Computer Graphics Summary

Adrian Hirt

January 2023

1 Light & matter

1.1 Light emission

A **incandescent** light source produces visible light from heat (e.g. an old-fashioned light bulb).

A **blackbody** is a hypothetical body that absorbs all wavelengths of thermal radiation. They do not reflect light and therefore appear black if their temperatures are low enough. When heated to a given temperature, they will emit thermal radiation with the same spectrum. In a perfect blackbody, the color spectrum is defined purely by the temperature of the material, given by Plank's law.

On the other hand, **luminescence** is emission of light by a substance not resulting from heat, e.g. from chemical reactions, electrical energy, subatomic motions or stress on a crystal.

1.2 Quantifying light

1.2.1 Radiometry

Visible light is any radiation capable of directly causing visual sensation, in the wavelengths of 390 to 700nm.

We assume light consists of photons with:

- x: Position
- $\overrightarrow{\omega}$: Direction of travel
- λ: Wavelength

Each photon has an energy of $\frac{hc}{\lambda}$ Joules with h being Plank's constant and c the speed of light in vacuum.

We also have the following quantities (which depend on the wavelength):

Flux (Radiant Flux, Power), which is the total amount of radiant energy passing through a surface or space *per unit time*. Examples would be number of photons hitting a wall per second or number of photons leaving a lightbulb per second:

$$\Phi(A) \qquad \left[\frac{J}{s} = W\right]$$

Irradiance is the *area density* of flux, i.e. flux per unit area *arriving* at a surface. An example is the number of photons hitting a small patch of a wall per second, divided by size of patch:

$$E(x) = \frac{d\Phi(A)}{dA(x)} \qquad \left[\frac{W}{m^2}\right]$$

Radiosity (Radiant Exitance) is again the *area density* of flux, however here it's the flux per unit area *leaving* a surface. An example is the number of photons reflecting off of a small patch of a wall per second, divided by size of patch:

$$B(x) = \frac{d\Phi(A)}{dA(x)} \qquad \left[\frac{W}{m^2}\right]$$

Radiant Intensity is the *directional density* of flux, i.e. the power (flux) per solid angle. An example is the power per unit solid angle emitting from a point source.

$$I(\overrightarrow{\omega}) = \frac{d\Phi}{d\overrightarrow{\omega}} \qquad \left[\frac{W}{sr}\right]$$

The **solid angle** is the "extension" of the measurement of an angle (for circles) to spheres. For a sphere, the angle is defined as the arclength l over the radius r: $\theta = \frac{l}{r}$. For a sphere, the solid angle is defined as the area A over the radius r squared: $\Omega = \frac{A}{r^2}$.

To get the flux, we can integrate the radiant intensity over the area of the sphere S:

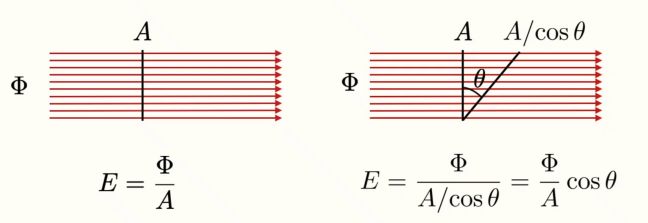
$$\Phi = \int_{S^2} I(\overrightarrow{\omega}) d\overrightarrow{\omega}$$

If we have an isotropic light source (emits light same in all direction) we have a radiant intensity of $I = \frac{\Phi}{4\pi sr}$ for each direction.

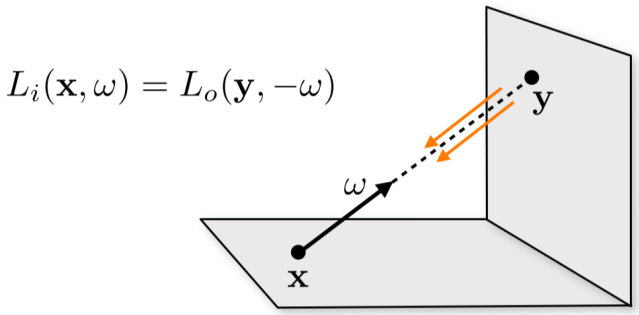
Radiance is the intensity per perpendicular unit area, i.e. the flux density per unit solid angle, per perpendicular unit area:

$$L(x, \overrightarrow{\omega}) = \frac{dI(\overrightarrow{\omega})}{dA^{\perp}(x, \overrightarrow{\omega})} = \frac{d^2\Phi(A)}{d\overrightarrow{\omega} dA(x) cos\theta} \qquad \left[\frac{W}{m^2 sr}\right]$$

Lambert's Cosine Law describes that an angled surface A has a smaller Irradiance E than an "unangled" surface, because the Flux is distributed over a larger area:



We can express the **incident radiance** L_i at one point as the **outgoing radiance** L_o at another point:



With this, we can express *irradiance* in terms of radiance by integrating over the hemisphere:

$$\int_{H^2} L(x, \overrightarrow{\omega}) cos\theta d\overrightarrow{\omega} = E(x)$$

2 Shape representation

2.1 Parametric representation

A **circle** is represented by the formula

$$p(t) = r(cos(t), sin(t)) \qquad t \in [0, 2\pi)$$

A **sphere** is represented by the formula

$$s(u, v) = r(cos(u)cos(v), sin(u)cos(v), sin(v)) \qquad (u, v) \in [0, 2\pi) \times [\frac{-\pi}{2}, \frac{\pi}{2}]$$

2.2 Implicit representation

Implicit representations of shapes are formulas, that when evaluate to zero, describe the points on the surface of their respective shape.
A **circle** is represented as:

$$f(x, y) = x^2 + y^2 - r^2$$

A **sphere** is represented as:

$$f(x, y, z) = x^2 + y^2 + z^2 - r^2$$

To compute the **normal vector** of the surface, we take the gradient of the implicit function:

$$\nabla f(x, y, z) = (\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz})^T$$

We can also have the following boolean set operations (for functions f_i):
Union: $\cup f_i(x) = min f_i(x)$
Intersection: $\cap f_i(x) = max f_i(x)$
Subtraction of f from g: $h = max(g, -f)$

3 Polygonal Meshes

3.1 Manifolds

A surface is a closed **2-manifold** if it is everywhere locally homeomorphic to a disk, i.e. you can place an infinitesimally small disk on any point of the surface and the surface would be equivalent to the disk. This means there are no "fold overs" or self-intersections.

A **manifold with boundary** means that the surface has a boundary, with each boundary point being homeomorphic to a half-disk.

In a **manifold mesh**, there are at most 2 faces sharing an edge. Boundary edges have one incident face, inner edges have two incident faces. If the manifold is closed and not intersecting, it divides the space into inside and outside.

The **Genus** of a manifold is 0.5× the maximal number of closed paths that do not disconnect the graph. Informally, the number of handles ("donut holes").

3.2 Triangulation

A **triangulated mesh** is a polygonal mesh where every face is a triangle. Simplifies operations & storage. Any polygon can be triangulated.

4 Appearance modelling

4.1 BRDF

A **Bidirectional Reflectance Distribution Function** provides a relation between incident radiance and differential reflected radiance. From this, we can derive the **Reflection Equation**:

$$\int_{H^2} f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i) cos\theta_i d\vec{\omega}_i = L_r(x, \vec{\omega}_r)$$

with H^2 meaning we integrate over the hemisphere, f_r being the BRDF, L_i the incident radiance and $cos\theta$ the term from Lambert's cosine law.
The Reflection Equation describes a *local illumination* model, i.e. the reflected radiance due to incident illumination from all directions.

