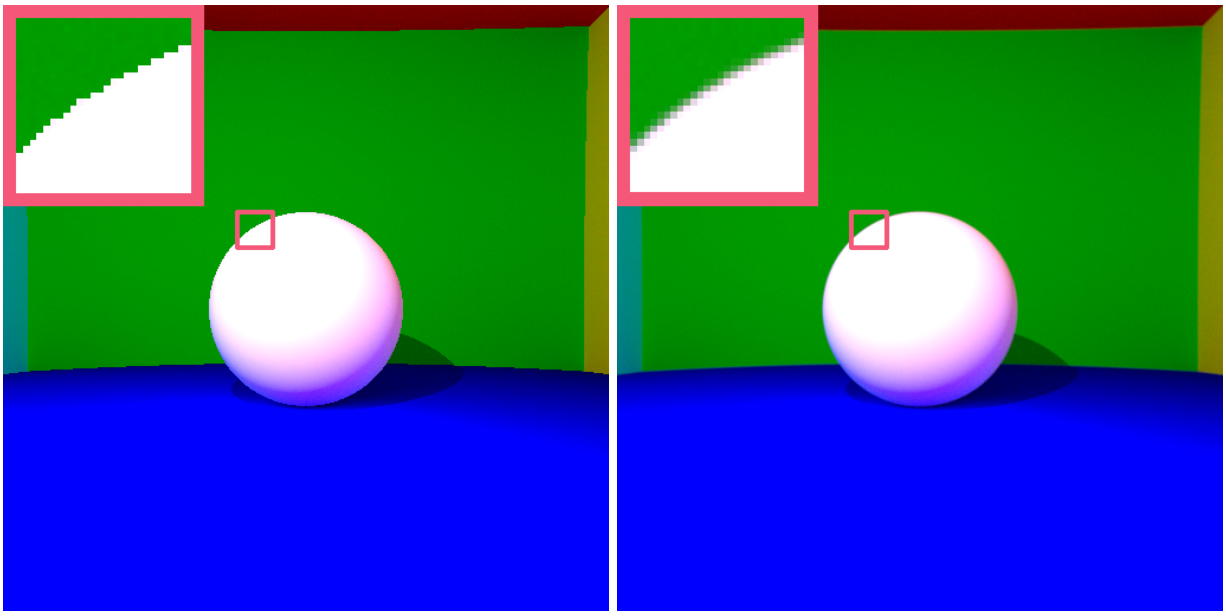


Figure 2.19: Image without (left) and with (right) antialiasing.

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 be to simply 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 would 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).

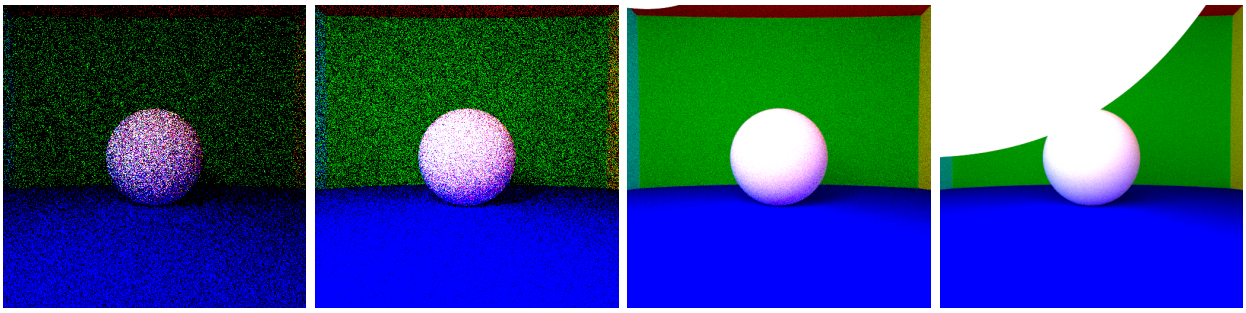


Figure 2.21: Naively handling soft shadows using spherical light sources of radius 1, 2, 10, and 20. As the radius increases, light paths have more chances to randomly reach light sources, which reduces noise. Also notice the soft shadows appearing. These renderings still have 1000 (uncorrelated) samples 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:

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

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 was an infinitesimally small domain, of zero measure.

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 an indirect lighting contribution off a diffuse surface but it nevertheless hits a spherical light source, 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). This involves tracking whether the ray we are currently dealing with was generated by an indirect bounce over a diffuse surface, or by any other way (sent from the camera, or reflected/transmitted by other surfaces). 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).

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:

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

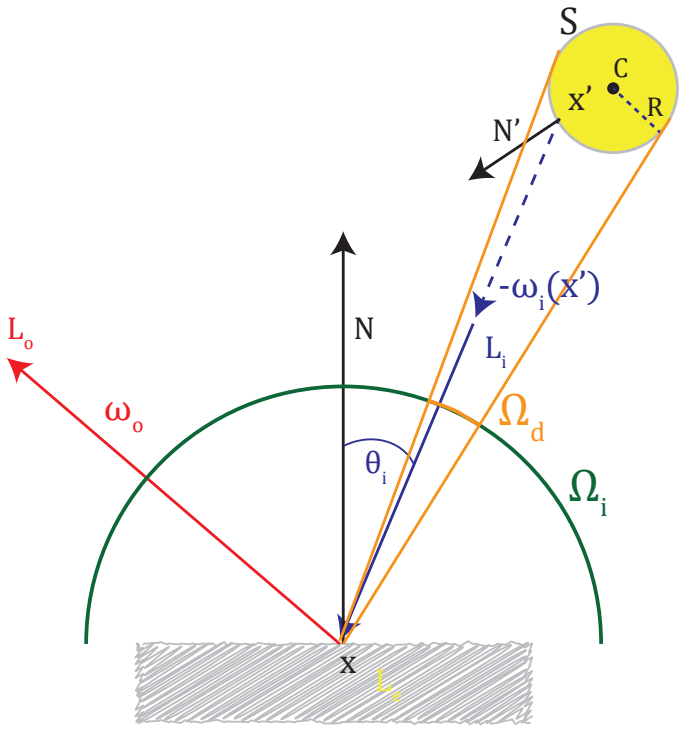


Figure 2.22: Notations for integrating over elements in the scene.

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. 2.1.5).

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 should put little budget in sampling near the "edge" of the spherical light, as this dot product will be close to zero. 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 a direction V 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 = random_cos(D)` to obtain a unit direction that has more chances of facing D, and finally obtain x' using $x' = RV + C$. The probability density function at x' is $p(x') = \frac{\langle V,D \rangle}{\pi} . \frac{1}{R^2}$, where $1/R^2$ is due to the samples being *stretched* 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 Watts.m^{-2}.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) but in general, (unidirectional) path tracing, as we implement it, is not ideal for caustics.

⚠ 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, notably due to various offsets being introduced. 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 offseted point above the surface.

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. 2.25): just a dark box of length f (called *focal length*) pierced with an tiny

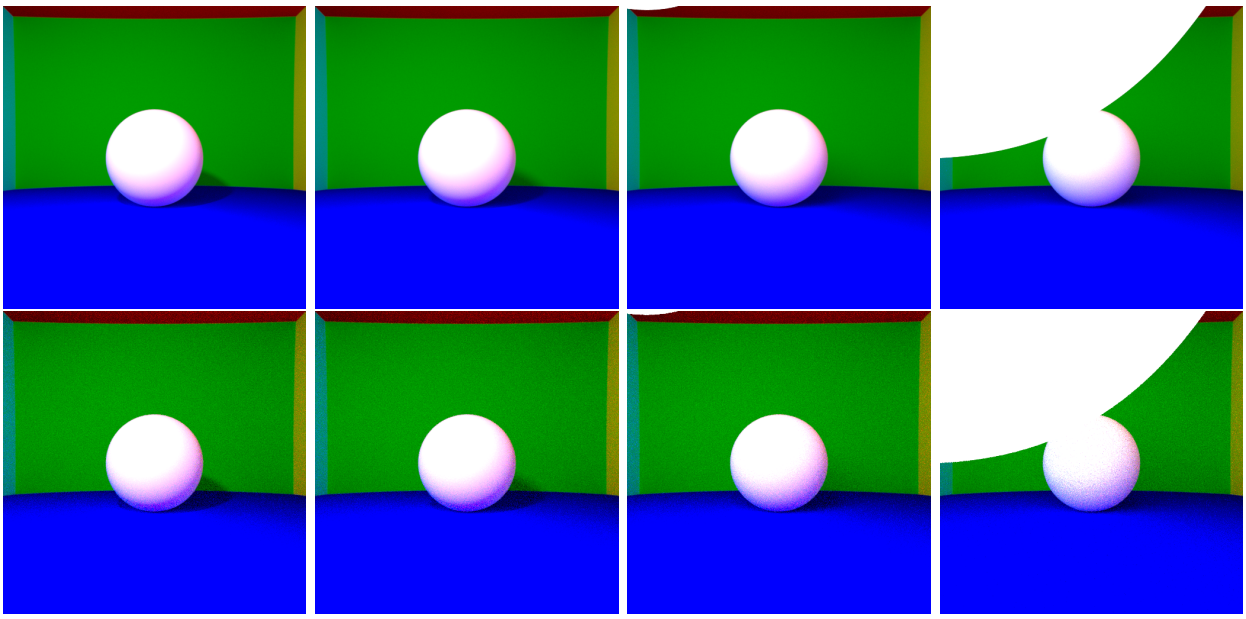


Figure 2.23: Soft shadows by directly sampling the spherical light source (the code is now about 320 lines) of radius 1, 2, 10 and 20. Using 1000 samples per pixel and 5 light bounces (top row), it takes about 1 minute per image. Using 32 samples per pixel (bottom row), about 2 seconds. Note that noise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

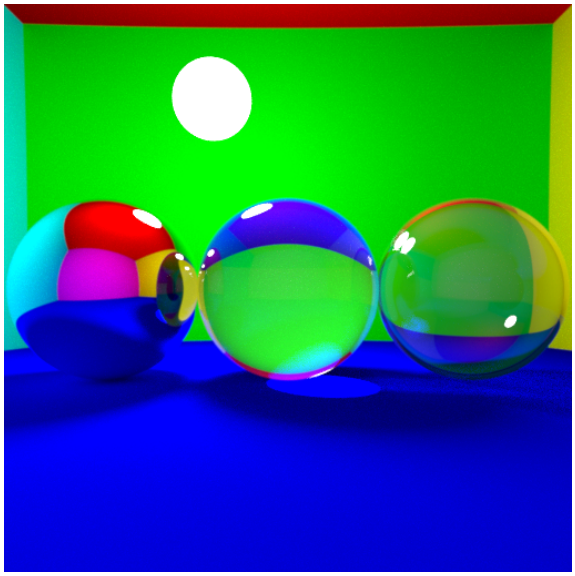


Figure 2.24: Moving the light a little bit reveals caustics in the transparent scene. Here the light sphere is at position $(-10, 25, -10)$ and of radius 5. These indirect specular bounces are hard to capture and thus produce much higher levels of noise (here, 5000 samples per pixel were used). Other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

hole (in practice, the optimal hole size is $d = 2\sqrt{f\lambda}$ for a wavelength λ). This kind of setup has been known for a long time. In fact, it is suspected that it was known since paleolithic times[10]. 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. 2.26)[11]. You may see one camera obscura setup in the Greenwich

[10] see `http://paleo-camera.com/` for discussions on suspected paleolithic and neolithic setups.

[11] 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, like Vermeer (1632-1675) `https://en.wikipedia.org/wiki/Hockney%E2%80%93Falco_thesis`.

observatory, used in the past for imaging the sun during eclipses, and now pointing towards London (the current one is from 1994, Fig. 2.27, though there have been other camera obscuras at Greenwich starting from the late 17th century).

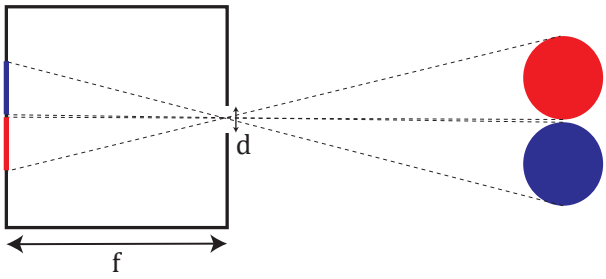


Figure 2.25: A pinhole camera is just a small hole in a dark chamber that lets light come in and displays a sharp view of the outside world on the screen. The image is flipped: in our path tracer, we have just put our sensor at a virtual location at a distance f outside the box for a more intuitive implementation and non-flipped renderings (in our setup, the *camera location* C is the hole, and the pixel grid is outside).

Figure 2.26: 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)

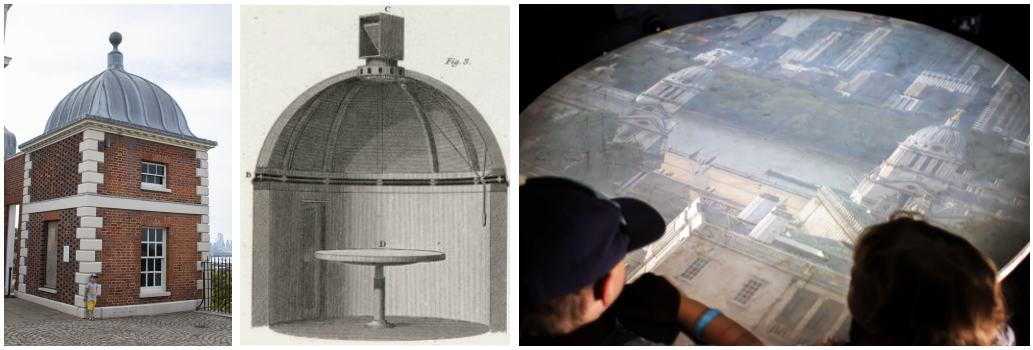


Figure 2.27: The last Greenwich pinhole, here, was installed in 1994 to see the city from the Royal Observatory, but there have been numerous pinholes in Greenwich dating back to the mid-17th century, often used to see solar eclipses without looking at the sun.

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. 2.28), 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 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 such, we will simply find new starting points for our rays that are slightly tangentially offseted 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).

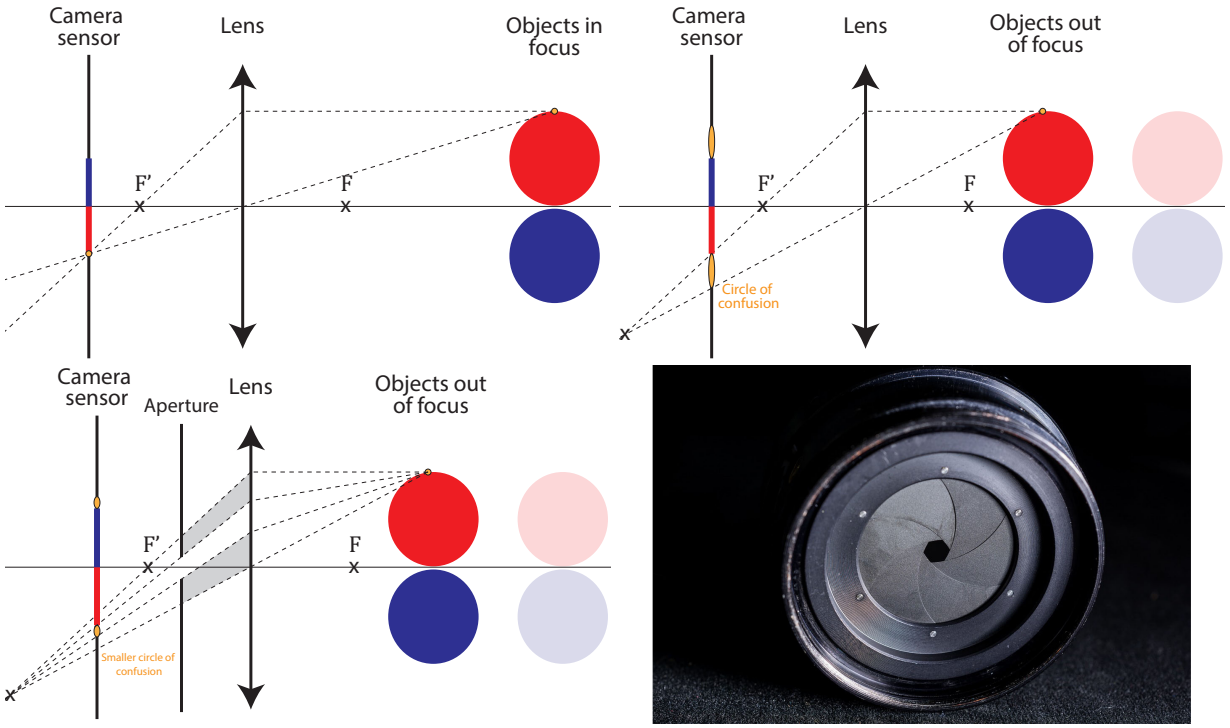


Figure 2.28: **Top row.** Using a camera lens, an object placed at the focus distance will appear sharp (left) as the image of a point of the object is a point on the sensor. However, moving the distance closer to (or away from) the camera makes the object appear blurry as the image of a point is a small 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).

To achieve that, we first generate a ray from the camera center Q (the pinhole center) as before (red ray in Fig. 2.28). 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. 2.29). 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. 2.30.

Similarly, while the shutter of the camera is open, objects may have moved. This produces another kinf 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 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. 2.31.

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

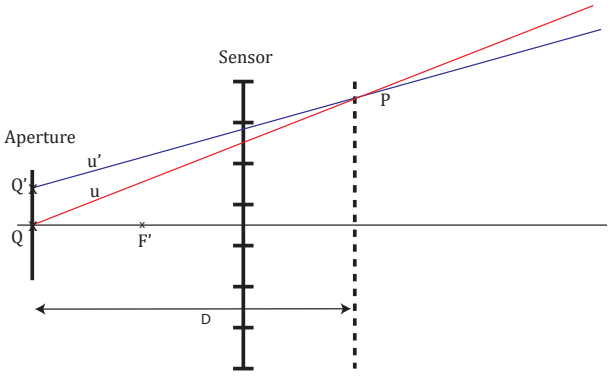


Figure 2.29: Depth of field can be obtained in our path tracer by starting rays from a point on the aperture shape (instead of the pinhole center) such that rays cross at the focusing distance D.

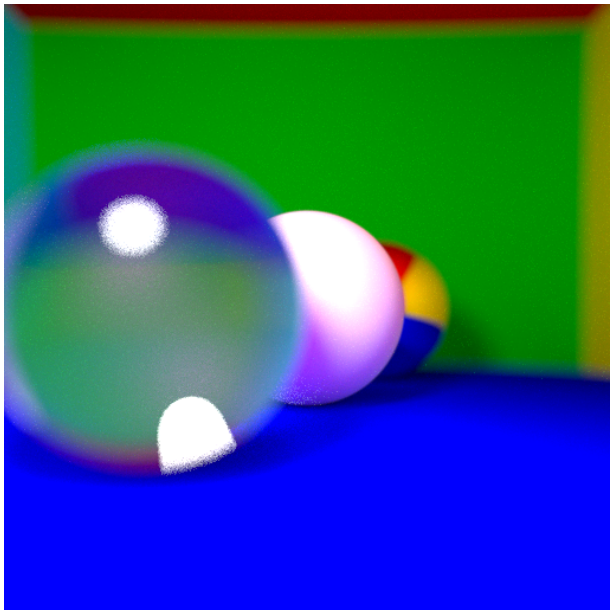


Figure 2.30: Depth of field result in our path tracer, adding less than 10 lines of code, bringing it to 330 lines. Here, 2000 samples per pixel were used because of specular paths, though depth of field often necessitate more samples.

Meshes

The next big thing in our path tracer is the support of triangle meshes. It is highly uninteresting 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:

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

We are still only interested in positive solutions.

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*

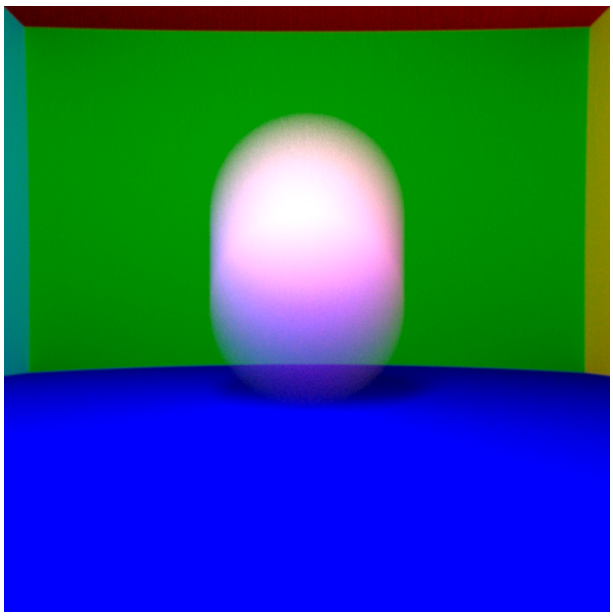


Figure 2.31: Motion blur is obtained by adding a time parameter to the rays. The time value is selected randomly within the interval of time the camera shutter is kept open. The ray-sphere intersection here considers a linear motion of the sphere. This merely adds 2 lines of code, and modifies a couple of others.

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:

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

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

$$\alpha = 1 - \beta - \gamma \tag{2.13}$$

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

$$\tag{2.15}$$

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, this kind of class cannot be instantiated (you cannot create an object of type `Geometry` because the implementation of some functions is missing). 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` ; Edit2: oh, as of 2024, cadnav is back after a long pause, and this model too!). It has 3954 polygons to test.

⚠ Unless you made a fancy GUI, normalized your models upon loading, or know or made your 3d model yourself, 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. 2.32.

⚠ 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 in considering the pairs of intersections between the ray and pairs of planes. Theses pairs of intersections

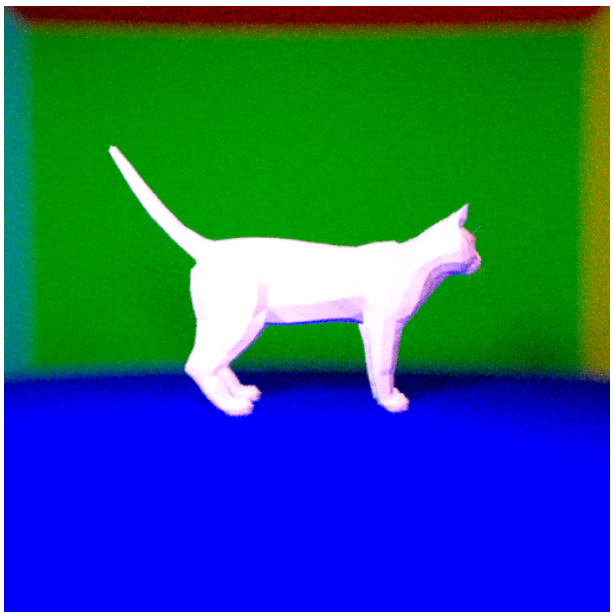


Figure 2.32: Our cat model, just scaled by a factor 0.6 and translated by the vector $(0, -10, 0)$. At 32 samples per pixel (spp) and 5 light bounces, it took 6 min and 20 seconds (in parallel) by naively testing all triangles using the *Möller–Trumbore intersection algorithm*. By adding a simple ray-bounding 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 !

define 3 intervals for the ray parameter t: 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 theses 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. 2.33 for notations in 2-D). It is also interesting to see that a ray-plane intersection with axis-aligned planes takes a particularly simple 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. 2.34).

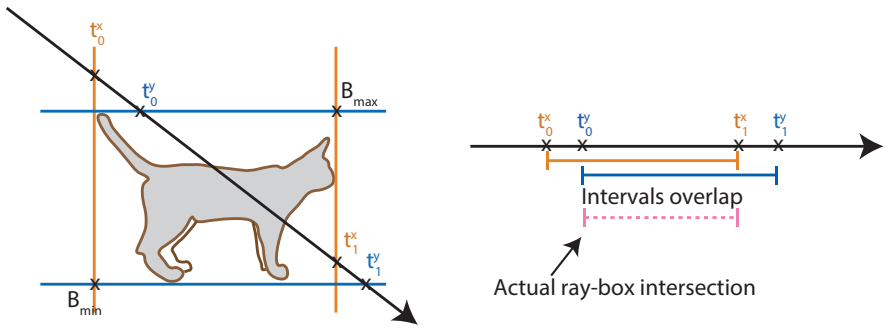


Figure 2.33: A ray-bounding box intersection can be performed by testing the overlap between intervals defined by pairs of ray-planes intersections.

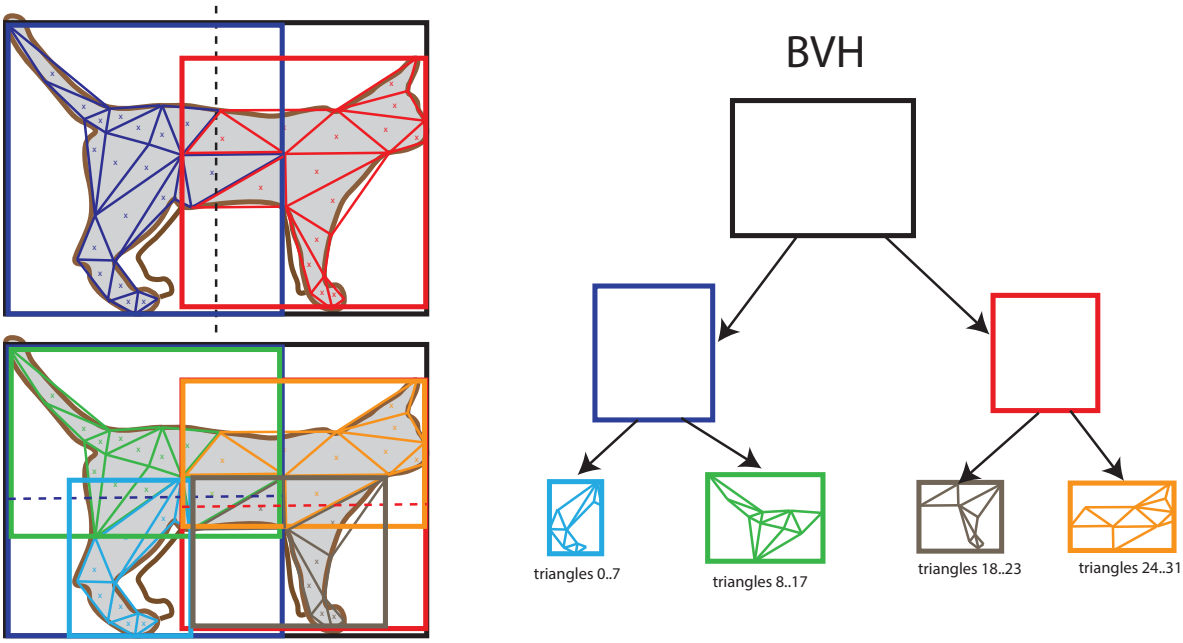


Figure 2.34: A BVH recursively computes bounding boxes. The overall bounding box (black) is split into 2 categories (blue and red) based a vertical split in the middle of the black box. Triangles are assigned to either the blue or red categories based on their centroid. The bounding boxes of these two 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 refering to a permutation of the original set of triangles.

In practice, building this tree can be performed using a method akin to *QuickSort*: instead of storing the indices of triangles belonging to one side of the split or the other side, triangles are simply reordered in a way that consecutive triangles belong to the same bounding box. This reordering 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 – similarly to QuickSort. This looks like:

```
node->bbox = compute_bbox(starting_triangle, ending_triangle); //BBox from ↩
    starting_triangle included to ending_triangle excluded
node->starting_triangle = starting_triangle;
node->ending_triangle = ending_triangle;
Vector diag = compute_diag(node->bbox);
Vector middle_diag = node->bbox.Bmin + diag*0.5;
int longest_axis = get_longest(diag);
int pivot_index = starting_triangle;
for (int i=starting_triangle; i<ending_triangle; i++) {
  Vector barycenter = compute_barycenter(indices[i]);
```

```
  // the swap below guarantees triangles whose barycenter are smaller than middle_diag are before "pivot_index"
  if (barycenter[longest_axis] < middle_diag[longest_axis]) {
      std::swap(indices[i], indices[pivot_index]);
      pivot_index++;
  }
}
// stopping criterion
if (pivot_index<=starting_triangle || pivot_index>=ending_triangle-1 || ending_triangle-starting_triangle<5 ) return;
recursive_call(node->child_left, starting_triangle, pivot_index);
recursive_call(node->child_right, pivot_index, ending_triangle);
```

⚠ In degenerate cases, during one of the recursive calls, *all* triangles may be on the same side of the split. This is due to the split being determined by the middle of the bounding box, and it is not difficult to construct cases where all triangles are on the same side. Make sure to stop the recursive calls in that case, since it will otherwise continue infinitely!

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 far[14]:

```
if (!root.bbox.intersect(ray)) return false;
std::list<Node*> nodes_to_visit;
nodes_to_visit.push_front(root);
double best_inter_distance = std::numeric_limits<double>::max();
while(!nodes_to_visit.empty()) {
   Node* curNode = nodes_to_visit.back();
   nodes_to_visit.pop_back();
   // if there is one child, then it is not a leaf, so test the bounding box
   if (curNode->child_left) {
      if (curNode->child_left->bbox.intersect(ray, inter_distance)) {
         if (inter_distance < best_inter_distance) {
            nodes_to_visit.push_back(curNode->child_left);
         }
      }
      if (curNode->child_right->bbox.intersect(ray, inter_distance)) {
         if (inter_distance < best_inter_distance) {
            nodes_to_visit.push_back(curNode->child_right);
         }
      }
    } else {
      // test all triangles between curNode->starting_triangle
      // and curNode->ending_triangle as before.
      // if an intersection is found, update best_inter_distance if needed
   }
}
```

[14] A 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.

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 !

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 computations[15]. 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 see in Fig. 2.35.

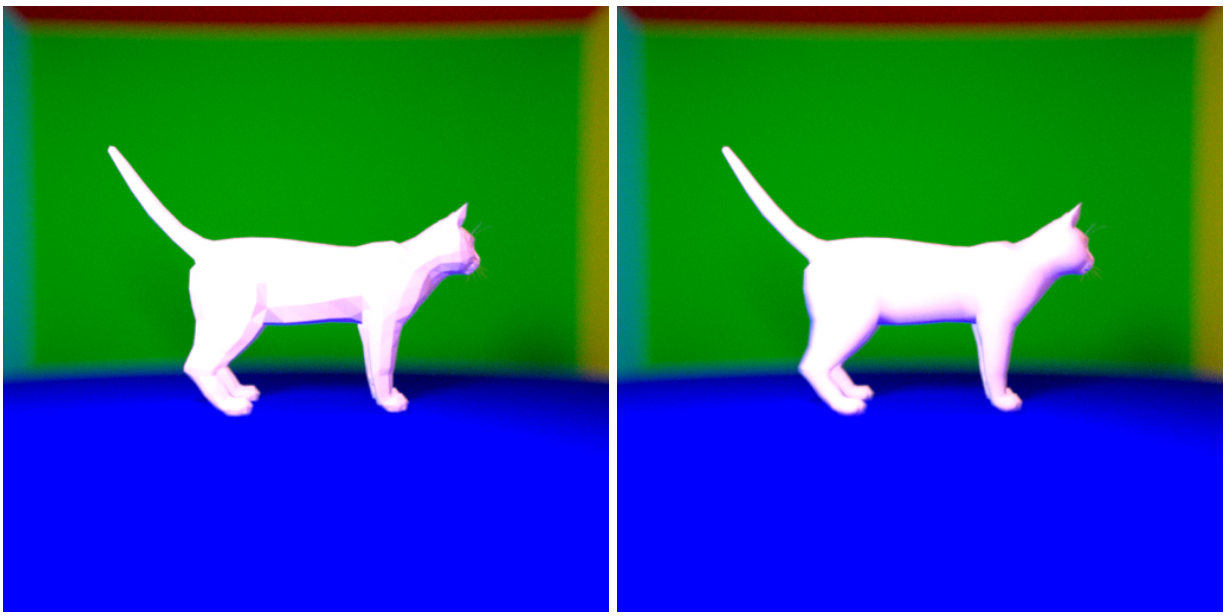


Figure 2.35: Cat model without (left) and with (right) interpolation of normals.

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. We will see in Sec. 4.6 how they can be obtained. 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 ob 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. 2.36). 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

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

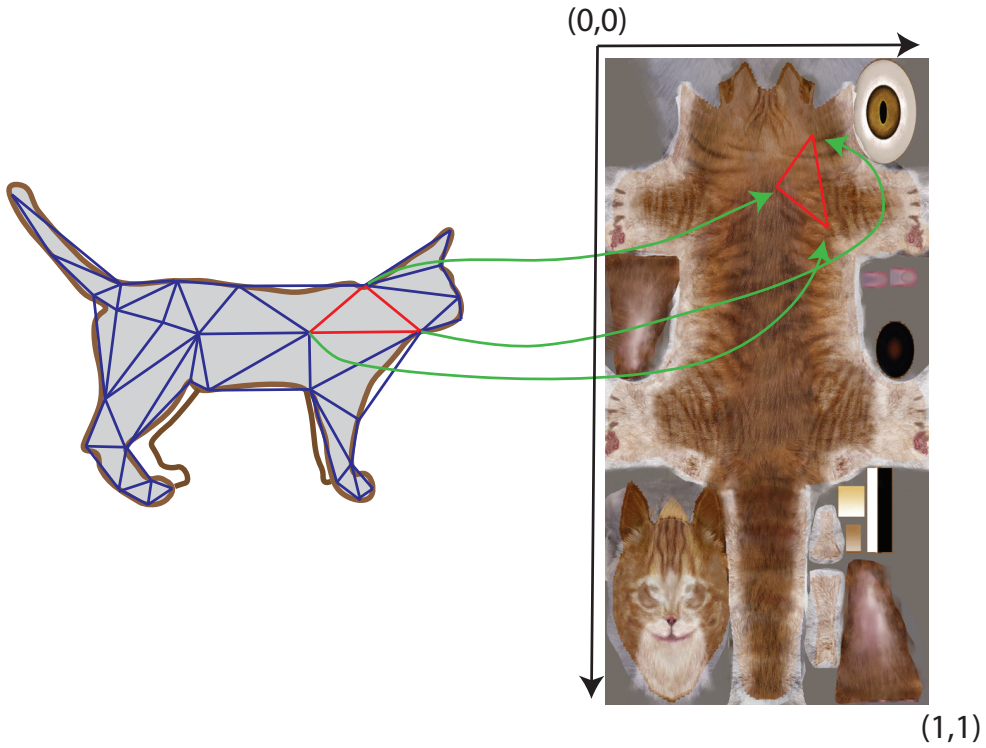


Figure 2.36: UV coordinates associate for each 3D vertex a 2D coordinate in the texture map, that can be interpolated using barycentric coordinates.

