

David Peicho

Physically-Based Rendering: Real-Time Implementation

Slides available at <https://davidpeicho.github.io/teaching/>
Found an error? Please contact me at david.peicho@gmail.com

Course Layout



Image from the Pixar short film "Piper"

01 **3H - Theory + Lab**

02 **3H - Theory + Lab**

03 **3H - Lab**

04 **3H - Lab**



Disclaimer

The rest of the course will assume that...



Light travels in vacuum



We deal only with
opaque surfaces



Interactions occur at
object surface



This course is based on several assumptions (listed above). Those assumptions will allow us to simplify computation and speed up rendering.



Before We Start

Don't get confused please!

There are a lot of ways to generate an image (**raytracing**, **rasterization**, etc...)

For each technique, the theory is similar but implementation differs

Throughout this course, we will focus on **real-time** with a rasterization pipeline

One thing to always remember in **real-time**: *fake it until you make it*



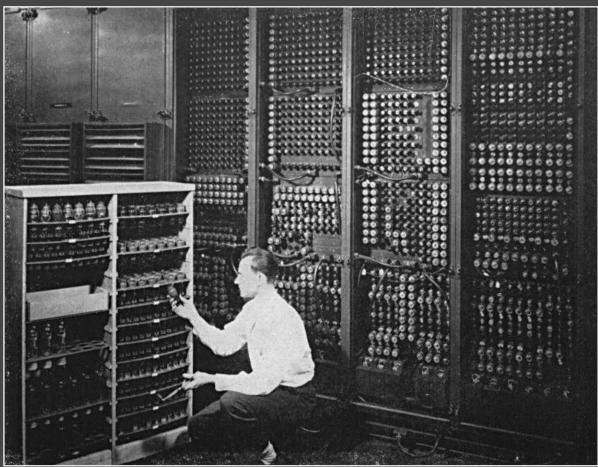
Before diving in this course, I want to mention a few things. Our main goal is to generate an image, it doesn't matter if you use a CPU, a GPU, or both. It doesn't matter if you use **raytracing** or a **rasterization** pipeline. All those things are just a means to an end.

However, because a lot of students are often more interested in video games, we will focus on the technology behind video games, i.e., real time rendering with a **rasterization** pipeline.

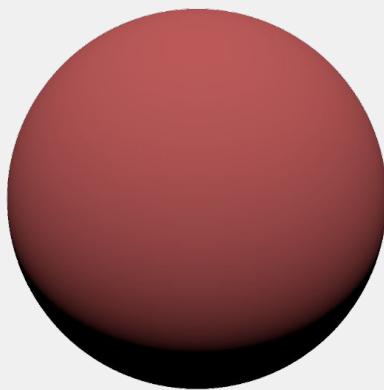
Rendering engines are crazy complex nowadays, and they even start to mix raytracing and rasterization for real-time rendering. However for the purpose of this course, we will stick to a simple OpenGL (WebGL) rasterization pipeline.

- 01 The (Good) Old Times
- 02 What? Why?
- 03 Microfacets Theory
- 04 Dielectrics vs Conductors
- 05 BRDF
- 06 Ponctual Lights
- 07 Image Based Lighting
- 08 Colorspace
- 09 Going Further
- 10 References





Old Times

Sphere rendered using a **Lambertian** BRDF

Lambert

I am sure you have heard about the **Lambertian model**

Purely diffuse, uniform scattering

Physically-based when not using **magic constant**
Magic constant would make the BRDF **non-conservative**

Unfortunately not exactly *realistic*
But still **good** and **sufficient**
... and **used!**

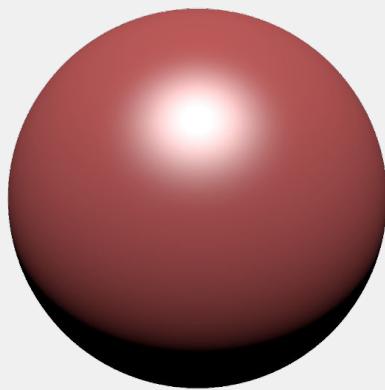


I am sure you have all used a **Lambertian model**, or at least something similar that was applying a constant to diffuse lighting.

The **Lambertian model** describes a perfectly diffuse surface that would reflect light uniformly in every directions.

If you apply it incorrectly, for instance using a magic constant, you might end up with a non-energy conservative **BRDF**, which means that the system is imbalanced and more energy is introduced. In this case, this BRDF would make the model non **physically based**, and less realistic, apart from some good tweaking..

The lambertian model isn't really plausible because no material is purely diffuse. However, it performs quite well for materials exhibiting a strong diffuse component, and is still used nowadays especially in real time rendering for its **simplicity / efficiency**.



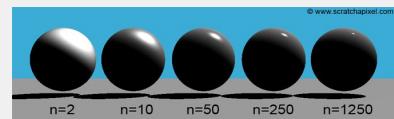
Sphere rendered using a Phong BRDF

Phong

Empirical model, from the 70s

Hard approximation of rendering equation

Not energy conservative



Big achievement! Drove the computer graphics field even further!



I am sure you all have worked on a Phong model implementation at some point as well.

The Phong (or the improved Blinn-Phong) model splits the lighting into two distinct components: **diffuse** and **specular**.

While easy to implement, this one doesn't respect the energy conservation rule. As you can see on the right image, it's possible to generate more radiance than the material received.

It's been used for years by a lot of applications, and is still used nowadays. The Phong model isn't *bad per se*, it just needs to constantly be manually **tweaked**.

Pseudocode

```
Lambert
void main()
{
    vec3 diffuse = kD * dot(normal, lightDirection) * color;
    gl_FragColor.rgb = vec4(diffuse, 1.0);
}

Phong
void main()
{
    vec3 r = reflect(- viewDirection, normal);
    vec3 diffuse = kD * dot(normal, lightDirection) * color;
    vec3 specular = kS * pow(max(dot(lightDirection, r)), exponent);
    gl_FragColor.rgb = vec4(diffuse + specular, 1.0);
}
```



Here I added some code that might look like what you have been previously written in other courses (or for fun!). The code is simple but leads to nice result with really few processor ops.

What & Why

Introduction

The need for Physically-Based Rendering

Non-physical models require a lot of **tweaking**, on the light side, on the material side, etc....

No standard was out there to also help **sharing content**.

But finally, **Physically-Based Rendering (PBR)** became the industry standard

Those issues aren't completely fixed, but the industry is in a much better place!



Before we introduce what PBR is, let's see the issues there were with the previous computer graphics techniques:

- Non-physical model requires tweaking. It's not easy for instance to make a day / night scene by simply changing the light color and intensity. Artists were used to constantly tweak and hack the rendering options to get the most appropriate results.
- It wasn't easy at all to share content. Some people were using different kind of inputs to feed the rendering equation. In addition, there was no exact standard for what equations were used.

Obviously, this has a high cost in an industry. A lot of redundant work was done, and sharing was a pain. Fortunately, **Physically-based Rendering (PBR)** became a thing. Even if those issues might not be completely fixed, PBR is and has been a big revolution in the computer graphics industry.

Introduction

What is Physically-Based Rendering (PBR)

Standard set of **mathematical models**, and **approximations** used to describe interactions between light and matter in graphics.

In addition, those standard models provide a common ground that simplifies how to feed the rendering equation, **i.e.**, that simplifies how to drive material appearance

PBR also helps standardize inputs to the equation: textures, light units, etc...





Image from the [Marmoset](#)

Why is it Popular?

More accurately represents the world

Physical values for light and material properties

Ensures **consistency**

Less (or no) manual tweaking

Big win for **engineers** and **artists**

Let's dive into **PBR!**



Physically-Based Rendering emerged because of all the advantages it brings.