4.2 Lambertian Diffuse BRDF

For **Lambertian Reflection**, the BRDF is a constant, which means the Reflection Equation can be written as:

$$L_r(x) = f_r E(x)$$

with $f_r = \frac{1}{\pi}$. For diffuse reflectance (**albedo**) $p \in [0..1]$ we have $f_r = \frac{p}{\pi}$, with $p = 0$ meaning that all energy is absorbed, and $p = 1$ meaning that all energy is reflected.

4.3 Ideal Specular Reflection

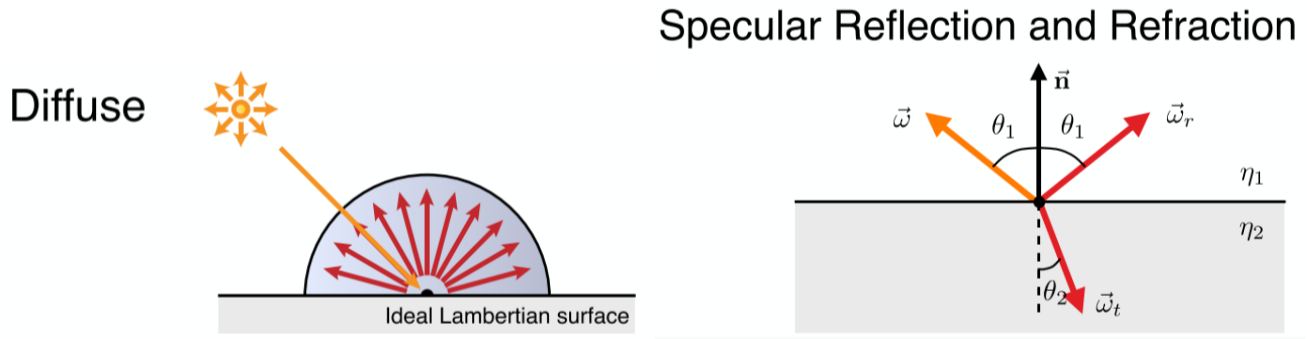
Here, we have angle of incident direction = angle of reflected direction (in relation to the surface normal).

4.4 Ideal Specular Refraction

Snell's Law describes the relation between the incident direction $\overrightarrow{\omega}$ with angle θ_1 and the refracted direction $\overrightarrow{\omega}_t$ with angle θ_2 as:

$$\eta_1 sin\theta_1 = \eta_2 sin\theta_2$$

with eta_1 and eta_2 being the **index of refraction** of the two medias.



5 Ray tracing

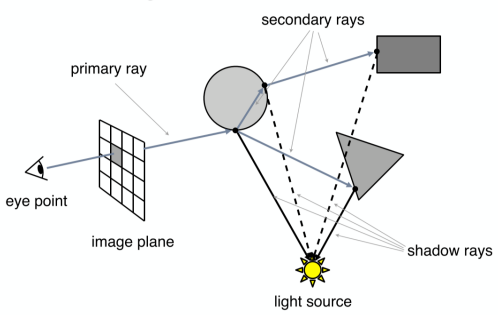
5.1 Assumptions

- No diffraction, no polarization, no interference
- Light travels in a straight line in a vacuum
- Discrete-wavelength approximation of color (RGB)

5.2 Basics

A **Ray** r is parametrized by an origin o and a direction d as: $r(t) = o + td$.

In **Light Tracing**, we shoot out rays from light sources and measure the rays hitting the camera point. This is inefficient, as most rays don't hit the camera, and therefore we use **Camera Tracing**. Here, we shoot out rays from the camera:



With **Basic Shading** with a **Point light** for a point x on a triangle, with normal n and direction $\overrightarrow{\omega}_l$ from x to the point light, we have:

$$E = \frac{\Phi}{4\pi r^2} cos\theta = \frac{\Phi}{4\pi r^2}(n \cdot \overrightarrow{\omega}_l)$$

with the term $\frac{\Phi}{4\pi r^2}$ being the point light source irradiance (point / sphere has surface area of $4\pi r^2$).

Combined with the diffuse shading from above, we get the following term for diffuse shading:

$$L_d(x, \overrightarrow{\omega}) = k_d E = k_d \frac{\Phi}{4\pi r^2}(n \cdot \overrightarrow{\omega}_l)$$

with k_d being the **material parameter**.

6 Direct Illumination

6.1 Rendering Equation

The **Rendering Equation** describes the energy equilibrium from the outgoing light to the emitted + reflected light:

$$L_o(x, \overrightarrow{\omega}_o) = L_e(x, \overrightarrow{\omega}_o) + L_r(x, \overrightarrow{\omega}_o) = L_e(x, \overrightarrow{\omega}_o) + \int_{H^2} f_r(x, \overrightarrow{\omega}_i, \overrightarrow{\omega}_o) L_i(x, \overrightarrow{\omega}_i) cos\theta d\overrightarrow{\omega}_i$$

In **direct illumination**, the L_i term comes directly from an emitter, which gives us:

$$L_i(x, \overrightarrow{\omega}) = L_e(r(x, \overrightarrow{\omega}), -\overrightarrow{\omega})$$

For simplicity, we will ignore the emission, i.e. $L_e = 0$, which then gives us the following term to compute direct illumination in the **hemispherical form**:

$$L_r(x, \overrightarrow{\omega}_r) = \int_{H^2} f_r(x, \overrightarrow{\omega}_i, \overrightarrow{\omega}_r) L_e(r(x, \overrightarrow{\omega}_i) - \overrightarrow{\omega}_i) cos\theta_i d\overrightarrow{\omega}_i$$

We can estimate the integral by using Monte Carlo Integration. However, by sampling from the hemisphere (H^2 above), we often have many samples evaluating to zero, because they don't hit a light source directly. Instead, we can integrate over the solid angle Ω subtended by the light (i.e. only sample from the part of the hemisphere that "sees" the light).

For integrating over surfaces, we need some different notation, setting:

- $L_i(x, \overrightarrow{\omega}_i) = L_i(x, y)$
- $L_r(x, \overrightarrow{\omega}_r) = L_r(x, z)$
- $f_r(x, \overrightarrow{\omega}_i, \overrightarrow{\omega}_r) = f_r(x, y, z)$
- $d\overrightarrow{\omega}_i = \frac{|cos\theta_o|}{||x-y||^2} dA$

With this, we can then write the integral in its **surface area form**:

$$L_r(x, z) = \int_A f_r(x, y, z) L_i(x, y) G(x, y) dA(y) = \int_A f_r(x, y, z) L_i(x, y) V(x, y) \frac{|cos\theta_i||cos\theta_o|}{||x - y||^2} dA(y)$$

with $G(x, y)$ being the *geometry term* consisting of the *visibility term* V which is 1 if y is visible from x and 0 otherwise. The fraction is the chance that a photon emitted from x will hit y, which decreases as the points move away from each other (denominator) or face away from each other (numerator).

With this, we can now simply use the area of the emitter A_e to integrate over emissive surfaces only, which is more efficient.

6.2 Light Sources

We can have various light sources, which can be categorized as **Delta Lights**, which create hard shadows (point light, spot light, directional light) and **Area / Shape lights**, which create soft shadows (quad lights, sphere light, mesh light).

6.2.1 Point light

A **point light** is usually defined using a point P and emitted power Φ. We have omnidirectional emission from a single point, and therefore can write the integral over the point light as:

$$L_r(x,z) = \frac{\Phi}{4\pi} f_r(x,p,z) V(x,p) \frac{|cos\theta_i|}{||x-p||^2}$$

6.2.2 Spot light

A **spot light** is usually defined using a point P and a directionally dependent radiant intensity I:

$$L_r(x,z) = I(p,x) f_r(x,p,z) V(x,p) \frac{|cos\theta_i|}{||x-p||^2}$$

6.2.3 Directional light

A **directional light** is defined using a direction $\overrightarrow{\omega}$ and radiance $L_d(\overrightarrow{\omega})$ coming from direction $\overrightarrow{\omega}$:

$$L_r(x, \overrightarrow{\omega}_r) = f_r(x, \overrightarrow{\omega}, \overrightarrow{\omega}_r) V(x, \overrightarrow{\omega}) L_d(\overrightarrow{\omega}) cos\theta$$

6.2.4 Quad light

A **quad light** has finite area, which creates soft shadows. Here we just integrate over the area of the quad, as above.