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. 2.37.

⚠ 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! It would be even better to interpolate instead, using bilinar or bicubic interpolation for example, but we will not do it here. Finally, beware that the origin $(0, 0)$ of UV coordinates is conventionally the bottom left of the texture, while most often textures are loaded from top to bottom.

Procedural textures. While textures stored as bitmap can be convient and easy to edit in image editing softwares, procedural textures are sometimes used. Procedural textures represent color values using mathematical expressions (involving the position, normal, or any other parameter, such as local curvature, depth, ...). They have the advantage that they are very compact to store, can represent functions of arbitrary resolution (similar to vector images), easily extend in size (e.g., to represent a kilometer-wide world), allows for variety (e.g., having many rocks looking different with a parameter in the formula rather than re-using the same bitmap image to texture each of them) and easily extend to 3-d, volumetric, textures (e.g., to represent the inside of a rock when it breaks, or to represent the density of a cloud). Simple textures can be obtained with simple expressions. For instance, a checkerboard procedural texture amounts to checking if $\lfloor \frac{x}{\alpha} \rfloor + \lfloor \frac{y}{\alpha} \rfloor \text{mod} 2 = 0$ and return either black

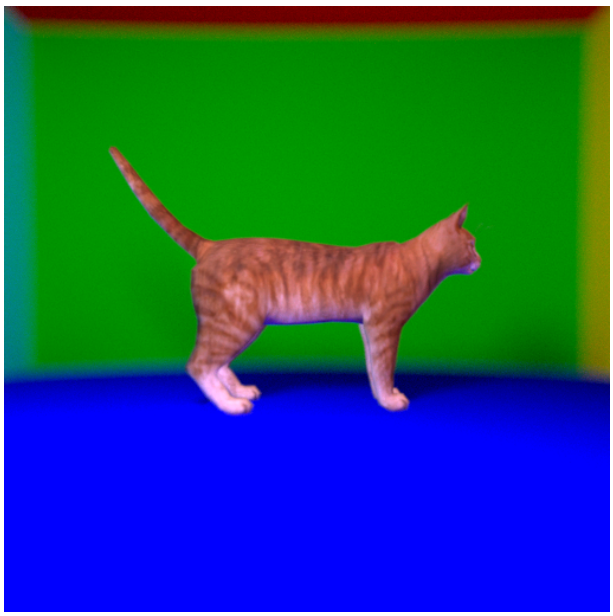


Figure 2.37: Cat model with textures, with a gamma function applied. The code is now about 700 lines long, including the 200 lines obj file reader.

or white. Many procedural textures can be obtained via the so-called *Perlin noise*. A 3-d Perlin noise stores precomputed unit random vectors at each corner of a voxel grid. Then evaluating the noise at a point P in space amounts to computing the 8 vectors between P and each of the 8 corners of the voxel P belongs, then computing the dot products between each of these 8 vectors and the corresponding precomputed random vector at each corner, and finally interpolating these 8 dot products based on the location of P within the voxel. Perlin noise functions are widely used, and are the main component of more complex textures. For instance, a *turbulence* texture computes the weighted sum of multiple Perlin noise functions at different scales. A marble function can be obtained by altering the phase of a sine function using a turbulence function. Perlin noise can also be used to generate procedural geometry, and in particular, terrains/mountains. The french company Allegorithmic founded in 2003 and bought by Adobe in 2019 is specialized in producing procedural textures, which become highly complex function graphs to achieve realistic, varied and parameterized textures.

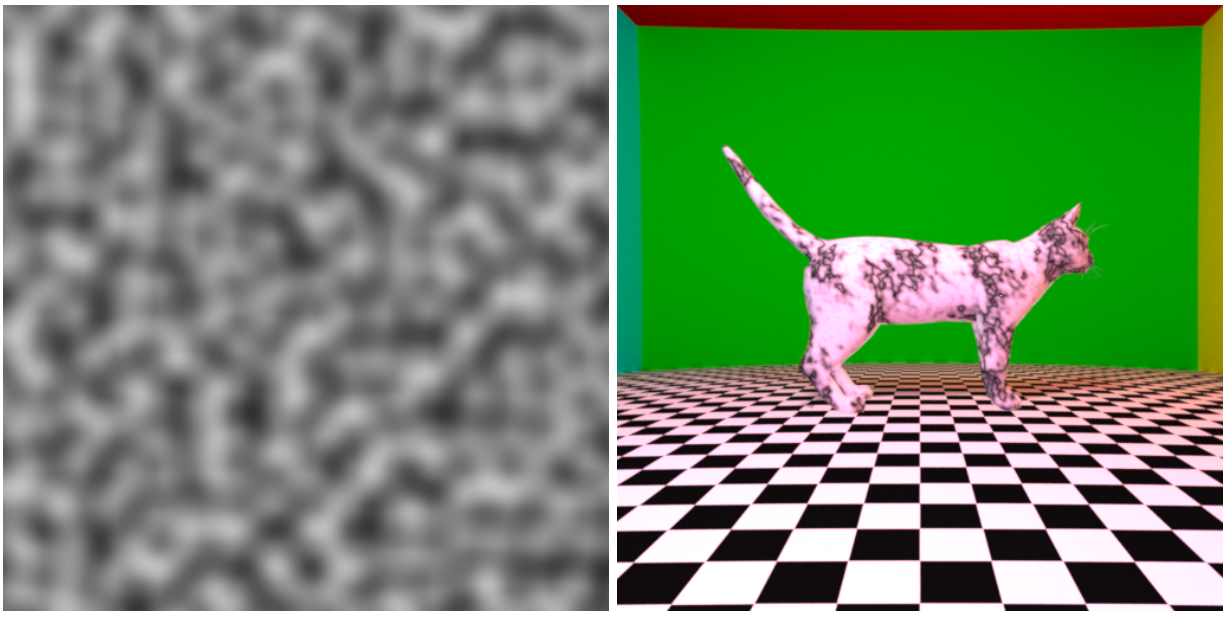


Figure 2.38: Procedural textures. **Left**. A Perlin noise in 2-d (from Wikipedia). **Right**. The cat has a marble texture based on Perlin noise functions, while the ground is a simple procedural checkerboard.

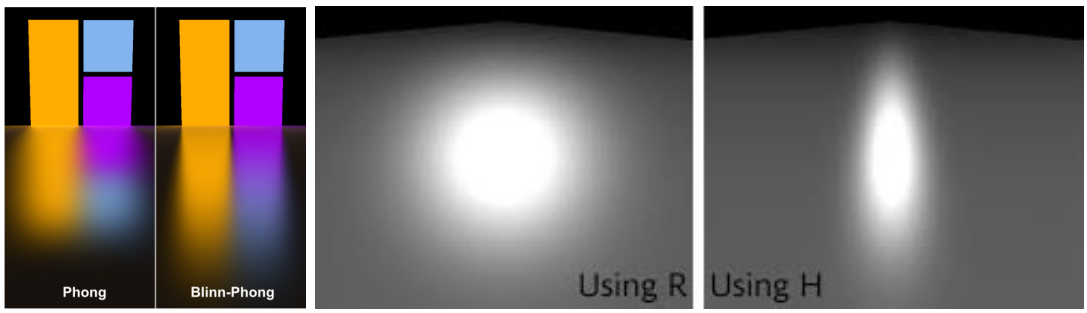


Figure 2.39: The original Phong model does not appropriately model the distortion of highlights at grazing angles (left of each pair) while this is solved by the Blinn-Phong model (right of each pair). *Left image pair by an unknown author. Right image pair by Lecocq et al. 2017.*

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. 2.40). However, this model does not model well highlight distortions at grazing angles (Fig. 2.39). 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 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. Note that a more modern BRDF would be the GGX model or the "Disney" BRDF, though more difficult to sample.

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 law[16].

[16]See 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 probability density function (pdf) is given by $p(X) = \frac{\alpha+1}{2\pi}\cos^\alpha\theta$, where here, θ is the angle to 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 $I = \int f(x)\mathrm{d}x = \int \sum w_i f_i(x)\mathrm{d}x$. 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 sample. 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 materials[17]. We end up with a code similar to:

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

    Vector Lo(0., 0., 0.);
   if (intersect(ray, P, N, sphere_id)) {
      if (spheres[sphere_id].mirror) {
        // handle mirror surfaces ...
      } else {
        // handle Phong materials
        // add direct lighting
        Vector xprime = random_point_on_light_sphere();
        Vector Nprime = (xprime-centerLight)/(xprime-centerLight).norm();
        Vector omega_i = (xprime-P)/(xprime-P).norm();
        double visibility = ... ; // computes the visibility term by launching a ray ↩
            of direction omega_i
```

[17]There 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.

```
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. 2.40.

Multiple Importance Sampling. Another variance reduction technique considers multiple strategies for importance sampling: *multiple importance sampling* (MIS). For instance, one may know how to sample a pdf that well approximates the BRDF, and another pdf that well approximates the incoming light distribution, but not their product. Both options are reasonable, and it may be interesting to benefit from both strategies. When multiple pdfs $\{p_k\}_{k=1..K}$ are available, one may combine estimators obtained with these different strategies. The *naive* estimator simply averages the results obtained with these estimators: $I = \sum_{k=1}^{K} w_k I_k$, where $\sum_{k=1}^{K} w_k = 1$ are fixed weights, and I_k is the classical importance sampling estimator obtained with pdf p_k. A better option is the *balanced heuristic*, which is optimal when considering convex sums[18], and is given by $I = \sum_{k=1}^{K} \sum_{i=1}^{n_k} \frac{f(x_{k,i}}{\sum_{\ell=1}^{K} n_\ell p_\ell(x_{k,i}}$. It allocates n_k samples for the strategy k which pdf is p_k, and produces the samples $\{x_{k,i}\}_i$ for this strategy. This amounts to a weighted average $I = \sum_{k=1}^{K} \frac{1}{n_k} \sum_{i=1}^{n_k} w_k(x_{k,i}) \frac{f(x_{k,i})}{p_k(x_{k,i})}$ where the weight for strategy k is given by $w_k(x) = \frac{}{n_k p_k(x)} \sum_{\ell=1}^{K} n_\ell p_\ell(x)$. Note that with this heuristic, contrary to the naive estimator, it is necessary to evaluate the pdf p_ℓ of a sample generated by another strategy $k \neq \ell$.

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 $\left(T^{-1}\right)^T = T^{-T}$. Doing so has several advantages over directly transforming the vertices of each object upon loading them. First,

[18] The paper of [Kondapaneni et al. 2019] *Optimal Multiple Importance Sampling* investigates the case where weights can be negative

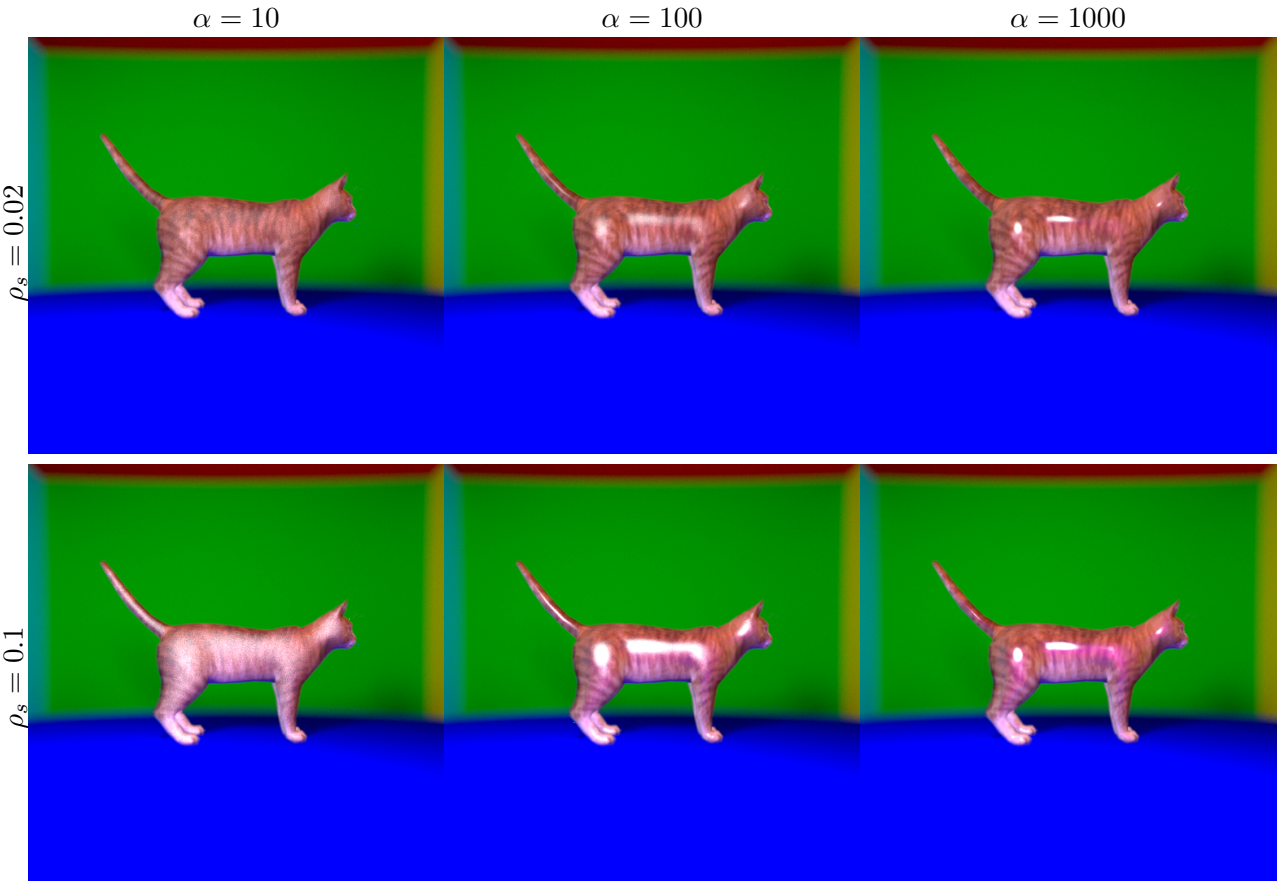


Figure 2.40: Cat model with a Blinn-Phong BRDF with varying α and ρ_s (here, ρ_s is not colored). α controls the roughness of the material (i.e., the size of highlights) while ρ_s controls the intensity of highlights. Note that since ρ_d is guided by a texture between 0 and 1 and ρ_s is a constant, this particular rendering may not preserve energy. The rendering takes 1min20 for 1000spp – the code is about 740 lines long.

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. It also has some other advantages, like being able to have simpler ray-object intersection routines (e.g., remember how to intersect an axis-aligned box? non-axis aligned boxes are supported by just transforming rays).

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 $y = s\,R\,x + t$ 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.

⚠ 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 !

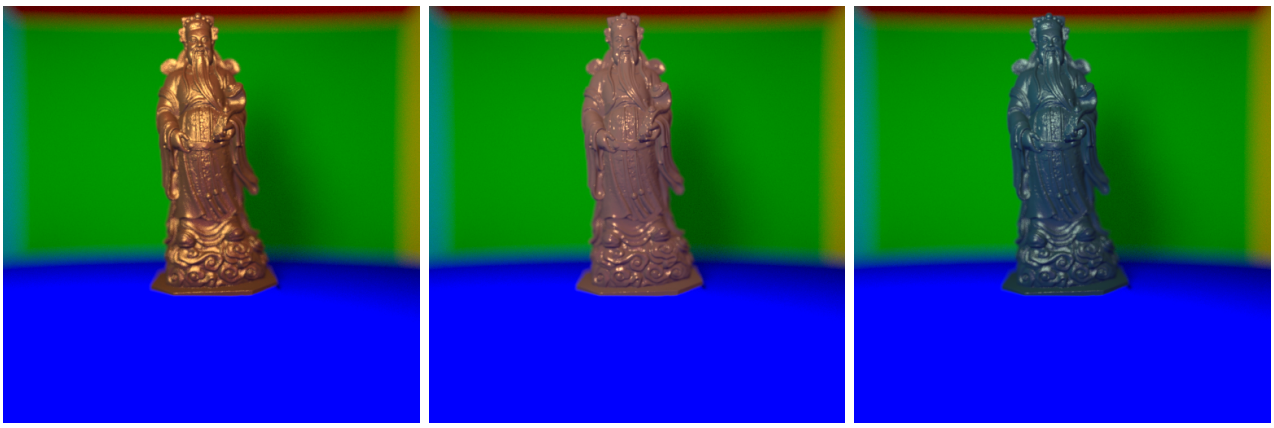


Figure 2.41: Another model (see `http://www.cadnav.com/3d-models/model-45798.html`) with Blinn-Phong BRDFs. The BRDF parameters can be found in the supplemental materials of "Experimental Analysis of BRDF Models" (`https://people.csail.mit.edu/wojciech/BRDFAnalysis/BRDFFits.pdf`), a document that contains 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 2.42: 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.

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 normals*, i.e., fake normals used during the shading computation in place of the geometric normal used for computing intersections. 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 vectors did not matter before since our reflectance model was isotropic, so we could take them arbitrarily. Conventionally, for normal maps, 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 $A^{-1} = \frac{1}{\det(A)} Cof(A)^T$ 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:

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

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

$$\det = (E_u - D_u)(F_v - D_v) - (F_u - D_u)(E_v - D_v) \tag{2.18}$$

Now, we can easily compute normalized T and B at each vertex of each triangle of the mesh[19], 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. 2.43 for the result.

[19] You 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.

[20] Similarly to the smooth shading normals we have implemented in Sec. 2.1.2, having a shading normal that is not exactly the geometric normals can lead to issues in energy conservation.

Figure 2.43: 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`, `body2_d.tga`, `gear_d.tga`, `gear_d.tga`, `body2_d.tga`

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. 2.44.

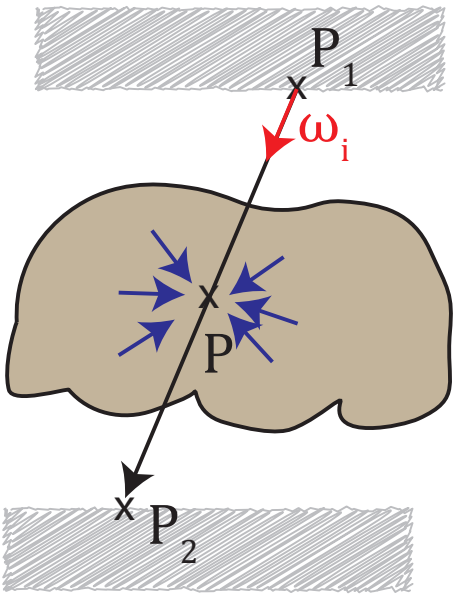


Figure 2.44: The light coming from direction ω_i is absorbed by the medium, but the medium also contributes positively (arrows in blue) to the light reaching point P_2.

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:

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

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

$$T(t) = \exp\left(-\int_0^t \sigma_t(P(r))\mathrm{d}r\right)$$

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 $T(t) = \exp\left(\frac{\alpha}{\beta\omega_{i,y}}\left(\exp(-\beta(P_y - y_0)) - \exp(-\beta(P_{1,y} - y_0))\right)\right)$ 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. 2.45.

Figure 2.45: 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:

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

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 Rayleight 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:

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

     Vector Lo(0., 0., 0.);
    if (intersect(ray, P, N, sphere_id)) {
       if (spheres[sphere_id].mirror) {
         // handle mirror surfaces ...
       } else {
         // handle Phong materials
       }
    }
    // return Lo;  // previous code without participating media

    double T = ....; // transmittance function (use closed form expression)
    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"
    double pdf = ...; // pdf for the choice of "r" and "v"
    return T*Lo + Lv/pdf;  // return the radiance modified by the participating medium