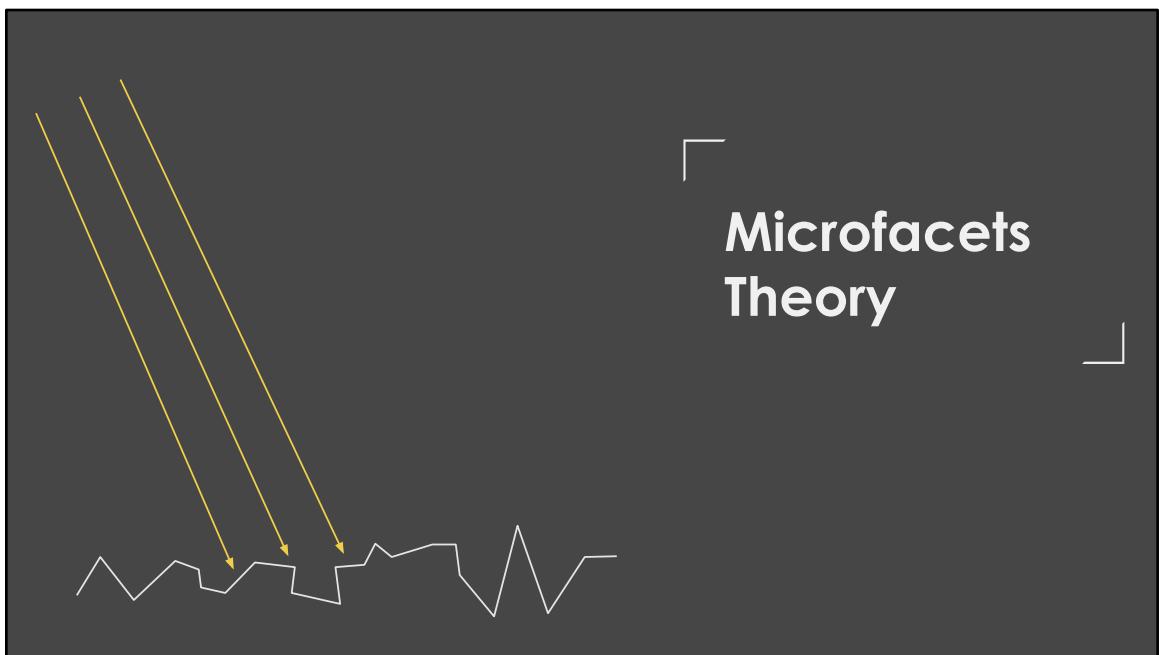
Expressing light with physical quantities (lumens, candelas, ...) has two advantages:

- We can setup scene realistically. We know the temperature of the sun in / out the atmosphere, etc...
- No need to modify the materials in a scene where the lighting would drastically change

All those advantages bring **consistency**. Consistency and less manual tweaking improve efficiency and thus reduce the money spent in the process 😊

Throughout this course, we will see how the PBR standard separates materials into two classes: **dielectrics / conductors**.

This separation allows to design material in a super friendly way, easier to setup and more intuitive for artists. In addition, we will see that the PBR equations are parametrized with other inputs that will help create different materials using the exact same equation.



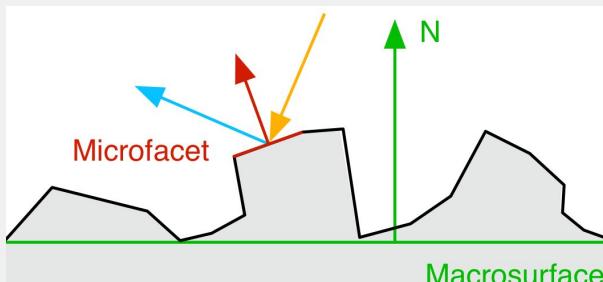


Image from the [filament PBR guide](#)

Microfacet Theory

Everything is about models and approximations

Geometry-optic based approach

Materials assumed to be made of **facets** at the **microscopic level**

Parametrized by **roughness**

How smooth / rough a surface is

Introduced in graphics by "**A reflectance model for computer graphics**" [Cook82]

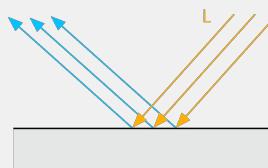
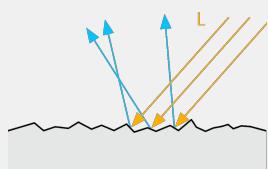


Our goal is to render something that looks close enough to reality. This is where the **Microfacet Theory** comes in.

the **Microfacet Theory** describes material by approximating reality using facets. Those models assumed that the material surface is made out of facets at a microscopic level. Every facets can be thought as as a perfect mirror.

Microfacets models are often parametrized by a few inputs, that describe the statistical orientation of those facets. They allow to represent surfaces ranging from perfectly **smooth** (basically mirrors), to **rough** surfaces (behaving like completely diffuse surfaces).

As far as I know, the theory has been introduced in graphics by: [\[Cook82\]](#). The idea goes back even beyond to [\[Torrance67\]](#).

Smooth surface, i.e. mostly **specular**Rough surface, i.e. mostly **diffuse**
Image from the [Filament PBR guide](#)

Microfacet Theory

Smooth surfaces behave like mirrors

Rough surfaces scatter light chaotically into **diffuse**

At macro level, surfaces considered **flat**

Not every facet is **represented**

Statistical view describes facets orientation

PBR is often based on **Microfacets Model**

Though not mandatory



The drawing above should give you an intuition about the model. A perfect mirror can be seen as a surface containing facets oriented in the same direction as the macrosurface normal. In this case, incident light would be reflected in an ideal specular lobe.

On the other side, a perfect diffuse surface can be seen as containing “chaotically” oriented facets. In this case, incident light is uniformly distributed in the hemisphere around the normal.

The microfacet model might not be good for every materials. However, it can represents a fair range of different materials and this is why it became the industry standard.

We are obviously not going to represent every materials with each of its facet. We are going to use a macroscopic statistical view, where the facets orientation will be describe by a simple value: the **roughness**. The roughness will be a probability of a material surface to be rough. **1** meaning very rough, and **0** very smooth / flat.



Image from [UE4](#)

Dielectrics vs Conductors

PBR; Real Time | Dielectrics / Conductors

18 | 10/2022

Dielectrics vs Conductors

Recall diffuse is the result of complexe **Subsurface Scattering**

Conductors **absorb** quickly **refracted light!**

Conductors as high as **60-90%** of reflectivity

Dielectrics often have **0-20%** of reflectivity
More often around **4%**

Some conductors are tainted due to the **wavelength** range they absorb

Using this knowledge, we can simplify our model by separating **insulant** from **conductors**

◀ | ▶

Image from the [Filament PBR guide](#)

One thing we haven't talked about and you will see everywhere online. PBR materials are often categorized between **dielectrics** and **conductors**.

Conductors absorb quickly refracted lights. It means that the visible light on a conductor is simply what's reflected!

Refracted light gets absorbed and isn't re-transmitted. This occurs because most of the absorption occurs in the first layers of atoms in the lattice. However if you recall, absorption also leads to emission (electrons will decay to a lower energy level) and so scattering occurs, meaning reflection here! This is counter intuitive, but it's true!

There is a catch to that: some metal however absorbs at specific **wavelength**, it's for instance the case of gold. Because it absorbs only some wavelength, the material appears tainted differently than the incoming light.

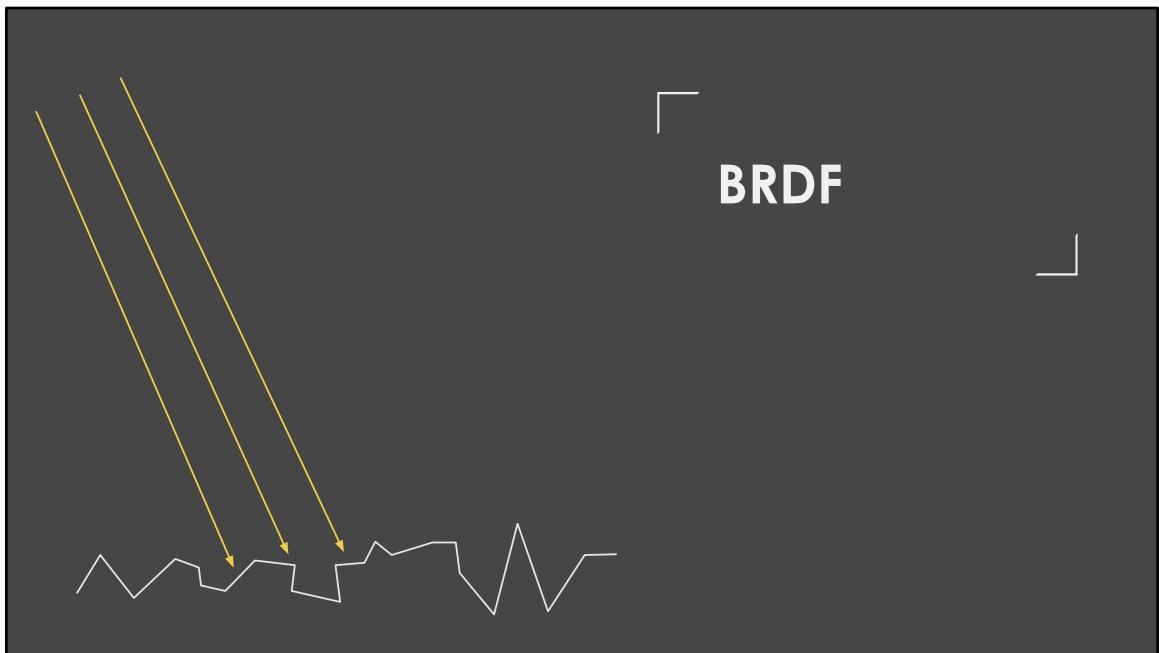
Using this knowledge, we can make our model take into account those differences. This will help us on two levels:

- Simplify artists life. They can simply say whether a material is a conductor / insulator, without the need

- to specify more physical data (absorption rate, etc...)
- Makes it easier to design materials quickly

It's possible to implement a PBR renderer without a **metallic workflow**. Actually, several engines and frameworks have a specular workflow that doesn't use any information about conductors.