6.2.5 Sphere light

A **sphere light** is defined using a point p, radius r and emitted power Φ. The sphere has a finite area of $4\pi r^2$. To sample the sphere, we either can sample the sphere uniformly over the whole area (many points not visible from shading point), over the area of the visible spherical cap (not ideal as emitted radiance is weighted by the cosine term) or uniformly sample the solid angle subtended by the sphere.

6.2.6 Mesh light

A **mesh light** is an emissive mesh where every surface point emits given radiance L_e. We have a total area of $\sum A(k)$.

6.3 Environment Lighting

Using **environment lighting**, we avoid having to model the whole environment by representing it by one or more images. These images "wrap" the virtual scene, serving as a *distant source* of illumination:

$$L_r(x, \overrightarrow{\omega}_r) = \int_\Omega f_r(\overrightarrow{\omega}_i, \overrightarrow{\omega}_r) L_i(x, \overrightarrow{\omega}_i) cos\theta_i d\overrightarrow{\omega}_i = \int_\Omega f_r(\overrightarrow{\omega}_i, \overrightarrow{\omega}_r) L_{env}(\overrightarrow{\omega}_i) V(x, \overrightarrow{\omega}_i) cos\theta_i d\overrightarrow{\omega}_i$$

When sampling from the environment map, we want to distribute the weights in such a way that we sample mostly from the "bright" parts of the environment map, as these parts represent "light sources".

7 Monte Carlo integration

We can approximate the integrals of the reflection equations using **Monte Carlo integration**:

$$\int_D f(x)dx = \int_D \Big[\frac{f(x)}{p(x)}\Big] p(x)dx \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} = F_N$$

where x_i are random variables distributed according to the PDF $p(x)$.

The **Monte Carlo Estimator** is an **unbiased** estimator, which means that the expected value equals the integral. Extension to higher dimensions is straightforward (as the number of samples is independent of dimension), but the convergence is slow, with a runtime of $O(\frac{1}{\sqrt{N}}$, which means reducing the error by a factor of 2 requires 4 times as many samples.

7.1 Sampling random variables

Usually, we have random variables uniformly distributed in the interval $[0, 1]$, i.e. if we sample 2 random variables, we sample uniformly from a square.

For shapes, we can use **Rejection Sampling**, which samples uniformly from $[0, 1]$ and then inserts these values into the formula for the shape (e.g. a sphere) and then checks if the result lies on the shape. This however is inefficient, as most sampled points are rejected!

For many shapes however, we can define a warping, which is a function which transforms the samples from the square to samples from the other shape (e.g. from a disk). This requires us to write this warp function, however it is much more efficient as we're not wasting any samples.

7.2 Importance sampling

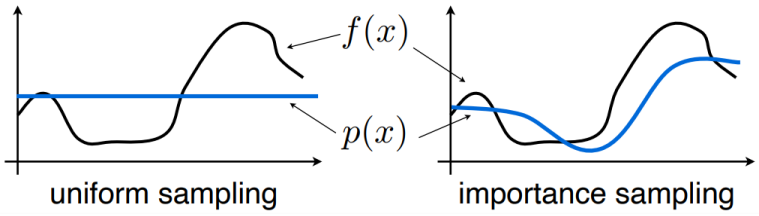
By using **importance sampling**, we can reduce the variance of our sampling method. Usually, we have our estimator $F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}$ for the integral $\int f(x)dx$.
When we assume that the PDF is proportional with some factor c to the function with $p(x) = cf(x)$, we get:

$$\int p(x)dx = 1 \Longrightarrow c = \frac{1}{\int f(x)dx}$$

with which we get the estimator $\frac{f(X_i)}{p(X_i)} = \frac{1}{c} = \int f(x)dx$, which means we have a zero variance estimator.

The issue with this approach however is, that setting $p(x) = cf(x)$ requires knowledge of the integral that we are trying to solve.

However, if the pdf is similar to the integrand, the variance can be significantly reduced. The common strategy here is to sample proportional to the integrand:



7.2.1 Combining multiple strategies

We can combine multiple importance sampling methods to get a better result, however we cannot simply average the two different estimators, as *variance is additive* and therefore does not work.
Instead, we sample from the average pdf:

$$\frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{0.5(p_1(x_i) + p_2(x_i))}$$

7.3 Sampling arbitrary distributions

7.3.1 Inversion method

The **inversion method** includes the following steps:

1. Compute the CDF $P(x) = \int_0^x p(x')dx'$
2. Compute the inverse $P^{-1}(x)$
3. Obtain an uniformly distributed random number ξ
4. Compute $X_i = P^{-1}(\xi)$

7.3.2 Jacobian method

We can use the **Jacobian method** for a random variable $X \sim p_x(x)$ and a transformation T to find the distribution of $Y_i = T(X_i)$. The new density is:

$$p_y(y) = p_y(T(x)) = \frac{p_x(x)}{|J_T(x)|}$$

where $|J_T(x)|$ is the absolute value of the determinant of the Jacobian matrix of T. The Jacobian matrix J is the matrix containing all first-order partial derivatives of the transformation matrix T.

7.4 Multiple Importance Sampling

When using Monte Carlo integration, variance is high when the PDF is not proportional to the integrand. In the worst case, we have *rare samples* with *huge contributions*, which lead to "firefly" artifacts. For example, in the following term, such a firefly would appear if $f(x_i)$ is large, but the corresponding PDF $p(x_i)$ is small:

$$\langle F^N \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$

The approach in **Multiple Importance Sampling** now is that if at least one sampling strategy covers each part of the integrand well, then combining them should reduce artifacts. For that, we can combine e.g. 2 strategies weighted:

$$\langle F^{N_1+N_2} \rangle = \frac{1}{N_1} \sum_{i=i}^{N_1} w_1(x_i) \frac{f(x_i)}{p(x_i)} + \frac{1}{N_2} \sum_{i=1}^{N_2} w_2(x_i) \frac{f(x_i)}{p(x_i)} \qquad \text{with } w_1(x) + w_2(x) = 1$$

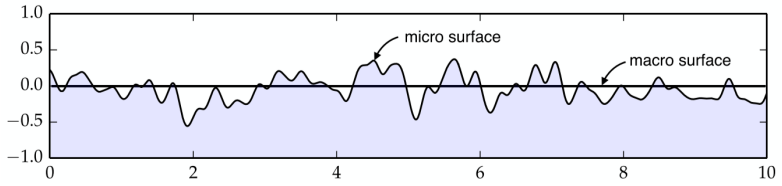
We can extend this term to arbitrary many strategies, the important restriction is that the weights always sum up to 1.

To choose the weights, we can use the **balance heuristic**:

$$w_s(x) = \frac{N_s p_s(x)}{\sum_j N_j p_j(x)}$$

8 Microfacet Theory

When using **microfacets**, we assume the surface consists of tiny **facets**, where the total area being illuminated is relatively large compared to the size of the microfacets. A facet can be perfectly *specular* or *diffuse*:



The **general microfacet model** is the following:

$$F(\vec{\omega}_i, \vec{\omega}_o) = \frac{F(\vec{\omega}_h, \vec{\omega}_o) \cdot D(\vec{\omega}_h) \cdot G(\vec{\omega}_i, \vec{\omega}_o)}{4|(\vec{\omega}_i \cdot n)(\vec{\omega}_o \cdot n)|} \qquad \vec{\omega}_h = \frac{\vec{\omega}_i + \vec{\omega}_o}{||\vec{\omega}_i + \vec{\omega}_o||}$$

In the above term, F is the **Fresnel coefficient**, D is the **Microfacet distribution** and G is the **Shadowing / masking** term. A good distribution to use for the D term is the *Beckmann Distribution*.
The shadowing / masking term depends on the chosen distribution, and it represents a microfacet being shadowed and / or masked by other microfacets.