}
```

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 sources[21]. 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. 2.46.

[21] See 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`

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

⚠ 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.

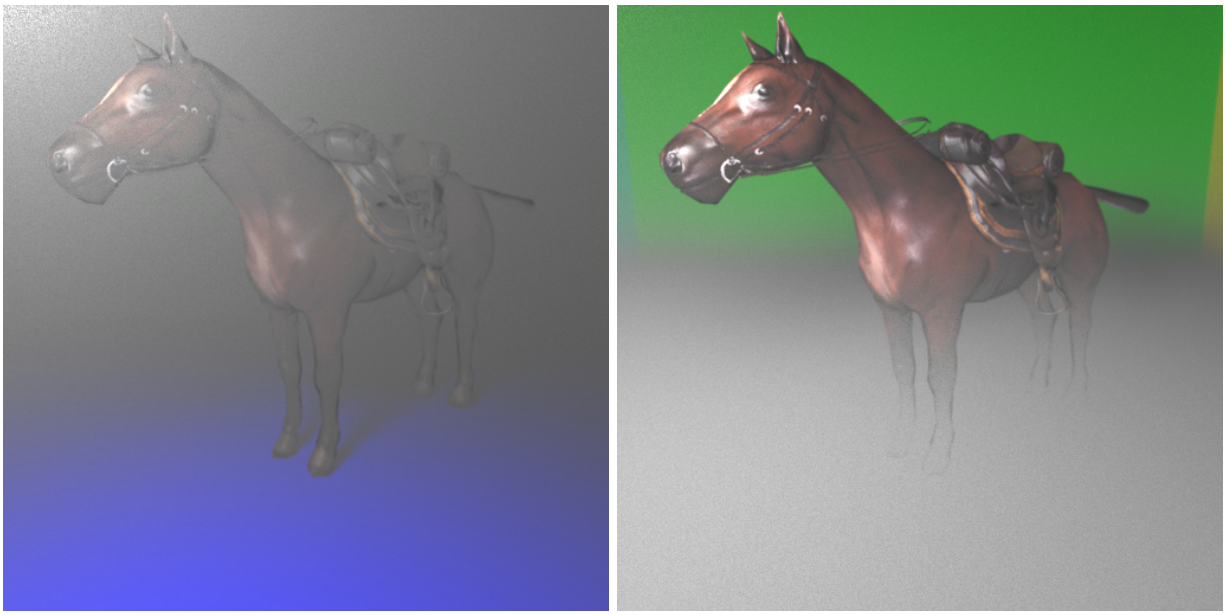


Figure 2.46: Adding the in-scattered radiance to the models presented in Fig. 2.45, with uniform (left, $\sigma_s = 0.004$) and exponential (right, $\sigma_s = 0.5\exp(-0.3(y+10))$) fog. I used $\lambda = 0.3t$. The code is 900 lines and renders in 3min 40sec.

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. 2.47;

Figure 2.47: 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 par pixel. Rendering time: 4 min. for 1000 spp.

2.1.3 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. 2.48). 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.

Figure 2.48: 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".*

2.1.4 Precomputed Radiance Transfer

Let's write the rendering equation without emissivity:

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

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:

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

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

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

$$\begin{aligned} L_o(\omega_o) &= \int_\Omega \sum_k \hat{f}^k_{\omega_o} \mathcal{F}_k(\omega_i) \sum_l \hat{L_i}^l \mathcal{F}_l(\omega_i) \mathrm{d}\omega_i \\ &= \sum_k \sum_l \hat{f}^k_{\omega_o} \hat{L_i}^l \int_\Omega \mathcal{F}_k(\omega_i) \mathcal{F}_l(\omega_i) \mathrm{d}\omega_i \end{aligned}$$

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

$$L_o(\omega_o) = \sum_k \sum_l \hat{f}^k_{\omega_o} \hat{L_i}^l$$

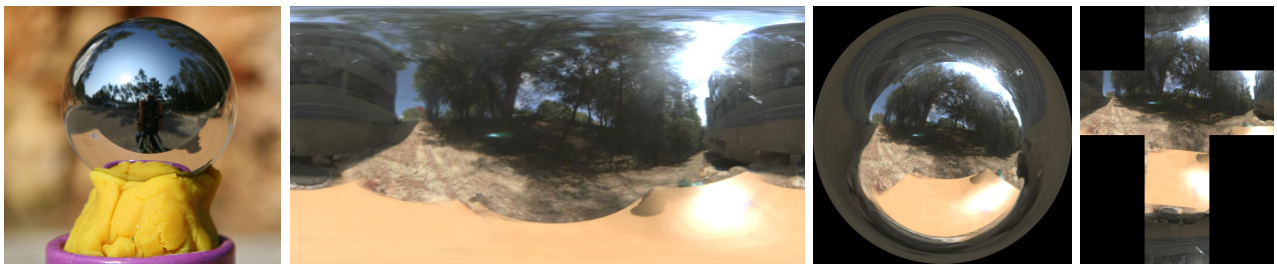


Figure 2.49: An environment map (or envmap) is simply a panoramic image representing the incident radiance at a point. It is often used for outdoor scenes since it well approximates distant illumination, and can be captured by mobile phone apps that stitch photographs into a panoramic image, or by taking 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.

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. 2.49 and 2.51), and performing the dot product in realtime.

Spherical Harmonics. Spherical Harmonics (SH) are commonly used orthogonal basis functions on the sphere (Fig. 2.50). They are analogous to the Fourier transform on the plane (eigenfunctions of the Laplacian operator are sine and cosines on the plane, and are spherical harmonics on the sphere). They hence represent a frequency decomposition of the signal. Similarly to the Fourier transform, they posess a discrete version, that can be efficiently evaluated using Fast Fourier Transforms. Additionally, they posess *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. 2.51. This is one reason why photographers use light diffusers: they will make skin more matte, and remove shiny reflections.

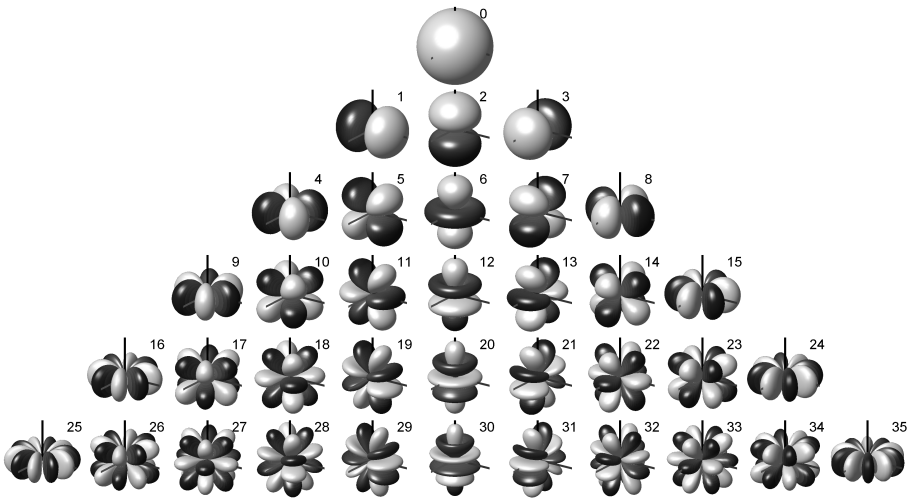


Figure 2.50: Spherical Harmonics up to degree 5 (Source Dr Franz Zotter, Wikimedia Commons).

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

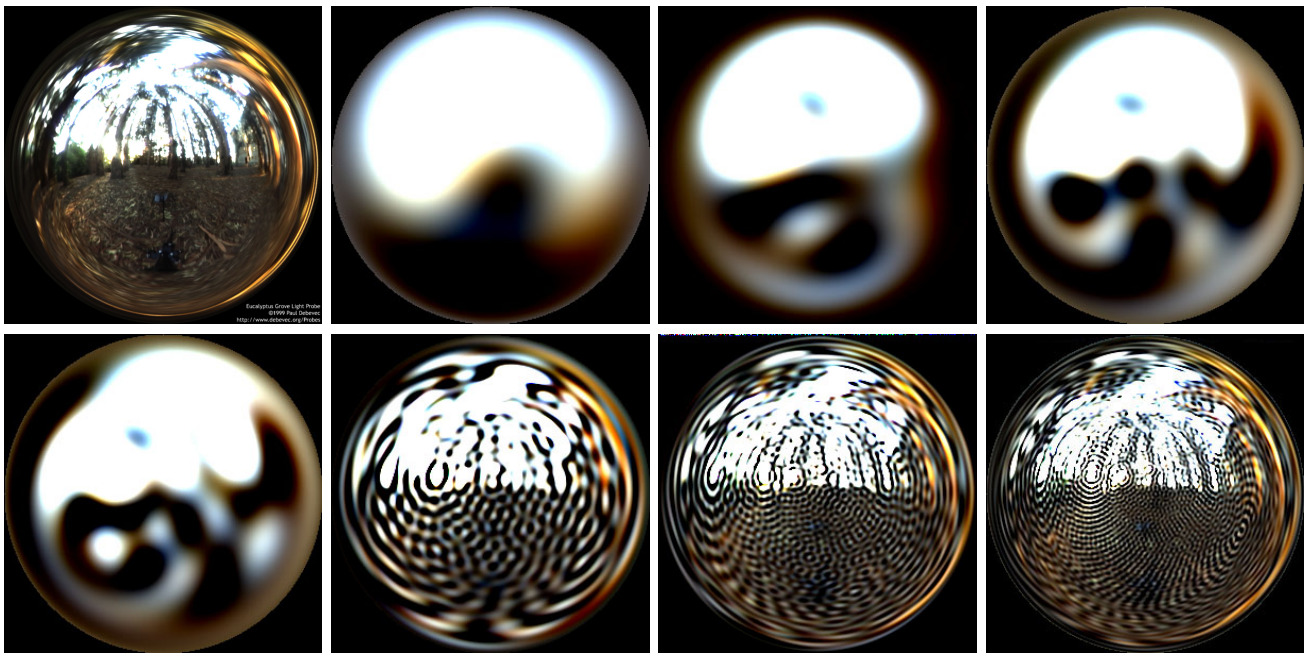


Figure 2.51: First image: input environment map, as a light probe. Other images: Progressively increasing the number of terms in the spherical harmonics decomposition increases accuracy. In that order: 3, 5, 6, 7, 30, 55, 80 spherical harmonic bands (N bands correspond to N^2 coefficients).

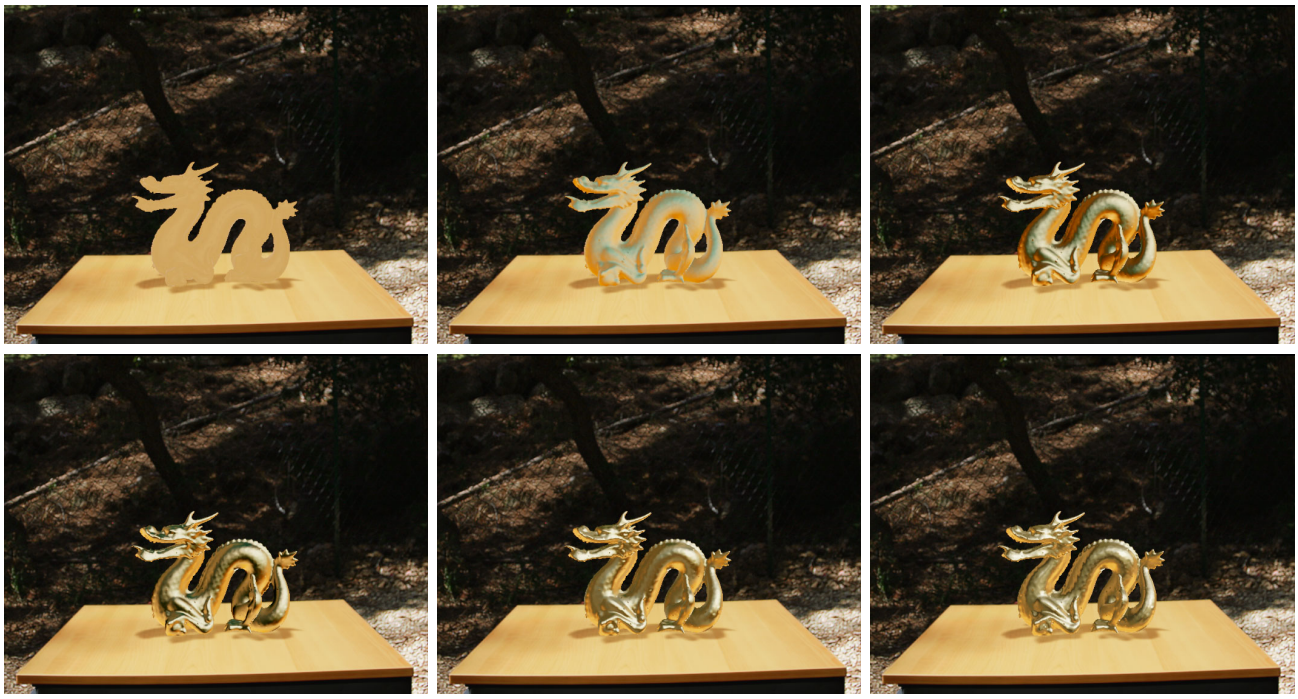


Figure 2.52: Progressively increasing the number of spherical harmonic bands to represent a gold BRDF makes it more shiny. In that order: 1, 2, 3, 4, 6, 12 SH bands (N bands correspond to N^2 coefficients).

they are non-local, and tend to induce ringing artifacts when clamped abruptely. 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 successibe 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/`.

2.1.5 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:

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

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. 2.1.2:

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

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. 2.1.2.

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:

$$L = L_e + \mathrm{diag}(\frac{\rho}{\pi}) G L$$

and by rearranging terms:

$$L = \Big(\mathrm{Id} - \mathrm{diag}(\frac{\rho}{\pi}) G \Big)^{-1} L_e = M^{-1} L_e$$

Solving linear systems in general is out of the scope of this class[23]. 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 $n \to \infty$. 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 $G^{i,j} = \int_{T_i} \int_{T_j} G(x, x') \mathrm{d}x \mathrm{d}x'$ 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

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

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. 2.53. More recent approaches allow for glossy materials[24].

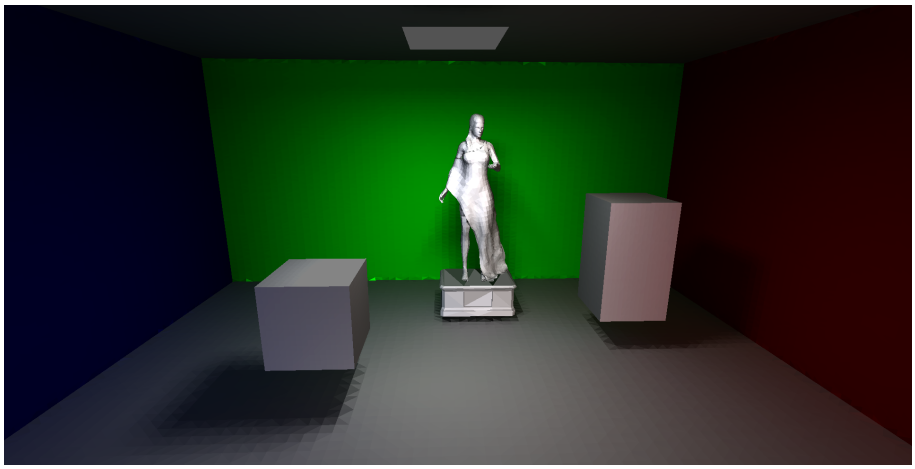


Figure 2.53: Radiosity result – my (very naive) implementation has 100 lines for building and solving the linear system + 450 lines for defining basic classes (`Vector`, `Triangle`, `Mesh`...), reading obj files, constructing the BVH, intersecting. There are 10 light bounces (i.e., Jacobi iterations), 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.

Remark: Nowadays, most work on radiosity has been abandonned: 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).

2.2 Image-Based Rendering

Image-based rendering tries to render a scene from a set of pre-recorded (precomputed renderings or, more commonly, photographed scenes) images rather than simulating light transport. Traditional approaches used to reconstruct a textured mesh from these images (which is only useful for the case of

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

photographed scenes!). Then came image-based rendering techniques that, when navigating inside the environment, tried to directly warp the input images so that they look like they have been rendered from the novel point of view. For instance, a crude implementation is akin to what is done in Google Streeet View, when walking in the street.

More recently Neural Radiance Fields (2020) and Gaussian Splatting (2023) have significantly changed the way we look into this problem as they were the first to produce truly impressive photo-realistic results. We we look into these more modern approaches.

2.2.1 Neural Radiance Fields

Given a set of input images of known camera poses (camera poses can be estimated with computer vision techniques, by matching points), the goal of Neural Radiance Fields is to train a machine learning model – a multilayer perceptron in this case – to predict a radiance and density given a 3-d point in the scene and a ray direction. To understand, you may need to come back to participating media (Sec. 2.1.2). In the context of Neural Radiance Fields, the entire scene is considered as a participating medium: a big volume with continous density and colors. It would consider for instance that a table is a "cloud" which is extremely dense inside the table and empty outside. Inside that table, at each point (x, y, z) in space, there is a high density $\sigma(x, y, z)$ and a (directional) color $c(x, y, z, \theta, \phi)$ that acts like a directional light source, though no scattering occurs and the phase function is kept constant. In this context, the corresponding volume rendering equation becomes:

$$C(\omega_o) = \int_0^\infty T(t)\sigma(P(t))c(P(t), \omega_o)dt$$

where $P(t) = X_0 + t\omega_o$ the position along the ray, and T, the transmittance, is still $T(t) = \exp(-\int_0^t \sigma(P(s))ds$.

Since there is no scattering involved, merely the absorption by the medium of the emitted light, the rendering can be very efficient, and performed in realtime on the GPU. Now, the benefit of this volumetric representation is that everything is continuous, and even smooth provided the density and color fields σ and c are smooth. So, the density and color fields are modeled by a multilayer perceptron (a deep fully connected neural network) taking position[25] and view angles as input. This makes the renderer *differentiable*, and the rendering parameters can now be optimized. The neural network is "trained"[26] to reconstruct the set of input images by adjusting the network parameters. The training requires rendering images with the current guess of network parameters, and the integral is discretized using Monte Carlo integration. Differentiable rendering is a relatively recent and very active trend within computer graphics.

Once the network is trained, the network "knows" the density and directional color at each point in space, and rendering a novel view merely is a matter of evaluating the integral above for a new set of rays originating from a novel camera. An interesting aspect of this approach is that the entire 3d (view-dependant) scene is represented by a single neural network. Since its publication in 2020, this approach has given birth to a multitude of competitors – the original paper by B. Mildenhall et al. has been cited more than 6,000 times in the past 4 years – including approaches relying on grids rather than neural networks.

2.2.2 Gaussian Splatting

More recently, a method called Gaussian Splatting have managed to produce even more realistic renderings based on photographs. The method starts with a point cloud obtained by computer vision

[25] In practice, a *positional encoding*: 3d coordinates are fed to a bunch of sine waves of different frequencies, and these sine waves are used as input instead.

[26] This is *one-shot learning*, i.e., there is no other dataset than the set of input photograph of the scene we want to represent. The word "training" is thus often criticized in this context.

techniques (e.g., Structure-from-Motion that tries to match feature points on images, and iteratively minimize a 3d-to-2d reprojection error). It then places small oriented anisotropic 3-d Gaussians at these points, with some opacities, and also attaches a set of Spherical Harmonics coefficients (see Sec. 2.1.4) for each color channel. The very general idea is similar to NeRF but there are also a number of differences. The rendering is performed by accumulating (i.e., "splatting") Gaussians on the screen with their opacities. The color of the Gaussians are merely determined by evaluating the sum of spherical harmonics for the desired view direction. Since Gaussians are explicitely represented, there is no need for a neural network anymore. At training stage, all parameters, such as Gaussian orientation (represented as quaternions[27]) and sizes, opacities, spherical harmonic coefficients, are all directly optimized within a non-linear optimization routine trying to match the current rendered image to the set of input photographs.

The rendering is usually much faster than NeRF and quality largely improved. Renderings from NeRF and Gaussian Splatting can be seen in Fig. 2.54.

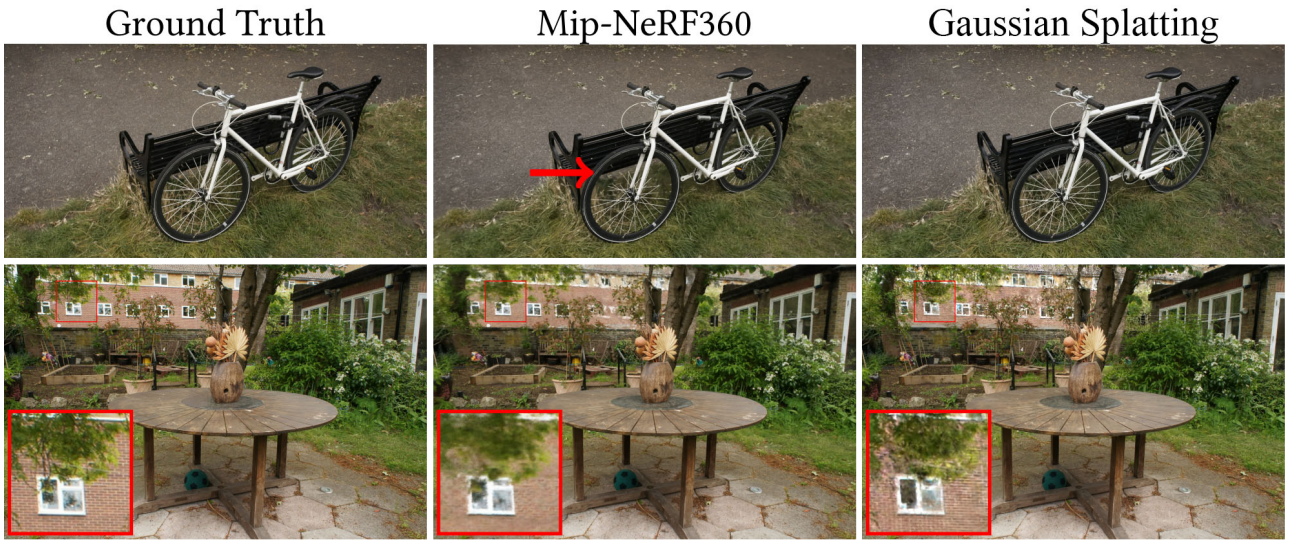


Figure 2.54: Comparison between a Nerf-like approach and Gaussian splatting, with the ground truth photograph from the exact same viewpoint. These approaches allow to move the viewpoint in this scene.

[27]If you are not familiar with quaternions, you can interprete that as a generalization to complex numbers but for storing 3-d rotations instead of 2-d rotations, by writing it as $\omega = a + b.i + c.j + d.k$.

Chapter 3

Image Processing

This chapter covers a couple of image processing techniques that are popular within the computer graphics community.

3.1 Filtering

This section will detail a couple of commonly used filtering operations in computer graphics. Filtering is a technique that denoises an image, and is, for instance, significantly used for rendering or photography. In photography, it can be used for low light conditions that do not allow for a significant number of photons to be averaged on the sensor or for cheap sensors. Raw photographs can be extremely noisy and should be denoised. For rendering, it can be used for realtime rendering. For instance, the few games that are currently capable of performing a full path tracing like we saw earlier, allow for a budget of 0.25 samples per pixel, i.e., 1 path is built for each block of 4 pixels, not hundreds like we did. In this context renderings are extremely noisy and a clean version should be recovered. In the context of rendering, additional information can be leveraged: for instance, it is easy to obtain a noise-free depth image (deph *buffer*) or normal buffer, which can guide the denoising process.

While I will explain basics of filtering operations that can be used outside of the context of image denoising, nowadays best solutions come from dedicated and trained convolutional neural networks.

3.1.1 Gaussian filtering

I will briefly go over Gaussian filtering. We have seen some ways of doing Gaussian filtering in the context of our path-tracer. Gaussian filtering amounts to performing a convolution between a signal and a Gaussian function. In 1-d the Gaussian function is expressed as:

$$h(x) = 1/(\sigma\sqrt{2\pi})\exp((x-\mu)^2/(2\sigma^2))$$

The convolution between the signal f and Gaussian h is defined as, in 1D again and continuously,

$$(f * h)(y) = \int_{-\infty}^{\infty} f(x)h(y-x)\mathrm{d}x$$

This operation averages values in the neighborhood of y with some Gaussian weight so that values far from y contribute less to the average and values close to y contribute more. This tends to blur the image, as can be seen in Fig. 3.1. Filtering a 2-d image with a Gaussian can be performed in several ways.

Noisy image	Gaussian filtered 0.1 s 25 lines	Bilateral filtered 0.24 s 25 lines	Non-local Means 0.4 s 45 lines

Figure 3.1: **Image filtering.** To handle RGB values, we perform 3 independent filterings, 1 for each color channel. **First row**. Our Davy Jones rendering (without fog) at 4spp contains much noise and is computed in 0.6 seconds in 512x512 (left). A simple Gaussian ($\sigma = 7$, 31x31 windows) aggressively blurs the result without accounting for edges. The bilateral filter ($\sigma_x = 7$, $\sigma_v = 45$, 31x31 windows) reduces noise while preserving edges. A naive bilateral filter implementation runs in 0.24s, though a (naive) bilateral grid already brings it to 10 ms and takes 80 lines of code instead. A non-local mean filtering ($h = 5^2 * 98$) similarly preserve edges. **Second row**. Same things at 32 spp – rendering time: 5 seconds. Gaussian: $\sigma = 5$. Bilateral filter: $\sigma_x = 5$, $\sigma_v = 15$ and 31x31 windows. The bilateral grid filtering time is 35 ms and is more expensive since the grid is finer. Non-local mean: $h = 15^2 * 98$. All timings performed using parallel code. *Recall that we did not correlate samples among neighboring pixels, so we could have (largely) reduced noise without post filtering.*

Discrete, separable, truncated, convolution

The first thing to realize is that a d-dimensional isotropic Gaussian function is the product of d 1-dimensional Gaussian functions. It happens that in most cases, we will be interested in isotropic Gaussians anyway. As such, the convolution in d dimensions for isotropic Gaussians is separable. Concretely, it means that a Gaussian convolution over a 2-d image can be performed by first convolving each rows independently with a 1-d Gaussian, and then, convolving each column of the result independently with a 1-d Gaussian.

The second thing to realize is that Gaussian functions drop quickly, such that truncating a Gaussian at 1 standard deviation σ preserves 68% of its integral, and at 2σ, 95% remains and only less than 5% is lost. For filtering purposes, it is often the case that loosing the last 5% is ok[1]. Additionally, in certain cases it can be simpler to use the fact that convolving with a Gaussian of standard deviation σ amounts to convolving twice with a twice cheaper Gaussian of standard deviation $\sigma/2$. Finally, we will see in Sec. 3.1.2 that since Gaussian functions are low pass filters (they smooth thing out and remove high frequencies), Gaussian filtering can be approximately computed on a coarser image resolution.

[1] A few algorithms such as the Sinkhorn algorithm for optimal transport are not robust to Gaussian truncation or other approximations.

Overall, for filters with relatively small σ, a separable truncated convolution can be cheaply performed. A typical algorithm looks like:

Algorithm 1: Compute a Gaussian convolution of image I with a Gaussian of standard deviation σ, assuming 0 outside of the image I.

Input: Image I of size $W \times H$, standard deviation σ
Output: Filtered image F

$S \longleftarrow 2\sigma$ // Support of the filter
$h[-S..S] \longleftarrow 1/(\sigma\sqrt{2\pi}) \exp([-S..S]^2/(2\sigma^2))$ // Precomputes a 1d filter
$T[0..W, 0..H] \longleftarrow 0$ // Temporary image
// Filter each row
for $i = 1..H$ **do** // For each row (in parallel)
  for $j = 1..W$ **do** // For each column
    for $k = \max(j - S, 1) - j.. \min(j + S, W) - j$ **do** // For each filter value
      $T(i, j) += h(k) * I(i, j + k)$
/* Transpose the image so that columns become rows. This can be done inplace if the image is square by swappings rows and columns. Transposing the image allows for better cache coherence rather than operating directly on columns since images are stored rows by rows. */
$T \longleftarrow \text{tranpose}(T)$
// Filter each column
for $i = 1..W$ **do** // For each column (in parallel)
  for $j = 1..H$ **do** // For each row
    for $k = \max(j - S, 1) - j.. \min(j + S, H) - j$ **do** // For each filter value
      $F(i, j) += h(k) * T(i, j + k)$
// Transpose back to get the original orientation.
return *tranpose*(F)

Using FFTs

An important identity states that a convolution in the spatial domain amounts to a product in the frequency domain. A such, denoting $\mathcal{F}$ the Fourier transform, and $\mathcal{F}^{-1}$ its inverse, we have $f * h = \mathcal{F}^{-1}(\mathcal{F}(f).\mathcal{F}(h))$. Assuming our images are periodic and stored as discrete pixels, the discrete analogous to the Fourier transform, called discrete Fourier transform, can be computed via the Fast Fourier Transform algorithm, and the theorem still holds.

The Discrete Fourier Transform. A detailed introduction to the Fourier transform is out of the scope of this class, but I will give some intuition about it. The idea is to decompose a signal into a sum of waves of different frequencies. With a discrete 1-d signal $\{x_n\}$, this can be achieved by computing:

$$X_k = \sum_{n=0}^{N-1} x_n(\cos(\frac{2\pi}{N}kn) - i.\sin(\frac{2\pi}{N}kn)) = \sum_{n=0}^{N-1} x_n e^{-i\frac{2kn\pi}{N}}$$

This amounts to projecting our sequence $\{x_n\}$ onto complex exponential basis functions of integer frequencies $e^{-\frac{i2\pi}{N}kn}$: for the 0^{th} frequency (for $k = 0$) we simply get the average of the signal, for the 1^{st} frequency, it is an average weighted by a complex exponential of 1 period etc. This transform hence well represents the different frequencies in the signal, and one can recover the initial signal based on its frequency decomposition:

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k(\cos(\frac{2\pi}{N}kn) + i.\sin(\frac{2\pi}{N}kn)) = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{i\frac{2kn\pi}{N}}$$

For 2-d images, formulas are similar, except that now the frequency is a 2-d vector representing directional waves. We can thus extract "vertical" frequencies, "horizontal" frequencies, "diagonal" frequencies and so on (see Fig. 3.2):

$$X_{k,\ell} = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x_{m,n}(\cos(\frac{2\pi}{N}(km+\ell n)) - i.\sin(\frac{2\pi}{N}(km+\ell n)))$$

and

$$x_{m,n} = \frac{1}{MN}\sum_{k=0}^{M-1}\sum_{\ell=0}^{N-1} X_{k,\ell}(\cos(\frac{2\pi}{N}(km+\ell n)) + i.\sin(\frac{2\pi}{N}(km+\ell n)))$$

This corresponds to projecting on directional complex exponentials $e^{-\frac{i2\pi}{N}\mathbf{k}.\mathbf{n}}$ where now $\mathbf{k}$ is a 2-d frequency $\mathbf{k} = (k, \ell)$ and n is a 2-d image coordinate $\mathbf{n} = (m, n)$.

This formula seems computationally intensive since each 2-d frequency involves a summation over the entire image plane. Fortunately, two strategies make that very cheap. First, similarly to the separable Gaussian convolution, one can see that a 2-d Fourier decomposition can be achieved by performing 1-d Fourier decompositions alongs rows and then columns of the image. Second is a particularly fast algorithm by Cooley and Tukey called the *Fast Fourier Transform*[2]. This algorithm recursively splits the summation in 2 parts (it hence works best for images of sizes that are power of 2), which results in an algorithm of complexity $\mathcal{O}(N\log(N))$ where (here) N is the number of pixels in the image for 2-d images, or the number of values for 1-d (or any-dimensional) data.

The filtering process hence consists in precomputing the FFT of the Gaussian and of the image, computing their pixel-wise product, and performing an inverse FFT. This technique is ideal for Gaussian filters of large σ. In fact, additional speedup can be obtained by considering the Discrete Cosine Transform, i.e., the restriction of the Discrete Fourier Transform to only cosines (so there is no imaginary part and the cosine transform is real) since the Gaussian function is even[3].

Using recursive approximations

The Deriche filter consists in considering that a Gaussian is composed of a sum of 2 half-Gaussians, that each can be approximated by a recursive causal or anticausal filter by approximating these half-gaussians with sum of (complex) exponentials. He developed the following approximation to filter in 1-d (and again, this can be performed on rows and then columns) that can be expressed, outside of boundaries, with the following :

$$\begin{aligned}
X_i^+ =& \frac{1}{\sigma\sqrt{2\pi}}\left[1.0051\,x_i + \left(\exp(-\frac{1.512}{\sigma})(-2.9031\cos(\frac{1.475}{\sigma}) + 1.021\sin(\frac{1.475}{\sigma})) + 0.8929\exp(-\frac{1.556}{\sigma})\right)x_{i-1}\right.\\
&\left. + \left(\exp(\frac{-3.068}{\sigma})\left(-0.8929\cos(\frac{1.475}{\sigma}) - 1.021\sin(\frac{1.475}{\sigma})\right) + 1.898\exp(\frac{-3.024}{\sigma})\right)x_{i-2}\right]\\
&+ \left(2\exp(\frac{-1.512}{\sigma})\cos(\frac{1.475}{\sigma}) + \exp(\frac{-1.556}{\sigma})\right)X_{i-1}^+\\
&- \left(2\exp(-\frac{3.068}{\sigma})\cos(\frac{1.475}{\sigma}) + \exp(-\frac{3.024}{\sigma})\right)X_{i-2}^+\\
&+ \exp(-\frac{4.58}{\sigma})X_{i-3}^+
\end{aligned}$$

[2]See `https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm`. A typical C++ library for that is *FFTW*

[3]See *A Survey of Gaussian Convolution Algorithms*, `https://www.ipol.im/pub/art/2013/87/article.pdf`. Note that FFT assumes periodic boundary conditions while DCT assumes symmetric boundary conditions, which may work better in most cases.

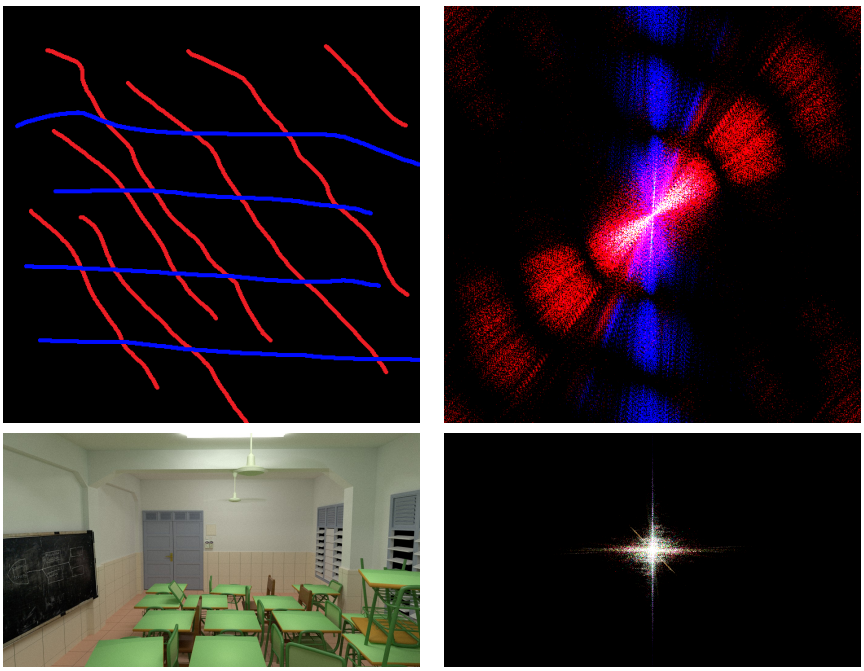


Image Discrete Fourier Transform (log-abs)

Figure 3.2: Discrete Fourier Transform results on images. The result is shown in the log-domain, and only the magnitude of the complex values are shown (the phase is harder to interprete). As can be seen, repeating edges in the images result in lines that are orthogonal to the edges in the Fourier domain. The Fourier Transform makes it easier to find repeating oriented structures.

for the causal part (i.e., values depend on previously computed values), and

$$\begin{aligned}
X_i^- = &\frac{1}{\sigma\sqrt{2\pi}}\left[\left(\exp(-\frac{1.512}{\sigma})(-0.8929\cos(\frac{1.475}{\sigma})+1.021\sin(\frac{1.475}{\sigma}))+1.898\exp(-\frac{1.556}{\sigma})\right)x_{i+1}\right.\\
&+\left(\exp(\frac{-3.068}{\sigma})\left(-2.9031\cos(\frac{1.475}{\sigma})-1.021\sin(\frac{1.475}{\sigma})\right)+0.8929\exp(\frac{-3.024}{\sigma})\right)x_{i+2}\\
&\left.+1.0051\ \exp(\frac{-4.58}{s})x_{i+3}\right]\\
&+\left(2\exp(\frac{-1.512}{\sigma})\cos(\frac{1.475}{\sigma})+\exp(\frac{-1.556}{\sigma})\right)X_{i+1}^-\\
&-\left(2\exp(-\frac{3.068}{\sigma})\cos(\frac{1.475}{\sigma})+\exp(-\frac{3.024}{\sigma})\right)X_{i+2}^-\\
&+\exp(-\frac{4.58}{\sigma})X_{i+3}^-
\end{aligned}$$

and $(x * h) = X^+ + X^-$ gives the final result[4]. A result can be seen in Fig. 3.3.

3.1.2 Bilateral filtering

An issue with Gaussian filtering is that it tends to blur edges, so it cannot be used for heavy denoising (Fig. 3.1). To alleviate this issue, the bilateral filter adds a weighting term that penalize spatial smoothing in places where edges occur. Bilateral filtering a 1-d signal $\{f_i\}$ can be performed using the following formula to compute the i^{th} value of the result:

[4]See *A Survey of Gaussian Convolution Algorithms* for longer, more accurate, recursions. They have accompanying code, but beware, they have typos in Algorithm 10 of the paper: for $k = K$, $b_K^- = -a_K b_0^+$ and the first sum of the anticausal filter runs from 1 to K : $\sum_{k=1}^{K} b_k^- f_{n+k} - \sum_{k=1}^{K} a_k q_{n+k}^-$ (i.e., $b_0^- = 0$

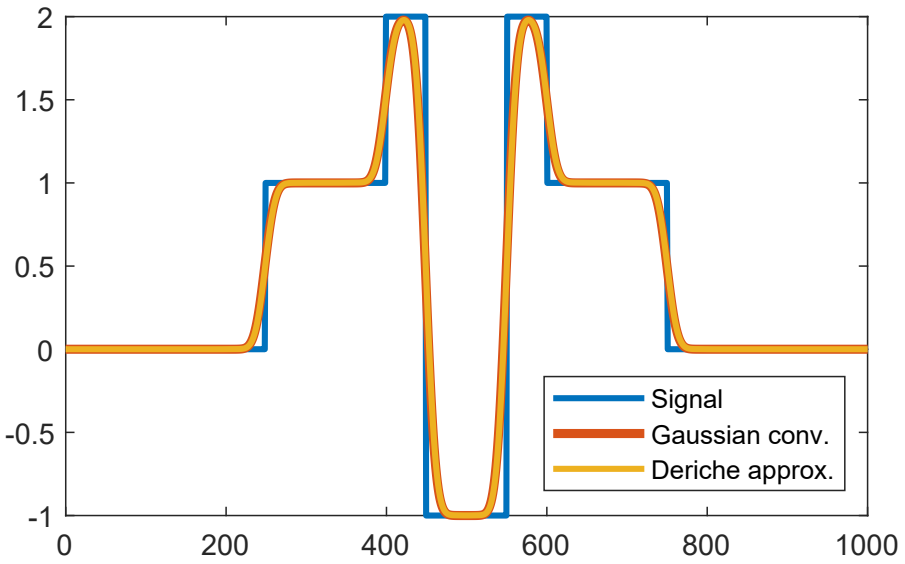


Figure 3.3: Filtering a signal with a Gaussian ($\sigma = 10$) vs. using Deriche's approximation (the 3-term recurrence of Sec. 13). You can't see both curves ? Sure, that means the approximation is good!

$$F_i = \frac{\sum_{j=-K..K} \exp(-\frac{(i-j)^2}{2\sigma_x^2}) \exp(-\frac{(f_i-f_j)^2}{2\sigma_v^2}) f_{i+j}}{\sum_{j=-K..K} \exp(-\frac{(i-j)^2}{2\sigma_x^2}) \exp(-\frac{(f_i-f_j)^2}{2\sigma_v^2})}$$

Without the second term involving σ_v, this would exactly be a (truncated) Gaussian filter. However, the extra term is such that large differences in the signal (i.e., edges) reduce the contribution of the neighboring pixels. With $\sigma_v = \infty$, this amounts to Gaussian blurring, while $\sigma_v = 0$ results in no denoising at all. This formula can again be easily generalized to 2-d images or higher dimensional signals[5]. This formula implemented naively can become quite costly for large images and neighbors: like for Gaussians, it results in a complexity of $\mathcal{O}(K^2MN)$ with K the width of the window of neighbors, and MN the number of pixels in the image. However, it cannot be further sped up with the same tricks as this filter is nonlinear: it cannot be directly computed via the Fourier Transform, and it is not separable, so it cannot be performed dimension by dimension.

An interesting trick has been found via the *Bilateral Grid* data structure[6]. This considers that the above filtering can be computed as a 3-D Gaussian filtering in the space $\times$ intensity domain with homogeneous coordinates (for the normalization), and that since this is a low-pass filter, it can be performed at a much coarser resolution. In fact, a filter of standard deviation σ can be performed on a grid that is σ times coarser, and this amounts to using a Gaussian function of standard deviation 1 on this coarser grid (assuming the size of the image can be divided by σ and σ is an integer). Specifically, a simple version of the algorithm is as follows:

[5] In fact, it can also be generalized by considering that the second term does not depend on $(f_i - f_j)^2$ but on $(g_i - g_j)^2$ with g a *different* image that serves as a guide. This is called the Cross Bilateral Filter.

[6] *A Fast Approximation of the Bilateral Filter using a Signal Processing Approach*, `https://people.csail.mit.edu/sparis/publi/2006/tr/Paris_06_Fast_Bilateral_Filter_MIT_TR.pdf`

Algorithm 2: Compute a bilateral filtering of image I using a bilateral grid structure, a spatial standard deviation σ_x, and intensity standard deviation σ_v. $[x]$ denotes the rounding of x to the nearest integer.

Input: Grayscale image I of size $W \times H$, standard deviations σ_x, σ_v
Output: Filtered image F

$\tilde{I} \longleftarrow zeros(W/\sigma_x, H/\sigma_x, 255/\sigma_v, 2)$ // downsampled homogeneous data // Downsample
for $i = 1..H$ **do** // For each row
  for $j = 1..W$ **do** // For each column
    $\tilde{I}([i/\sigma_x], [j/\sigma_x], [I(i,j)/\sigma_v], 1) += I(i,j)$
    $\tilde{I}([i/\sigma_x], [j/\sigma_x], [I(i,j)/\sigma_v], 2) += 1$
/* Standard isotropic truncated Gaussian convolution with $\sigma = 1$. A 5x5 window can be used. */
$\tilde{F} \longleftarrow \text{GaussianFilter}(\tilde{I}, \sigma = 1)$
/* Upsampling to the original resolution. A tri-linear interpolation is recommended ; a nearest filtering somewhat works, so let's do that. */
for $i = 1..H$ **do** // For each row
  for $j = 1..W$ **do** // For each column
    $F(i,j) = \tilde{F}([i/\sigma_x], [j/\sigma_x], [I(i,j)/\sigma_v], 1) / \tilde{F}([i/\sigma_x], [j/\sigma_x], [I(i,j)/\sigma_v], 2)$
return F

3.1.3 Non-local means

A similar idea shared with non-local means is to weigh pixels that are more *similar* to the current pixel when spatially averaging. In non-local means filtering, the idea is to perform a weighted average of a large neighborhood (the original paper[7] mentions a 21x21 window), where the weight is computed as a Gaussian function of a similarity metric. While in bilateral filtering, this similarity metric is just the difference between pixel values, here the similarity metric is an ℓ^2 distance between 7x7 neighborhoods. The formula becomes (directly in 2D) :

$$F_{i,j} = \frac{\sum_{k=-10..10,\ell=-10..10} \exp(-\frac{\sum_{m=-3..3,n=-3..3}(f_{i+k+m,j+\ell+n} - f_{i+m,j+n})^2}{h}) f_{i+k,j+\ell}}{\sum_{k=-10..10,\ell=-10..10} \exp(-\frac{\sum_{m=-3..3,n=-3..3}(f_{i+k+m,j+\ell+n} - f_{i+m,j+n})^2}{h})}$$

Again, this can become quite costly to evaluate (each pixel is seen $21^2 * 7^2 = 21609$ times). Accelerations and generalizations have been proposed, for instance by using fast nearest neighbor features datastructures such as PatchMatch[8]).