Metallic will be the first input to our rendering function. We will use the metallic information to compute the reflectivity of our materials to deduce their base color.



Throughout this course, we will try to understand how to implement a microfacet model for **real time purposes**.

Implementing a microfacet model in an offline renderer works similarly. However, real-time constraints force us to either perform pre-computation ahead of rendering, or to approximate our equations more coarsely.

Before Starting

Remember we said that a physical **BRDF** needs to respect 2 conditions.

Physically based rendering models must use **physical BRDF**, and so:

$$\int_{\Omega} f(p, \omega_o, \omega_i) = 1$$
$$\int_{\Omega} f(p, \omega_o, \omega_i) = \int_{\Omega} f(p, \omega_i, \omega_o)$$



We haven't really talked about what makes a model **Physically-based** or not. I don't think there is an easy answer, and I feel what I am going to tell you could be opinionated?

Some people describes Physically-Based Rendering as based on microfacets **BRDF**. Other people describes PBR as any rendering method that follows the Rendering Equation more accurately. To me, I would say that PBR is about:

- Having an energy conservative BRDF
- Having inputs that are somehow based on physical inputs

This slide also gives a quick reminder about what we mean by physical **BRDF**: it should be energy conversative and symmetrical.

BRDF Simplification

$$f_r(p, \omega_o, \omega_i) = f_d(p, \omega_o, \omega_i) + f_s(p, \omega_o, \omega_i)$$

[\[Shofer 84\]](#)

We already said that in this course we care about

Interactions at object's surface

Diffuse as an approximation of **Subsurface Scattering**

In addition: Specular can be thought as an approximation of **scattering on first layer**

We can then work with BRDF splitting **diffuse** and **specular** components

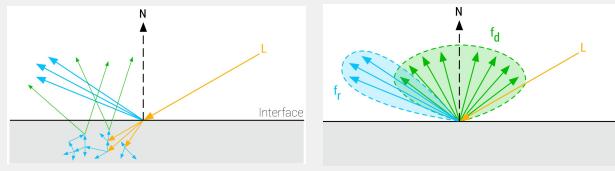


Image from the [Fragment PBR guide](#)



Remember that this course only cares about:

- Light propagating in vacuum
- Light-matter interactions **at the object surface**

This assumptions allow us to simplify the equation complexity and to take shortcuts. We will approximate **Subsurface Scattering** as diffuse. Light scattering inside the material and reaching the object surface as different points can be approximated as a diffuse component. Instead of simulating the interactions inside the material, we only care about light getting uniformly distributed in the hemisphere around the normal.

This is not 100% **physically accurate**. However, remember that everything is about trade-off. Using such an approximation will lead to really good results for a **large range of materials**.

This is what you have been doing intuitively until now: splitting your computations in two parts: computing the **diffuse** and **specular** components.

$$f_r(p, \omega_o, \omega_i) = k_d f_d(p, \omega_o, \omega_i) + k_s f_s(p, \omega_o, \omega_i)$$

$$k_d + k_s \leq 1$$

Ensures energy conservation

Implementation Notes

Diffuse and specular weighted **proportionally**

Rule of energy conservation

Obtained using ratio of **reflected** vs **transmitted** light

BRDFs changes between implementation

Diffuse and specular components can be changed independently

Always think in terms of Trade-Off:

Complexity / Simplicity / Performance

Real-time vs Offline

Trade-off quality vs Speed



We will use the reflectivity of the material to ensure **energy conservation**. The k_d and k_s terms will be used to weight both the diffuse and specular components. If we know the reflectivity of the material (k_s term), we can then find the k_d term and weight the diffuse lobe accordingly.

Before diving in concrete equations for the diffuse / specular parts of the BRDFs, I want to clarify something. Online, you will find a **lot** of different names for microfacets **BRDFs** (GGX, Oren-Nayar, etc...). Each of those **BRDFs** are made for either the diffuse or the specular components.

Do not get confused: **BRDFs** are plug and play. You can replace any of the two terms by any equation you like, as long as you ensure the few conditions are still met!

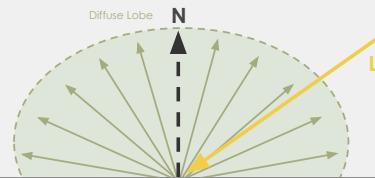
Real-time rendering for instance has strong time constraints. Because of that, real-time implementations will often use faster to compute **BRDFs** that leads to less accurate results.

$$f_d(p, \omega_o, \omega_i) = \frac{\rho}{\pi}$$

Reflectance spectrum, i.e. **Albedo**

Diffuse Lobe

Lambert, Oren-Nayar, etc...

For simplicity, let's stick to the **Lambertian model!**

Lambert: uniform reflectance

The **albedo** is the base color of the material, i.e., the **spectrum**
It's an input to our shader: **uniform color** or **texture**

Unreal Engine 4 is an example of the Lambertian model

There are multiple famous diffuse BRDF we could use: Lambert, Oren-Nayar, etc...
We will stick to Lambert for now!

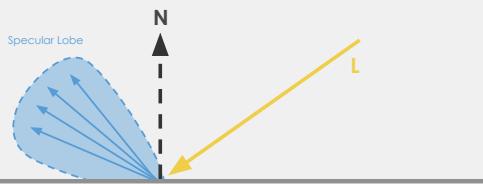
Albedo is the true color of your object. It's the color of the light leaving the material once it's been absorbed and re-emitted.

You may already have worked on some material on which you applied a texture to get more color information. The albedo is basically the base color of the object. Just remember that it shouldn't contain **any extra** lighting information, i.e., no ambient occlusion, no shadowing, etc...

Albedo will be passed as an input to our shader. As any input, it can either be a constant, or fed via a texture.

Normal Distribution Function Fresnel Function Geometric Function

$$f_s(p, \omega_o, \omega_i) = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$



Specular Lobe

Cook-Torrance GGX

Each function approximates a specific reflective effect

*D, F, and G must be **normalize***

Energy conservation again

Unreal Engine 4 is an example of the GGX modelTo be consistent, let's implement the **Cook-Torrance GGX model**

One of the most used specular BRDF is the Cook-Torrance. It's made out of three swappable terms.

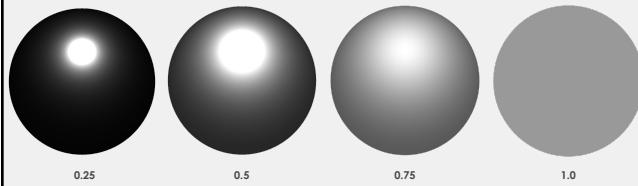
Just remember: the *D*, *F*, and *G* functions **must be** normalized as well. No energy should be created by those functions.

We will talk about the Cook-Torrance BRDF for the rest of this course and for the assignment.

$$D_{GGX}(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2}$$

$$h = \frac{\omega_i + \omega_o}{\|\omega_i + \omega_o\|} \quad \xrightarrow{\text{Orange}} \text{Halfway vector}$$

$$\alpha \quad \xrightarrow{\text{Green}} \text{Roughness}$$



Specular BRDF:

Normal Distribution Function $D(\omega_o, \omega_i)$.

Estimates the area of microfacets aligned to give perfect specular

As usual, lots of different NDF equations...

To be consistent, let's implement the **Trowbridge-Roach** equation

Low roughness means few samples contributing a lot to specular



The **Normal Distribution term** computes “*how much*” of the microfacets are aligned to the normal, maximizing specularity when the viewing angle is a reflection of the incoming light direction.

Low roughness will make the equation concentrate all the energy in a small spot, while high roughness will diffuse the light over the surface. The principle of energy conservation is easily visible here.