9 Advanced Camera Models

In the most basic version, we use a **pinhole camera** for the ray tracer, i.e. a camera with an infinitesimally small aperture. With that, we don't have a focal plane, and every point in the scene is in focus. When using a **thin lens camera**, we have points which are on the focal plane, and others which are not on the focal plane

and therefore appear blurry.

The **depth of field** is the range of distances from which objects appear in focus. Objects outside this range appear blurry. To compute the image, we need to integrate over pixel area and the *aperture area*. Typically, the lens geometry is defined by the *radius* of the aperture and the *distance* to the focal plane.

10 Global Illumination

10.1 Light paths

We want to express light paths in terms of the surface interactions that have occurred. A light path is a chain of linear segments, joining at event "vertices".

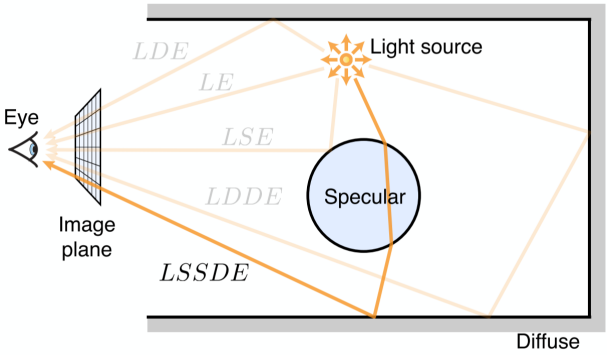
For this, we can use **Heckbert's Classification**, which classifies the events as follows (using a regex-style for classifying the number of occurences of the events):

- L: a light source
- E: the eye / camera
- S: a specular reflection
- D: a diffuse reflection
- $k+$: one of more of event k
- $k*$: zero or more of event k
- $k?$: zero or one k events
- $(k|h)$: a k or an h event

From this, we get the following types of illumination:

- Direct illumination: $L(S|D)E$
- Indirect illumination: $L(D|S)(D|S) + E$
- Classical(Whitted-style) ray tracing: $LDS * E$
- Full global illumination: $L(D|S) * E$

An example of various light paths:



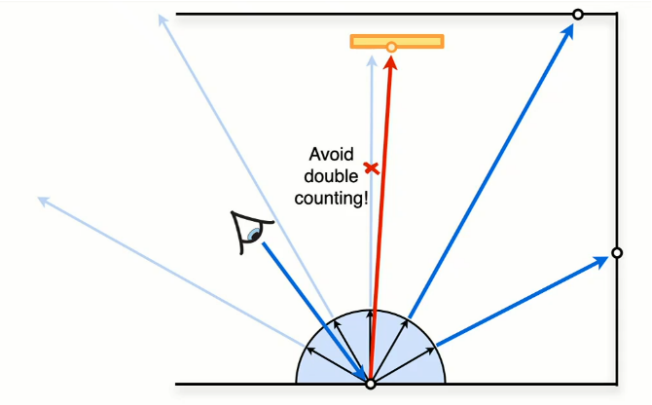
10.2 Recursive Monte Carlo ray tracing

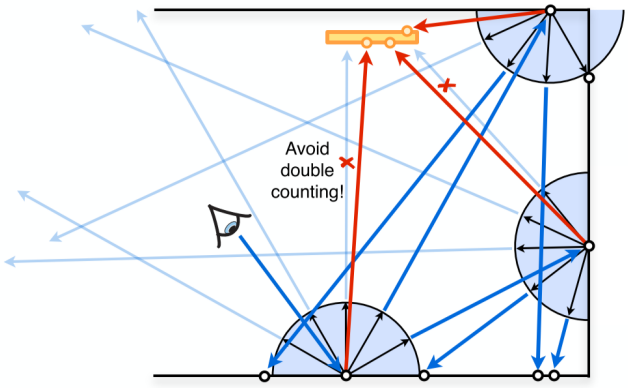
For direct illumination, it's sometimes better to estimate it by sampling emissive surfaces. As such, we estimate direct illumination separately from indirect illumination and then adding the two. For this, we shoot shadow rays (direct lighting, towards light sources) and gather rays (indirect lighting). This gives us the term we want to estimate:

$$L(x, \overrightarrow{\omega}) = L_e(x, \overrightarrow{\omega}) + L_d(x, \overrightarrow{\omega}) + L_i(x, \overrightarrow{\omega})$$

where L_e is the emitted color at x, L_d is the direct lighting and L_i the indirect lighting. For example, the first "iteration" of the recursive algorithm looks like the figure on the left, and the figure on the right contains the next iterations:

This approach however has the downside that the formulation is recursive and grows geometrically.



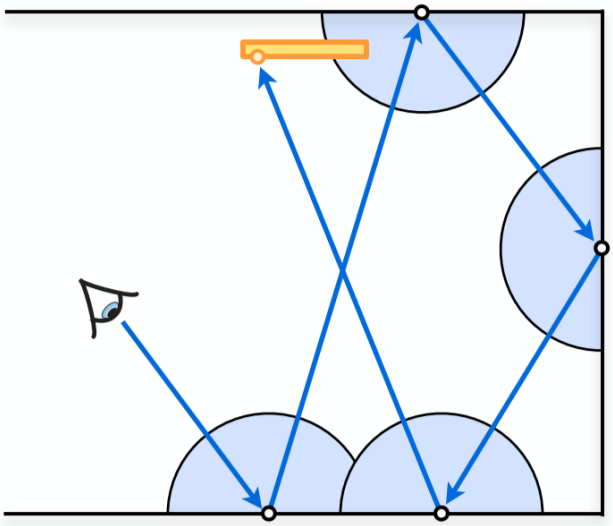


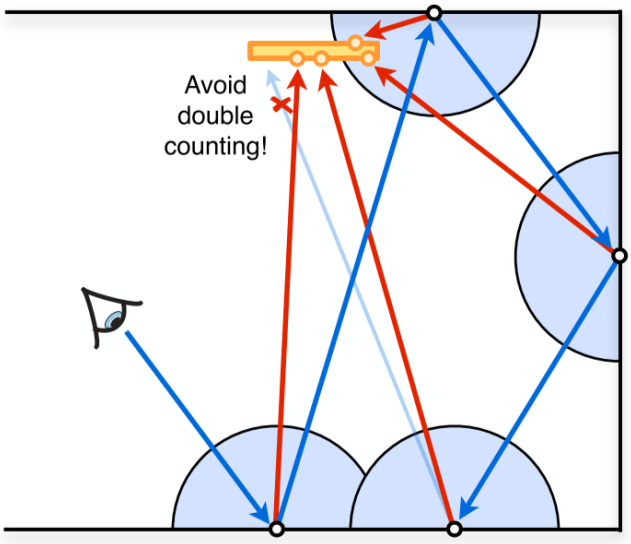
10.3 Path tracing

To remedy the previously mentioned issue, we can use **path tracing**, where we don't shoot n rays at each hit location to get the recursive values, but only 1 ray. This means we estimate the integral with an one -sample estimator:

$$L(x, \overrightarrow{\omega}) \approx L_e(x, \overrightarrow{\omega}) + \frac{f_r(x, \overrightarrow{\omega}', \overrightarrow{\omega}) L(r(x, \overrightarrow{\omega}'), -\overrightarrow{\omega}') cos\theta'}{p(\overrightarrow{\omega}')}$$

We can also include the shadow rays as before, to sample the direct illumination (right image below).