In the context of denoising, more recent restoration algorithms make use of deep convolutional neural network to learn noise models.

3.2 Color Matching

A typical problem for colorists is to get a good color palette in an image. While this can be performed manually[9], we will see popular techniques for transfering the color palette from one image to another.

[7] *A non-local algorithm for image denoising*: `https://www.iro.umontreal.ca/~mignotte/IFT6150/Articles/Buades-NonLocal.pdf`

[8] *PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing*: `https://gfx.cs.princeton.edu/pubs/Barnes_2009_PAR/patchmatch.pdf`

[9] and I urge you to get familiar with color spaces. We have seen the RGB color space that directly maps to displays, but other exists such as Lab (that is perceptually uniform and ideal to compute perceptual distances between colors ;

We will call *model* image the image from which we want to extract the color style, and the *input* image the image we want to transform.

Input image	Model image	Mean/stdev 80 ms 50 lines	Sliced 5s 20 lines

Figure 3.4: **Color matching**. Matching the mean and standard deviation does not precisely respect the color distribution (e.g., see the brighter ground on the second row) but gives the overall atmosphere. A sliced optimal transport approach precisely respects the color distribution, but this can produce artifacts such as too large contrasts (e.g., the background wall in both results). I used 100 iterations for the sliced optimal transport approach.

3.2.1 Simple mean/standard deviation matching

A pioneer work on color transfer is a simple procedure that matches means and standard deviations of pixel values, in some $l\alpha\beta$ color space[10]. The $l\alpha\beta$ color space represents some luminance value l and chrominances α (yellow-blue) and β (red-green) and are computed using a linear transform of the log of the LMS color space. The LMS color space represents the eye's response to light relative to Long, Medium and Short cones. Converting RGB values (normalized in $[0, 1]$) to $l\alpha\beta$ values can be done with the following transforms:

$$\begin{pmatrix} L \\ M \\ S \end{pmatrix} = \begin{pmatrix} 0.3811 & 0.5783 & 0.0402 \\ 0.1967 & 0.7244 & 0.0782 \\ 0.0241 & 0.1288 & 0.8444 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$\begin{pmatrix} l \\ \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} 0.5774 & 0.5774 & 0.5774 \\ 0.4082 & 0.4082 & -0.8165 \\ 0.7071 & -0.7071 & 0 \end{pmatrix} \begin{pmatrix} \log_{10}(L) \\ \log_{10}(M) \\ \log_{10}(S) \end{pmatrix}$$

Once all the pixels of the input and model images have been transformed to this $l\alpha\beta$ color space, means and standard deviations[11] are matched for each color channel independently. Specifically,

supposedly, a Euclidean distance of 1 represents a "just noticeable difference" in term of colors), HSV (that is intuitive and good for colorists), XYZ (that is good for spectral rendering), CMYK (that is good for printing), LMS (that represent eye's photoreceptor responses) and many others.

[10] *Color Transfer between Images*: `https://users.cs.northwestern.edu/~bgooch/PDFs/ColorTransfer.pdf`

[11] Recall that computing a standard deviation can be done via $\sigma_X = \sqrt{\left(\frac{1}{N}\sum_i x_i^2\right) - \left(\frac{1}{N}\sum_i x_i\right)^2}$

denoting ν_α^i and σ_α^i the means and standard deviation of the α color channel of the input image, called α^i , and μ_α^m σ_α^m the means and standard deviation of the α channel of the model image called α^m, the α channel of the transformed image reads $\alpha^t = \frac{\sigma_\alpha^m}{\sigma_\alpha^i}(\alpha^i - \mu_\alpha^i) + \mu_\alpha^m$. And similarly for the l and β channels. The assumption behind this model is that pixels (in $l\alpha\beta$) follow an isotropic Gaussian distribution.

Finally, the transformed values are converted back to RGB using:

$$\begin{pmatrix} \log_{10}(L) \\ \log_{10}(M) \\ \log_{10}(S) \end{pmatrix} = \begin{pmatrix} 0.5774 & 0.4082 & 0.7071 \\ 0.5774 & 0.4082 & -0.7071 \\ 0.5774 & -0.8165 & 0 \end{pmatrix} \begin{pmatrix} l \\ \alpha \\ \beta \end{pmatrix}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 4.4679 & -3.5873 & 0.1193 \\ -1.2186 & 2.3809 & -0.1624 \\ 0.0497 & -0.2439 & 1.2045 \end{pmatrix} \begin{pmatrix} L \\ M \\ S \end{pmatrix}$$

Results can be seen in Fig. 3.4.

3.2.2 Sliced optimal transport matching

Another approach for transfering colors consists in considering an optimal transport problem[12]. Here, we will only consider images with the same number of pixels, and this so called optimal transport problem becomes, in this case, a linear assignment problem. The goal here is to find a one to one assignment between pixels of the input image (for instance in the RGB color space) and pixels of the model image minimizing some cost. We hence try to match two point clouds of pixels that live in a 3-dimensional (RGB) space. Once the matching is done, we can simply move the pixels of the input image towards their assigned pixel in the model image, and the color distributions will perfectly match.

Such linear assignment problem can be quite costly to compute and the *sliced* approach consists in projecting the initial problem of matching pixels in 3-d onto 1-d lines where optimal transport is trivial. The overall algorithm consists in, iteratively, first find a uniformly random direction on the sphere. For that, we can go back to the Global Illumination Compendium, and use the formula:

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

Then, we project the input and model point clouds (i.e., pixel RGB values) onto this direction (that is, simply computing the dot product between this random direction and the pixel coordinate). Once projected, the optimal matching that is of interest to us simply consists in matching the first projected point of the input point cloud to the first projected point of the model point cloud, the second with the second and so on. To do that, we simply sort the two projected point clouds according to their computed dot product, while keeping track of the pixel index (for instance, using a `std::pair<>`: by default, `std::sort` will sort according to the first element in the `pair`). Finally, we advect each point of the input point cloud in the randomly chosen direction by the projected distance to its matched

[12]Somewhat similar results can be found in *N-dimensional probability density function transfer and its application to color transfer*: `https://github.com/frcs/colour-transfer/blob/master/publications/pitie05iccv.pdf` but the technique is that of *Sliced and Radon Wasserstein Barycenters of Measures*: `https://hal.archives-ouvertes.fr/hal-00881872/document` which has a similar framework to that of *Wasserstein Barycenter and its Application to Texture Mixing*: `https://hal.archives-ouvertes.fr/file/index/docid/476064/filename/TexturesECCV10.pdf`

point in the model image. We finally iterate with newly chosen random directions. To summarize the algorithm:

Algorithm 3: Sliced optimal transport color transfer algorithm.

Input: Color input image I and model M, both consisting of n pixels.
Output: Color matched image: I is modified in-place

for $iter = 1..nbiter$ **do**
  $v \longleftarrow$ random_direction()
  `// Project. We store the dot product and pixel index as a pair of values`
  for $i = 1..n$ **do** `// For each pixel`
    $projI(i) \longleftarrow (\langle I(i), v\rangle, i)$
    $projM(i) \longleftarrow (\langle M(i), v\rangle, i)$
  `// Sort according to the dot product`
  sort($projI$)
  sort($projM$)
  `// Advect initial point cloud.`
  for $i = 1..n$ **do** `// For each pixel`
    $I(projI(i)[2]) += (projM(i)[1] - projI(i)[1])v$
return I

Results can be seen in Fig. 3.4[13].

3.3 Image Retargeting

Input image | Cropped | Stretched | Seam Carving

Figure 3.5: **Image Retargeting**. We would like to make the input image square. Cropping can loose useful information or, like here, the image composition. Stretching, here, distorts the scultpure. The Seam Carving approach removes vertical seams that are least useful. It here removed 450 vertical seams on a 1800x1350 image. The seam carving result was computed in 22 seconds and less than 50 lines of C++ code. Image by Jean-Pol Grandmont, CC-BY 3.0.

Another common problem in image and video processing is that of resizing an image/video so that it matches a certain display size. For instance, adapting a 4:3 movie to a 16:9 screen, or going from a portrait to landscape image mode (without rotating). Simple solutions involve cropping (but that can loose particularly important information) or stretching (but that can significantly distort the image).

We will now see a popular approach to downsizing images while preserving as much content as possible: the *Seam Carving* method[14]. Without loss of generality, I will focus on the case of horizontal

[13] This approach precisely matches histograms: it is thus sensitive in the image content. If a landscape input image with 40% sky and 60% grass is recolored from a beach model image of 60% sky and 40% sand, the recolored image will have 20% unpleasant beach-colored sky. We have published a partial sliced optimal transport framework to solve this issue: *SPOT: Sliced Partial Optimal Transport*: `https://hal.archives-ouvertes.fr/hal-02111220/document`

[14] *Seam Carving for Content-Aware Image Resizing*: `http://www.eng.tau.ac.il/~avidan/papers/imretFinal.pdf`

downsizing. As this name says, this approach finds a seam – a vertical 1-pixel wide path – in the image that would go unnoticed if removed. Repeating this operation multiple times allows to stretch the image down while preserving most of the interesting content and reducing deformations.

The **first step** is to determine what kind of features need to be preserved, ie., what would not go unnoticed if removed. A simple heuristic is that we want to preserve edges: any flat uniform surface can be stretched down without producing much artifacts, but stretching down a tree may pose more problems. The first step is thus to compute an *energy map* that detects edges. A simple approach for that is to consider that the value of the energy map $E(x, y)$ at the pixel (x, y) is given by

$$E(x, y) = \text{abs}(I(x + 1, y) - I(x - 1, y)) + \text{abs}(I(x, y + 1) - I(x, y - 1))$$

where $I(x, y)$ is the intensity of the original image. For color images, a simple approach is to take

$$I(x, y) = R(x, y) + G(x, y) + B(x, y)$$

where R, G, and B are respectively the red, green and blue channels of the image (Fig. 3.6).

Figure 3.6: Original photo and the associated energy that consists of edges.

The **second step** is to compute a cumulated energy map, $C(x, y)$, that describes optimal paths. Such methods come from the field of dynamic programming, but are relatively easy to understand. This map can be computed with a rather simple formula:

$$C(x, y) = \min(C(x - 1, y - 1), C(x, y - 1), C(x + 1, y - 1)) + E(x, y)$$

where E is the energy map above. This function tells you that if you currently are at the position (x, y) in the image, on row y, your best move to go to row $y - 1$ is to either go toward the pixel $(x - 1, y - 1)$ or $(x, y - 1)$ or $(x + 1, y - 1)$ (choosing the one that minimizes this cumulative energy map). And the value at pixel (x, y) is the minimum cost of reaching this pixel via a vertical seam. This map is called *Value Function* (Fig. 3.7).

The **third step** is to go to the very last row of the cumulative energy map, and select the pixel with the lowest cumulative energy. This is the starting point of the seam.

The **fourth step** is to start from the pixel chosen in step 3, and progressively build the seam. For that, you will use the insight given in step two: if your current pixel is (x, y), your best move to build the seam is to choose the pixel that minimizes the cumulative energy $C(x, y)$ among pixels $(x - 1, y - 1)$, $(x, y - 1)$, and $(x + 1, y - 1)$. This operation is called *backtracking* (Fig. 3.8).

The **final step** is to remove the seam. A simple way to see that is that all pixels on the left of the seam remain unchanged, while all pixels on the right of the seam are translated by one pixel to the left. After this step is done, you can finally crop your image by one pixel: you now have successfully rescaled the original image width by one pixel. Repeat the operation as necessary to reduce the image width by the desired amount[15].

[15]It is also possible to enlarge an image by considering multiple seams at once, and duplicating them.

Figure 3.7: From the energy (left) we compute the cumulative energy (here, $y = 0$ is the top row).

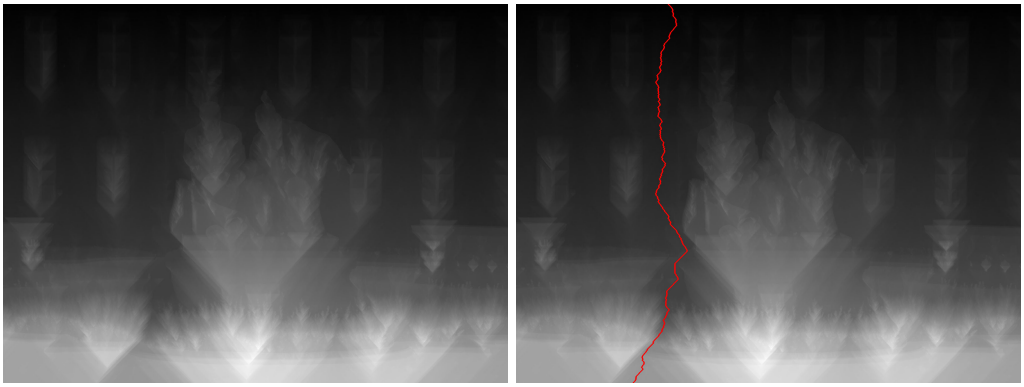


Figure 3.8: Fom the cumulative energy, we compute a seam that goes through consecutive pixels of lowest cumulative energy.

3.4 Poisson Image Editing

It is quite common as well to integrate (part of) a photo onto another photo, for compositing purpose (Fig. 3.9). This kind of problem arose in early photography when photographic plates were very sensitive to blue-violet wavelengths leading to over-exposed sky. In 1852, Hippolyte Bayard proposed to combine two negatives – this technique was first used by William Lake Price in 1855 and popularized by Gustave Le Gray in 1856–1858 (Fig. 3.10. Most photos of skies from the XIX's century were faked. In modern digital photography era, we are able to copy-paste photographic elements[16]. However, naively copy-pasting does not always produce realistic results, and may require precise matting/detouring. While matching colors is a good start, it does not solve the matting problem.

The idea behind Poisson image editing[17] for seamless cloning is that the human eye is more sensitive to color differences than absolute values (see Adelson's checkerboard, Fig. 3.11). As such, it tries to preserve gradients from an input image to be pasted onto another image.

3.4.1 A simple approach

Preserving gradients can be expressed as the minimization of the following functional:

$$\min_u J(u) = \min_u \int_\Omega \|\nabla u(x) - \nabla f(x)\|^2 \mathrm{d}x$$

[16] For a direct application to sky manipulation, see *Sky is Not the Limit: Semantic-Aware Sky Replacement*, `https://sites.google.com/site/yihsuantsai/research/siggraph16-sky`

[17] *Poisson Image Editing*: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.451.1843&rep=rep1&type=pdf`

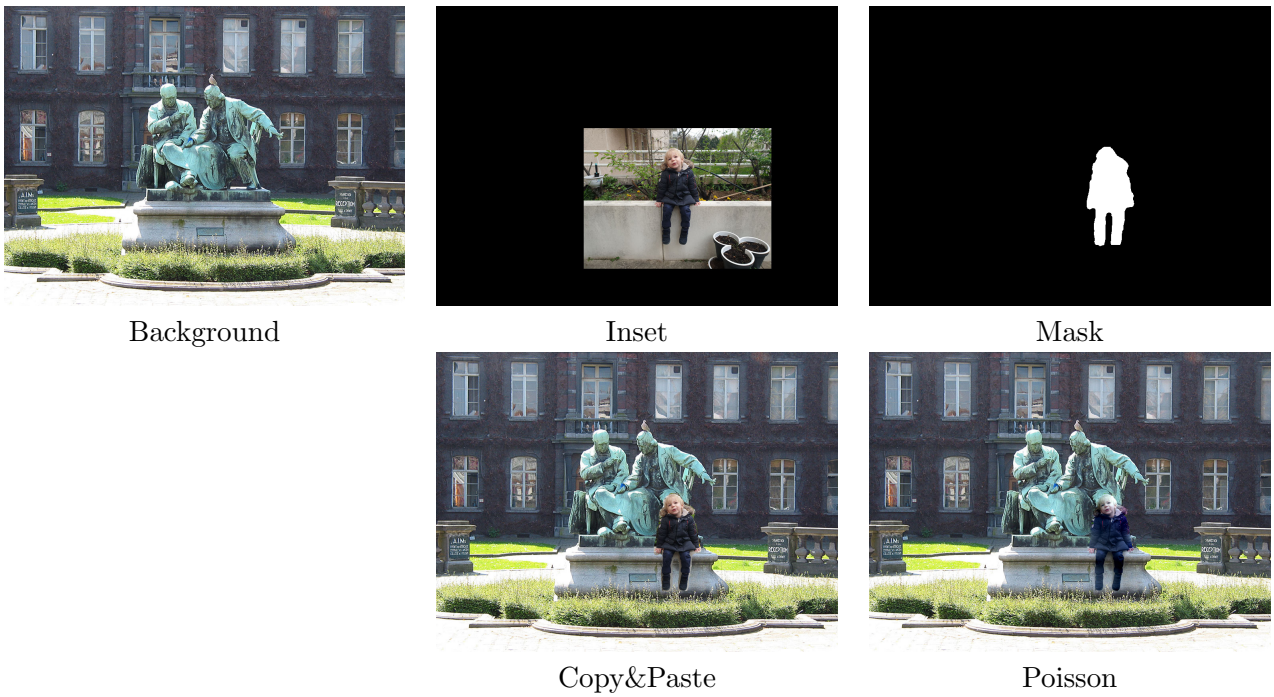


Figure 3.9: **Seamless Cloning**. We would like to insert the kid image onto the background. Copy-pasting the image using a crude mask leads to unnatural results: boundaries are highly visibles, and colors do not match. Using Poisson Image Editing for the task of seamless cloning reduces these artifacts. This took 200ms to compute on a 1008x752 image, and 80 lines of code using a multiscale strategy, or just 25 lines without multiscale. Statue image by Jean-Pol Grandmont, CC-BY 3.0.

$$u(x) = g(x) \qquad x \in \partial\Omega$$

where Ω is the inpainted area, u is the solution we are looking for, f is the image to paste and g is the *background* image.

We will denote $h = u - f$ for simplicity. Minimizing this functional amounts to solving the following PDE, called the Poisson equation[18]:

$$-\Delta h = 0 \qquad x \in \Omega$$
$$h = g - f \qquad x \in \partial\Omega$$

This can be demonstrated by considering a small variation around h[19]: $h + \varepsilon v$, where $v = 0$ on the boundary $\partial\Omega$ (so that $h + \varepsilon v$ still respects the original boundary conditions). Minimizing the functional J amounts to having all variations of J equals 0 around the minimizer. We thus compute the variation of J:

$$\frac{J(h+\varepsilon v) - J(h)}{\varepsilon} = \frac{1}{\varepsilon}\int_\Omega \|\nabla h + \varepsilon \nabla v\|^2 - \|\nabla h\|^2 \mathrm{d}x \tag{3.1}$$
$$= \frac{1}{\varepsilon}\int_\Omega 2\varepsilon\langle \nabla h, \nabla v\rangle + \varepsilon^2 \|\nabla v\|^2 \mathrm{d}x \tag{3.2}$$
$$= 2\int_\Omega \langle \nabla h, \nabla v\rangle \mathrm{d}x + \varepsilon \int_\Omega \|\nabla v\|^2 \mathrm{d}x \tag{3.3}$$

We are interested in infinitesimally small variations, to get a notion of derivative of J:

$$\lim_{\varepsilon \to 0} \frac{J(h+\varepsilon v) - J(h)}{\varepsilon} = 2\int_\Omega \langle \nabla h, \nabla v\rangle \mathrm{d}x$$

[18] We often use minus the Laplacian to keep a symmetric positive definite linear system instead of negative.
[19] or by applying Euler-Lagrange formulas!

Figure 3.10: A photography by Le Gray in 1857 (*La Grande Vague*) featuring an early combination of two negatives, one for the sea, one for the sky.

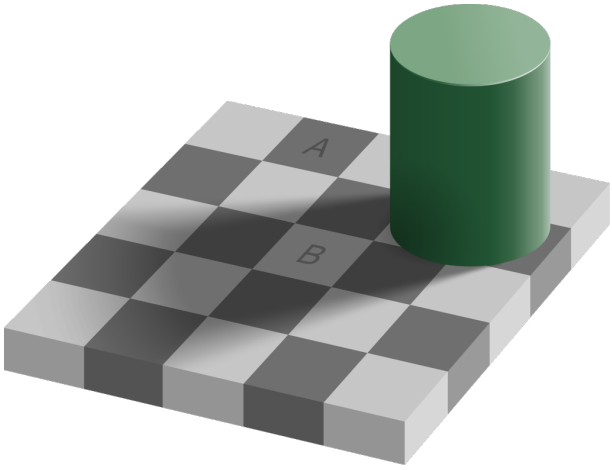


Figure 3.11: In Adelson's checkerboard, the squares flagged A and B are of the same absolute intensity: the human eye is more sensitive to intensity differences than to absolute intensities.

By applying Green's identity (i.e., integrating by parts!):

$$\lim_{\varepsilon \to 0} \frac{J(h+\varepsilon v) - J(h)}{\varepsilon} = 2 \int_{\partial\Omega} v \langle \nabla h, \vec{n} \rangle \mathrm{d}\Gamma - 2 \int_{\Omega} v \Delta h \mathrm{d}x$$

where $\vec{n}$ is the normal of the boundary $\partial\Omega$. Since we have taken v to be 0 on the boundary, and setting this infinitesimally small variation to 0, we have:

$$\int_{\Omega} v \Delta h \mathrm{d}x = 0$$

In this context, the fundamental lemma of calculus of variations says that $\Delta h = 0$ (intuitively, if the integral of the product of a function F with all tests functions is zero, then F ought to be zero).

We can discretize this equations by realizing that $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$, and using centered finite differences discretization of the second derivatives. We will denote $h_{i,j}$ the discretized value of h at pixel (i, j). This yields:

$$-\Delta h(x_i, y_j) \approx 4h_{i,j} - h_{i+1,j} - h_{i-1,j} - h_{i,j+1} - h_{i,j-1}$$

This leads to a linear system, that we can again solve with Jacobi iterations (see Sec. 2.1.5). Taking more time to detail the Jacobi method here, the idea is that a square matrix M can be decomposed in a sum of a diagonal matrix D and two upper and lower triangular matrices (minus the diagonal) E and F such that $M = D - E - F$. Solving $Mx = b$ corresponds to solving $Dx = (E + F)x + b$, so $x = D^{-1}(E + F)x + D^{-1}b$. The idea behind Jacobi iterations is to consider the iterations $x^{k+1} = D^{-1}(E + F)x^k + D^{-1}b$ which converge if the largest eigen value of $D^{-1}(E + F)$ is smaller than 1 (a sufficient condition is for the matrix to be strictly diagonal dominant).

In our case, this leads to the iterates:

$$h_{i,j}^{n+1} = \frac{1}{4}\left(h_{i+1,j}^n + h_{i-1,j}^n + h_{i,j+1}^n + h_{i,j-1}^n\right) \qquad (i,j) \in \Omega$$

$$h^{n+1} = g - f \qquad (i,j) \in \partial\Omega$$

Performing these iterations (in parallel!) should yield a solution h, and u can be recovered using $u = h + f$.

However, these iterations converge extremely slowly in the context of the Poisson equation. In fact, so slowly that the difference between two consecutive iterations can reach machine precision, and the iterations get stuck way before mathematical convergence. To alleviate this issue, we will proceed in a multiscale strategy (see Fig. 3.12)[20]. We first downsample the images by some power of two (on this result, I downsampled by a factor 16 in both width and height), iteratively solve the problem at this resolution, then upsample the result by a factor of 2, then iteratively solve the problem but using the upsampled result as a starting point, and so on.

In practice, for dowsampling the mask, we can use a nearest neighbor downsampling (i.e., querying the pixel value at the nearest integer pixel) as we cannot average binary mask values. For dowsampling RGB values, simple averaging is sufficient. For upsampling, linear interpolation would be good – in my results, I used a nearest neighbor for simplicity.

3.4.2 Possible improvements

The solution heavily relies on boundary values of $g - f$ – the rest is merely a diffusion process that does not involve the input images. As such, artifacts that can occasionally be observed mostly come from boundary values. It has been suggested to find an optimal boundary using dynamic programming, which reduces bleeding artifacts in practice[21].

While multigrid approaches converge in linear time complexity, it can still remain costly in practice. It has been suggested to use mean value coordinates to solve a similar approximate problem[22].

Finally, we have considered minimizing a squared ℓ^2 norm, resulting in a smooth membrane interpolation. However, many other approaches allow to account for edges, minimize other norms, add anisotropy etc. Notably:

ℓ^1-norm minimization. The goal is here to minimize $\int |u(x) - f(x)|\mathrm{d}x$ which penalizes less extreme values. The corresponding PDE becomes:

$$\operatorname{div}\left(\frac{\nabla u(x) - \nabla f(x)}{|\nabla u(x) - \nabla f(x)|}\right)$$

[20] The *correct* way to do it would be a geometric multigrid approach – however, proceeding as we will do works reasonably well in practice.

[21] *Drag-and-Drop Pasting*: `https://www.cse.cuhk.edu.hk/~leojia/all_project_webpages/ddp/dragdroppasting.pdf`

[22] *Coordinates for Instant Image Cloning*: `https://www.cse.huji.ac.il/~danix/mvclone/files/mvc-final-opt.pdf`

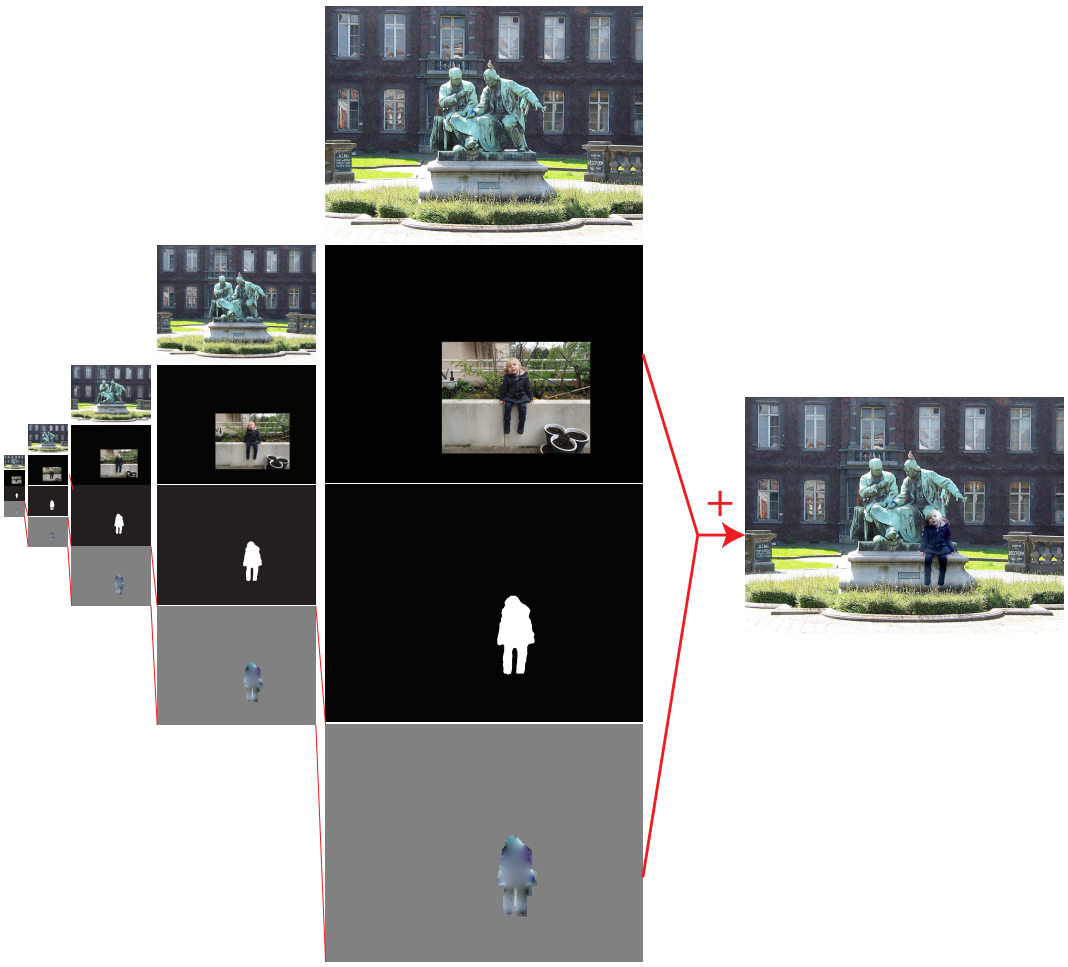


Figure 3.12: A multiscale strategy for solving the Poisson image editing problem.

It is of course much more difficult to solve as it is non-linear.

Spatial weighting. It can sometimes be interesting to weigh differently some parts of the image, using a weight map w. We thus want to minimize $\int w(x)\|u(x) - f(x)\|^2 \mathrm{d}x$. The corresponding PDE to solve is:

$$\operatorname{div}\left(w(x)(\nabla u(x) - \nabla f(x))\right)$$

General formula. When minimizing a general form of the problem $\int F(|u(x) - f(x)|)\mathrm{d}x$ for some functional F, the resulting PDE becomes:

$$\operatorname{div}\left(\frac{F'(|\nabla u(x) - \nabla f(x)|)}{|\nabla u(x) - \nabla f(x)|}(\nabla u(x) - \nabla f(x))\right)$$

Chapter 4

Geometry Processing

This chapter will give basic tools of geometry processing and modeling.

4.1 Representing shapes

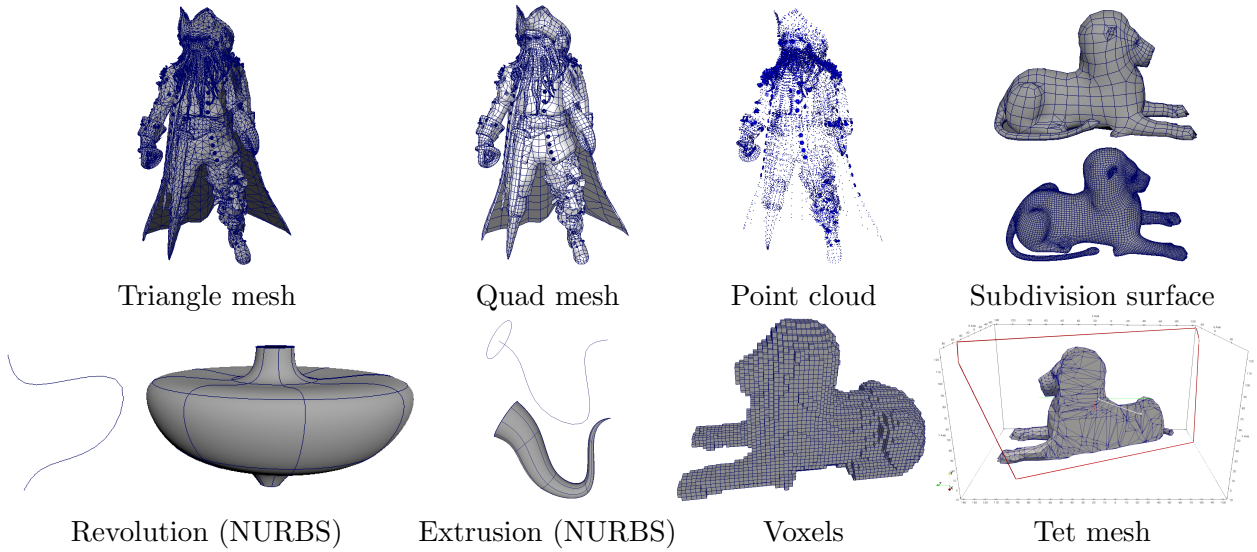


Figure 4.1: **Geometry representation**. Surfaces can be represented by triangle meshes, quad meshes, subdivision surfaces (here, Catmull-Clark on a quad mesh), or parametric surfaces such as surfaces of revolution or extrusion (here, using NURBS). Volumes are often represented by tetrahedral meshes, voxel grids or hexahedral meshes (voxel grids are particular instances of hexahedral meshes). In-between are point set representations. Other representations include cell complexes, triangle (or polygon) soups or implicit representations.

There are various ways to represent shapes (Fig. 4.1), the two main categories being implicit and parametric representations. Parametric representations represent geometries with parametric equations of the form $(x, y, z) = f(u, v)$. As such, a centered unit sphere, for instance can be described by the parametric equation:

$$x = \sin\theta \cos\phi \tag{4.1}$$
$$y = \sin\theta \sin\phi \tag{4.2}$$
$$z = \cos\theta \tag{4.3}$$

It is however not always the case that complex geometries have nice equations of this form. A common way to represent parametric shapes is via tensor products of curves, such as **B-splines**, **Bezier curves**

or **Non-Uniform Rational B-Splines** (NURBS). For instance, a Bezier curve is a polynomial curve defined by a set of control points $\{P_i\}_{i=0..n}$ and is defined as $P(t) = \sum_{i=0}^{n} B_i^n(t)P_i,\ t \in [0,1]$ and $B_i^n(t) = \binom{n}{i}t^i(1-t)^{n-i}$ are Bernstein polynomials – it can easily be constructed via De Casteljau's algorithm. From this definition in 1D, we can define a **Bezier surface** by instead using an array of control points $\{P_{i,j}\}_{i=0..n,j=0..m}$ as $P(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)P_{i,j}$ hence defining a smooth, polynomial, parametric surface. Similarly, **subdivision surfaces** gives recursive subdivisions schemes (e.g., Catmull-Clark, Doo Sabin, Loop...) to smoothly refine coarse triangular or quadrangular meshes – these coarse meshes can be seen as "control points" as well. But surfaces need not be smooth, can be defined by piecewise polynomial functions (e.g., **splines** – the simplest of them being piecewise linear functions), ... and actually, a **triangle mesh** is a parametric surface (finding the parameters u and v is a problem called mesh parameterization). More related to triangle meshes are **quadrilateral meshes** (or quad meshes for short), or **triangle soups** (sets of triangles without any connectivity stored between them).

Alos, mostly for rendering, surfaces can be stored as **point clouds**. For rendering, usually, a small disk or sphere is rendered at the location of each point. This is typically the representation of 3d scanners such as LiDaR, that scan an environment using lasers and place a point at the location where light has been reflected.

The other class of representation is implicit representations, where shapes are defined as solutions of some equations. For the case of a sphere, this amounts to representing a centered unit sphere as the set of points P solutions to:

$$\|P\|^2 - 1 = 0$$

In practice, representing surfaces with implicit functions can be occasionally useful. This is done for shapes like **meta-balls** (or blobs) defined as the surface solution to $\sum_{i=1}^{n} \frac{1}{\|P-P_i\|^2} - c = 0$ where $\{P_i\}$ is a set of points defining the centers of these blobs, and c is the isosurface parameter. A generalization of meta-balls are **convolution surfaces**, that are roughly equivalent to meta-balls but using curves instead of points to define the center of each blob. It can also be useful to represent surfaces using **distance fields** (for instance stored on voxel grids, it can be signed or unsigned), for example when doing fluid simulation, to facilitate collision detection[1] or boolean operation of surfaces (e.g., computing the intersection of two geometries with implicit surfaces is way easier than using meshes: the intersection of two implicit surfaces defined by equations f and g is simply $\max(f,g)$ and one can build trees of boolean operators to represent complex objects, a process called *Constructive Solid Geometry*).

To represent volumes, implicit representations are also easily used (e.g., instead of using the solution $f(P) = 0$, one can use $f(P) < 0$ to define a volume – **voxel grids** can also be used to define which voxels are inside or outside of the geometry, which is widely used for fluid simulations), as well as parametric representations. Typically, triangles meshes are extended in 3d to form **tetrahedral meshes**, while quad meshes are extended to 3d to form **hexahedral meshes** (a voxel grid is a particular case of a hexahedral mesh). **Cell complexes** represent volumes (or surfaces, or geometries in any dimension) by cells of any shapes partitioning the geometry – this is particularly the case of Voronoï diagrams that we will manipulate.

Finally, **procedural shapes** are represented via algorithms that produce them. This is for instance widely used to generate complex terrains, cities, clouds etc. The final models obtained via these procedures can be either parametric or implicit.

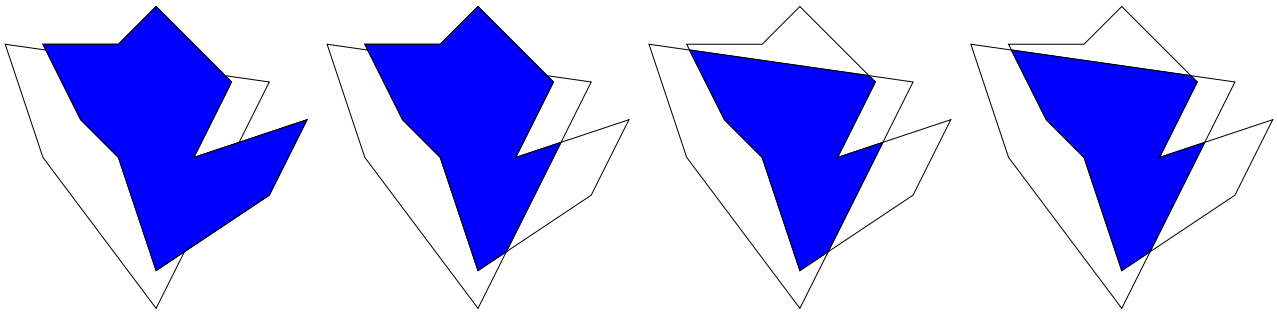


Figure 4.2: Iterations of the outer loop of the Sutherland-Hodgman algorithm for clipping the polygon in blue by a quadrilateral. The algorithm iteratively removes half spaces.

4.2 Polygon clipping

An important tool that we will use later relates to the problem of clipping polygons (or polyhedras) by convex polygons (resp. convex polyhedras). This corresponds to restricting a given polygon called the *subject polygon* to the *inside* of a convex polygon called the *clip polygon*. This is used for many applications. For instance, it is used for rendering: all polygons in the scene can be clipped to the view frustum (i.e., strictly restricting the scene to what is within the field of view for efficiency), polygons can be cut when using acceleration structures such as regular grids or kd-trees, etc. We will use it later to build Voronoï diagrams (Sec. 4.3.3).

A well-known algorithm for this task is the Sutherland-Hodgman algorithm described next, in 2-D first. The basic operation in this algorithm is the clipping of the subject polygon by an infinite line (or rather, a half-space delimited by an infinite line). Then this basic operation is simply repeated for all edges of the convex clip polygon (Fig. 4.2), thus considering all half spaces delimited by each edge of the clipping polygon.

To clip a polygon by a line, the algorithm traverses all edges of the subject polygon and progressively builds a new updated polygon. For a given edge E, either it is completely outside of halfspace of interest and this edge is simply ignored, or this edge is completely inside and it is added to the new polygon, or this edge is partly inside and partly outside of the considered halfspace, and only part of this edge is added.

The algorithm reads:

[1]See how it can be used for character skinning in *Implicit Skinning: Real-Time Skin Deformation with Contact Modeling*: `http://rodolphe-vaillant.fr/permalinks/implicit_skinning_project.php`

Algorithm 4: Clips the subjectPolygon by a convex clipPolygon. The corresponding C++ code has 32 lines, plus the basic operators and classes for `Vector`.