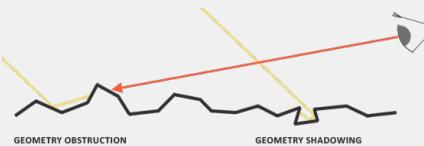


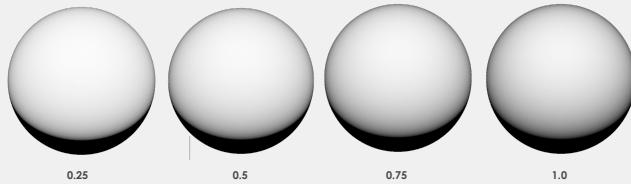
Image from the [Joey De Vries](#)

$$G(n, v, l, k) = G_{SchlickGGX}(n, v, k)G_{SchlickGGX}(n, l, k)$$

Obstruction

Shadowing

$$G_{SchlickGGX}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k}$$



Specular Lobe:

Shadowing Term $G(\omega_o, \omega_i)$

Approximates **occlusion**

Orientation of facets might **trap** light

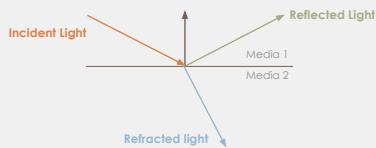
Shadowing in geometry vs shadowing towards camera

Equation based on [Smith67]



The **Shadowing term** computes the probability of light rays to be occluded. There are two types of “*occlusion*”, the light can either be trapped and bounce in the geometry, or the visibility can be “*masked*”.

Those two occlusion form can be represented using the Smith masking function. The smith function gives a normalized value with **0** meaning that maximum shadowing occurs.



Specular Lobe:
Fresnel 1/3

Describes light's behaviour at the material □ air interface

Computes how much light is **reflected** vs **refracted**

Light follows the [Fermat Principle](#)

Remember: scattering due to re-emission of radiation

Energy conversation between the two

Take into account viewing angle and IOR

Developed by **Augustin Fresnel** (1788-1827)

Fresnel term is in fact the **weight** of the **specular lobe**: k_s



The **Fresnel effect** is the last piece of the equation. It describes how light reacts at a plane between two medium.

The fresnel effect is visible almost everywhere around ourselves, we just don't pay attention to it anymore but our brain knows it exist!

Imagine you are sitting on the beach and look at the sunset like the image shown here. You would see the reflection of the sky clearly on the water. However, if you go in the ocean and look straight down into the water, you wouldn't see anything (except the sand maybe!).

The Fresnel equation exactly describes this effect: looking at an object at grazing angles gives “*maximum*” specular reflectance.

But why is that?

Light always travel to the fastest path (**Fermat Principle**, or **Principle of Least Time**). Basically, the path that has the most constructive interferences (remember that light is an electromagnetic radiation).

When it reaches the interface between two mediums, light thus undergo a change of

direction (scattering). It's possible to compute how much of the light is reflected / refracted using the Fresnel Equation.

We have been talking a lot about energy conservation. The amount of energy **must remain constant** between what's reflected and what's refracted. Thus, we end up seeing the specular at grazing angle (i.e., the reflected light), and less of the diffuse (i.e., the transmitted light).

The Fresnel term is in fact the weight of the specular lobe. It gives us how much light is reflected, and we will be able to use it to deduce how much of the diffuse lobe should be applied.

$$F_{shlick}(v, h, f_0, f_{90}) = f_0 + (f_{90} - f_0)(1 - v \cdot h)^5$$

$$F_{shlick}(v, h, f_0) = f_0 + (1 - f_0)(1 - v \cdot h)^5$$

$$F_0(ior) = \frac{(ior - 1)^2}{(ior + 1)^2}$$



Sphere example with grazing reflectivity

Specular Lobe:
Fresnel 2/3

Need to solve the **Fresnel equation**

For real time, we use the Shlick's approximation [Shlick94]

f0 is the base reflectivity at normal incidence
Calculated using material's IOR

f90 is the base reflectivity at grazing angle
Almost always 1 for **dielectrics** and **conductors**

f0 requires different equations for **conductors** and **dielectrics**



The fresnel effect is normally computed using the Fresnel equation.

However, one of the most used implementation is the Shlick's approximation. It allows to calculate with a few operations the reflected light.

The **f0** parameter is the base reflectivity, which is the ratio of reflected light at **normal incidence** (0 degrees), i.e. when looking straight at the normal of the surface. At the opposite, **f90** is the base reflectivity of the material at **90 degrees**, i.e: at grazing angles.

f0 is computed per material using the equation (3). Unfortunately, the function using IOR can't be used to compute the **f0** term for conductor materials. In order to avoid having a special path in the code for dielectric and one for material, it's common to use pre-computed values for **f0** that can then just be used with the Shlick's approximation.

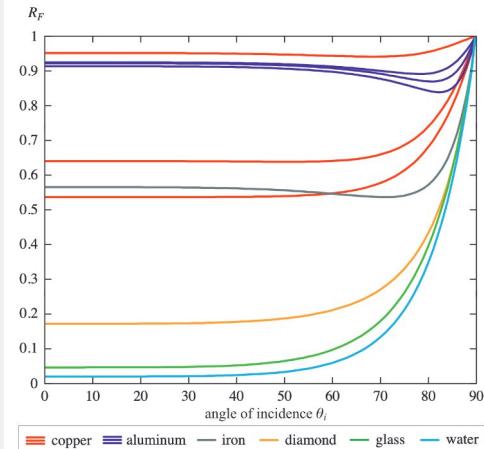


Image from "Real-Time Rendering, 3rd Edition"

Specular Lobe:
Fresnel 3/3

Fresnel **reflectance** for common materials

For dielectrics, **f0** is often approximated with **0.04**

Some materials **f0** are tinted (gold, copper)

Implementation note:

For dielectrics, pick **0.04 f0**

For conductors, store **f0** in albedo texture

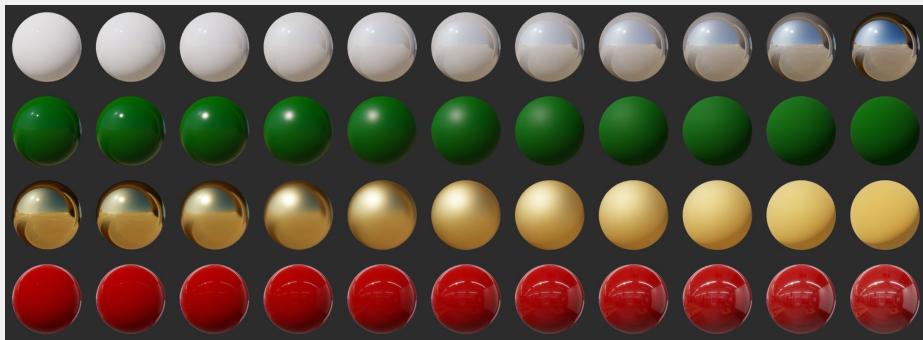
Use **metallic** input to lerp between the two



When thinking about the specular component, it should only be tinted based on the light spectrum. Specular is indeed a reflection with no subsurface scattering, the light color should then be unaltered.

However, some metals are tinted and we have said that no subsurface scattering occurs in conductors! There must be something wrong somewhere. It turns out some metal have low reflectivity but only for short wavelength, that's the case for instance for gold and copper.

On the above drawing, you can see how the Fresnel reflectance changes with the viewing angle for different materials. You may be wondering why copper and aluminum are splitted into 3 curves? Copper and aluminum are metal that have different reflectivity for different wavelength! Thus, those metals will have different tint and it will even change based on the viewing direction.

Parameters DemoImage from the [Filament PBR guide](#).

Direct-Lighting Pseudocode

```
vec3 irradiance = vec3(0.0);
for(int i = 0; i < NB_LIGHTS; ++i)
{
    vec3 w_i = lights[i].direction;

    vec3 kS = FresnelShlick(f0, wi, wo);
    vec3 specularBRDFFeval = kS * f_s(p, w_i, wo);
    vec3 diffuseBRDFFeval = (1.0 - kS) * f_d(p, w_i, wo);
    diffuseBRDFFeval *= (1.0 - metallic);

    irradiance += (diffuseBRDFFeval + specularBRDFFeval) * sampleLight(lights[i], p, w_i) * dot(normal, w_i);
}
```



Material Parameters

Material can be fed using:

Global uniforms

Textures

Textures allow per-fragment changes

Textures aren't only for **albedo**!

Albedo, roughness, metalness, AO, etc...

Packed together: **roughness** in red, **metalness** in green, etc...

Production-level models have detailed textures



One thing we haven't talked about: how are the inputs fed per material?

For simplicity, you can feed your shader with uniform values for **roughness**, **metalness**, and **albedo**. In order to get more complex rendering and to be able to represent a broader range of materials, you will need to create some changes on a per-fragment basis. The best way to do that is to use **textures**.

Some inputs are scalar, that's the case of the roughness and metalness. Because of that, it's common in rendering engines to ask for textures where several inputs are packed on the same texture. For instance, you could create a texture where the **red** channel contains the **roughness**, and the **green** channel the **metalness**.