To improve quality of the indirect illumination estimation, we then can shoot multiple rays per pixel. To stop recursion, we use **russian roulette**, where we chose some termination probability $q \in (0, 1)$ and a random number ξ. We terminate is $\xi < q$, and otherwise compute a new probability based on the value of ξ (such that the probability decreases for increasing path length).

10.4 Light tracing

For **light tracing**, we shoot multiple paths from light sources and search for the **importance** (which is "emmited" from sensors and describes the *response of the sensor* to radiance traveling within a differential beam). The name can be a bit misleading, it refers to what we follow (trace) & where we start, not what we search for. This works especially well for caustics. However, e.g. glass spheres appear black with this approach, and therefore we wish to combine path tracing and light tracing to get the best of both.

10.5 Photon mapping

Photon mapping is a two-pass algorithm:

- Pass 1: Tracing of photons from light sources and caching the photons in a **photon map**
- Pass 2: Tracing from the eye and approximating indirect illumination using the previously gathered photons

In the first pass, we shoot out photons from the light sources, that carry power (flux), which is a fraction of the light sources power. When a photon hits a diffuse surface, we store it in the photon map. To end the recursion, we again use russian roulette to terminate the recursion. The photons keep their flux if they are allowed to continue, but only a certain number of photons are allowed to continue. E.g. if we have 300 photons with power 1.0 W, which hit a surface with reflectance of 50%, we will have 150 photons continuing with 1.0 W.

In the second pass, for each shading point we hit during the path tracing, we find the closest k photons and approximate the radiance using the density of the photons (by using the area that these k photons were found in. If we have a larger area, the photons are less dense and the indirect lighting is less intense).

10.5.1 Progressive photon mapping

To improve the quality of images rendered with photon mapping, we can use **progressive photon mapping**. Here, we repeat the previously mentioned two passes, however in each iteration we use a radius that is smaller than the radius from before to gather photons from (please note that for this, we need to use the variant where we have a fixed radius to gather photons from, and not the variant where we gather a fixed number k of photons).

11 Participating media

When modeling a **participating medium**, we usually do not model the individual particles of the medium explicitly, but describe the properties statistically using various coefficients and densities. This is a concept similar to that of the microfacet models, where we use a distribution to model the facets instead of explicitly modeling them in the surfaces.

We can have **homogeneous** media (with coefficients staying the same through the medium) which are either infinite or bounded by a surface, and we can have **heterogeneous** media (with spatially varying coefficients), which are often procedural generated.

The main quantity we're interested in is *radiance*. In contrast to previous models, where the radiance of a ray stayed constant, here the radiance changes along the ray if it goes through a participating medium, and as such we have:

$$L_i(x, \overrightarrow{\omega}) \neq L_o(r(x, \overrightarrow{\omega}), -\overrightarrow{\omega})$$

11.1 Transmittance

We can define the **expression of transmittance** between x and y as follows.

First, we have the scattering coefficient $\sigma_s(x)$ as well as the absorption coefficient $\sigma_a(x)$, which can be added together to get the **extinction coefficient** $\sigma_t(x) = \sigma_s(x) + \sigma_a(x)$.

Then, for a **homogeneous** volume, we have:

$$T_r(x, y) = e^{-\sigma_t ||x-y||}$$

For a **heterogeneous** volume (with spatially varying σ_t), we have:

$$T_r(x, y) = e^{-\int_0^{||x-y||} \sigma_t(t)dt}$$

12 Camera models

12.1 Depth of field

The **depth of field** is the distance between the nearest and the farthest objects that appear in acceptably sharp focus:

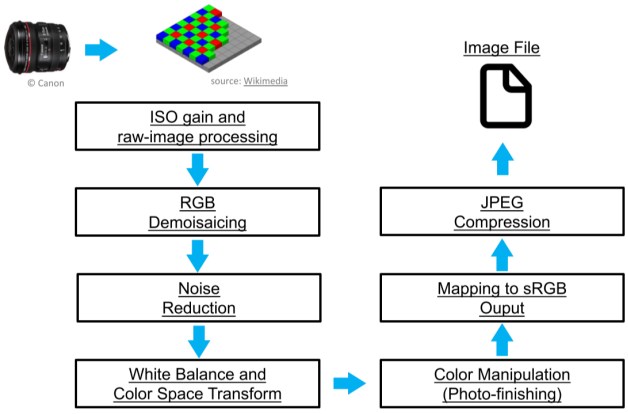
$$\text{DOF} \approx \frac{2u^2 Nc}{f^2}$$

where c is the given circle of confusion, f the focal length, N the F-number (aperture) and u the distance to subject.
The **circle of confusion** describes the optical spot on the sensor caused by a cone of light rays not coming to perfect focus when imaging a point source.

12.2 Chromatic aberration

Using lenses consisting of spherical surfaces introduces aberration, which means that different rays or wavelengths have different focus points. For chromatic aberration, we have the issue that the different wavelengths have other focus points, which means some wavelenghts will appear to be "closer" to the lens and as such appear smaller.

12.3 Typical color imaging pipeline



Please note that the objective of a camera is *to create good looking images*. Generally, there are many non-linear effects and unknown elements in the image pipeline, which is why in general, cameras *cannot be used as light-measuring devices*.

13 Multi-view geometry and 3D reconstruction

13.1 Pinhole camera

A **camera projection model** for a pinhole camera is a transformation using homogenous coordinates, mapping 3D points to coordinates in image space (2D coordinates):

$$p_{2D} = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} p_{11} & \cdots & p_{14} \\ \vdots & \ddots & \vdots \\ p_{31} & \cdots & p_{34} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = P_{3D}$$

The transformation can also be decomposed into intrinsic parameters (first part) and extrinsic parameters (second part). f denotes the focal length, c is the optical center (intersection of the optical axis with the image plane), r denotes rotation and t translation of the camera:

$$p_{2D} = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = P_{3D}$$

13.2 Camera calibration

To calibrate a camera, we usually use a calibration pattern (e.g. a checkerboard), from which we know the structure and size of. All the points on this pattern are on the same plane, and as such we can simplify the equations from above by setting $z = 0$. This also means that we can remove the corresponding column in the extrinsic matrix, from which we now obtain a simplified version:

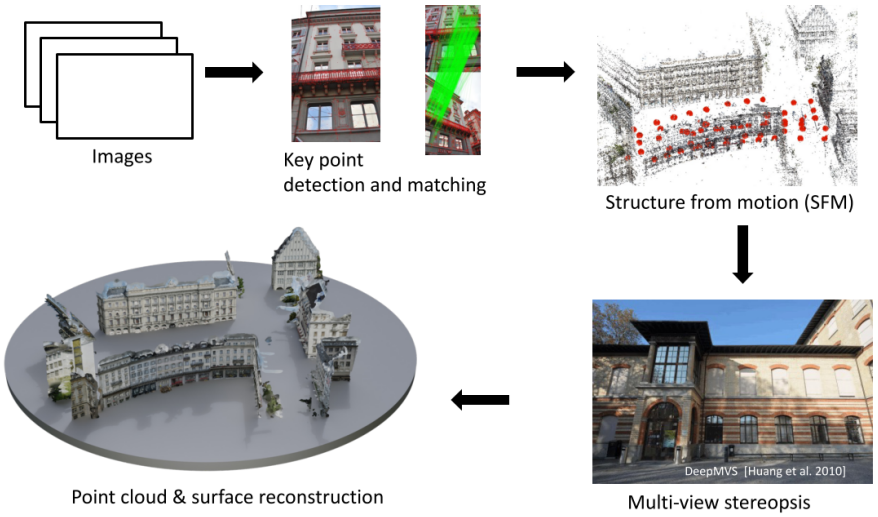
$$p_{2D} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

The product of the simplified intrinsic and extrinsic parameters is also called a **Homography**. This term can be used to relate two images on the same plane in space. For two points a_1 and a_2, we have the following relation:

$$a_2 \sim H a_1 \implies \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}$$

To be able to perform a camera calibration, we need at least 3 different views of the plane, and for each view we need at least 4 points to compute the homography.