```
Input: subjectPolygon, and a convex clipPolygon
Output: outPolygon
1  for clipEdge in clipPolygon do // For each edge of the clip polygon
       // Clip the subjectPolygon by a half-space
2      outPolygon = new Polygon();
3      for i ← 0 to subjectPolygon.vertices.size()-1 do // For each vertex of the subject polygon
           // Test the subject polygon edge with vertices (i-1, i)
4          Vector curVertex = subjectPolygon.vertices[i];
5          Vector prevVertex =
            subjectPolygon.vertices[(i > 0)?(i − 1):(subjectPolygon.vertices.size()-1)];
           // Compute inter.  between the infinite line supported by clipEdge and edge (i-1, i)
6          Vector intersection = intersect(prevVertex, curVertex, clipEdge);
7          if curVertex inside clipEdge then
8              if prevVertex not inside clipEdge then
                   // The subject polygon edge crosses the clip edge, and we leave the clipping area
9                  outPolygon.vertices.add(intersection);
10             outPolygon.vertices.add(curVertex);
11         else if prevVertex inside clipEdge then
               // The subject polygon edge crosses the clip edge, and we enter the clipping area
12             outPolygon.vertices.add(intersection);
13     subjectPolygon = outPolygon;
14 return outPolygon
```

The point of intersection between the (finite) edge [A, B] and the (infinite) line for which two points u and v are known can be computed similarly to the line-plane intersection in Sec. 2.1.2, that is $P = A + t(B - A)$ with $t = \frac{\langle u-A,N\rangle}{\langle B-A,N\rangle}$ where N is the normal to the line (u, v) and has coordinates $(v_y - u_y, u_x - v_x)$. If $t < 0$ or $t > 1$, no intersection exists with this segment (since we made sure both ends of the edge are on opposite sides of the plane, this should not happen). Similarly, the test *inside* tells on which side of the clipEdge the vertex is. It returns true for a point P if $\langle P - u, N\rangle \leq 0$ for u a point on the clipEdge (here, make sure N is the outwards normal to the clipEdge!).

This algorithm can be easily extended to clip polyhedral domains or triangle meshes (in 3D). One simply needs to perform the Sutherland-Hodgman to each facet of the domain (note that clipping a triangle mesh will produce a mesh that does not *only* contain triangles), by iterating over all facets of the clipping mesh, over all facets of the subject mesh, and finally over each edge of each facet of the subject mesh. This will iteratively remove 3d half-spaces delimited by infinite planes, and the intersection point between each edge of each facet and the cutting plane is found exactly like the line-plane intersection formula of Sec. 2.1.2. However, after cutting half of the space, the hole needs to be filled. This can be done by realizing that the new added polygons correspond (only) to consecutive vertices that have been added while cutting each facet. These new vertices should be traversed in the correct order so as to add the filled area (Fig. 4.3).

4.3 Voronoï diagrams and Delaunay triangulations

Given a set of samples $\{P_i\}$, generally in 2-d or 3-d space, we would like to triangulate them. Delaunay triangulation is one possible triangulation that has interesting properties (Fig. 4.4). Delaunay triangulations also happen to be the dual of Voronoï diagrams, a decomposition of the domain into convex cells called Voronoï cells – i.e., the triangulation produced by connecting adjacent cells by a triangle edge – where Voronoï diagram vertices are Delaunay triangles circumcenters. It is uniquely

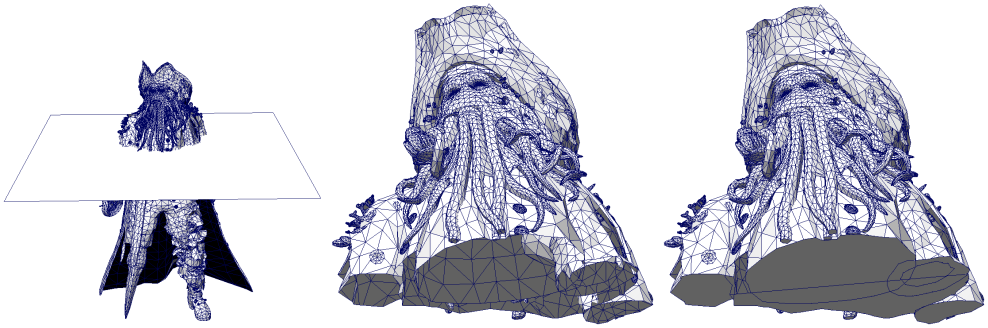


Figure 4.3: Cutting Davy Jones with a plane. Applying Sutherland-Hodgman's algorithm to each facet allows for clipping in 3D, but an additional step fills the hole produced (here, the hole does not form only closed loops).

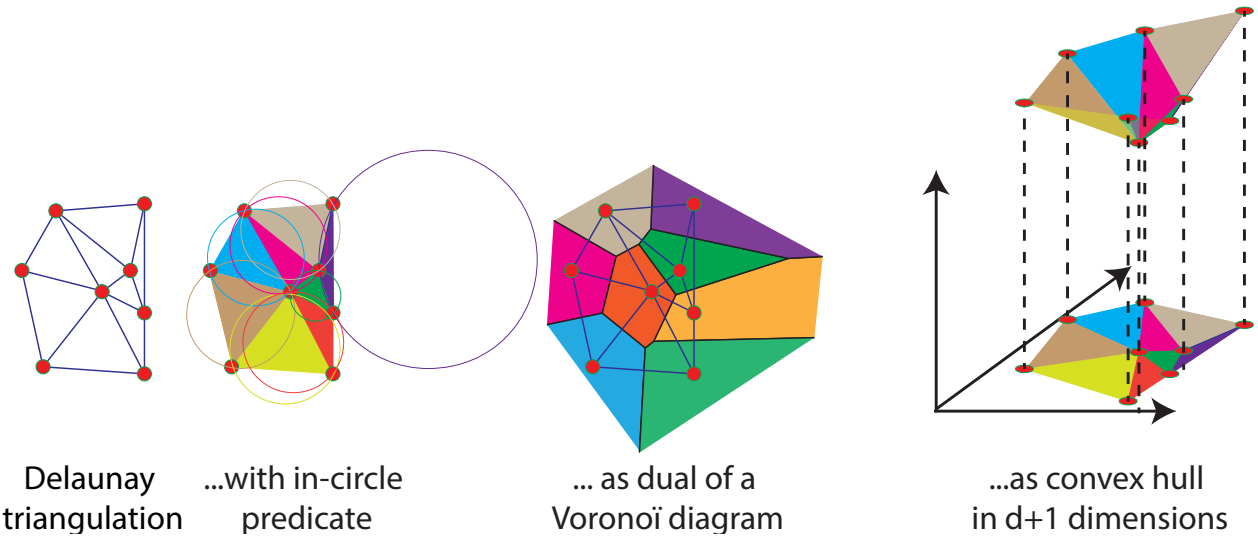


Figure 4.4: A Delaunay triangulation of a set of samples (left) is a triangulation that respects the in-circle property, is the dual of a Voronoï diagram, and the convex hull of these samples lifted with a parabola. Here, the Voronoï diagram has 2 closed cells, and 6 infinite open cells (truncated for display purpose).

defined for samples that are in *general positions*, that is, when adjacent triangles do not have cocyclic vertices (vertices on the same circle, e.g., two triangles forming a rectangle).

Properties that define Delaunay triangulations and Voronoï diagrams are:

- The circumcircle of each Delaunay's triangle does not encompass any other vertex. We usually implement this property using an *in-circle* predicate, that checks whether a given sample is within a given triangle's circumcircle. The circumcircle of triangle ABC of center K and radius r is a bisector of all edges. Denoting $u = B - A$ and $v = C - A$, and $M = \frac{A+B}{2}$ and $N = \frac{A+C}{2}$ the middle of AB and AC, then K, the intersection of both bisectors is such that $\langle u, K - M\rangle = \langle v, K - N\rangle = 0$, so that $\langle u, K\rangle = \langle u, M\rangle$, and $\langle v, K\rangle = \langle v, N\rangle$. This defines two linear equations:

$$\begin{pmatrix} u_x & u_y \\ v_x & v_y \end{pmatrix} \begin{pmatrix} K_x \\ K_y \end{pmatrix} = \begin{pmatrix} \langle u, M\rangle \\ \langle v, N\rangle \end{pmatrix}$$

  Using Cramer's rule, and denoting $u^\perp = (u_y, -u_x)$ and $v^\perp = (v_y, -v_x)$, this yields the circumcenter

$$K = \frac{\langle u, M\rangle v^\perp - u^\perp \langle v, N\rangle}{u_x v_y - u_y v_x}$$

The radius is then trivially obtained, and the *in-circle* predicate merely compares distances to the circumcenter.

Similarly in 3d, the circumsphere of a tetrahedron $ABCD$ can be obtained the same way, by considering the intersection of 3 planes passing through the middles M, N and O, of each edge $u = B - A$, $v = C - A$ and $w = D - A$, leading to the solution of a 3×3 linear system:

$$K = \frac{\langle u, M\rangle(v \times w) - \langle v, N\rangle(u \times w) + \langle w, O\rangle(u \times v)}{\langle u, v \times w\rangle}$$

The consequence of that is that if 4 points A, B, C and D (in 2d, here ordered clockwise) are cocyclic (i.e., they belong to the same circle), the Delaunay triangulation is not unique, since 2 triangulations are equally valid: the first consists of the triangles ABC and ACD, the second consists of the triangles ABD and BCD.

- Each point within the Voronoï cell associated with sample P_i is closer to P_i (usually using the Euclidean norm[2]) than to any other sample P_j:

$$\|P - P_i\|^2 \leq \|P - P_j\|^2, \quad \forall i \neq j$$

  In this context, P_i are often called "sites". An intuition is that if your samples represent the location of bakeries, you would belong to the Voronoï cell of your nearest bakery. As such, each vertex of a Voronoï diagram is the intersection of bisectors, and the Voronoï diagram of a set of points consists of Voronoï cells that are polygonal and convex. This can be implemented with the help a predicate determining on which side of a bisector a sample resides.

- The Delaunay triangulation of $\{P_i\}$ is the convex hull of $\{(P_i, \|P_i\|^2)\}$ (or of any other isotropic parabolic lifting).

We will review an algorithm for Delaunay triangulation and two for Voronoï diagrams (though both could be used in the two contexts since it is trivial to go from the primal to the dual structure). In general, Voronoï diagrams are important in computer graphics, but also as a general data structure as it is an acceleration structure to find closest points within a dataset.

4.3.1 Bowyer–Watson algorithm

The Bowyer–Watson algorithm is an algorithm to compute a Delaunay triangulation in arbitray dimension. I will describe it in 2d, but it easily extends to higher dimension.

In 2d, you would start with a gigantic triangle encompassing the entire point set to triangulate (for instance, take an equilateral triangle whose basis is slightly below the bottom of the bounding box of all points but of length $W + 3H$ with W and H the width and height of the bounding box – this should create a triangle that largely encompasses the bounding box). Then we will progressively add each point to the triangulation. For that, we iterate over each point P_i, and for each P_i we check if any triangle from the current triangulation has its circumcircle encompassing P_i. We remove all these triangles from the triangulation, thus creating a hole in the mesh. We then fill this hole by creating new triangles connecting P_i to all edges forming the border of this hole. When the algorithm terminates, we simply clean up the triangulation by removing triangles connected to the vertices of the initial gigantic triangle (Fig. 4.5).

Stated like that, the algorithm sounds quite simple, but this hides small details that make it fast. In fact, a naive implementation in 2D would make it $\mathcal{O}(N^2)$ (for each inserted point, you would look for all triangles whose circumcenter encompasses this point), although it is possible to bring that to

[2]See how using L_p norms can help produce hexahedral meshes for example, in *Lp Centroidal Voronoi Tessellation and its applications* `https://hal.inria.fr/inria-00600251/PDF/LpCVT.pdf`

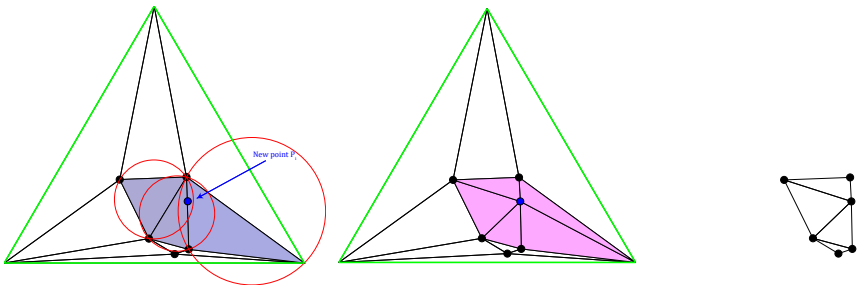


Figure 4.5: Computing the Delaunay triangulation of these 6 samples requires first to add a gigantic triangle (in green) encompassing all points. Then, assuming the triangulation of 5 of these points was done previously (left), we want to insert the 6th point (in blue) in the triangulation. We first determine which triangle it belongs to and then we progressively propagate to adjacent triangles testing for the in-circle predicate. Here, 3 triangles have their circumcircle encompassing P_i. We remove these triangles and instead create new triangles between P_i and the boundary of this hole (middle ; the hole will not contain other points than P_i). Ultimately, we remove the big green triangle to obtain the final triangulation. My quick'n dirty monothreaded code is around 140 C++ lines, takes about 1.3 seconds for 100K vertices, and around 1min30 for 1M vertices (most of the time is spent locating vertices on the mesh – fast libraries using quad trees would do that 10-100x faster). Adding about 35 lines for a regular grid acceleration structure (rasterizing triangles to store all triangles that fall inside each cell of a 100x100 grid) brings timings down to 600ms for 100K triangles and less than 8 seconds for 1M triangles.

$\mathcal{O}(N\log(N))$. The thing is to realize that only the triangles around the newly added P_i will need to be altered. The solution is to first find in which triangle from the existing triangulation P_i belongs. And then, to only verify the *incircle* predicate of neighboring triangles, possibly propagating over a larger neighborhood while the predicate is true. To allow for efficient propagation and navigation in the triangulation, one needs to have quick access to the neighbors of a triangle. For that, the triangle datastructure should now contain the 3 indices of triangles adjacent to each edge in addition to the 3 indices referencing the array of vertices.

To find the triangle in which P_i belongs, a simple solution is to randomly pick any triangle T in the triangulation[3]. We launch a ray from the barycenter Q of T towards P_i and check which of the 3 edges of T it intersects. If no intersection is found, it means P_i belongs to T (otherwise, since the barycenter is inside T and P_i would be outside, there would be an intersection). If an intersection is found, we go to the triangle at the opposite of the intersected edge and repeat the operation. This efficiently navigates in the mesh triangles (though quad-tree based solution are generally faster).

Here, "launching a ray" means testing if a segment $[P_iQ]$ intersects an edge (e.g., $[AB]$), one solution could be to compute the point of intersection between infinite lines and check whether parameters lie within the $[0,1]$ range. A simpler (and more efficient) option is to consider that P_i and Q should be on opposite sides of the infinite line (AB) and that A and B should be on opposite sides of the infinite line (P_iQ). After having computed the normal $AB^\perp$, checking if P_i and Q are on opposite sides means verifying that $\langle P_i - A, AB^\perp\rangle\langle Q - A, AB^\perp\rangle < 0$, and similarly for the other condition.

Locating the triangle t containing a point P is thus performed with the following snippet (assuming an appropriate datastructure storing `triangles`):