Textures

Shows material variations based on different texture inputs



Roughness = **0.0**



Roughness = **0.5**



Roughness = **1.0**



Metalness = **0.0**



Metalness = **0.5**



Metalness = **1.0**



High-Quality Model

Example



◀ | ▶

To Remember!

Diffuse is an approximation of **Subsurface Scattering**, visible for dielectric materials

(Most) **Conductors** absorb all the refracted light

PBR implementations often (always?!) make use of this distinction

Simplify artist workflow and simplify the process of creating meaningful materials





Direct Lighting

We talked about BRDF, material parameters, and we even have seen some pseudo-code showing how to apply this BRDF. We are still missing a big piece of the equation: lights.

Point Light

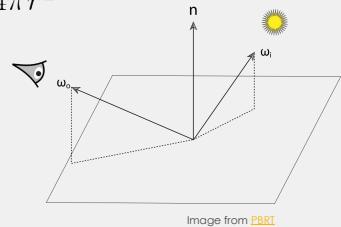
1 / 2

Infinitely small

Isotropic

Describe only by a **position****Simple** to code and **fast** to sampleNot perfectly accurate, but worth the **speed / simplicity****Units** and **photometry** are important, but not addressed

$$L_i(p, \omega_i) = \frac{\phi}{4\pi r^2} n \cdot \omega_i$$



◀ | ▶

A point light can be seen as an infinitesimally small point emitting light in a sphere. The light is isotropic and the irradiance at point from this light falls with a square law.

As you can see from the equation, it should be really easy to implement. Even though point lights are approximations and don't exist in the real world, it's worth implementing them for the simplicity and quick computation.

To get really good punctual light, they should be setup using proper units and not magic constant. For the sake of simplicity, this course will unfortunately not cover that. However, you are advised to have a look at the amazing [Moving Frosbite to PBR](#) that goes into really great details about that!

Point Light

2 / 2

Power unit should be set using **Lumens**

How to select a proper value?

Not as accurate as **Area Light**

$$L_i(p, \omega_i) = \frac{\phi}{4\pi r^2} n \cdot \omega_i$$

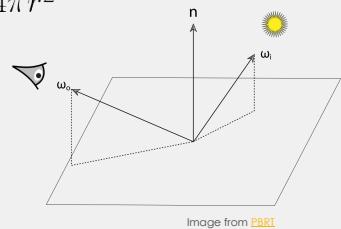


Image from [PBR](#)



Something we haven't talked about at all: unit for lights. You all have seen on light bulbs

Lumens, Lux, Candella. Unfortunately, we haven't studied at all **Photometry**, which is the equivalent

of **Radiometry** but tailored to the human visual system. There exists conversion between Radiometric <> Photometric quantities.

For the purpose of this introduction to PBR, just assume that you feed **Lumens** to your point lights.

For curious readers, the publication [Moving Frosbite to PBR](#) is a must to see how to deal better with light units in a PBR renderer.

Sun / Directional Light

Far away point light approximated as a **direction**

$$L_i(p, \omega_i) = \phi(n \cdot \omega_i)$$

Crude approximation: ignore **falloff**

Units and **photometry** are important, but not addressed

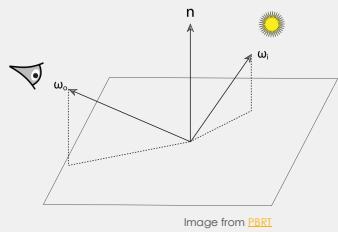


Image from [PBR](#)

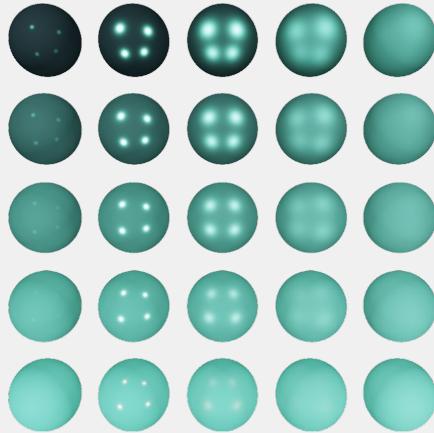


It's possible to create sun light the same way, by considering it as a really distance light. Because it's so far, any modification to the scene wouldn't affect the visible energy change significantly. Thus, only the direction is used and the fall off can be ignored.

It's not an amazing approximation, but is good enough for our course. There are ways to improve sun lighting using other shapes.

Punctual Lighting

Result with four points



```

vec3 radiance = vec3(0.0);
for(int i = 0; i < NB_LIGHTS; ++i)
{
    vec3 w_i = lights[i].direction;

    vec3 kS = FresnelShlick(f0, wi, w_o);
    vec3 specularBRDFEval = kS * f_s(p, w_i, w_o);
    vec3 diffuseBRDFEval = (1.0 - kS) * f_d(p, w_i, w_o);
    diffuseBRDFEval *= (1.0 - metallic);
    vec3 inRadiance = sampleLight(lights[i], p, w_i);

    float cosTheta = dot(normal, w_i);
    radiance += (diffuseBRDFEval + specularBRDFEval) * inRadiance * cosTheta;
}

```

Pseudocode for direct lighting taken from [Slide 3](#).



Here you can enjoy our first result! You will be able to render something like this during the lab :)

From left to right: varying level of roughness, from **0.0** to **1.0**

From top to bottom: varying level of metalness, from **1.0** to **0.0**

Note

One thing to note: you might be stuck with different light units

Point lights aren't as fidele as **Area Lights**

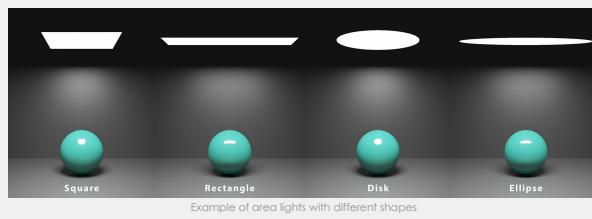


Image From [Radeon Pro](#)

To get better results, it's common to use **Area Lights**, which aren't infinitesimal and represent better the type of lights we use in the daily life.

Area lights bring better shadowing and smoother rendering. However, sampling area light doesn't have analytical solution, which makes the process:

- Hard to implement
- Computational heavy

Because of those two reasons, we will stick to point lights for this course, which I think are enough to get you started with beautiful renderings :)

Indirect Lighting

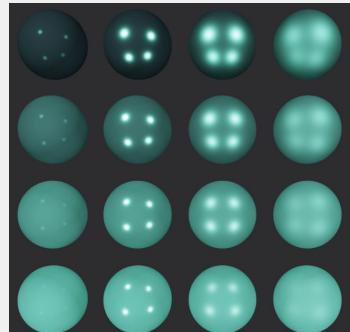
Recall that...

I introduced PBR with **this** image...



Expectation

And that's **our result** so far...



Reality

◀ | ▶

For those paying attention, you might feel fooled by my beautiful words. I started this course showing you all those beautiful screenshots, all taken from amazing sources such as [Filament](#), or [Unreal Engine](#). However, what do we have here? Some bowling balls barely reflecting anything! What is wrong with this course?!

So, what's happening?

Recall our rendering equation:

$$L_o(p, \omega_o) = \int_{\Omega} f_r(p, \omega_o, \omega_i) L_i(p, \omega_i) n \cdot \omega_i d\omega_i$$

But we already mentioned that the equations is **recursive**

Unfortunately, we have been so far integrating only the radiance coming **straight from a light** to the **point we shade**

Light bounce have been **ignored!**



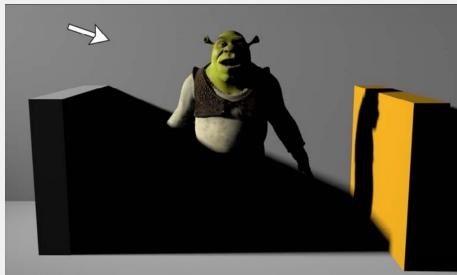
When in doubt always go back to our source of truth: the **Rendering Equation**. Recall that the equation is recursive, computing the incoming radiance is dependent on the irradiance entering the hemisphere at the point of interest. So far, we have been... approximating a **little**. When I say "*a little*", please actually read: "*a lot!*".

We have only been simulating primary rays and we have been integrating over punctual lights. We have completely ignored secondary light bounces!

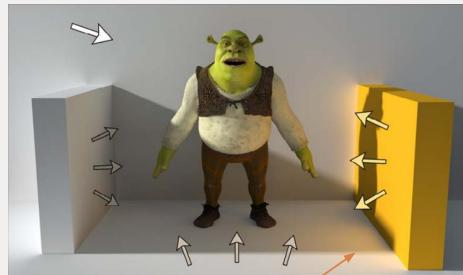
On the next slide, let's have a visual comparison of what we have been doing versus what we should have been doing.