13.3 3D reconstruction pipeline



In **keypoint detection**, we find points of interest in the image, which are at well-defined positions in image space, *stable* under local and global perturbations and can be detected efficient. Usually, we use points where the brightness changes significantly or edges of geometries in the images.

In the **structure from motion** (SFM) step, we want to be able to estimate the 3D structure from multiple images. The issue here is, that we might have outliers, which can negatively impact the quality of the reconstruction. We can use the **RANSAC** (Random Sample Consensus) algorithm to remove outliers. This is done by fitting some points to a model (e.g. 2 points to a line) and then solving which points are in these model parameters (e.g. which points are within a certain threshold to the line).

14 Image-based rendering / Light fields

Modeling complex scenes can be quite time-consuming, and as such we'd like to be able to capture real objects with computer vision, and display these models in 3D rendering. However, CV usually fails to capture every detail of models, and even if we manage to capture a perfect representation of an object, we still face the issue that the rendering is only an approximation of the real world, and as such also cannot render a perfect representation.

14.1 Textures from images

When we want to generate textures from images, we first need to reconstruct the 3D representation of the object, such that we can map a point on the image to the 3D representation, and then map this point to the UV coordinate of the texture. However, we have some issues, for example a point on the surface is obstructed in some images.

Basically, we can formulate this as a minimization problem, where we want to minimize the difference between the estimated color value and all the values obtained from the color images:

$$min \sum_i ||T(u,v) - T_i(u,v)||$$

where T is the estimated values and the T_i values are the values obtained from the color images.

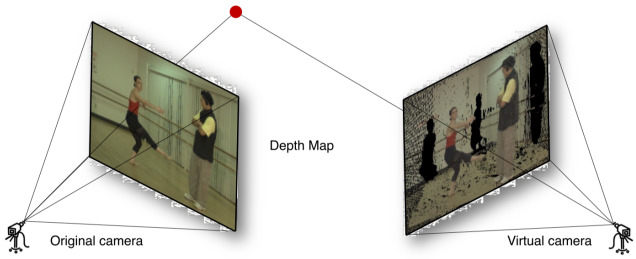
As mentioned before, we don't have the same values for all T_i values, for example we might have more information in the image from a certain viewing angle. For this, we introduce a general penalty λ, which takes into account the view-dependent effects. We also introduce a smoothing loss to the end of the term, such that we end up with:

$$min \sum_i \lambda||T(u,v) - T_i(u,v)|| + ||\nabla T||$$

14.2 Depth based rendering

14.2.1 Forward mapping

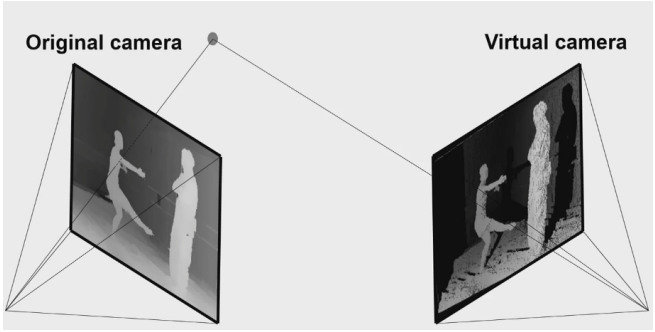
In **forward mapping**, we use the depth map of the original input image to project every pixel (color value) to a virtual scene, and we then can project every pixel in that virtual 3D scene to the new camera, which is located at a different position:



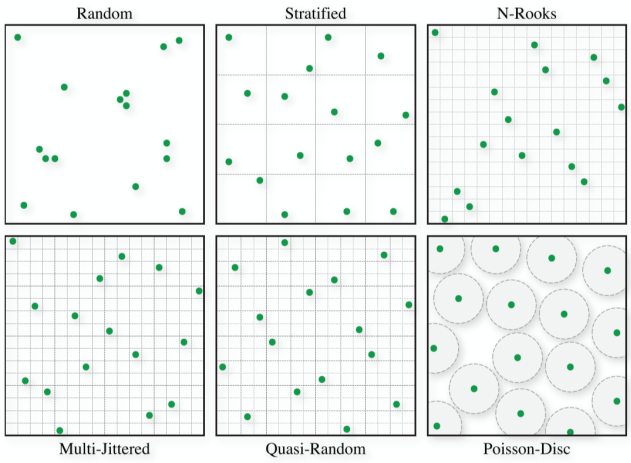
This however introduces some issues, as we have a non-surjective map, especially because of occlusions and perspective changes.

14.2.2 Backward mapping

Resolving the issues that arise in forward mapping would be possible if we had a depth map at the virtual camera. As such, we could backtrack every pixel and discretization would become a non-issue. We can do this by forward projecting the depth map (instead of the color map as in forward mapping), merging the depth map of several cameras and applying some filtering.
This gives us a depth map for the virtual camera, which we then can use to map the depth-point of the virtual camera to the color value of the original image.



15 Low-discrepancy point distributions



16 Denoising Monte Carlo renderings

Usually, images rendered using Monte Carlo techniques have some noise, especially when using a low sample count. Increasing the sample count usually improves the quality of the final image, but also increases the runtime by a lot. We therefore wish to be able to apply a denoising procedure to the images to remove such noise.

To do this, we want a **flexible** (arbitrary filter support) and **robust** (few denoising artifacts) filter. To do this, we need to be able to compute weights for the denoising.

16.1 Bilateral filter

The **bilateral filter** is not a robust filter, as it still leaves some noise after denoising. In general, we have a pixel p and want to compute the weight $w(p, q)$ of a neighbouring pixel q.

In the bilateral filter, we use a generalized distance d, which has a spatial (in pixels, exact) and range (difference in value u of the pixel, noisy) component:

$$d^2(p, q) = \frac{(u(p) - u(q))^2 - 2\sigma^2}{\epsilon + k^2 2\sigma^2}$$

The spatial component defines the range r of neighbouring pixels we want to include, pixels outside this range have weight 0. With this formula, weights of neighbouring pixels decrease exponentially with the distance to the pixel p. σ is the variance, which we remove from the numerator, as otherwise the term is biased by the variance (such that we get an unbiased difference of the values of both pixels). k^2 is an user-defined parameter, and ϵ is a very small number to avoid division by zero errors.

16.2 Non-local means filter

The **non-local means filter** is a generalization of the bilateral filter, where instead of computing the distance between two pixels p and q, we define a radius f around these pixels, and for each pair of pixels from the patches $P(p)$ and $P(q)$, we compute the distance. We then average this, to get the final result for distance between p and q. The computation of the distance between a pair of pixels is still done the same as in the bilateral filter (which basically is a special case of this filter with $f = 0$.

17 Material acquisition

17.1 Colorimetric calibration

To calibrate the colors of a camera, we take a picture of a color calibration checker, which has a number of colors printed on it, where we have a ground truth value for each of the colors (i.e. we know what each color should map to):

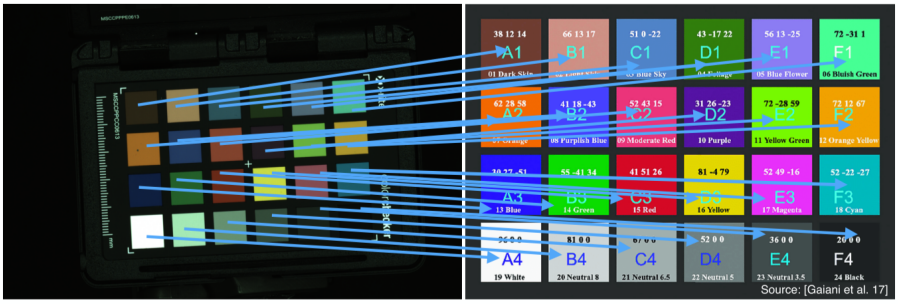


Photo (camera raw colour space) Target values

We then can build a linear system to estimate the function that maps the measured values (by the camera) to the ground truth values:

$$A_{24\times3} \cdot x_{3\times3} = b_{24\times3}$$

where A are the ground truth X, Y, Z values, and b are the measured R, G, B values. By solving for x. we obtain the function to map the measured values to the color space we have.