```
1  int locateTriangle(const Vector& P) {
2     int i, t = 0;
3     do {
4        Vector Q = barycenter(t);
```

[3]In my implementation, I use a 100x100 regular grid to help locate a reasonably close starting triangle. I also do not remove bad triangles but merely invalidate them, to simplify memory management.

```
        for (i = 0; i < 3; i++) {
            if (intersect(Q, P, vertices[triangles[t].vtx[i]], vertices[triangles[t].vtx[(i + 1) % 3]])) { // segment-segment intersection ?
                t = triangles[t].neighbor[i];
                break;
            }
        }
    } while (i != 3);
    return t;
}
```

4.3.2 Jump Flooding

The jump flooding algorithm[4] is a simple and efficient algorithm to compute Voronoï diagrams on pixel grids, that is embarrassingly parallel, and that can also be used to propagate any information in the grid (e.g., to compute distance maps).

The idea of this algorithm is to see the Voronoï diagram as a fire spreading from the input sites. After some time, the various fire fronts will meet in the bissectors. However, spreading fire iteratively one pixel at a time would require at most as many iterations as the size of the input image. The idea of Jump Flooding is to spread first at very large distances (at a distance of $W/2$ with W the width of the image), and at each iteration to divide the step size by two, hence performing $\log_2(W)$ iterations (Fig. 4.6).

So, at the begining the image is entirely black (i.e., consisting of invalid data) except at the locations of the sites from which fire will start, where the site index is recorded. The algorithm then scans all pixels $p_{i,j}$ and checks whether a site index has been stored at $p_{i+k,j+l}$ with k and l in $\{-s, 0, s\}$, where s, the step length, is initially set to $W/2$. The shortest distance between the current pixel $p_{i,j}$ and the 9 sites whose indices are stored at pixels $p_{i+k,j+l}$ is computed, and the corresponding closest site is stored in a **second** image at $p'_{i,j}$. The process is repeated after having halved the step length s and swapped image p and p', until $s = 1$. In general, the approximation error is already extremely small. But if higher accuracy is needed, a few additional iterations at $s = 1$ can be performed. The approach is very fast and its speed does not depend (or almost) on the number of seeds (Fig. 4.7). Note that this algorithm usually converges even faster in higher dimension as fire propagates faster. And since we only read values from image p and write in image p', all pixels can be computed in parallel, which makes this algorithm ideal for GPU. Finally, in addition to propagating the seed indices, it can propagate other information such as the distance to the nearest seed, which can be used to compute distance maps very efficiently.

```
void JFA(int step, const int* prevIter, int* curIter) {
#pragma omp parallel for
    for (int i = 0; i < H; i++) {
        for (int j = 0; j < W; j++) {
            Vector2D p(j, i);
            double minDist2 = std::numeric_limits<double>::max();
            int bestSite = -1;
            for (int k = -1; k <= 1; k++) {
                for (int l = -1; l <= 1; l++) {
                    int i2 = i + k*step, j2 = j + l * step;
                    if (i2 < 0 || j2 < 0 || i2 >= H || j2 >= W || prevIter[i2*W + j2] <0) continue;
                    double dist2 = (seeds[prevIter[i2*W + j2]] - p).getSquaredNorm();
                    if (dist2 < minDist2) {
                        minDist2 = dist2;
```

[4] *Jump Flooding in GPU with Applications to Voronoï Diagram and Distance Transform* https://www.comp.nus.edu.sg/~tants/jfa/i3d06.pdf

```
                    bestSite = prevIter[i2*W + j2];
                }
            }
        }
        curIter[i*W + j] = bestSite;
        distance[i*W + j] = minDist;  // optionally stores a distance map
      }
    }
  }