Direct vs Direct + Indirect

Example 1/2



Images from: "Global Illumination Across Industries", by Eric Tabellion



See how the floor gets lit by the wall
Materials from the surrounding affect an entire scene

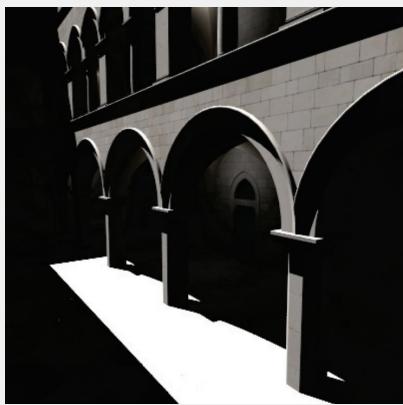


On the left, you have a Shrek rendered only with **direct lighting**, coming from a single directional light source. On the right you have a more accurate light transport estimation taking into account **indirect lighting**.

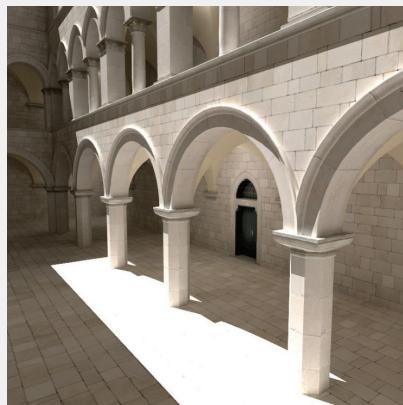
These examples show more clearly how surrounding materials participate to the final amount of energy. If you look closely at the image on the right, you can see that the bright yellow wall affects its surroundings!

Direct vs Direct + Indirect

Example 2/2

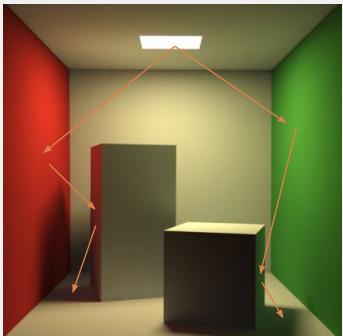


Images from "[Global Illumination Across Industries](#)", by Jaroslav Křivánek



◀ | ▶

Another example where the image on the left is lit only with direct light from a single directional light. Every occluded primitives end up not receiving any energy and so most of them appear completely dark. This is obviously **unrealistic** and is a consequence of the current sampling strategy we have apply so far: sampling **direct lighting only**.



Demonstrates the effect of Global Illumination on a scene

From [wikipedia](#)

Chandelier shot from Toy Story 4. Some frames took 1300h to render.

Global Illumination (GI)

How?

Can't we just compute everything?

Would be too easy if we could (I wouldn't have a job I guess)

Naïve implementation:

Shoot random ray and recurse

Accumulate

Physical and beautiful but... converges in **ages**

Lots of existing techniques!

Each has its **pros** and **cons**

Some are screen-space: SSAO, SSR, etc...

Pre-computed ones: **Image-based Lighting**, **Lightmap**, etc...

Dynamic: **Voxel Cone Tracing**, **Raytracing**

Most techniques rely on some sort of **caching**

This course is about **real time**, let's talk about those ones



So what's the catch? How do we get beautiful images like above? Is it hard to implement?

A naïve implementation is actually relatively simple to implement. Perform the recursion over multiple bounces, shoot random rays, and average the result. You will get the correct value however... You might have to wait a **lot** of time. When I say a lot... I mean, a **lot!**

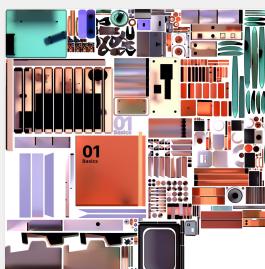
Just to let you imagine: In Toy Story 4, there is a scene with a chandelier in an antique store. A single frame from this scene took 1300 hours to render (proof: <https://twitter.com/Pixar/status/1380671797226393605>). This was rendered on a Pixar renderfarm, we aren't even talking of a user's desktop.

There are many techniques to achieve coarse or precise indirect lighting. There is no perfect world: Every technique has its own **pros** and **cons** and what techniques suit your need will depend on your use case.

Let's have a look at a few of them, but we will only pick one for this course!



Scene with a single lightmap and no punctual lights

lightmap used to lit the left scene
From [Bruno Simon](#)

Because everything is pre-computed,
you can't ride and move this board 😞

Global Illumination (GI)

Lightmap

Store lighting information in a **texture**

Generated using more **accurate** rendering algorithm

Can be generated in third-party software: **Blender**, etc...

Process of pre-computing is often called "baking"

Pros

Quality can be good if pre-computed correctly

Implementation simplicity

Cons

Static objects

Memory consumption



One of the simplest one is **Lightmapping**. The idea is relatively simple:

- Use a complex rendering algorithm ahead of runtime and store the results in textures
- At runtime, fetch those textures and treat them as global illumination, i.e., an extra source of light

Just as a quick note: the process of rendering and storing the result as a cache is called **baking**.

This is really easy to implement (if you use a powerful third party software to actually generate the lightmaps). Besides, the rendering quality can be amazing for an extremely cheap cost (1 texel fetch).

Lightmapping allows you to render **complex scene** even on mobile hardware at high frame rate.

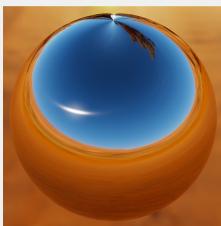
If that was it, computer graphics would be a solved field and no more effort would be put into it. I am sure you are already realizing the biggest downside this technique has: no object can be **dynamic**.

You want a light to rotate at runtime? Can't happen (except if you accept visual

artefacts). You want a baked object to move in your scene? Can't happen.

This technique comes with a major downsides and this is why there are tons of other way to go GI floating around.

At the end of the day: lightmaps might be enough. It all depend on what your use case is and what constraints you have.



Global Illumination (GI)

Image-based Lighting

Spherical image acting as a light source

Assumes rendered object is **centered in the image**

Cache looked up using **normal** and material params

Cache **irradiance** and **BRDF** parameters

Pros

High quality

Objects can be rotated

Cons

Static objects (translation only)

Probe placement



Pre-computed environments, also called **Image-Based Lighting**, is the technique often assimilated to PBR rendering.

This technique is similar to lightmapping, but differs in how it's precomputed and evaluated. Here, objects are assumed to be at the center of the light probe (environment) and the pre-computed environment is fetched using the geometry information (normal) as well as the materials parameters: roughness, metalness, etc...

Compared to lightmaps, the environment is **pre-integrated**, meaning that part of the rendering equation has been pre-computed using this environment. Because each texel contains the pre-integrated environment based on the direction from the center of the sphere to the texel, it's then possible at runtime to shade the object even if a rotation occurs. More operations are needed to shade the surface compared to lightmapping, but this little extra cost comes with the possibility to rotate freely our models!

If you didn't understand exactly how this technique can work, don't worry! We will deep dive in this method later in the course.



Image from [Unreal Engine](#)

Global Illumination (GI)

(Hardware) Raytracing

Many techniques in this domain as well!!

Those techniques still use **rasterization**

Offload some pieces to **raytracing**

Close to **100%** raytraced, but give it a few more years

RTXGI: continuously generate probe at runtime

Pros

Raytracing algorithm are often "simpler"

Flexibility and efficiency with the dedicated hardware

Cons

Needs dedicated hardware



I didn't want to make this course a shopping list of all the cool stuff happening in the real-time global illumination field. However, I couldn't do this course without mentioning hardware raytracing!

Hardware raytracing is newish (few years old, meaning centuries on the scale of tech evolution speed) but is **likely** to change real-time rendering over the next few years. If you don't know what hardware raytracing is, it's basically all in the title! Raytracing... directly in the hardware. There are many algorithms used to compute the radiance, but raytracing has always been appreciated because:

- The algorithm can handle optical effects, think refraction / reflection
- A lots of filmic effects are easy to implement such as **Depth of Field**
- It fits "naturally" to light's behavior.

I added on this slide an example of Global Illumination algorithm created by NVIDIA: [RTXGI](#). This algorithm works hand-in-hand with **IBL** by re-computing light probes at runtime using raytracing. This algorithm is just one example of what can be achieved. It's also possible to use raytracing only for **shadows** or for **ambient occlusion** effects (can be considered part of **GI**).