We can use this computed matrix by mapping the measured values to the X, Y, Z color space, by inverting

it: $C_{RBG'2XYZ} = x^{-1}$. Please note that this transformation maps the RBG values to the XYZ space, and not to a specific space such as rRGB for example. We do this because any RGB color space may be obtained by a simple (known) linear transformation, i.e. we can then just use these known transformations to map our XYZ values to any RGB space we want.

17.2 High dynamic range

When taking a picture of a scene, it might be overexposed (bright areas lack details as everything is white) or underexposed (dark areas lack details). To remedy this, we can use **high dynamic range** images, where we take multiple pictures of the scene with varying exposure time, and then fuse the images together to get the final exposure of the images.

18 Inverse rendering

Inverse rendering is the term given to the estimation of scene characteristics given a single photo or a set of photos of the scene.

In the scene characteristics, we usually know the **Transformations** and **Cameras** (positions) from geometric calibration (color calibration [17.1] and camera calibration [13.2]). What we only know partially are the **lights**, **materials** and especially the **geometry**.

In the **light transport equation** (see also 6.1), we leave out the term for the emitted light for now, which gives us the following term to work with:

$$L_o(x, \vec{\omega}_o, \lambda, t) = \int_{H^2} f_r(x, \vec{\omega}_i, \vec{\omega}_o, \lambda, t) L_i(x, \vec{\omega}_i \lambda, t)(\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

Please note that $(\vec{\omega}_i \cdot \vec{n})$ is equal to the $cos\theta$ term from before. λ is the wavelength of the light we're looking at, and t is time.

We have to make some assumptions about the light, which is called **active illumination**:

- The incident radiance is known and controllable (i.e. we can control the light sources)
- Assumptions about the incident radiance: $L_i(x, \vec{\omega}_i, \lambda, t) \rightarrow L_i(\vec{\omega}_i(t), \lambda_{RGB}(t))$
  - x is removed from term, i.e. the lighting does not depend on the spatial position
  - The incident direction and radiance of the lights are known
  - Indirect illumination is considered negligible

18.1 Photometric stereo

Photometric stereo is the term used for the process of getting geometry from photos. We have the following situation with out equation from before:

$$L_o(\mathbf{x}, \vec{\omega_o}, \lambda_{RGB}) = \int_{H^2} \underbrace{f_r(\mathbf{x}, \vec{\omega_i}, \vec{\omega_o}, \lambda_{RGB})}_{\text{Unknown}} \underbrace{L_i(\mathbf{x}, \vec{\omega_i}, \lambda_{RGB})(\vec{\omega_i}}_{\text{Known}} \cdot \vec{n}) d\vec{\omega_i}$$

As mentioned before, the L_i and $\vec{\omega}_i$ terms are know because we control the incident light. The BRDF and the normal $\vec{n}$ are unknown.

We want to vary the direction of the incident light L_i and from the measurements at a certain point to be able to calculate the surface normal $\vec{n}$ at that point.

In addition to the assumptions above, we need some more assumptions:
First, we want **delta-directional lighting** $L_i(x, \vec{\omega}_i, \lambda_{RGB} = L_i(\vec{\omega}_i) = \mathbb{1}_3$, i.e. the light is normalized with strength 1 at every place.
Second, we assume the surface has a **lambertian BRDF**, i.e. a perfect diffuse reflectance, i.e. the BRDF is constant (and only depends on the incoming light): $f_r(x, \vec{\omega}_i, \vec{\omega}_o, \lambda_{RGB} = f_r(\lambda_{RGB}) = \frac{\rho_{d,RBG}}{\pi}$

With these assumptions, the equation from above drastically simplifies to the product of the lambertian BRDF with the dot product of the normal and the light direction:

$$I = L_o(x, \overrightarrow{\omega}_o, \lambda_{RGB}) = \frac{\rho_{d,RBG}}{\pi}(\overrightarrow{n} \cdot \overrightarrow{\omega}_i)$$

As we assume that we don't have any indirect lighting or subsurface scattering, the value of every pixel is independent. Therefore we can solve the following linear system per channel and per pixel, with n being the number of light sources (i.e. "pictures taken with light at a different angle"):

$$A_{n\times 3} \cdot x_{3\times 1} = b_{n\times 1} \qquad \text{with } A = \begin{bmatrix} \omega_{i,1}^T \\ \vdots \\ \omega_{i,n}^T \end{bmatrix} \; b = \begin{bmatrix} I_{\lambda,1} \\ \vdots \\ I_{\lambda,n} \end{bmatrix} \; x = \frac{\rho_{d,\lambda}}{\pi} \cdot \overrightarrow{n}^T$$

i.e. A contains the directions of all the n light sources, b are the measured intensities for the pixel, and x is the normal, scaled by the color of the measured value.

To solve this, we need at least 3 lighting conditions (3 photos with different light), as the normal is made of 2 unknowns $n \to (\theta, \phi)$ and the third unknown is the scaling ρ_d.

From the $x = \frac{\rho_{d,\lambda}}{\pi} \cdot \overrightarrow{n}^T$ from before, we can extract the normal and the $\rho_{d,\lambda}$ as:

$$\rho_{d,\lambda} = ||x|| \qquad \overrightarrow{n} = \frac{x}{||x||}$$

18.2 Photometric stereo with arbitrary BRDFs

In the previous section, we made the assumption that the BRDF is a lambertian BRDF, i.e. we have a perfect diffuse reflection. However, this is usually not the case, and we have a non-linear equation, which does not have a closed-form solution.

18.2.1 Gradient descent

To solve such a system, we can use **gradient descent**, for example to use the *Blinn-Phong BRDF*. For this, we need to be able to calculate the derivative. We can do this manually (symbolic), which is possible, but it's tedious and error-prone. For more complex BRDFs, it gets really complicated and therefore is not really recommended.

Other approaches to compute the derivative are using **finite differences** or **automatic differentiation**.

Using **finite differences** is however quite error-prone, and suffers from numerical errors.

19 Acceleration Data Structures

Ray-surface intersection is at the core of every ray tracing algorithm. We can use the **brute force** approach, where we check the intersection of every ray with every primitive, however this leads to many unnecessary ray-surface intersection tests.

In such a naive approach, we have a cost of $O(N_x \cdot N_y \cdot N_o$, i.e. (number of pixels) $\cdot$ (number of objects), which assumes 1 ray per pixel. Typically, the runtime is measured per ray, i.e. we have a naive runtime of $O(N_o)$, i.e. it's linear with the number of objects. This is extremely inefficient, and leads to really long rendering times.

19.1 Acceleration techniques

We can either use **spatial subdivision** or **object subdivision** to accelerate the ray-primitive lookups. In the following sections, some will be described with a bit more detail.