  void compute(std::vector<int> &curIter) { // curIter will receive the result
    std::vector<int> prevIter(W*H, -1);
    curIter.resize(W*H);
    for (int i = 0; i < seeds.size(); i++) { // seeds an array of sample coordinates
      prevIter[((int)seeds[i][1])*W + (int)(seeds[i][0])] = i; // puts seed numbers in the image
    }
    for (int k = W/2; k >= 1; k/=2) { // log2 W iterations of JFA
      JFA(k, &prevIter[0], &curIter[0]);
      prevIter.swap(curIter);   // this just swaps pointers and is in O(1)
    }
    if ((int)(log2(W))%2 == 1)  // makes sure the result is in the correct std::vector
      prevIter.swap(curIter);
  }
```

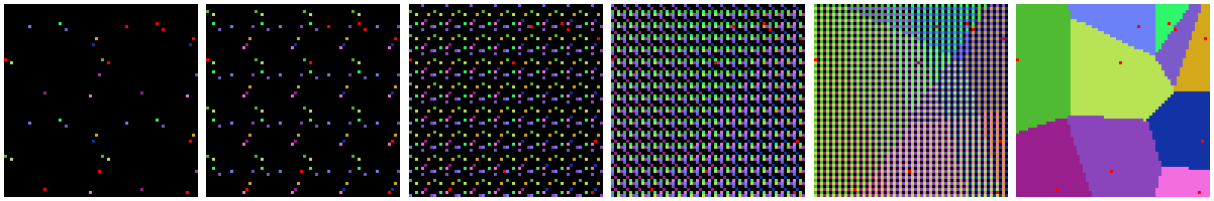


Figure 4.6: Iterations of the Jump Flooding Algorithm. Seeds are displayed as red pixels. In this 64x64 pixel grid, 6 iterations are needed (log2(64)).

Fig. 4.7 also shows a result where seeds are placed within all pixels belonging to the cat shape of Fig. 2.33 (all brown pixels in Fig. 2.33). You may notice that some interior edges form the **medial axis**. The medial axis is the set of points P such that placing a sphere of maximal radius centered at P that remains inside the shape will touch the shape in at least two points. Medial axes are also important in computer graphics – they allow to build skeletons of objects, allow for topological analysis, or can be used as shape descriptors. It can be shown that in 2-d, if (the boundary of) a shape is sampled with a set of points, the set of edges of the Voronoi Diagram that are completely inside the shape form the medial axis[5].

4.3.3 Voronoï Parallel Linear Enumeration

Clipping half-spaces. We have seen that Sutherland-Hodgman's polygon clipping algorithm is an efficient way to clip a polygon by iteratively removing half-spaces defined by infinite lines. The idea of Voronoï Parallel Linear Enumeration [6] is to treat each Voronoï site independently and compute their Voronoï cell independently in parallel. To compute the Voronoï cell of site P_i, we start with a large shape largely enclosing the entire point set (e.g., an extremely large quadrilateral), and we use

[5] Note that this is note the case in 3D – see *Stability and Computation of Medial Axes: a State-of-the-Art Report*, `https://hal.archives-ouvertes.fr/hal-00468690/document` – nor on our 2D pixel grid since the shape boundary has a certain width

[6] *Variational Anisotropic Surface Meshing with Voronoi Parallel Linear Enumeration*, `https://members.loria.fr/Bruno.Levy/papers/vorpaline_IMR_2012.pdf`

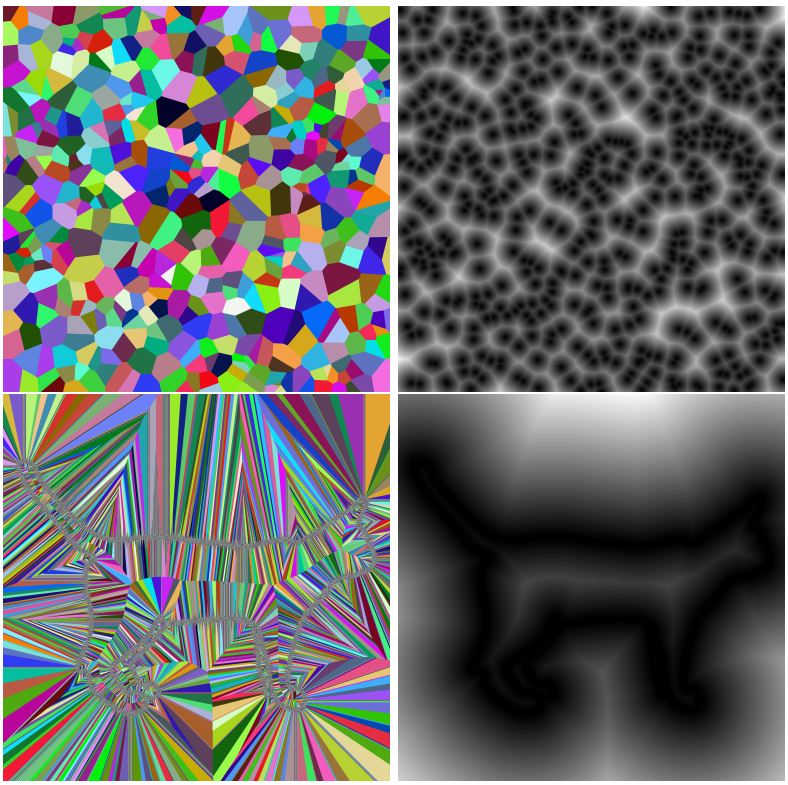


Figure 4.7: The Jump Flooding algorithm computes a Voronoï diagram of these 2048x2048 images in 80ms in parallel (it is almost independent on number of seeds) and about 40 lines of code. At the same time, it can compute a distance map or propagate any other information at no additional cost. Seeds need not be isolated points: the bottom result shows the algorithm run on the cat of Fig. 2.33. As you can see, the medial axis is a subset of the Voronoï "edges". Also note that sites can be grouped if desired, in which case multiple sites would have the same ID.

Sutherland-Hodgman polygon clipping algorithm to remove all half-spaces defined by the infinite lines that are bissectors between P_i and all P_j. Specifically, we cut our big quadrilateral removing the space defined by the set of points X such that $\|X - P_i\|^2 > \|X - P_j\|^2$ (see Fig. 4.8). The only modification to Sutherland-Hodgman algorithm is that the point of intersection P between the bissector of P_iP_j (that passes through the middle M of P_iP_j) and the current edge $[AB]$ to be clipped is computed using the fact that $\langle P - M, P_i - P_j \rangle = 0$ and $P = A + t(B - A)$, so that $\langle A + t(B - A) - M, P_i - P_j \rangle = 0$ and so $t = \frac{\langle M-A, P_i-P_j \rangle}{\langle B-A, P_i-P_j \rangle}$, and a point X is *inside* the clip edge if $\langle X - M, P_j - P_i \rangle < 0$. By performing this operation for all $P_j \neq P_i$, we obtain the Voronoï cell of P_i, and this operation can be performed in parallel and independenly for all P_i.

However, proceeding that way would make the algorithm $\mathcal{O}(N^2)$ since for all P_i, the Voronoï cell of P_i necessitates cutting half spaces defined by all P_j[7]. This can quickly become prohibitive. To alleviate this issue, one needs to realize that a site P_j that is very far from P_i has little chance to contribute to the Voronoï cell of P_i. In fact, if the distance from P_i to P_j is greater than twice the distance from P_i to the farthest point of its current polygon estimate of the Voronoï cell (built with a subset of the samples $\{P_k\}$), the bissector of $[P_iP_j]$ will not clip anything of the polygon (Fig. 4.9). It is thus more interesting to start by clipping within bissectors of $[P_iP_j]$ for P_j close to P_i than far from it. To achieve that, we query the k-nearest neighboring sites of P_i ordered by increasing distance, and iteratively clip the current polygon estimate (starting with our gigantic quad) with these sites until we find one site that is not contributing to the Voronoï cell (in which case the next sites will not contribute either). If k nearest neighbors are not sufficient (i.e., the k^{th} nearest neighbor still

[7]The cost of Sutherland-Hodgman also depends on the number of edges of the Voronoï cell – however, the number of vertices in the entire Voronoï diagram in 2d is $2N - 5$ and the number of edges $3N - 6$; per Voronoï cell, this number is thus a constant(with an average of 6 edges per Voronoï cell).

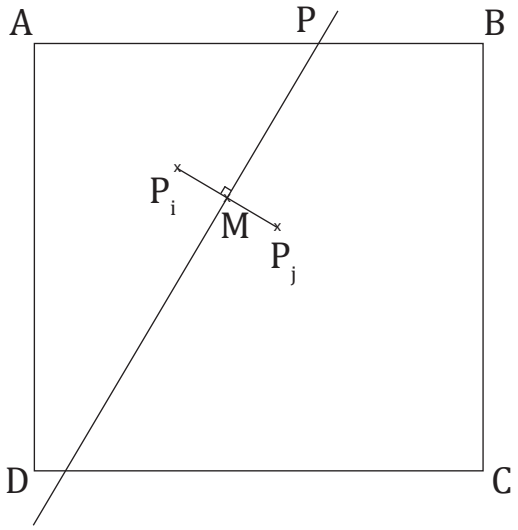


Figure 4.8: To compute the Voronï cell of site P_i, Voronoï Parallel Linear Enumeration clips a big quadrilateral by all half-spaces defined by bissectors of P_i and all other P_j.

contributes to the Voronoï cell), then we perform a new $2k$-nearest neighbors query.

⚠ Beware that the very first nearest neighbor in the point set is P_i itself ! Make sure to ignore the very first nearest neighbor since it does not make sense to clip with the bisector of $[P_iP_i]$

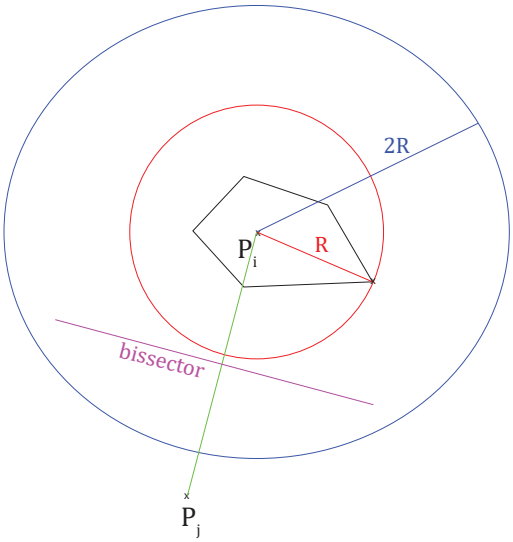


Figure 4.9: Clipping a polygon (in black) representing the current estimate of the Voronoï cell of P_i with a bissector (magenta) of $[P_iP_j]$ of a P_j that is more than twice further than the furthest vertex of the polygon estimate will not change the polygon estimate, and can thus be ignored.

K-d trees. The k-nearest neighbor queries can be performed in $\mathcal{O}(k\log(N))$ using a kd-tree. A kd-tree is an acceleration structure ideal for nearest neighbor queries which represents a partition of the space (contrary to our previous BVH in our renderer!). A kd-tree is a binary tree and is built by recursively splitting points into 2 subsets, ideally of equal sizes, alternately along each dimension (Fig. 4.10). It is constructed by sorting points (or a subset of them) along one dimension, and using the point with median value to split the set into two subsets, recursively until leaves have only one point. Each node stores the index of the median point, as well as the extent of the domain (the root contains the entire bounding box of the point set).

Searching for the closest point given a query point Q consists in first checking the distance between Q and the root of the tree, and then taking the branch where Q is (in Fig. 4.10, the left branch) since it will *more likely* contain the actual nearest neighbor. The process is repeated while keeping track of the smallest distance R encountered while going down the tree. However, children are visited if and only if their domain overlap with the disk $\mathcal{D}$ centered at Q and of radius R. Once we cannot go down anymore, our depth-first traversal will go back up: we will also visit children branches whose domain overlap $\mathcal{D}$ (while continuing updating R, possibly making it smaller during the traversal, hence elaging more branches).

Similarly, searching for the k nearest neighbors can be performed by keeping a max heap of points, visiting each branch that overlap with the disk whose radius is the k^{th} closest distance found so far. In practice, efficient libraries exist in C++ such as Flann, NanoFlann (header-only) or ANN. Flann and ANN support approximate nearest neighbor searches, by using a larger disk to prune branches.

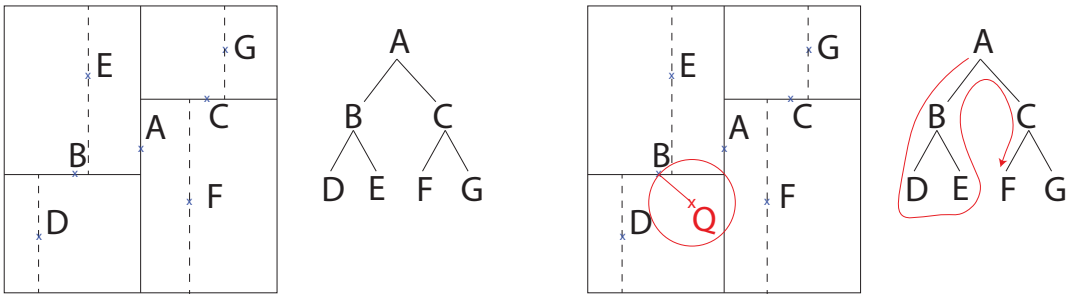


Figure 4.10: **Left.** Building the kd-tree of points $\{A, B, C, D, E, F, G\}$ consists in splitting the set of points alternately in the horizontal and vertical directions. **Right**. Searching for the closest point to Q results in first going down the tree to search in which leaf Q is located while recording the closest distance R from each node to Q. Then, we go back up and down the tree to traverse each branch whose domain overlap with the disk centered at Q and of radius R (that gets updated).

The Voronoï Parallel Linear Enumeration algorithm is fast[8] (see Fig. 4.11), though in large dimension the number of nearest neighbors contributing to each Voronoï cell increases exponentially.

4.4 More than Voronoï

This section describes applications and variants of Voronoï diagrams used in computer graphics.

4.4.1 Centroidal Voronoï Tessellation

A Centroidal Voronoï Tessellation is a Voronoï diagram in which sites coincide with cell barycenters. This produces Voronoï diagrams that have useful properties. Specifically, the dual of a Centroidal Voronoï Tessellation is a Delaunay triangulation with triangles that are as close as possible to equilateral. Such triangulations are useful for simulation: for instance, the speed of convergence of several iterative schemes to solve the heat equation depends on the anisotropy of triangles and are fastest with isoceles triangles. They also distribute the "geometric budget" more equally on the mesh, hence representing smooth surfaces better given the same number of triangles (note that this is not the case for non-smooth surfaces, where anisotropic triangles better represent sharp features). Finally, they also produce uniformly spread point sets that can be used for dithering/stippling or for quasi-Monte Carlo integration (recall, footnote in Sec. 2.1.2).

[8]It has also been made faster in some edge cases, notably in the context of optimal transport in the paper *Restricting Voronoi diagrams to meshes using corner validation*: `https://hal.archives-ouvertes.fr/hal-01626140/file/corner-validated-rvd.pdf`

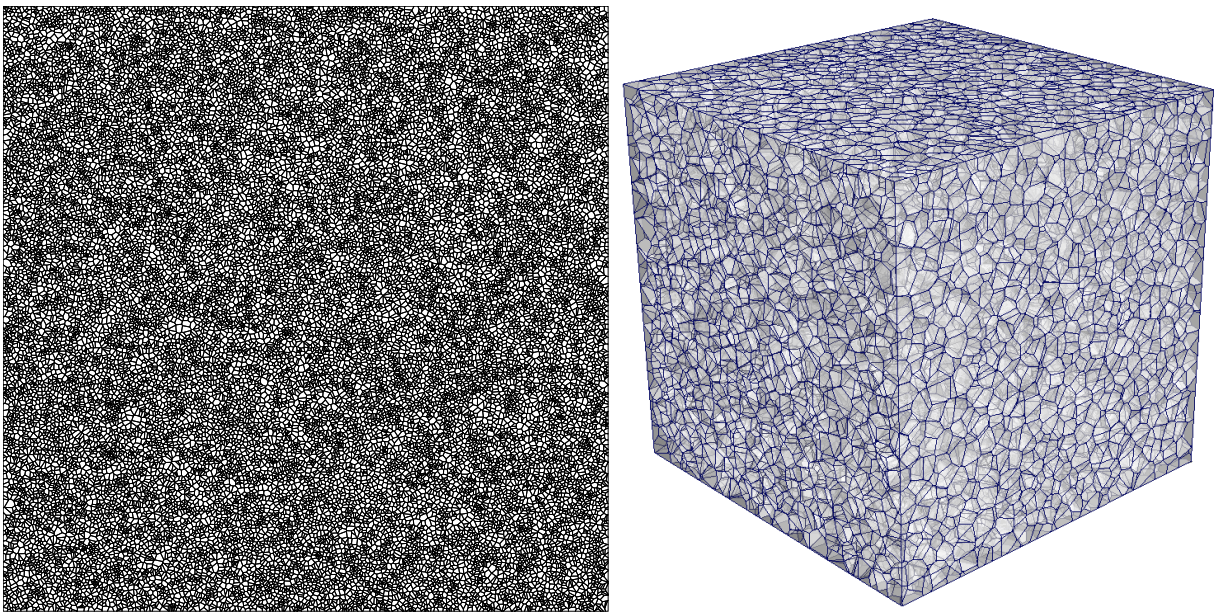


Figure 4.11: **Left.** For this 30k point 2D example, the algorithm runs in 19s (in parallel) using the naive $\mathcal{O}(N^2)$ algorithm coded in 56 lines of code, but runs in 30ms using the Nanoflann library and 35 additional lines of code. It then runs in 16 seconds to generate the Voronoï diagram of 10M points. In my implementation, it starts by searching for 20 neighbors, and doubles it each time it is not sufficient. **Right.** Extending the code to 3D, I run this 30k point 3D example in 3min30s using the naive $\mathcal{O}(N^2)$ algorithm (using a quick'n dirty inefficient 3D Sutherland-Hodgman) and brings that down to 300ms using Nanoflann. The 3d code is about 200 (dirty) lines.

Denoting the sites $X = \{x_i\}$, these triangulations minimize the energy:

$$\min_X E(X) = \min_X \sum_i \int_{Vor(x_i)} \|x - x_i\|^2 \mathrm{d}x \tag{4.4}$$

where $Vor(x_i)$ is the Voronoï cell of x_i. One can indeed see that this energy (called the Lloyd's energy) is minimal when the x_i's are at the barycenter of their Voronoï cell. In fact, Gersho's conjecture (proved in 2D by Gruber in 2001) states that after minimization, the resulting cells are hexagonal and each cell will have the same contribution to the energy.

A simple strategy to compute a Centroidal Voronoï Tessellation is to start with randomly placed x_i's, compute their Voronoï Diagram, then move the each x_i to the centroid of its Voronoï cell, and iterate. This process is called "Lloyd's iterations", and are similar to those used for clustering in the k-means algorithm.

Remark: Computing a Delaunay triangulation from a Voronoï diagram is relatively easy: one just needs to check all vertices of the Voronoï diagram shared by 3 sites, and form a triangle connecting these 3 sites.

The centroid C of a non self-intersecting polygon in 2D whose N vertices are $\{(x_i, y_i)\}_{i=0..N-1}$ is

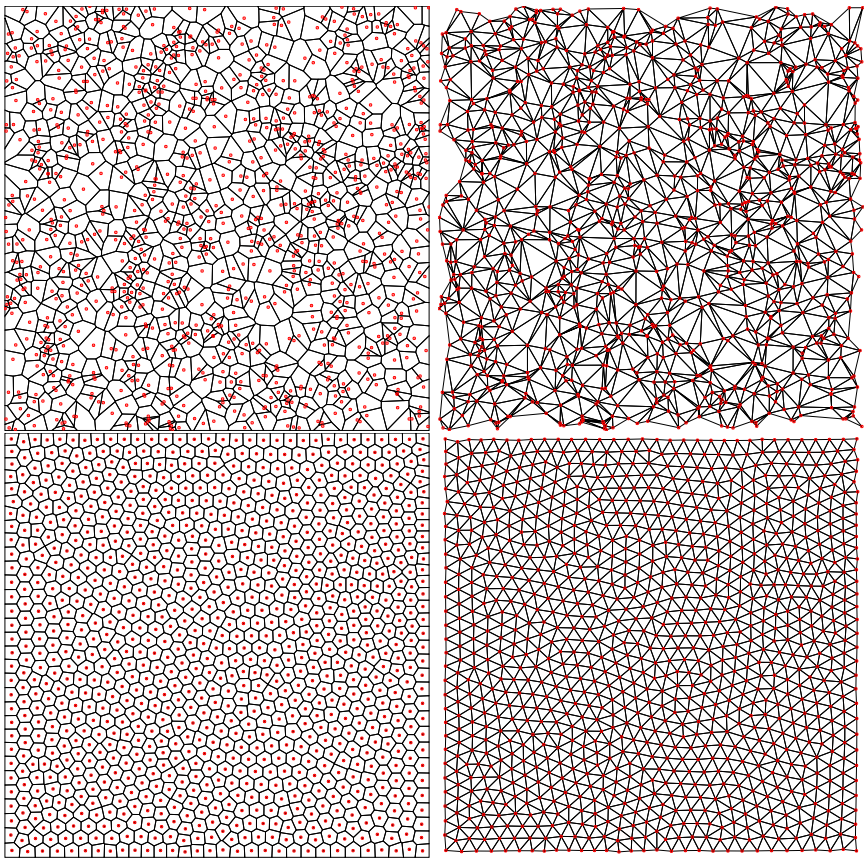


Figure 4.12: Lloyd's iterations bring a Voronoï diagram of random sites (top) such that sites coincide with cell centroids by moving them (bottom). The Delaunay triangulation of a Centroidal Voronoï Tessellation is such that triangles are near-equilateral.

given by[9]

$$C_x = \frac{1}{6A} \sum_{i=0}^{N-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \tag{4.5}$$

$$C_y = \frac{1}{6A} \sum_{i=0}^{N-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \tag{4.6}$$

where indices are taken modulo N, and where A, the area of the polygon, can be computed by

$$A = \frac{1}{2} \sum_{i=0}^{N-1} (x_i y_{i+1} - x_{i+1} y_i)$$

In 3D, the centroid of a polyhedron is obtained[10] by considering that polyhedron facets can (trivially) be subdivided into triangles, such that the entire boundary of the polyhedron only consists of triangles with vertices are $\{(a_i, b_i, c_i)\}$. Denoting $n_i = (b_i - a_i) \times (c_i - a_i)$ the non-normalized normal,

[9] see `https://www.seas.upenn.edu/~sys502/extra_materials/Polygon%20Area%20and%20Centroid.pdf`
[10] see `https://wwwf.imperial.ac.uk/~rn/centroid.pdf`

we have

$$C_x = \frac{1}{48V} \sum_{i=0}^{N-1} n_{i,x}((a_{i,x} + b_{i,x})^2 + (b_{i,x} + c_{i,x})^2 + (c_{i,x} + a_{i,x})^2) \tag{4.7}$$

$$C_y = \frac{1}{48V} \sum_{i=0}^{N-1} n_{i,y}((a_{i,y} + b_{i,y})^2 + (b_{i,y} + c_{i,y})^2 + (c_{i,y} + a_{i,y})^2) \tag{4.8}$$

$$C_z = \frac{1}{48V} \sum_{i=0}^{N-1} n_{i,z}((a_{i,z} + b_{i,z})^2 + (b_{i,z} + c_{i,z})^2 + (c_{i,z} + a_{i,z})^2) \tag{4.9}$$

where V, the volume of the polyhedron is given by

$$V = \frac{1}{6} \sum_{i=0}^{N-1} a_i.n_i$$

A faster strategy is to consider the minimization problem in Eq. 4.4 in order to use quasi-Newton approaches, such as l-BFGS (a type of algorithm that minimizes convex energies without explicitly computing a Hessian, but that tries to approximate a Hessian from the gradient). More generally, using an underlying density ρ that controls how packed should samples be:

$$\min_X E'(X) = \min_X \sum_i \int_{Vor(x_i)} \rho(x) \|x - x_i\|^2 \mathrm{d}x \tag{4.10}$$

and denoting $m_i = \int_{Vor(x_i)} \rho(x) \mathrm{d}x$ (e.g., computed via numerical integration), the gradient of the energy E' is:

$$\frac{\partial E'}{\partial x_i} = 2m_i(x_i - C_i)$$

with C_i the centroid of $Vor(x_i)$[11].

4.4.2 Restricted Voronoï Diagrams

A **Restricted Voronoï Diagram** (RVD) is, most commonly, the intersection of a 3D Voronoï diagram with a triangle mesh. This defines cells on the surface of the mesh (though possibly non-connected) that can be used for remeshing, by computing the triangulation dual to these cells called the Restricted Delaunay Triangulation (RDT). It has the advantage over **geodesic Voronoï diagrams** (that can be computed via front propagation on the surface of the mesh[12]) to be much cheaper to compute.

A way to compute the RVD is to clip the mesh triangles using Sutherland-Hodgman's algorithm by the bisector of each pair of sites. Again, a k-nearest neighbor search with the same criterion (maximum distance between the seed and the RVD cell) can be used to discard sites that will not contribute to the RVD cell.

Also, the RVD can benefit from Lloyd's iterations to obtain a Centroidal RVD, that can be used to remesh meshes with equilateral triangles (Fig. 4.13). By slightly modifying the minimized energy, this can be used to produce minimal surfaces (surfaces of constant mean curvature)[13].

As a parenthesis: to randomly sample sites on the surface of a triangle mesh, it is not sufficient to naively randomly select a triangle uniformly, and then generate a random point within this triangle.

[11] See *On Centroidal Voronoï Tessellation – Energy Smoothness and Fast Computation*, `https://dl.acm.org/doi/pdf/10.1145/1559755.1559758`

[12] See Gabriel Peyré's Numerical Tours: `https://www.numerical-tours.com/matlab/fastmarching_4_mesh/`

[13] *Robust Modeling of Constant Mean Curvature Surfaces*, `https://haopan.github.io/papers/cmc_surface.pdf`

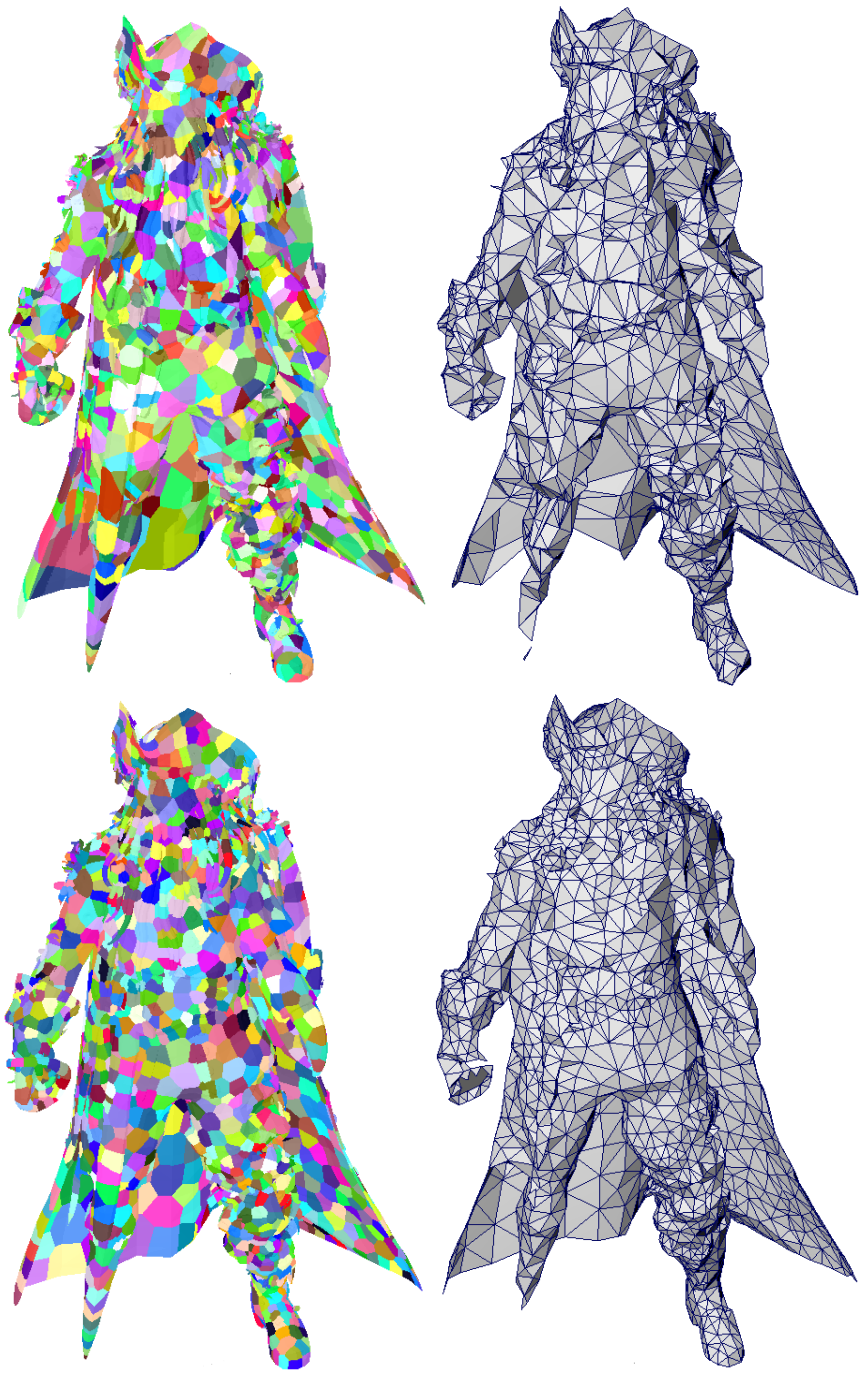
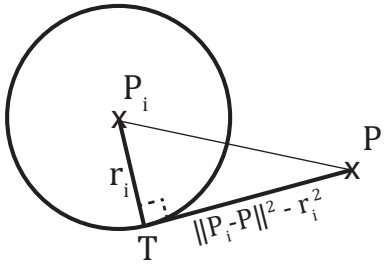


Figure 4.13: The Restricted Voronoï Diagram (RVD) is the intersection between Voronoï cells and a mesh (left) which dual, the Restricted Delaunay Triangulation (RDT) can be used to remesh shapes. Performing Lloyd's iterations (bottom) result in more isotropic remeshing – here with 3000 sites (*note that this one may not have yet fully converged*).

Indeed, this would ignore the triangle areas and would favor places where there are many small triangles. Instead, you should compute the area of all triangles and store them in some array, and store the total area of the mesh. For each new site you want to generate, you generate a random number between 0 and the total mesh area, then scan the area array and progressively accumulate areas until you have reached your random number. Once you have reached the random number, you stop scanning the array and you obtained a correct uniformly sampled triangle. This strategy is an inverse CDF random sampling (or inverse transform sampling) method and is used in much broader context than computer graphics.

4.4.3 Power diagrams

A power diagram (or Laguerre diagram) is an extension of the Voronoï diagram that allows for controlling the size of each cell via a set of weights. In fact, **any** partition of the space into convex polygonal cells is the Power Diagram of some sites and some weights (Aurenhammer 1987). It is alternatively defined as a Voronoï diagram where instead of taking the classical distance $\|P - P_i\|$ from a point P to a site P_i, we take the distance between P and a point T tangent to a circle centered at P_i and of radius r_i.



This modified distance is thus $\|P - P_i\|^2 - r_i^2$ (by Pythagorean theorem), and, denoting $w_i = r_i^2$, the *power cell* associated to sample P_i is defined by

$$\|P - P_i\|^2 - w_i \leq \|P - P_j\|^2 - w_j \qquad \forall j \neq i \tag{4.11}$$

More intuitively, as w_i increases relative to others weights, the area of the corresponding power cell increases. It is important to note that when all weights are equal, this power diagram coincides with the Voronoï diagram, and that the power diagram is invariant by an additive factor to all weights (adding the same value to both sides of the inequality does not change the result).

More importantly, it can be easily seen that a power diagram in dimension d can be obtained from a Voronoï diagram in dimension $d + 1$. Indeed, denoting $P_i' = (P_i, \sqrt{m - w_i})$ the sites in dimension $d + 1$ where a coordinate $\sqrt{m - w_i}$ has been added, with m any sufficiently large value such that $m - w_i \geq 0$ (for instance, $m = \max_i w_i$) and denoting $P^0 = (P, 0)$, we see that Eq. 4.11 is equivalent to

$$\|P^0 - P_i'\| \leq \|P^0 - P_j'\| \qquad \forall j \neq i$$

This, in fact, precisely describes the Voronoï diagram of $\{P_i'\}$ restricted to the hyperplane defined by all $(P, 0)$ in $\mathbb{R}^{d+1}$ – this is an RVD and can be obtained by the previous algorithm (Sec. 4.4.2). It is important to note that, contrary to Voronoï cells, power cells can be empty or may not encompass their associated site (Fig. 4.14).

Alternatively, it can be obtained by removing half-spaces as before in $\mathbb{R}^d$. Indeed, it can be easily seen that cutting planes are parallel to bisectors, and pass through the point $M' = M + \frac{w_i - w_j}{2\|P_i - P_j\|^2}(P_j - P_i)$ where $M = (P_i + P_j)/2$ is the middle point (and thus, when $w_i = w_j$, we see that cutting planes pass through M and are thus bisectors). The main change to the algorithm presented in Section 4.3.3 is thus merely a replacement of M by M' in the formulas. The second change is for the criterion to prune non-contributing hyperplanes in the k-nearest neighbor search. For this criterion, it is easier to perform the k-nearest neighbor search in the $d + 1$ dimensional space (recall the the constructed polygon has its last coordinate set to 0, but the sites have their last coordinates set to $\sqrt{m - w_i}$), and keep the previously used criterion (take the distance from the $d + 1$-dimensional site to the furthest point in the polygon, multiply it by 2, and you get the critertion for rejecting further sites that will not contribute to the power cell).

It is however not very practical to control the area of each power cell via its weight w_i (or the radius r_i). We will see next how semi-discrete optimal transport can alleviate this issue.

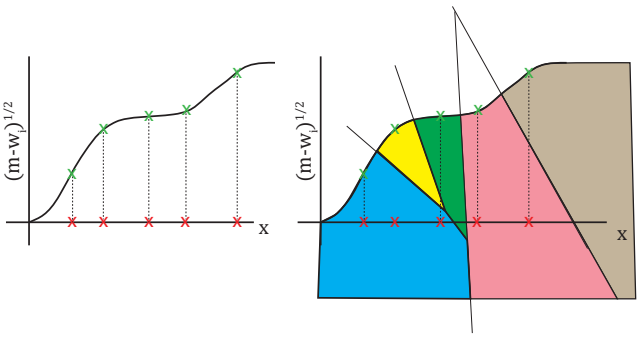


Figure 4.14: Simple example of a 1-d power diagram seen as a 2-d Voronoï diagram. By adding a second dimension to each site on the left (here, the second dimension has been chosen on the graph of a function, but this is not necessary), one can compute a Voronoï diagram (right) and consider its restriction to the $y = 0$ line. Here, the resulting diagram consists of the first power cell in blue that encompasses 3 sites ; the second power cell (in yellow) is empty, the third cell (in green) has a small area and does not encompass any site, the fourth cell encompasses 2 sites, and the last cell (in gray) does not encompass any site.

4.4.4 Semi-discrete Optimal Transport

Going back now to optimal transport, which we briefly introduced in Sec. 3.2.2. The optimal transport problem is the problem of matching a probability distribution with another probability distribution at minimal cost. Specifically, one considers that a probability distribution is a heap of sand, and the second probability distribution is a hole in the ground, and one would like to find how to move the sand into the hole using a little spoon, and minimizing the travelled distance with the spoon (or alternatively, minimizing the sum of squared distance travelled with the spoon). The paths borrowed by all these spoons define a *transport plan*, which tells what amount of sand from location x should go to location y.

It turns out that when the hole consists in a sum of "Dirac holes", the resulting transport plan can be represented by a Power Diagram. A better analogy in this case is that of bakeries located in a city whose population density is described by a probability density function f, and each bakery located at position y_i can serve λ_i pieces of bread (e.g., per day). An additional assumption is that all bread will be sold (e.g., at the end of the day). Given that the cost for someone located at position x to travel to any bakery at position y_i is $\|x - y\|^2$, what is the optimal global strategy to sell that bread. It can be shown[14] that the solution to this optimal transport problem is a partition of the space into convex polyhedra, which can thus be precisely modeled by the power diagram of the $\{y_i\}$ for some set of weights $\{w_i\}$ that need to be found.

Another way to see that is that instead of controlling the size of each cell via some weights $\{w_i\}$ that are hard to control, we want to directly control the mass of the Voronoï cells so that they are equal to $\{\lambda_i\}$ (if the underlying "population density" is uniform, then this mass exactly corresponds to the cell area).

In order to find the optimal set of weights $\{w_i\}$, it can be shown[15] that one needs to maximize the following functional:

$$g(W) = \sum_i \int_{Pow_W(y_i)} (\|x - y_i\|^2 - w_i) f(x) \mathrm{d}x + \sum_i \lambda_i w_i$$

[14] *Minkowski-Type Theorems and Least-Squares Clustering*: `https://link.springer.com/content/pdf/10.1007/PL00009187.pdf`

[15] For an intuitive explanation, see *A numerical algorithm for L_2 semi-discrete optimal transport in 3D*: `https://arxiv.org/pdf/1409.1279.pdf`

whose gradient can be expressed as:

$$\nabla g(W) = -\int_{Pow_W(y_i)} f(x)\mathrm{d}x + \lambda_i$$

and Hessian[16] as:

$$\frac{\partial^2 g}{\partial y_i \partial y_j} = \int_{Pow_W(y_i)\cap Pow_W(y_j)} \frac{f(x)}{2\|y_j - y_i\|}\mathrm{d}x \quad \forall i \neq j$$

$$\frac{\partial^2 g}{\partial y_i^2} = -\sum_{j\neq i} \frac{\partial^2 g}{\partial y_i \partial y_j}$$

Using only the gradient, one can easily perform a gradient **ascent** (g should be maximized!) by iterating:

$$w_i \leftarrow w_i + \epsilon g(W)$$

for some step size ϵ (which can be adjusted via line search). Typically, a gradient descent starts from an initial point and walks along the direction of steepest descent (i.e., the direction of the gradient of the function) to find a local minimum. A gradient ascent walks the other way around. However, this can be very slow to converge. Alternatively, one can use a quasi-Newton solver such as L-BFGS[17], in which case you will need to provide $-g$ and $-\nabla g$ to the library. A Newton solver considers a Taylor series expansion of g in the form $g(x+d) \approx g(x) + \langle \nabla g(x), d\rangle + \frac{1}{2} d^T H d$, where H is the Hessian of g (its matrix of second derivatives). Each iteration tries find the next point x^{n+1} that minimizes the Taylor expansion of g given the current x^n. This amounts to finding a good direction d^n, i.e., one that minimizes $E(d) = g(x+d) \approx g(x) + \langle \nabla g(x), d\rangle + \frac{1}{2} d^T H d$ among all d. One thus finds the gradient of $E(d) : \nabla E(d) = \nabla g(x) + \frac{1}{2} H d$, and the best direction d^n cancels this gradient, i.e., $\nabla g(x) + \frac{1}{2} H d^n = 0$. This amounts to finding $d^n = 2H^{-1}\nabla g(x)$. To maximize g, one thus iterates $x^{n+1} = x^n + \epsilon d^n$ with $d^n = 2H^{-1}\nabla g(x)$ and some well chosen ϵ. Unfortunately, in some cases, H is difficult to obtain and quasi-Newton solvers merely approximate H.

In general, faster solutions are obtained with a Newton solver. However, this is not entirely trivial: the Hessian cannot be computed if there are empty power cell along the execution of the optimizer. Starting with an initial guess with no empty cell (e.g., a Voronoï diagram, with constant weights), it was shown that if you halve the step size as soon as you encounter an empty cell, then Newton steps will converge.

When the density f is constant, the expression of g makes use of $\int_{Pow_W(y_i)} \|x - y_i\|^2 \mathrm{d}x$. In 2-d, denoting $\{(X_i, Y_i)\}$ the vertices of the power cell, this can be computed analytically using[18]

$$\int_{Polygon(\{(X_i,Y_i)\})} \|P - P_i\|^2 \mathrm{d}P = \frac{1}{12}\sum_{k=1}^{N} (X_{k-1}Y_k - X_k Y_{k-1})(X_{k-1}^2 + X_{k-1}X_k + X_k^2 + Y_{k-1}^2 + Y_{k-1}Y_k + Y_k^2 \tag{4.12}$$

$$- 4(P_{i,x}(X_{k-1} + X_k) + P_{i,y}(Y_{k-1} + Y_k)) + 6\|P_i\|^2) \tag{4.13}$$

Alternatively, an equivalent expression[19] considers that the power cell has been triangulated into triangles, and each triangle $T = (c_1, c_2, c_3)$ contributes to the integral as:

$$\int_T \|P - P_i\|^2 \mathrm{d}P = \frac{|T|}{6} \sum_{k\leq l\leq 3} \langle c_k - P_i, c_l - P_i\rangle$$

[16] *Convergence of a Newton algorithm for semi-discrete optimal transport*: `https://arxiv.org/pdf/1603.05579.pdf`

[17] You can use the library available at `https://github.com/chokkan/liblbfgs` – it merely consists of two header files (`lbfgs.h` and `arithmetic_ansi.h`) and one .c file (`lbfgs.c`), so it is quite easy to integrate it into your project ; also take a look at the sample file.

[18] *Polygon Integrals – Arbitrary Moments of a Polygon*: `https://people.sc.fsu.edu/~jburkardt/cpp_src/polygon_integrals/polygon_integrals.html`

[19] *Fitting Polynomial Surfaces to Triangular Meshes with Voronoi Squared Distance Minimization*: `https://members.loria.fr/Bruno.Levy/papers/VSDM_IMR_2011.pdf`

This formula can be extended in arbitrary dimension – see *The Multi-Dimensional Version of* $\int_a^b x^p dx$ by Lasserre and Avrachenkov. See Fig. 4.15 for results.

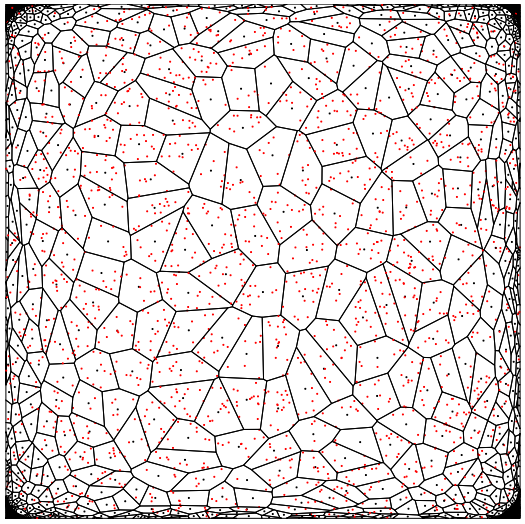


Figure 4.15: Power-diagram of sites in red, optimized using semi-discrete optimal transport so that the cell associated to a site at position y_i has an area proportional to $\exp(-\|y_i - C\|^2/0.02)$ where C is the center of this unit square (here $f = 1$). This was optimized using L-BFGS. The analogy with bakery would be a square city with a uniform population density, and 2000 bakeries. The bakeries close to the center are able to produce more bread than those far from the center, so they attract people from a larger area.

Semi-discrete optimal transport can be used similarly to Lloyd's algorithm to produce well, uniformly, distributed point set[20] (their Fourier spectrum has a peculiar form and we call this property of these point sets "Blue Noise") or non-uniformly distributed point sets for image stippling (Fig. 4.16). As we shall see in Sec. 5.4, it can be extended to the partial optimal transport context, where in our analogy, this amounts to having more people in our population than the number of breads bakeries can produce. This will be used for fluid simulation.

4.5 The Marching Cubes algorithm

While it is possible to directly render implicit functions (e.g., via *ray marching* that performs raytracing by computing the intersection via trial and errors along the ray), it is sometimes more convenient to transform them into a triangular mesh. A simple algorithm called **marching cubes** allows that[21].

The input of the algorithm is an implicit function that can be evaluated at any vertex of a voxel grid, where negative values represent the inside of the volume and positive values the outside. The algorithm considers a voxel grid, and for each voxel taken individually, tries to determine if part of the surface traverses this voxel (in which case, triangles should be computed and added to the mesh).

To do that, the algorithm checks the value of the implicit function at each of the 8 vertices of the current voxel. Of course, if all the 8 vertices have negative values or if all the 8 vertices have positive values, it means the voxel is outside of the surface and can be ignored. Now, the sign of the implicit function at each of the 8 vertices produces $2^8 = 256$ combinations (in fact, due to symmetries, only 15 are really different, but we will not use this fact). Given the following numbering of the vertices and edges

[20] *Blue Noise Through Optimal Transport*: `https://graphics.stanford.edu/~kbreeden/pub/dGBOD12.pdf`

[21] *Marching cubes: A high resolution 3D surface construction algorithm*: `https://dl.acm.org/doi/abs/10.1145/37402.37422`

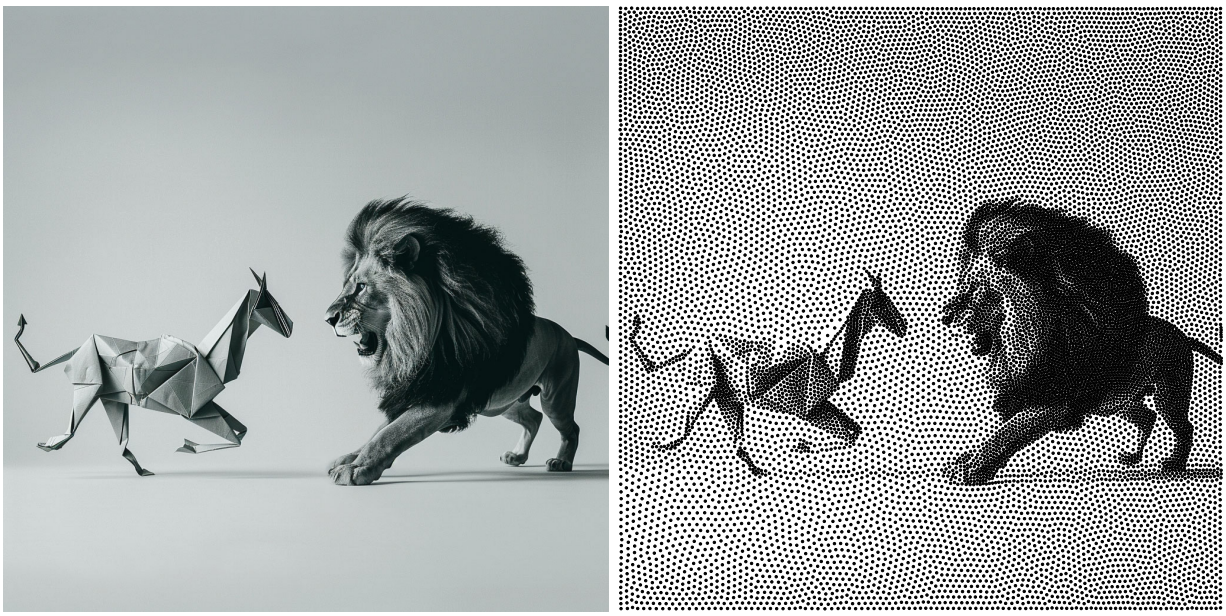
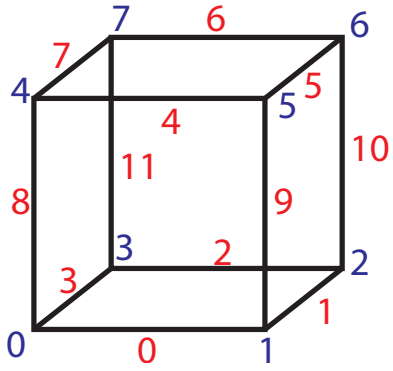


Figure 4.16: By recentering samples to the centroid of each power cell obtained by semi-discrete optimal transport, and by accounting for the underlying density, one can produce image stippling similarly to Lloyd's algorithm.



the first step is to convert our voxel into an single scalar value (a hash). This is performed by considering the scalar

$$k = \sum_{i=0}^{7} S(i)2^i$$

where $S(i) = 1$ if the implicit function is negative at vertex i, and $S(i) = 0$ otherwise. This simply is a binary representation of our voxel.

The authors of the method have then built a large table that tells, given this scalar value, which triangles should be added to the triangulation[22]. The table looks like

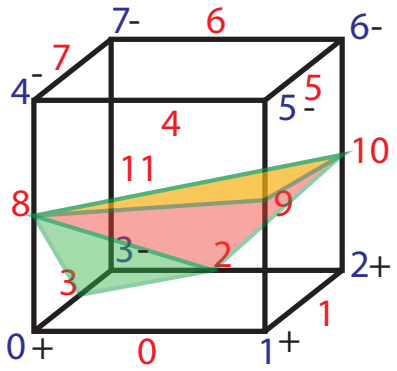
```
int triTable[256][16] =
{ {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 2, 10, 0, 2, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 8, 3, 2, 10, 8, 10, 9, 8, -1, -1, -1, -1, -1, -1, -1},
```

[22] I've put it here: `https://pastebin.com/Bbmt1u4Y`