One limitation there is today is the need for dedicated hardware, which is already changing. I mean, you still need dedicated hardware, but many other companies are starting to roll-out hardware raytracing capabilities.



Global Illumination (GI)

So much more!

So many more methods

Unfortunately, those topics outside the course's scope 😢

Simply remember that there are many techniques, all with pros and cons

For this course, let's focus only on **Image-Based Lighting**!



There are **many** more methods... And picking one is always dependant on your use case and your computing power available.

This course is about **PBR** and not GI techniques in general. It's really hard to start talking about **PBR** without evoking those anyway.

Without further ado, let's go deeper in the topic of **IBL** and let's see how it fits our previously implemented **PBR** setup.



City of Monaco

Monte Carlo (MC)

This section isn't really going to be about Monaco, but I was lacking new image ideas

Before entering the beautiful world of Image-Based Lighting and how to pre-compute our environments, we will need to go back to some theory. Let's talk about statistics for bit 😊

$$F = \int_a^b f(x) dx$$

Integral to solve

$$F_N = \frac{1}{N} \sum_{i=0}^N \frac{f(X_i)}{pdf(X_i)} \quad X_i \in [a, b]$$

Discretized using **Monte Carlo** method

Random variable

Example of Pathtracer before (left) and after (right) convergence. Pathtracer from [David Peicho](#).**Monte Carlo**

The location? what?

Need to solve the rendering equation, i.e., **integrate**
How to go discrete?

Monte Carlo: use random sampling to **estimate** an **integral**It's thus **non-deterministic****Intuition:**

Imagine you are looking for the average height in a country
Instead of asking everyone, pick **N** samples and average

The larger **N** is, the more accurate the output isFormal name: the [law of large numbers](#)Noise in **Pathtracing** is the consequence of **Monte Carlo**

You might be wondering: Why is he even talking about Monte Carlo and Monaco?

It turns out in this context, **Monte Carlo** has nothing to do with the French Riviera. We have discovered together that we can approximate the rendering equation by discretizing it over the direct lights of our scenes (points, directional, etc...). We also talked about how this is a crude approximation of the equation.

There is a known method to solve integral, and this method is called... **Monte Carlo** (big surprise, I know). The idea behind Monte Carlo is to perform **stochastic integration**. In other words, the technique discretizes an integral and sample from the function **N** times using **N** randomly picked samples.

We can get a pretty good intuition about it using a simple example:

- Imagine a function **f**, that associates to each person its height
- We would like to get the average height of the population
- Instead of sampling every single person, we can pick a subset of **N** randomly picked person, and average their height

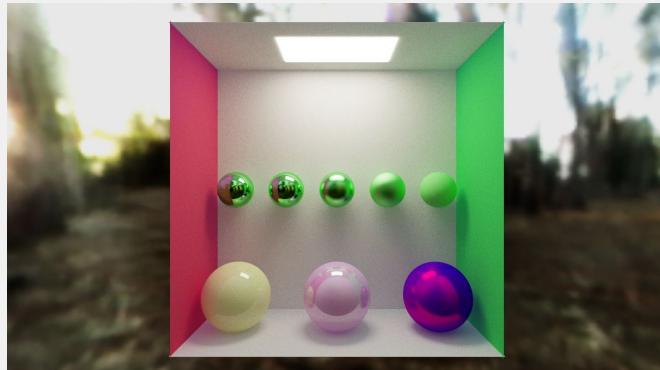
The result of this experience will be biased, but choosing a sufficiently large **N** should

bring you to the appropriate result. What I am describing here has a name: the [Law of Large Numbers](#).

For those of you who know about **Pathtracing**, Monte Carlo is the reason why noise appears! Instead of solving the entire rendering equation, a **Monte Carlo Pathtracer** will use statistics and random samples, resulting in noise at low frame count, due to high variance (i.e., “errors”).

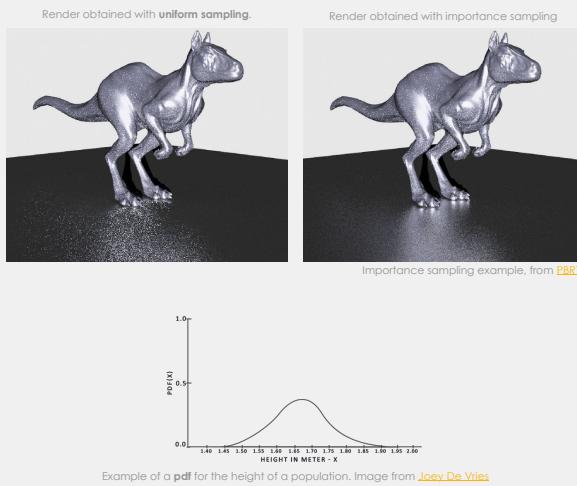
Alan Wolfe's Pathtracing Example

<https://www.shadertoy.com/view/WsBBR3>



◀ | ▶

We just talked about **Pathtracing** and how visible the noise can be. By clicking this ShaderToy link, you can see a live pathtracer converging slowly toward the good result.



Monte Carlo Importance Sampling

Uniform sampling has a slow **convergence rate**

N needs to be very large for good result

We haven't yet talked about one term: $pdf(X_i)$

This is the [Probability Density Function](#)

Dividing by the **pdf** allows to sample from non-uniform distribution

Samples with less likelihood will weight more

This is basically what **Importance Sampling** is about
Reduce **variance** by performing sample selection



We said that for large enough **N**, the result should converge to the expected result. However, picking very large **N** isn't really nice, because we end up performing a lot of computations.

Our goal is to speed up the rendering process as much as possible, i.e., converge as fast as possible. How can we achieve that? The answer is **Importance Sampling**.

Two slides before, I showed you how to discretize the function. If you recall, the function was actually divided by another function: a **pdf (Probability Density Function)**. Using **Importance Sampling**, the samples will be picked from **proposal** distribution and will be **weighted** to target the original distribution we originally wanted to sample from.

Let's briefly talk about our population's height example. The **pdf** would most likely look like the above diagram. Thus, instead of generating useless random samples below 1.60m and above 1.85m, we could sample from a function biased toward the 1.6-1.7 height range. Dividing by the **pdf** would then re-project the result in our original space: a **uniform distribution**.

To sum up: the goal of importance sampling is to generate meaningful sample in

order to converge at a faster rate.

Quick Proof

Mean $\longrightarrow E[h(X)] = \int_{-\infty}^{+\infty} h(t) f_X(t) dt$

PDF

$$E[h(X)] = \int_{-\infty}^{+\infty} h(t) f_X(t) \frac{g_Y(t)}{g_Y(t)} dt$$
$$E[h(X)] = \int_{-\infty}^{+\infty} \left(\frac{h(t) f_X(t)}{g_Y(t)} \right) g_Y(t) dt$$
$$E[h(X)] = E\left[\frac{h(t) f_X(t)}{g_Y(t)}\right]$$



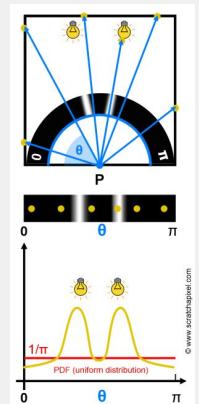
How does it help us?

We can turn rendering into a statistical problem by applying the **Monte Carlo** method

Importance Sample the BRDF: generate samples based on **roughness**



Shows the different lobe based on material parameters. Image from [Joey De Vries](#)



With this in mind, we can accelerate **convergence** big times

You can even go further with **Multiple Importance Sampling**



We can basically apply **Monte Carlo** to our rendering equation, using uniform sampling or using importance sampling.

Importance sampling requires some knowledge about the scene: materials, lights positions etc... There are multiple techniques you could apply. One simple enough technique is to importance sample the **BRDF**.

When using a Cook-Torrance BRDF, we have some knowledge about where the energy is mostly coming from based on the material roughness.

In the case of a smooth material, we know that most of the energy is concentrated in a single direction (the reflected direction based on the view direction). Thus, we can bias the sample to this direction. In the case of a rough material, any direction in the hemisphere could be a sample.

Importance sampling the **BRDF** is already good enough for some use cases. In a pathtracer, we often need to go the extra steps and use [Multiple Importance Sampling](#).

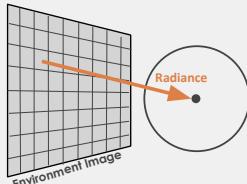


Image from [Filament](#)

Image Based Lighting

Now that we are done with our side explanation of Monte Carlo, let's get deeply into the Image-Based lighting topic. This section is going to be much harder than what we have seen so far. However, I don't want you to panic, this section is mostly going to be bonus points for the project to hand in.