19.2 Axis Aligned Bounding Box

An **axis aligned bounding box** (AABB) is a struct containing the minimal coordinate ("lower left corner") and the maximal coordinate ("upper right coordinate") of the bounding box of the object (aligned to the xy axis). To then compute the intersection, we can compute the intersection of "slabs", as outline in the following algorithm:

$$\mathbf{o}_x + t_{x1}\mathbf{d}_x = x_{min}$$
$$\mathbf{o}_x + t_{x2}\mathbf{d}_x = x_{max}$$

$\downarrow$

x slabs: solve for t_{x1}, t_{x2}

$$t_{x1} = \frac{x_{min} - \mathbf{o}_x}{\mathbf{d}_x}, t_{x2} = \frac{x_{max} - \mathbf{o}_x}{\mathbf{d}_x}$$

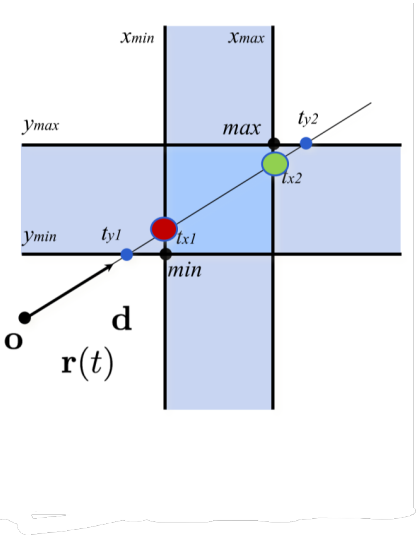
if $t_{x1} > t_{x2}$: swap(t_{x1}, t_{x2})

repeat for : $t_{y1}, t_{y2}, t_{z1}, t_{z2}$

$t_{min} = \max(t_{x1}, t_{y1}, t_{z1})$

$t_{max} = \min(t_{x2}, t_{y2}, t_{z2})$

hit if: $t_{min} < t_{max}$



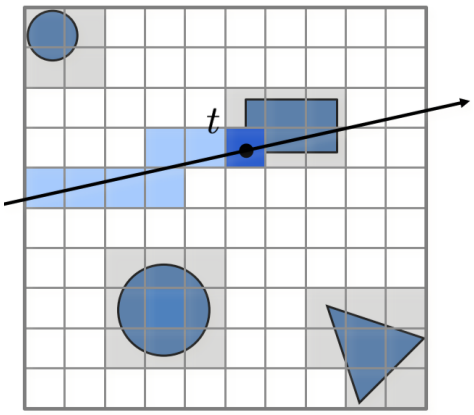
19.3 Spatial sorting

Using **spatial sorting**, we can reduce the number of cells we have to look at. First, we *preprocess* the geometry by decomposing the *space* into disjoint regions, storing pointers to overlapping objects within each region. During rendering, we traverse through the regions overlapping the ray, and intersect the objects in each region until a hit is found.

19.3.1 Uniform grids

In an uniform grid, we apply the following steps for preprocessing:

1. Compute the bounding box of the whole scene
2. Determine the resolution of the grid (usually $\sim 3\sqrt[3]{n}$)
3. insert the objects into the cells
4. Rasterize the bounding box of each object
5. Prune the empty cells (i.e. cells without any objects)
6. Store the reference for each object in the cell



The algorithm to traverse the uniform grid then is:

1. Incrementally rasterize the ray
2. Compute the intersection with objects in each cell
3. Stop when intersection in current voxel is found

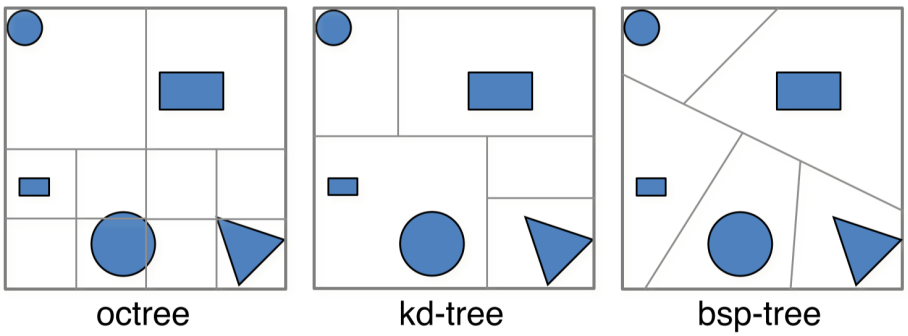
While uniform grids are easy to implement and building the data structure is fast, they do not adapt to non-uniform scenes (e.g. when we have one huge object and one small object). We can use **hierarchical grids** to resolve this, however there are better approaches which will be explained in the following sections.

19.4 Spatial hierarchies

Instead of using an uniform grid, we use **spatial hierarchies**, where we apply a classical divide-and-conquer approach. The most commonly used variations are the **octree**, the **kd-tree** and the **bsp-tree**:

19.4.1 KD-tree

In a **KD-tree**, we compute the bounding box, and then recursively split the cells using axis-aligned planes, until we hit a termination criteria (e.g. maximum depth or minimum number of objects). This results in a binary tree structure, where only the leaf nodes store references to geometry. The internal nodes store the split axis (x, y or z), the split position and references to the child nodes.



To traverse such a KD-tree, we can simply traverse the binary tree. If we're at an internal node, we're at a split and go to the correct child node, and if we're in a leaf node we can perform the intersection test on the items in the leaf node.

19.4.2 Octrees

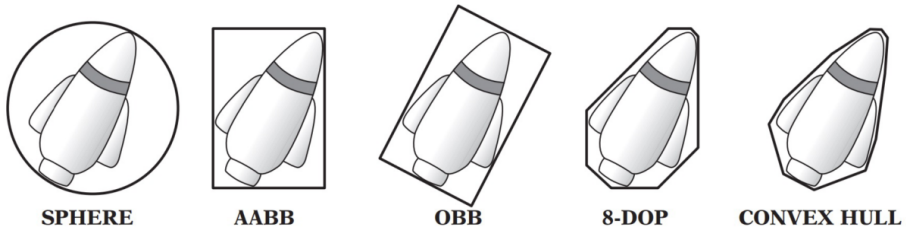
Similar to KD-trees, however for **octrees** we recursively divide the cells into 8 equal sub-cells (i.e. the split is always at the center of the cell, and as such we do not need to store the position of the split). Traversal is similar to KD-trees, however octrees are easier to implement and have cheaper costs for insertion and deletion. On the downside, they usually divide the space less efficient than KD-trees.

19.4.3 BSP-tree

Finally, in a **BSP-tree**, we split the space by arbitrary planes. This gives us the most flexibility abd we need to check for fewer intersections, but it's also more complex to implement than Octrees of KD-trees.

19.5 Bounding volume hierarchies

Instead of using spatial hierarchies, we can also use **bounding volume hierarchies** (BVH) around the objects using simple volumes, such that we can reject the intersection checks faster. We can choose from bounding volumes with many shapes, however it's always a trade off between simplicity of the shape and the "wasted space" inside of the volume (e.g. a sphere is a really simple shape, where the intersection check is easy, however it might waste a lot of space if the object inside of it is not spherical). An example of some possible shapes:



We then can build a hierarchical tree (BHV), which we can traverse to find intersections (e.g. a sphere around all objects in the scene, which then contains child spheres and so on).

20 Point based representations

So far, we've seen representation of geometry using either *triangles* or *polynomials*. However, both approaches have their drawbacks, which is why we want a different representation using **points**.

The main advantage is that we don't separate geometry and appearance (e.g. textures) and that it's a natural representation for many 3D acquisition systems.

20.1 Surface model

Surface modeling is the computation of a continuous surface from a discrete set of points P.

To compute such a surface, we can use the **moving least squares** (MLS) approximation. The surface is defined as a stationary set of projection operator $\Psi_P \Rightarrow$ implicit surface model:

$$S_P = \left\{ x \in R^3 | \Psi_P(x) = x \right\}$$

i.e. the set of all points which Ψ_P (MLS operator) projects onto themselves. Any point in "outer space" (not on the MLS surface) will be projected onto the surface.

To compute this, we use a weighted least squares optimization, using a Gaussian weighting function.

One downside of using this approach is that we lose details and the solution might degrade.

To remedy this, we can apply the **spherical MLS**, which fits a sphere into the points instead of fitting a plane into the points (like for normal MLS).