```
10 // ......
```

⚠ The original table published by the authors is incorrect: it (occasionally) produces surfaces that are not watertight.

The most complex possible voxel contains 5 triangles – this table hence has 3*5 = 15 values at most for each possible voxel (in fact the table contains 16 column, but that last one is always −1). The indices in this table represent triplets of edges that are crossed by the 0 level of the implicit surface. For instance, the very first voxel has index 0, which in binary, means that the implicit function is positive at all of the 8 vertices, so no triangle should be formed (hence the first row contains only −1 which represent null values). For another example, taking voxel 7 = 00000111 (in binary) means vertices 0, 1 and 2 are positive (so, outside of the volume of interest) and all the other vertices are negative (i.e., are inside). For this voxel, three triangles should be created. The first of these three triangles has its vertices somewhere on the edges 2, 8 and 3 of the current voxel. The second one has its vertices on edges 2, 10 and 8. And the third triangle on edges 10, 9 and 8. The figure below illustrates this example:



The last detail is to decide where to put the vertices exactly on that edge. A simple solution is to put it right in the middle, though it can produce jaggy results (Fig. 4.17). Instead, one can compute a more accurate location by linearly interpolating the implicit function value at the vertices to determine approximately where it should cross the 0 isovalue. Given the implicit function value $f(A)$ at vertex A of the voxel and $f(B)$ at vertex B of the voxel, the mesh vertex P that should be added along edge AB is thus $P = A + (B - A)\frac{f(A)}{f(A)-f(B)}$. Results can be seen in Fig. 4.18.

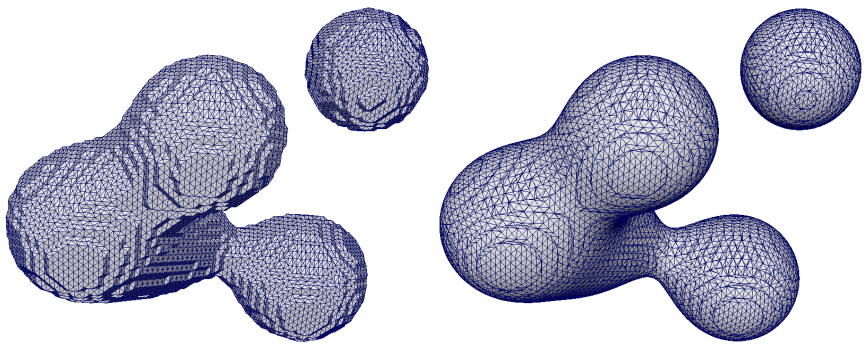


Figure 4.17: Placing the triangle vertex at the middle of the voxel's edge (left) results in jaggy reconstructions, while using a linear interpolation (right) produces smoother reconstructions.).

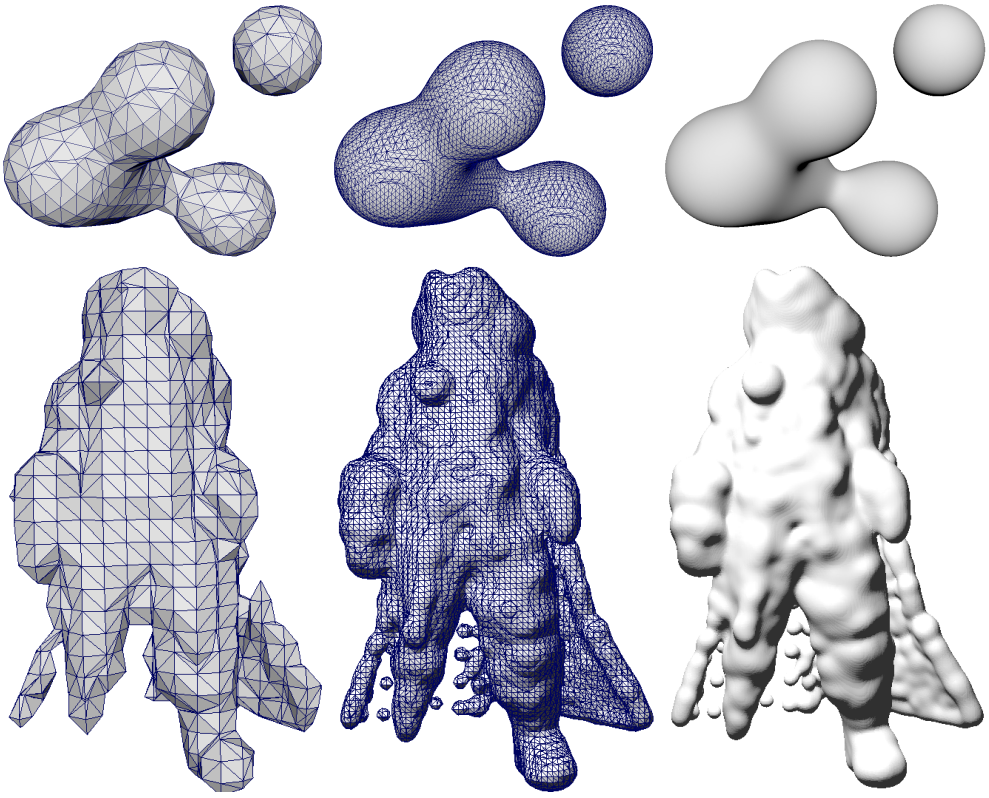


Figure 4.18: Reconstructions using 32×32, 128×128, and 512×512 voxel grids. Bottom row: I used the centroidal Voronoï tessellation of Davy Jones in Fig. 4.13 to place Gaussian kernels at each site, and reconstructed some level set of this sum of Gaussians (*we can see in the cape that Lloyd's iterations have not yet converged as the density of samples is obviously lower there*). The blob example in 128×128 runs in 18 ms (monothreaded) and the code is about 40 lines (excluding the 300 lines lookup table provided by the authors!).

4.6 Surface parameterization

Input mesh Tutte's embedding Harmonic mapping Conformal mapping

Figure 4.19: Different parameterizations to a disk of Goethe's life mask (`https://www.turbosquid.com/fr/3d-models/free-obj-mode-scan-lifemask-johann-wolfgang/1035699`). *The conformal mapping has not fully converged yet, and is neither exactly a disk nor exactly bijective.*

Parameterizing a triangle mesh has many applications, and in particular, texture mapping as we have seen in Sec. 2.1.2. Parameterizing a surface means that we want to uniquely assign each point of the surface to a point of a reference domain – a 2d texture map in our case of interest. There are many properties such a map could possess, and among them:

- Isometric: There are two definitions of isometries on manifolds. The first one is a map that preserve (global) distances on the manifold. Bijective isometric mappings are affine (Mazur–Ulam

th.), so they are of little interest for texture mapping. Up to some distortions, there are algorithms (e.g., SMACOF[23]) trying to find mappings that *best preserve* length. They are still barely used in computer graphics, but are used for drawing graphs. The other definition is a map that preserves entirely the Riemannian metric tensor (and hence local angles, distances, areas). These two definitions are equivalent (Myers and Steenrod th.[24]).

- Isoareal: These maps preserve areas. These maps are also not used for texture mapping as even a nice equilateral triangle on a mesh could be mapped to a very long but extremely thin triangle of the same area in the texture map. However, area preservation is a property that can be enforced on top of conformal maps (see next), notably using semi-discrete optimal transport[25]. In term of metric, it preserves the area element if the determinants of the first fundamental form are preserved (see below, Sec. 4.6.2).

- Conformal: A conformal map preserves angles, and that are the maps that are mainly of interest in texture mapping. In term of metric, it means the metrics are proportional. As such, an isometric map is necessarily conformal. More generally, an isometry is a map that is both conformal and isoareal.

- Harmonic: This map is such that $\Delta\phi = 0$. Interestingly, a conformal map is necessarily harmonic (but not the converse). More details in Sec. 4.6.2.

Example of them are shown in Fig. 4.19. Before delving into maps that posess these properties, we will see a simpler embedding, Tutte's mapping.

4.6.1 Tutte's mapping

Tutte was interesting in laying out graphs on a plane. Translated in the language of triangular meshes (from the book *Polygon Mesh Processing*):

"Given a triangulated surface homeomorphic to a disk, if the (u, v) coordinates at the boundary vertices lie on a convex polygon, and if the coordinates of the internal vertices are a convex combination of their neighbors, then the (u, v) coordinates form a valid parameterization (without self-intersections)."

This gives a pretty simple algorithm to produce such mappings.

[23] *Multidimensional Scaling Using Majorization: SMACOF in R*: `https://www.jstatsoft.org/article/view/v031i03/v31i03.pdf`

[24] see *Foundations Of Differential Geometry, vol. 1* (p. 169) for a proof: `http://tomlr.free.fr/Math%E9matiques/Math%20Complete/Differential%20Geometry/Foundations%20of%20Differential%20Geometry%20vol%201%20-%20Kobayashi,%20Nomizu.pdf`

[25] *Area-Preservation Mapping using Optimal Mass Transport*: `https://www.researchgate.net/publication/256837514_Area-Preservation_Mapping_using_Optimal_Mass_Transport`

Algorithm 5: Tutte's embedding algorithm.

Input: Triangular mesh M homeomorphic to a disk with vertices $\{v_i\}$.
Output: Tutte's embedding

$\partial M \leftarrow \{b_0, b_1, ..., b_{n-1}\}$ // identify ordered boundary vertices ($b_n = b_0$).
$s \leftarrow \sum_{i=0}^{n-1} \|b_{i+1} - b_i\|$ // boundary length.
$cs \leftarrow 0$
$v_i^0 \leftarrow v_i \quad \forall i$
// Layout boundary vertices on a circle.
for $i = 0..n-1$ **do**
  $\theta_i \leftarrow 2\pi \frac{cs}{s}$
  $v_i^0 = (\cos\theta_i, \sin\theta_i)$
  $cs \leftarrow cs + \|b_{i+1} - b_i\|$
// Layout internal vertices.
for *iter* $= 0..nbiter - 1$ **do**
  for *all interior vertex indices* i **do**
    $v_i^{n+1} \leftarrow \frac{1}{K} \sum_{j \sim i} v_j^n$ // With K: number of adjacent vertices, $i \sim j$ if v_j shares an edge with v_i
  for *all boundary vertex indices* i **do**
    $v_i^{n+1} \leftarrow v_i^n$
return $\{v_i^{nbiter}\}$

Note that this amounts to perform Jacobi iterations to solve a linear system that we will discuss in Sec. 4.6.2. See Fig. 4.19 for an example result.

4.6.2 Conformal mapping

This section describes conformal mapping, but again starts with simpler premises.

Laplace-Beltrami and Cotan Laplacian

We want to define a notion of Laplacian over the surface of a triangular mesh. To do that[26], let's consider that we want to solve the following equation on the mesh:

$$\Delta f = g$$

The weak formulation reads:

$$\int \Delta f(x)\, \phi(x) \mathrm{d}x = \int g(x)\, \phi(x)\, \mathrm{d}x$$

for all ϕ belonging to some suitable function space. We consider a set of *hat* test functions $\{\phi_i\}_{i=1..N}$, that are piecewise linear on the mesh (Fig. 4.20), such that $\phi_i(v_i) = 1$ and $\phi_i(v_j) = 0, j \neq i$, and hence consider $\phi(x) = \sum_j \lambda_j \phi_j(x)$. While it would not make sense to directly take the Laplacian of ϕ (since ϕ is piecewise linear, its second derivatives are 0!), we can nevertheless integrate by part (Sec. 3.4.1) to obtain:

$$-\int \nabla f . \nabla \phi \mathrm{d}x = \int g(x)\, \phi(x)\, \mathrm{d}x$$

We also write f (and g) in the $\{\phi_i\}$ basis, such that $f = \sum f_i \phi_i(x)$, and so, by bilinearity:

$$-\sum_{i,j} f_i \lambda_j \int \nabla \phi_i . \nabla \phi_j \mathrm{d}x = \sum_{i,j} g_i \lambda_j \int \phi_i(x)\, \phi_j(x)\, \mathrm{d}x$$

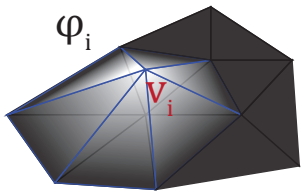


Figure 4.20: The piecewise linear basis vector associated to vertex v_i.

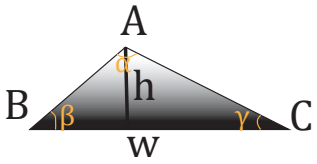


Figure 4.21: Notations for our triangle ABC.

We are now left with evaluating $L_{i,j} = \int \nabla\phi_i.\nabla\phi_j \mathrm{d}x$ for all (i,j). The matrix $L = [L_{i,j}]$ is the discretization of the Laplacian operator.

Given a triangle ABC with $\phi_A = 1$ on A and 0 on B and C (Fig. 4.21), it is easy to see that $\nabla\phi_A = \frac{BC^\perp}{2a}$ with a the area of the triangle, and $BC^\perp$ denotes a rotation of BC by 90 degrees counter-clockwise. Indeed, the gradient should be constant over the triangle since the function is linear, the function is constantly 0 on the edge BC so the gradient is necessarily orthogonal to BC, and the function goes from 1 to 0 over the height h of the triangle. The area a of the triangle is $a = BC.h/2$ (where I denote BC both the length and so vector – that should be clear from the context), and since $BC^\perp$ has length BC, $\frac{BC^\perp}{2a}$ has length $1/h$ as expected.

Then $\int_{ABC} \nabla\phi_A.\nabla\phi_A \mathrm{d}x = a.\|\frac{BC^\perp}{2a}\|^2$ since we are integrating a constant over a triangle. And using the fact that $\frac{BC^\perp}{2a}$ has length $1/h$, we obtain $\int_{ABC} \nabla\phi_A.\nabla\phi_A \mathrm{d}x = \frac{BC}{2h}$. Given the angles (A,α), (B,β) and (C,γ), and given the definition of the tangents $\tan\beta$ and $\tan\gamma$, it is easy to see that $\int_{ABC} \nabla\phi_A.\nabla\phi_A \mathrm{d}x = \frac{1}{2}(1/\tan\beta + 1/\tan\gamma) = \frac{1}{2}(\mathrm{cotan}\beta + \mathrm{cotan}\gamma)$. We can similarly see that $\int_{ABC} \nabla\phi_A.\nabla\phi_B \mathrm{d}x = -\frac{1}{2}\mathrm{cotan}\gamma$.

By summing over all triangles adjacent to vertex i (we denote $i \sim j$ if vertex i shares a triangle edge with vertex j), we can now build our Laplacian matrix[27] L:

$$L_{i,j} = \begin{cases} -\frac{1}{2}(\mathrm{cotan}\alpha_j + \mathrm{cotan}\beta_j)\gamma & i \sim j \\ \frac{1}{2}\sum_{i\sim j}(\mathrm{cotan}\alpha_j + \mathrm{cotan}\beta_j) & i = j \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

Regarding the right hand side $\sum_{i,j} g_i\lambda_j \int \phi_i(x)\,\phi_j(x)\,\mathrm{d}x$, we also need to evaluate $\int \phi_i(x)\,\phi_j(x)\,\mathrm{d}x$. An approximation by a diagonal matrix M, called the lumped mass matrix, can be obtained by taking as $M_{i,i}$ one third of the summed areas of triangles incident to i.

[26] more details at `https://graphics.stanford.edu/courses/cs468-13-spring/assets/lecture12-lu.pdf`

[27] Note that the sign of the Laplacian may differ in the litterature. The *true* Laplacian is semi-definite *negative*, but to simplify notations since algorithms usually work for semi-definite *positive* matrices, many people just change the sign of the Laplacian operator (like in the presented formulas).

Cauchy-Riemann equations

The idea behind a conformal mapping is that angles are preserved via the transformation. Without loss of generality, a straight angle remains straight. Given a point $X(u,v)$ on the surface[28] given its conformal parameterization (u,v), and let $N(u,v)$ be the (unit) normal vector at $X(u,v)$, we have that $N \times \frac{\partial X}{\partial u} = \frac{\partial X}{\partial v}$ to enforce orthogonality of isolines on the surface.

Writing locally this equality within a single triangle T in its local frame (aligning the triangle's normal to the z axis), and using complex numbers to represent positions: $X = x + iy$, we obtain that $i\frac{\partial X}{\partial u} = \frac{\partial X}{\partial v}$ (where we rotated $\frac{\partial X}{\partial u}$ by multiplying by i). Or written equivalently, without complex numbers:

$$\begin{cases} \frac{\partial x}{\partial u} &= \frac{\partial y}{\partial v} \\ \frac{\partial y}{\partial u} &= -\frac{\partial x}{\partial v} \end{cases} \tag{4.15}$$

These equations are called Cauchy-Riemann equations and form the basis of holomorphic/analytic functions. If a (complex) function is analytic (i.e., obeys Cauchy-Riemann's equations) with non-zero (complex) derivative, then it defines a conformal map, and conversely.

Since we will often see conformal mapping results onto disks, it is interesting to note that a Möbius transform conformally maps a disk to a disk. A Möbius transform is a map of the form $\phi(z) = \frac{az+b}{bz+\bar{a}} = e^{i\theta}\frac{z-c}{1-\bar{c}z}$ (c is the image of 0 and θ is a rotation angle). Möbius transforms define the $PSL(2,\mathbb{R})$ group, called the Möbius group. As such, a conformal map to a disk is only unique up to these 3 degrees of freedom. The Riemann Mapping theorem states that given a simply connected[29] domain D (whose boundary has more than one point) of the complex plane and z_0 a point inside it, there exists a unique conformal mapping ϕ from D to a unit disk such that $\phi(z_0) = 0$ and $\phi'(z_0) > 0$.

More generally, regarding the target space, Riemann uniformization theorem states that any simply connected surface can be embeded into the complex plane, the complex projective line or the hyperbolic plane. Unfortunately, for computer graphics purposes, only genus-1 surfaces (with one hole, e.g., a torus) or surfaces with boundaries can be mapped to the complex plane, i.e., the space that is most relevant to store textures. To map more complex surfaces, it is required to cut the mesh either into different charts (that are each topological disks), or to add seams to open the mesh. Adding seams can also be used to reduce area distortions: the process involves adding *cone singularities*, i.e., identifying a highly distorted point, and cutting to the nearest boundary (see Fig. 4.24).

Harmonic mapping

We can further differentiate both Cauchy-Riemann equations w.r.t. x and y to obtain:

$$\frac{\partial^2 x}{\partial u^2} = \frac{\partial^2 y}{\partial v \partial u} \tag{4.16}$$

$$= -\frac{\partial^2 x}{\partial v^2} \tag{4.17}$$

$$\frac{\partial^2 y}{\partial v^2} = \frac{\partial^2 x}{\partial u \partial v} \tag{4.18}$$

$$= -\frac{\partial^2 y}{\partial u^2} \tag{4.19}$$

$$\rightarrow \quad \Delta X(u,v) = 0 \tag{4.20}$$

[28] In fact, we will not deal with higher-dimensional objects. Liouville's theorem states that in dimenion (strictly) greater than 2, only Möbius transformations are conformal – those are extremely rigid and are thus pretty much uninteresting.

[29] Simply connected = any loop can be contracted to a point

Now, Δ is to be understood locally on a surface (it is not just the Euclidean Laplacian – unless the surface is perfectly flat – since we worked on a local frame!). This is the Laplace-Beltrami operator we have seen in Sec. 4.6.2 (using, for example, cotan weights[30]) which generalizes the Laplacian operator to surfaces.

This implies that conformal maps are harmonic, i.e., they satisfy $\Delta X = 0$. Also, a harmonic mapping of a topological disk to a subset of $\mathbb{R}^2$ with a *convex* boundary is bijective (Rado-Kneser-Choquet th.), i.e., there will not be flipped triangles in the mapping.

Perhaps the simplest algorithm for harmonic mapping a mesh that is a topological disk to an actual disk in the complex plane merely consists in solving Poisson equations. Specifically, the boundary of the mesh is mapped to the unit circle by simply converting edge length to angles. Then, the UV coordinates of the interior points are obtained by solving $\Delta U = 0$ and $\Delta V = 0$ with the boundary vertices as Dirichlet boundary condition. The resulting linear system is symmetric positive definite, and efficient solvers can be used (such as the Conjugate Gradient, see Sec. 5.2). The process would also work for other (convex) boundary configurations, such as squares, as often used for textures. It can even be used without imposing values on the boundaries, which reduces distortions. See Fig. 4.19 for a result on a disk.

While conformal maps are harmonic, the converse is not true, and harmonic maps may not preserve angles.

Algorithms for Conformal mapping

Conformal mapping can be obtained using a slightly more involved algorithm[31]. This algorithm is based on another property of conformal maps, that is, an homeomorphism ϕ between Riemannian manifolds $\phi : (S_1, g_1) \to (S_2, g_2)$ is conformal iff $\phi^* g_2 = e^{2u} g_1$. For reminder, a Riemannian metric g (here g_1 and g_2) gives a notion of dot product on a surface. For instance, $g(X, Y) = \langle X, Y \rangle$ for the Euclidean space. You do not need to know g for all possible pairs of vector: since it is bilinear (and symmetric positive definite), it is sufficient to know it for all pairs of basis vectors, so, on our 2d surfaces, only $g(X, X)$, $g(X, Y)$ and $g(Y, Y)$ are required for two independent vectors X and Y (basis vectors). In fact, the corresponding SPD matrix is called the first fundamental form. Like the regular Eucliean dot product, the metric gives a notion of angle (using an *acos*), length (using $\sqrt{g(u, u)}$) and area (using the square root of the determinant of the first fundamental form). The notation $\phi^* g_2$ means that we *pullback* the metric by the function ϕ, which means that we will evaluate the metric g_2 on the surface S_1 using the mapping ϕ. Specifically, we define this pullback metric as $(\phi^* g_2)_p(X, Y) = (g_2)_{\phi(p)}(d\phi_p(X), d\phi_p(Y))$ where p is the point where we evaluate the metric. The intuition behind $\phi^* g_2 = e^{2u} g_1$ is that, while we want to preserve angles, we allow lengths to be uniformly stretched. This stretching corresponds to a scaling factor in front of the metric, and this scaling factor may change at each point. To ensure positivity, we use the exponential. And to relate to areas (which we also do not care about in the context of conformal mapping) rather than length, we use the factor of 2. The factor e^{2u} is called the conformal factor and tells how much areas are stretched.

In the discrete setting, we will instead consider edge lengths, and write $\tilde{l}_{i,j} = e^{(u_i + u_j)/2} l_{i,j}$, where $l_{i,j}$ is the edge length between vertices i and j, and $\{u_i\}$ are scaling factors stored *per vertex*. It can be shown that in a triangle $t_{i,j,k}$ a triangle angle at vertex i can be expressed via its lengths (Fig. 4.22) as

$$\alpha^i_{jk} = 2 \tan^{-1} \sqrt{\frac{(l_{ij} + l_{jk} - l_{ki})(l_{jk} + l_{ki} - l_{ij})}{(l_{ki} + l_{ij} - l_{jk})(l_{jk} + l_{ki} + l_{ij})}}$$

[30] You will notice how the algorithm introduced for Tutte embeddings resemble that of a graph Laplacian (a Laplacian where each edge of the mesh graph is assigned a weight of 1) equals zero... however it is unfortunately not conformal.

[31] The simplest I could describe is based on *Conformal Equivalence of Triangle Meshes*: `https://dl.acm.org/doi/pdf/10.1145/1399504.1360676`

Further, for a planar surface $\Theta_i = \sum_{t_{ijk} \sim v_i} \alpha^i_{jk} = 2\pi$. The goal of the method is to minimize a function of the scaling factors $\{u_i\}$ such that the sums of the angles above are as close as possible to $\Theta_i = 2\pi$ for interior vertices (for boundary vertices, we would either fix $u_i = 0$ or prescribe the sum of the angles – for example, prescribing the sum of these angles to $\Theta_i = \pi/4$ for 4 of the boundary vertices and $\Theta_i = \pi/2$ for the other boundary vertices produces a mapping to a square ; prescribing the boundary Θ_i to $\frac{N-2}{2}\pi$ with N the number of boundary vertices results in a disk).

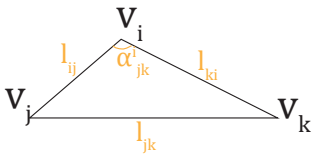


Figure 4.22: Notations for the angles.

The authors designed an energy E (see paper), whose gradient is given by

$$\partial_{u_i} E = \frac{1}{2}(\Theta_i - \sum_{t_{ijk} \sim v_i} \alpha^i_{jk}$$

and Hessian applied to some δu:

$$(\text{Hess}E.\delta u)_i = \frac{1}{2}(\Delta \delta u)_i = \frac{1}{4} \sum_{e_{i,j} \ni v_i} L_{ij}(\delta u_i - \delta u_j)$$

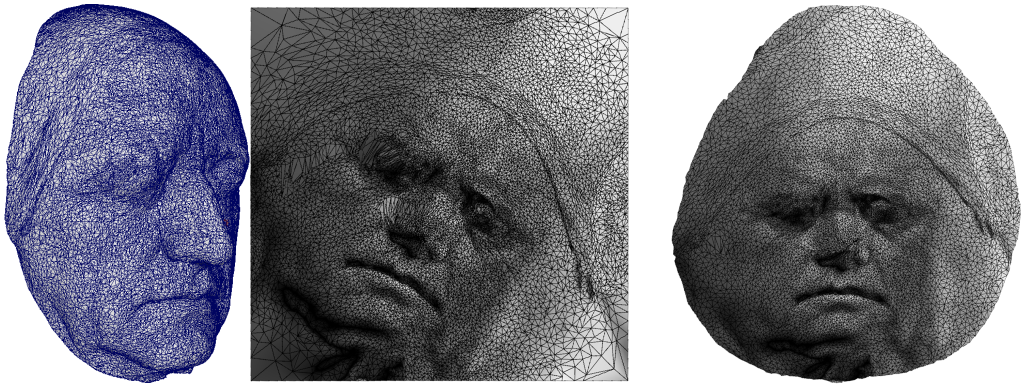
with L_{ij} the coefficients of our cotan Laplacian (Sec. 4.6.2). It is easy to see that when the gradient is zero, the sum of angles is equal to the desired result Θ_i, leading to a flat mapping.

With the gradient and Hessian, it becomes easy to develop a Newton solver (Sec. 4.4.4) to obtain optimal $\{u_i\}$ and thus, optimal edge length and triangle angles. Note that the optimization could lead to edge lengths that do not respect the triangle inequality (i.e., an edge longer than the sum of the two others), which messes up with angle formulas. When a triangle does not respect the triangle inequality, the angles are merely artificially set to 0 (for the angles between the small edges and the long edge) and π (for the angle between the two small edges).

To layout the triangle on a plane, one starts by placing the first triangle, and progressively propagate to neighboring triangles. Each time a new triangle (sharing an edge with an existing triangle) is placed, the coordinate of the new vertex on the plane can be obtained either by computing the intersection of two circles of known radii (the edge lengths) or by rotating the existing edge by the known angle and scaling it according to the known length.

See Fig. 4.19 for a result on a disk, Fig. 4.23 for a result on a square and a result that instead imposes edge lengths on the boundary, and Fig. 4.24 for the effect of introducing seams.

Aside from texturing, conformal maps have applications in remeshing (one can easily remesh in the 2d UV map domain, for instance using 2-d Centroidal Voronoi Tesselations), shape interpolation (which becomes easier once both meshes share the same 2d domain) and in fluids (an irrotational+incompressible flow is harmonic, and such fluid simulation can thus be conformally mapped to another domain).



Input mesh Mapped to square No boundary angular constraint

Figure 4.23: We can obtain a square by imposing 4 boundary angles to sum to $\pi/4$ and the rest to $\pi/2$, or we can reduce distortion by not imposing angles on the boundary but instead edge lengths.

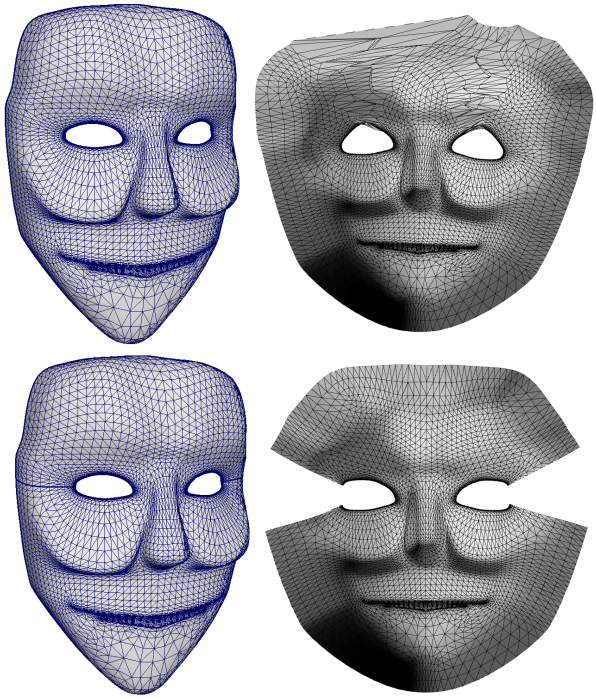


Figure 4.24: We can reduce distortions by introducing seams, and in particular to make the mesh homeomorphic to a disk. Left, the input meshes are shown, without and with seams that cut the mesh along highlighted edges. Right, the corresponding conformal parameterizations. The 3d model can be found here: `https://free3d.com/3d-model/mask3d-facemask-wall-maskfor-decorative-or-face-character-973366.html` (though I triangulated it, and there is a tiny hole in the mouth that I stiched).

Chapter 5

Fluid simulation

This chapter is dedicated to a few approaches used in computer graphics for solving incompressible Euler equations, a simplification of Navier Stokes equations. Note that other approaches exist, for example solving for vorticity[1]. While fluids can refer to liquids, gases or smoke, we will be mostly interested in liquids.

5.1 Principles

5.1.1 Helmoltz-Hodge decomposition

As a preliminary to fluid simulation, we will have a look at an important theorem in vector calculus, Helmoltz-Hodge decomposition (Fig. 5.1).

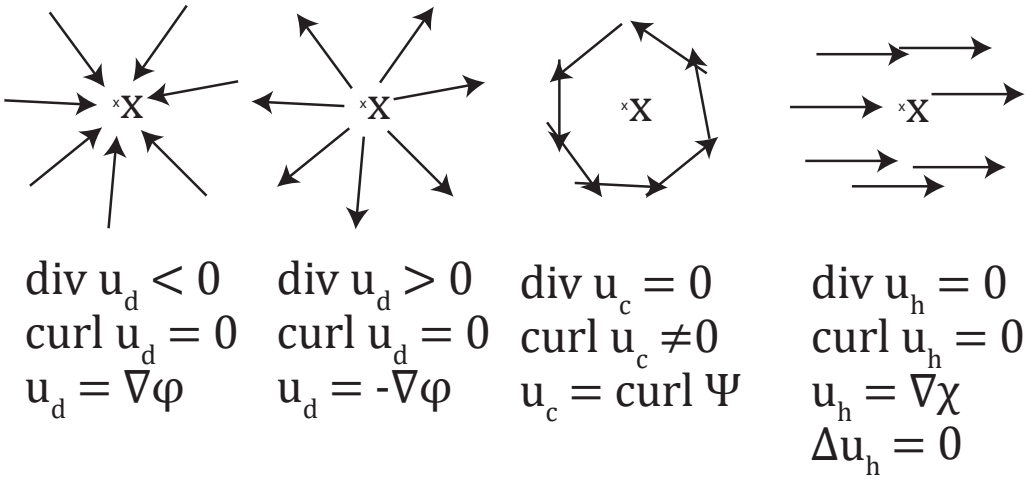


Figure 5.1: A vector field can be expressed as a sum of vector fields that are divergent free, curl free and harmonic.

This theorem can be stated in various ways, but it essentially says that any vector field can be decomposed into a sum of divergent-free, curl-free and harmonic vector fields. Specifically, one can write a vector field u:

$$u = u_d + u_c + u_h$$

[1]For instance to simulate smoke, *Simulation of Smoke based on Vortex Filament Primitives*: `http://www-evasion.imag.fr/Publications/2005/AN05/paper0132.pdf` , or *Lagrangian Vortex Sheets for Animating Fluids*: `https://dl.acm.org/doi/pdf/10.1145/2185520.2185608`

with u_d a curl-free vector field ($\operatorname{curl} u_d = 0$) and so, $\operatorname{div} u_d \neq 0$; u_c a divergence-free vector field ($\operatorname{div} u_c = 0$) with $\operatorname{curl} u_c \neq 0$; and u_h a harmonic vector field, which is both curl-free and divergence-free ($\operatorname{curl} u_c = \operatorname{div} u_c = 0$) with $\Delta u_c \neq 0$.

The other way to see this decomposition is to realize that a curl-free vector field and a harmonic vector field are both the gradient of some potential function, and that a divergence-free vector field should be the curl of another vector field. In short:

$$u = \nabla\phi + \operatorname{curl}\Psi$$

Intuitions about the shape of these vector fields are shown in Fig. 5.1. The divergence of a velocity field indicates whether there is a *sink* or a *source* pulling or pushing matter around. There typicially isn't any harmonic field when dealing with a subset of $\mathbb{R}^3$ (there could be if you consider boundary conditions that allow mass to freely flow in and out, for instance modeling the flow inside an infinite tube by studying a section of it ; there could also be harmonic fields if you deal with flows on the surface of a torus or higher genus surfaces).

5.1.2 Navier-Stokes and Incompressible Euler

Incompressible Navier-Stokes equations govern the motion of fluids and reads[2]:

$$\frac{\partial u}{\partial t} + u.\nabla u + \frac{1}{\rho}\nabla p = g + \nu\Delta u \tag{5.1}$$

$$\operatorname{div} u = 0 \tag{5.2}$$

There is a quite intuitive explanation of this equation in term of Newton's second law: $\sum_i F_i = ma$, the sum of the forces applied (locally) to the fluid is equal to its mass (or locally, its density, ρ) times the acceleration. First, let's assume that you are looking at a fluid particle whose position is described as $x(t)$, but consider a static point x of the domain. The velocity of the particle located at x and at time t is thus $u(x,t) = \frac{\partial x(t)}{\partial t}$. Now, its acceleration is the derivative of the velocity with respect to time. By applying the chain rule, $\frac{Du(x(t),t)}{Dt} = \frac{\partial x}{\partial t}.\frac{\partial u}{\partial x} + \frac{\partial u}{\partial t}$. Noting that $v = \frac{\partial x}{\partial t}$ and $\nabla u = \frac{\partial u}{\partial x}$, it reads $\frac{Du}{Dt} = \frac{\partial u}{\partial t} + u.\nabla u$. This is simply the expression of the acceleration of the fluid as seen from a fixed domain (when following particles in time using an acceleration $\frac{Du}{Dt}$ we call the approach *Lagrangian*, while we call approaches that see how velocities evolve on a static grid *Eulerian*).

So, with $\rho(\frac{\partial u}{\partial t} + u.\nabla u)$ the mass times the acceleration, this should be equal to the sum of external forces, that is, the force of gravity ρg (the mass, expressed locally, times the Earth's gravitational force g) and other forces. Among other forces is the pressure. If an object underwater is subject to pressure, but the pressure is the same everywhere, it will not make the object move – in fact, only differences in pressure matter: hence the term ∇p. In fact, the pressure p is often seen as a Lagrange multiplier to make fluid incompressible – it should take any value that makes the fluid incompressible (one can deduce p from u by taking the divergence on both sides and imposing incompressibility). And finally, viscosity. The intuition is that the Laplacian Δ measures how much a function at point x differs from its value within a neighborhood (see the second derivative as a measure of cuvature of the graph of a function). Highly viscous fluids will tend to have more homogeneous velocity fields. The coefficient in front of Δu that we will call η is the dynamic viscosity, while $\nu = \frac{\eta}{\rho}$ is the kinematic viscosity. Divide everything by ρ, and you obtain the *Momentum Equation* of the Navier-Stokes equations above.

As for $\operatorname{div} u = 0$, this simply states the incompressibility (Fig. 5.1).

[2]See the excellent book by Robert Bridson, *Fluid Simulation for Computer Graphics*. Robert's course notes are available at `https://www.cs.ubc.ca/~rbridson/fluidsimulation/fluids_notes.pdf`

Considering now an inviscid fluid (one for which the viscosity is zero), one obtain the Incompressible Euler equations:

$$\frac{\partial u}{\partial t} + u.\nabla u + \frac{1}{\rho}\nabla p = g \tag{5.3}$$

$$\operatorname{div} u = 0 \tag{5.4}$$

– the one we will be studying. Note that often, numerical schemes tend to introduce undesirable viscosity (called numerical viscosity), so it will not matter much at the moment from a computer graphics perspective.

Regarding boundary conditions, one has $p = 0$ on free surfaces (interfaces between the fluid and the air – ignoring surface tension), and the velocity at fluid-objects (or walls) interface is that of the object (or 0 for walls... unless walls move, you know!).

5.1.3 Chorin's projection

Chorin has proposed a splitting approach for Navier-Stokes in 1967. The idea is that given a PDE of the form:

$$\frac{\partial u}{\partial t} = f(u) + g(u)$$

one could build a first order finite difference discretisation in time and explicit Euler integration to obtain

$$u^{n+1} - u^n = dt(f(u^n) + g(u^n))$$

This can be split into two substeps:

$$u^* - u^n = dt(f(u^n))$$

$$u^{n+1} - u^* = dt(g(u^*))$$

So the idea of splitting is to decouple the different terms in the Navier-Stokes (or incompressible Euler) equation, and solve them separately (and not necessarily via explicit Euler schemes). The incompressibility constraint is just one step in this splitting.

More precisely, many fluid solvers compute the next time step of the velocity field:

Algorithm 6: Classical fluid solver time stepping using splitting.

Input: Current velocity field u^n
Output: Next velocity field u^{n+1}

1 $u^* = \text{advect}(u^n)$ by solving $\frac{Du^n}{dt} = 0$
2 $u^{**} = \text{addGravity}(u^*) = u^* + dt * g$
3 $u^{***} = \text{addViscosity}(u^{**}) = u^{**} + dt\nu\Delta u^{**}$
4 $u^{n+1} = \text{project}(u^{***})$ `// Make the fluid incompressible`
5 **return** u^{n+1}

5.2 Marker-and-Cell Method

The historical (1965) and most classical approach to solving Navier-Stokes is via the Marker-and-Cell (MAC) method and its variants. This approach is a semi-Lagrangian method, since it tracks particles for the advection (the advect method of the algorithm) since solving $\frac{Du}{dt} = 0$ using particles is much easier than dealing with the non-linear PDE $\frac{\partial u}{\partial t} + u.\nabla u$ on a grid, but it also makes use of a grid to deal with the other terms and the incompressibility step.

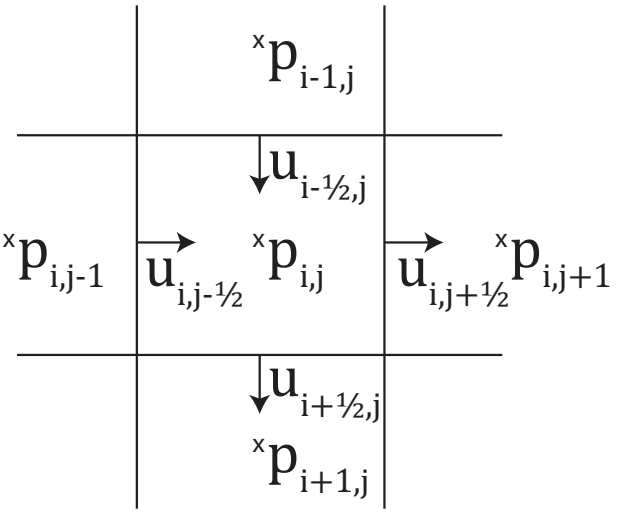


Figure 5.2: The MAC grid is a staggered grid where velocities are stored on the edges of the pressure grid.

The grid they used is a staggered grid: it is simply a grid structure which stores pressure and velocities on different offsetted grids (Fig. 5.2), to gain an order of approximation for free.

For instance, with the staggered grid, one gets $\frac{\partial p}{\partial x}$ using a second order centered finite differences to obtain the value at the grid location of $u_{i,j+1/2}$ (assuming j represents the x coordinate) by computing $\left(\frac{\partial p}{\partial x}\right)_{i,j+1/2} \approx \frac{p_{i,j+1}-p_{i,j}}{dx}$. And one would obtain it at the location (i,j) by averaging its value computed at $(i,j-1/2)$ and $(i,j+1/2)$.

The second important thing is to advect particles. Solving $\frac{Dq^n}{dt} = 0$ for some quantity q (q could be the velocity u, but also the color, or any other quantity) amounts to transporting that quantity along the flow, unchanged. That is, the variation (i.e., derivative) of that quantity is 0 when transporting it on a particle. A robust way to solve $\frac{Du^n}{dt} = 0$ at a grid position (i,j) is to check what the value of u was by tracking back a particle that moved backward in time by the velocity $u_{i,j}$ and interpolating when appropriate. So, assuming the grid position (i,j) is at spatial coordinate $(j/N_x, i/N_y)$, take the velocity $u^n_{i,j}$ and set $u^*_{i,j} = \text{interp}(u^n, (j/N_x, i/N_y) - dt * u^n_{i,j})$. Higher accuracy can be obtained via a Runge-Kutta method instead of forward Euler. Also note that while backtracking the particle, you may arrive at a grid point where no velocity was computed (e.g., due to numerical errors, $(j/N_x, i/N_y) - dt * u^n_{i,j}$ is outside of the fluid, in the air) – you may need to extrapolate your velocity field by a couple of voxels first.

The third important trick is the incompressibility step, called *project* for a reason. Recall the Helmoltz-Hodge decomposition. Imposing $\operatorname{div} u = 0$ corresponds to projecting the current velocity field estimate to one that has no curl, and the degree of freedom we have is by playing on the pressure field. I.e., one solves for a pressure field p such that the resulting velocity field u^{n+1} is incompressible. To do that, we realize that the fourth step of the algorithm can be written $u^{n+1} = u^{***} - \frac{dt}{\rho}\nabla p$, and one want to find p such that $\operatorname{div} u^{n+1} = 0$. So, taking the divergence (and using the fact that the divergence of the gradient is the Laplacian), one obtains $\operatorname{div} u^{***} - \frac{dt}{\rho}\Delta p = 0$. This simply amounts to solving a Poisson equation of the form $\Delta p = F$ (with the appropriate boundary conditions given previously) !

We have already seen a few approaches to solve linear systems. In practice, I use a Conjugate

Gradient algorithm which I briefly summarize for completeness but won't explain any further[3]:

Algorithm 7: Conjugate Gradient method to solve a linear system.

Input: A, b
Output: The solution x of $Ax = b$

$p_0 = r_0 = b - Ax_0$
for $k \leftarrow 0$ **to** *K-1* **do** // iterates
  $\alpha_k = \frac{\langle r_k, r_k \rangle}{\langle r_k, Ar_k \rangle}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k - \alpha_k A p_k$
  $\beta_k = \frac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle}$
  $p_{k+1} = r_{k+1} + \beta_k p_k$
return x^K

Finally, the computed velocity field can be used to advect particles (Fig. 5.3) or an implicit function representing the air-fluid interface.

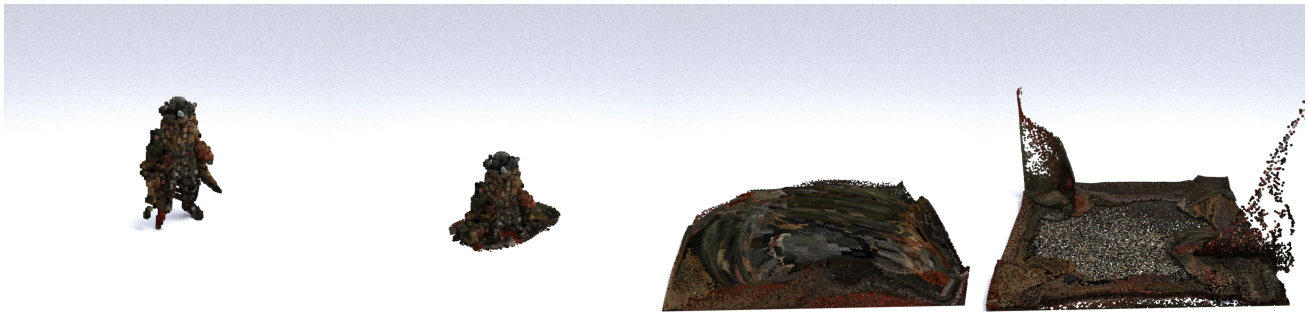


Figure 5.3: Our Davy Jones represented as particles moving according to the incompressible Euler equations. It simulated (very) approximately one frame in 15 seconds using a Jacobi preconditioned Conjugate Gradient, in parallel, on a 128^3 grid.

5.3 Smoothed particle hydrodynamics

Smoothed particle hydrodynamics is a Lagrangian way to solve PDEs, i.e., based on particle advection, and without requiring a grid (though some people use a grid for the incompressibility step), introduced by Lucy in 1977. The idea is to consider a set of particles moving in space, but instead of considering them like Dirac distributions (like infinitesimally small point masses), we consider a small radial function that describes these particles. Typicially, a spline can be used such as the cubic spline:

$$W(r,h) = \alpha_d \times \begin{cases} 2/3 - r^2 + 1/2r^3, & \text{if } 0 < r < 1. \\ 1/6(2-R)^3, & \text{if } 1 < r < 2. \\ 0 & \text{if } r > 2. \end{cases} \tag{5.5}$$

where α_d is a normalizing constant, where $\alpha_d = 1/h$ in 1-d, $\alpha_d = 15/(7\pi h^2)$ in 2-d and $\alpha_d = 3/(2\pi h^3)$ in 3-d. Here, r is the distance to the particle's center, h is called the smoothing length and support domain is $2h$ (the support of the function).

In general, any function f can be trivially written as $f(x) = \int_{\mathbb{R}^3} f(x')\delta(x - x')dx'$ where δ is the Dirac distribution (and abusing notations). The goal of our cubic spline kernel W is to approximate a

[3] I still have some slides at `https://projet.liris.cnrs.fr/origami/math/presentations/matrices.pdf` ; if you read french, Bruno Levy et al. have written a nice introduction here `https://ejcim2018.sciencesconf.org/data/pages/ejcim2018.pdf`

Dirac, for sufficiently small h, while being of compact support. The idea is thus to replace the trivial identity above by $f(x) = \int_{\mathbb{R}^3} f(x')W(\|x - x'\|, h)dx' + \mathcal{O}(h^2)$, or equivalently $f(x) = \int_{\mathbb{R}^3} \frac{f(x')}{\rho(x')}W(\|x - x'\|, h)\rho(x')dx' + \mathcal{O}(h^2)$ with ρ the density of mass.

Discretizing this identity on our set of particles located at $\{x_i\}_{i=1..N}$ leads to

$$f(x) \approx \sum_i \frac{m_i}{\rho_i} f(x_i)W(\|x - x_i\|)$$

where m_i is the (prescribed) mass of particle i ($m = \rho(x')dx'$). And by linearity of differential operators, we have:

$$\nabla f(x) \approx \sum_i \frac{m_i}{\rho_i} f(x_i)\nabla W(\|x - x_i\|)$$

$$\text{div } f(x) \approx \sum_i \frac{m_i}{\rho_i} \langle f(x_i), \nabla W(\|x - x_i\|)\rangle$$

and several Laplacian estimators have been proposed, such as:

$$\Delta f(x) \approx \sum_i \frac{m_i}{\|x - x_i\|^2 \rho_i}(f(x) - f(x_i)) * \langle (x - x_i, \nabla W(x - x_i, h)\rangle$$

In practice, more accurate estimators can be used, for instance:

$$\nabla f(x) \approx \sum_i \frac{m_i}{\rho(x_i)}(f(x_i) - f(x))\nabla W(\|x - x_i\|)$$

or, a more commonly used symmetric approximation to estimate the pressure gradient:

$$\nabla f(x) \approx \frac{1}{\rho(x)} \sum_i m_i \left(\frac{f(x)}{\rho^2(x)} + \frac{f(x_i)}{\rho^2(x_i)} \right) \nabla W(\|x - x_i\|)$$

We can then compute the quantities we need using this formula. For instance, the density of particles reads $\rho(x)$ used above can be estimated as:

$$\rho(x) = \sum_i m_i W(\|x - x_i\|, h)$$

which can as well be improved near free surfaces by further normalizing:

$$\rho(x) = \frac{\sum_i m_i W(\|x - x_i\|, h)}{\sum_i \frac{m_i}{\rho_i} W(\|x - x_i\|, h)}$$

In practice, all those quantities are *only* evaluated at (other) particle locations, such that $x = x_j$ for some j, and the summation is performed over a neighborhood of particle j, and especially since our particle support is compact, of radius $2h$. This also calls for fast neighborhood queries – you may use a regular grid for that, or re-use your favorite kd-tree library if its supports querying a fast neighborhood.

⚠ NanoFlann's `radiusSearch` method, although documented to require the maximum distance as a parameter, actually requires the *squared* distance ! (and also returns squared distances as a result).

A typical solver will then simply add the various forces to the particle velocities, and advect particles according to their velocity. But among those forces are again pressure forces that make the

fluid incompressible. There is again the solution to discretize the same Poisson equation as before ($\operatorname{div} u - \frac{dt}{\rho}\Delta p = 0$) either on a grid (the scheme would be semi-Lagrangian!) or directly over particles using the above discretization of the Laplacian operator. This would make the fluid incompressible.

There is however a simpler option if one tolerates a small loss of incompressibility (about 1%): the weakly compressible model[4]. In this model, one can merely apply a force that is repulsive when the density of particles is higher than what it should be, or an attractive force otherwise. For instance, if, for water, the density estimate $\rho(x) = 2000kg/m^3$, it means there are twice as many particles as required here, and one should strive to break them appart. This is done by computing the pressure as:

$$p(x) = B\left(\left(\frac{\rho(x)}{\rho_0}\right)^\gamma - 1\right)$$

Here, $\gamma = 7$, $B = \frac{\rho_0 c_s^2}{\gamma}$, c_s the speed of sound in the fluid (assumed to be at least 100x larger than the maximum velocity of the fluid), and ρ_0 the expected density of the fluid (i.e., $\rho_0 = 1000kg/m^3$ for water). Since this does not require solving any linear system, this makes the approach extremely fast.

Finally, collisions with obstacles or boundaries can be performed in a similar way, by imposing a repulsive force between the fluid particles and *ghost* particles sampling the obstacles. This force is computed as a sum over neighboring ghost particles:

$$F(x) = \sum_i \frac{m_i}{m(x) + m_i}\Gamma(x, x_i)\frac{x - x_i}{\|x - x_i\|}$$

where $\Gamma(x,y) = 0.02\frac{c_s^2}{\|x-y\|}\begin{cases} 2q - 1.5q^2, & \text{if } q < 1. \\ 2/3, & \text{if } 1 < q < 2/3. \\ 0.5(2-q)^2, & \text{if } 2/3 < q < 2. \\ 0 & \text{if } q > 2. \end{cases}$ with $q = \frac{\|x-y\|}{h}$.

Also, since one obtains a density of particles, one can easily reconstruct a surface using the marching cube algorithm we previously saw in Sec. 4.5 (Fig. 5.4).

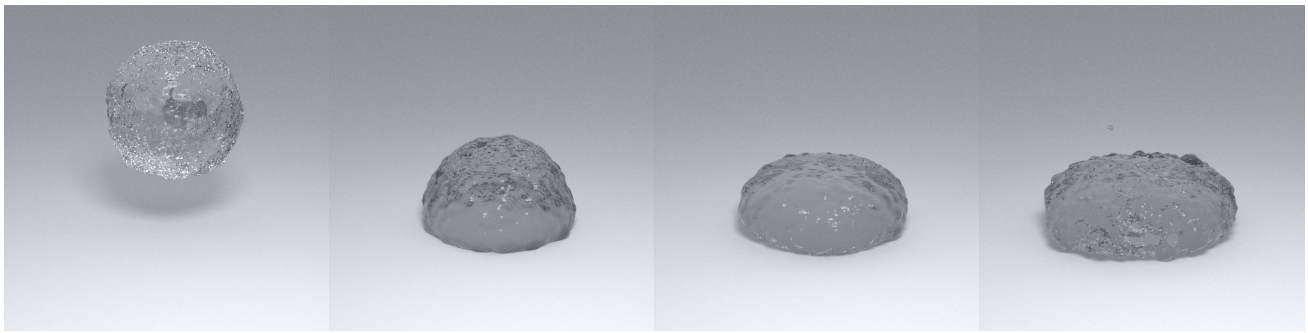


Figure 5.4: Simple water drop falling using weakly compressible SPH. The simulation (not the rendering!) is near realtime.

5.4 Using optimal transport

The last technique[5] we will see to simulate fluids via incompressible Euler's equations will make use of the semi-discrete optimal transport method developed in Sec. 4.4.4.

[4] *Weakly compressible SPH for free surface flows* `https://cg.informatik.uni-freiburg.de/publications/2007_SCA_SPH.pdf`

[5] inspired by *A Lagrangian Scheme à la Brenier for the Incompressible Euler Equations*: `https://hal.archives-ouvertes.fr/hal-01425826/document`

A few intuitions help motivate the use of optimal transport. First a theorem by Brenier[6] that relates optimal transport to Helmoltz-Hodge decomposition and polar decomposition of vector fields. The consequence of this theorem is that if you take a set of particles at position X and advect them with any velocity field, you will obtain particles at new positions Y. Now, if you compute an optimal transport map between X and Y, the map will produce a velocity field that is the closest divergence-free velocity field to the original velocity field in the least square sense. In short, optimal transport can enforce the incompressibility constraint. A second intuition is brought by going back to our repulsive/attractive forces in SPH to weakly enforce incompressibility. The goal was to force areas of high density to by repulsed towards areas of lower density such that the resulting distribution of mass is uniform. And this can be optimally achieved by computing an optimal transport map between a uniform density and the set of particles, exactly as we did in Sec. 4.4.4. A third intuition is that the semi-discrete optimal transport approach we implemented merely allowed to control the volume of each cell of a power diagram. By imposing these cells to remain of constant volume across the simulation, at least we are sure the fluid is incompressible!

The Lagrangian approach of Gallouët and Mérigot hence considers the semi-discrete optimal transportation problem between a set of fluid particles and a uniform density, leading to a Laguerre's cell for each particle. It then considers a spring force pushing each particle to the centroid of its Laguerre's cell. Doing so, particles that are too tightly packed will get spread closer to a uniform density at each time step.

Formally, the splitting scheme now reads:

Algorithm 8: One time-step of the Gallouët Mérigot scheme.

Input: Positions X, velocity v and mass m of particles
Output: New positions X' and velocity v' of particles

$V_W = \text{OptimalTransport}(X, \text{Uniform})$ `// optimize weights W of the Laguerre's cells of all particles`
for $i = 1..N$ **do** `// For each particle`
  $F^i_{spring} = \frac{1}{\epsilon^2}(Centroid(V^i_W) - X_i)$
  $F^i = F^i_{spring} + m_i\vec{g}$
  $v'_i = v_i + \frac{dt}{m_i}F^i$
  $X'_i = X_i + dt\, v_i$
return X', v'

Particles going outside of the domain can bounce back inside if needed.

Now, the interesting part is how to simulate free surface fluids with this scheme (as here, we considered a uniform fluid density in the entire simulation domain).

The solution is to consider a set of particles for the fluid and a set of particles for the air, while only moving the fluid particles. The air particles should cover the entire domain (in my example in Fig. 5.5, I performed a few Lloyd iterations to have all the air particles uniformly spaced in the domain). And then instead of considering an optimal transport only between the fluid and a uniform density, now we enforce the size of each fluid Laguerre's cell to have a constant mass m_i (or a constant volume since all particles have the same mass) as before by optimizing the weight w_i of each Laguerre's cell of each fluid particle, but we also consider an new single additional weight $\tilde{w}$ that will be shared by all air particles and is optimized such that the sum of all volumes of all Laguerre's cells of all air particles is a constant (equals to the initial air volume). This only slightly changes the semi-discrete optimal transport formulation, as now, there is an additional weight that counts for several Laguerre's cells – we are not optimizing N weights anymore but $N + 1$.

[6] *Polar factorization and monotone rearrangement of vector-valued functions*: `http://www.math.toronto.edu/~mccann/assignments/477/Brenier91.pdf`

At the limit case, when considering infinitely many air particles, the above scheme still considers $N+1$ weights to optimize (N for the fluid particles, 1 for the infinite number of air particles). In its final form, this amounts to considering a partial semi-discrete optimal transportation problem that can be rewritten similarly to equations for the non-partial semi-discrete optimal transport problem in Sec 4.4.4 (and we consider a uniform density $f = 1$ and Dirac masses of weight $\lambda_i = \frac{desired\_fluid\_volume}{N}$):

$$\begin{aligned}\min g(W) = \min \sum_{i=1}^{N} \int_{Pow_W(y_i)} (\|x - y_i\|^2 - w_i)\mathrm{d}x + \sum_{i=1}^{N} \frac{desired\_fluid\_volume}{N} w_i \\ + w_{air}(desired\_air\_volume - estimated\_air\_volume)\end{aligned}$$

where $desired\_fluid\_volume + estimated\_air\_volume = 1$ and whose gradient can be expressed for fluid particles as:

$$\frac{\partial g(W)}{\partial w_i} = \frac{desired\_fluid\_volume}{N} - Area(Pow_W(y_i))$$

and for the air variable:

$$\frac{\partial g(W)}{\partial w_{air}} = \frac{desired\_air\_volume}{N} - estimated\_air\_volume$$

with $estimated\_air\_volume = 1 - \sum_{i=1}^{N} Area(Pow_W(y_i))$

Within this context, it can be seen that to handle the infinite number of air particles, one simply need to intersect each power cell of the fluid particle with a disk of radius $R_i = \sqrt{w_i - w_{air}}$. This can be seen because we now have the definition of a power cell for fluid particle as:

$$\|x - x_i\|^2 - w_i \leq \|x - x_j\|^2 - w_j$$

for all $j \in \{1, .., N\}$ (i.e., regarding all other fluid cells), but since there are infinitely many air particles everywhere in the domain, when accounting for air, this adds the constraint:

$$\|x - x_i\|^2 - w_i \leq 0 - w_{air}$$

which amounts to intersecting the usual power cell with a disk. In practice, we can use a Sutherland-Hodgman algorithm to intersect the fluid polygon with a discrete approximation of a disk (beware of taking sufficiently many sides for that disk). A result can be seen in Fig. 5.6.

A simple potential acceleration is the warm-restart of Laguerre's weights. At each frame of the simulation, this consists in starting the optimization of the Laguerre's weights (for the semi-discrete optimal transport) by the last weights found at the previous time step. Results can be found in Fig. 5.5.

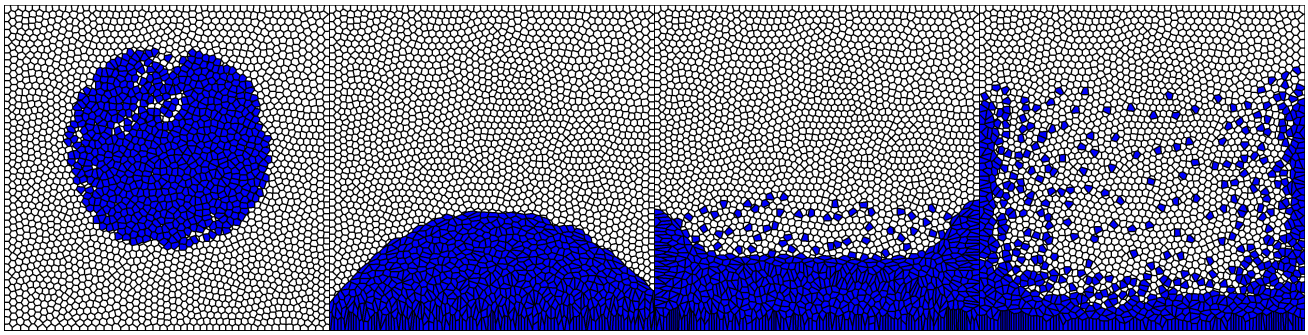


Figure 5.5: Visualization of a free-surface liquid simulated with semi-discrete optimal transport. The sum of all air Laguerre's cell has a prescribed mass, with each individual liquid Laguerre's cell has a prescribed mass. Here the optimization takes about 1 second per frame while there isn't much fluid-boundary interaction and about 30s–1 minute per frame afterwards (default L-BFGS settings) with 700 fluid particles and 2500 air particles (this could be made *much* faster, notably using the Newton's optimizer instead of quasi-Newton, and using other variants of power diagram computations! Early universe reconstructions are being performed with millions of particles with this approach on the GPU - see `https://twitter.com/BrunoLevy01/status/957552532661915649` and `https://twitter.com/BrunoLevy01/status/1242942393474670592` !). I used $\epsilon = 0.004$, $dt = 0.002$ and each particle has a mass of 200.

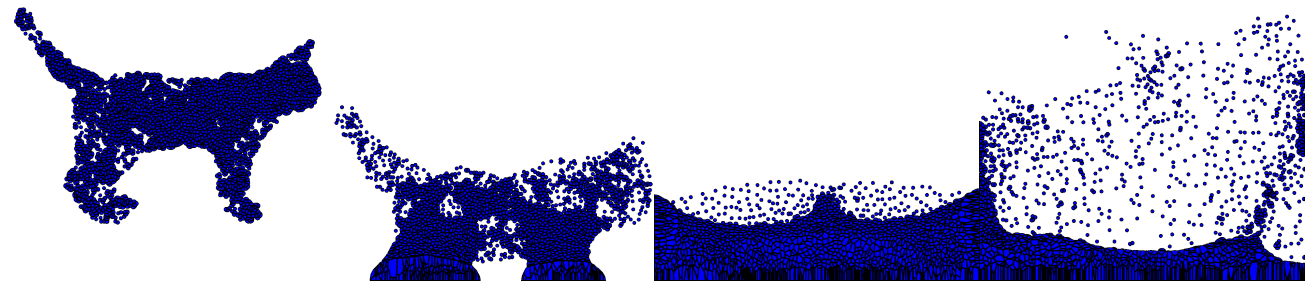


Figure 5.6: Visualization of a free-surface liquid simulated with semi-discrete optimal transport. The limit case when the number of air particles tend to infinity amounts to intersecting fluid Laguerre cells with a disk.