However, if you like computer graphics and plan to have a career in it, I strongly advise you to research those concepts, at least on a high level.



Beautiful drawing (by myself) to show you what we want to achieve

$$L_o(p, \omega_o) = \int_{\Omega} (k_d \frac{\rho}{\pi} + k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p, \omega_i) (n \cdot \omega_i) d\omega_i$$

$$L_o(p, \omega_o) = \boxed{k_d \frac{\rho}{\pi} \int_{\Omega} L_i(p, \omega_i) (n \cdot \omega_i) d\omega_i} + \\ \boxed{k_s \int_{\Omega} \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} L_i(p, \omega_i) (n \cdot \omega_i) d\omega_i}$$

Specular

Image Based Lighting

Again, but better this time!

Every **texel** treated as an **emitter**

wi (texel to sphere center) needs to be evaluated

The radiance from w_j is affected by other environment texels

Diffuse & specular are independent

Solve them separately

Changing **BRDF** means **re-convoluting** the image!

We have talked briefly about Image-Based Lighting treating every texel as an emitter. We thus want to pre-compute the image, for each direction from the sphere texel to the center of the sphere, where the object is supposed to located.

To understand how to pre-compute the environment image, we will need once again to go back to our equation.

Looking at the equation, we can see that the diffuse and specular components are actually unrelated to each other, and can be easily separated. Besides, a lot of terms are constant to the integral, and can also be moved out of it.

One **super important** thing to note: we will pre-compute the equation for **a given** diffuse and specular BRDFs. This means that you can't simply used the pre-computed environment with **any** BRDF when rendering. If you find some pre-computed environment online for different BRDFs that you aren't using, you technically shouldn't be using them.

PBR; Real Time

$$k_d \frac{\rho}{\pi} \int_{\Omega} L_i(p, \omega_i) (n \cdot \omega_i) d\omega_i$$



Pre-compute Diffuse

One note on the above image: you can see how it looks like an unwrapped cube. This is because probes are often represented as cubemap. To reduce the overhead of creating cubemap in this course, I simply preferred to use a equirectangular maps.

$$L_o(p, n) = \int_{\Omega} \frac{\rho}{\pi} L_i(p, \omega_i) n \cdot \omega_i d\omega_i$$

$$L_o(p, n) = \frac{\rho}{\pi} \int_{\Omega} L_i(p, \omega_i) n \cdot \omega_i d\omega_i$$

$$L_o(p, n) = \frac{\rho}{\pi} E_n(p)$$

Apply **spectrum** information at **runtime**

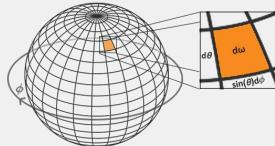


Image from [Joey de Vries](#)

Diffuse
Convolution

Convolution for the Lambertian BRDF

Isolate constant term that doesn't need **pre-integration**

Hard to integrate over solid angle, either use given approximation or integrate in **Spherical Coordinates**

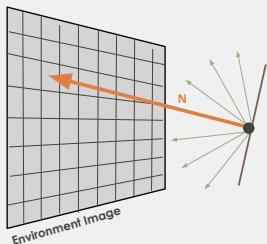


In order to convolute the diffuse lobe, we first isolate constants that can be applied at **runtime**.

Because it's not easy to integrate the irradiance as-is based on the solid angle, we can either:

- Use a coarse approximation of the solid angle
- Integrate over spherical coordinates

For more information about how to integrate radiometric integrals, please have a look at the [PBRT book](#).



```

vec3 acc = vec3(0.0);
int count = 0;
for(float phi = 0.0; phi < 2.0 * PI; phi += 0.25)
{
    for(float theta = 0.0; theta < 0.5 * PI; theta += 0.25)
    {
        // Direction must be updated using phi and theta.
        vec3 direction = ...
        acc += texture(environment, direction).rgb * cos(theta) * sin(theta);
        count++;
    }
}
acc = PI * irradiance * (1.0 / float(count));

```

Pseudo code demonstrating how to compute the irradiance

Diffuse
Convolution algorithm

Discretize integral using **Riemann Sum**

Convert integral over spherical coordinates

Accumulate radiance (i.e., compute **irradiance**)

The **more sample**, the **better** the approximation will be

Importance Sampling not needed here

Diffuse lobe has a uniform distribution

No samples are more **important** than other



We can then use a Riemann Sum in order to compute an approximation of the integral. The idea is to use discrete weighted samples. For the case of the irradiance map, it's enough to select uniformly distributed samples.

The two drawings on this slide explains how the algorithm work. For every texel of the environment map (cubemap, ...), you should compute the oriented hemisphere with normal the direction to the currently processed texel. The rendering equation is then applied to neighboring texels that will contribute to the final **irradiance**.

At **runtime**, the convoluted environment is fetched using the normal and used as an ambient occlusion term. It's up to the runtime shader to determine whether the environment lighting must be occluded or not.

We do not use Monte Carlo here because the diffuse is integrated from all over the hemisphere. Thus, using a Riemann sum is possible.

Result



Input Image



Convolved Image

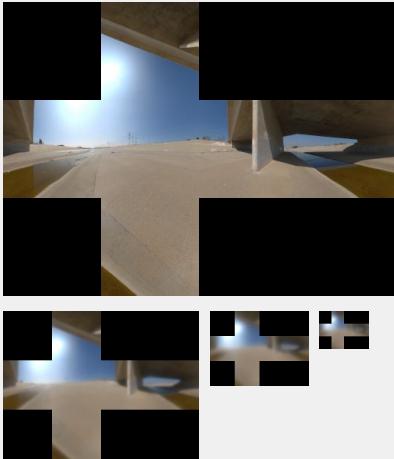
◀ | ▶

For those paying attention, you might feel fooled by my beautiful words. I started this course showing you all those beautiful screenshots, all taken from amazing sources such as [Filament](#), or [Unreal Engine](#). However, what do we have here? Some bowling balls barely reflecting anything! What is wrong with this course?!

PBR; Real Time

$$k_s \int_{\Omega} \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} L_i(p, \omega_i)(n \cdot \omega_i) d\omega_i$$

IBL Specular



That's all you needed to pre-integrate the diffuse part. This process was actually not too hard and is pretty fast to compute. The hard part of the job comes now with the specular integration.

$$\int_{\Omega} \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} \quad \text{← Depends on } \omega_o \text{ and } \omega_i$$

Split Sum Approximation:

$$L_o(p, \omega_o) = \int_{\Omega} L_i(p, \omega_i) d\omega_i * \int_{\Omega} \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} (n \cdot \omega_i) d\omega_i$$

- Same as the diffuse
- roughness as extra parameter

Specular
Split Sum Approximation

Dependent on pair of **light direction / view**
direction = much harder to deal with!

Solving for combinations of ω_o / ω_i is too much

Solution: Integral splitted using the Split Sum Approximation [\[Karis14\]](#)

Not 100% accurate, but good for most environment

At runtime, fetch both cache and multiply
them together to get full **specular component**

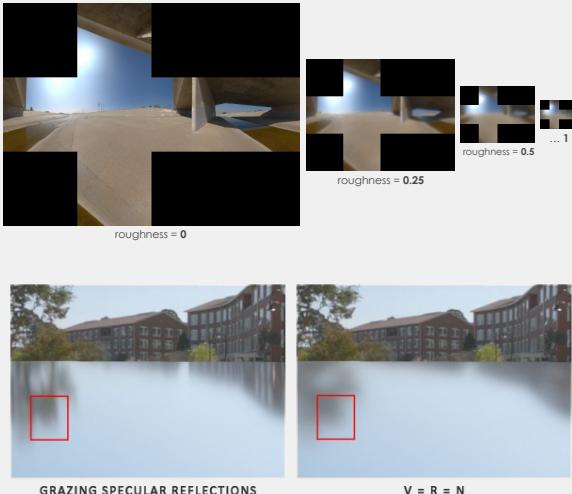


Pre-computing the specular component is much harder because it depends on more variables. Trying to generate all combination isn't feasible and wouldn't make sense in a real-time application.

The idea used by most (everyone?) application is to approximate the integral into two simpler form that can be computed separately. This is called the Split Sum Approximation and has been presented [in this paper\[Karis14\]](#).

In the next two slides, we are going to have a look at how each integral is pre-computed and stored in a texture.

$$\int_{\Omega} L_i(p, \omega_i) d\omega_i$$



Shows how the $V=R=N$ approximation affects crisp specular at grazing angle. Image from [Frostbite PBR](#)

Specular
Pre-filtered environment

Similar to diffuse irradiance, but uses **roughness**

Because high roughness blurs the contribution, store in lower mips (lower resolution)

Saves **memory**
Improves **cache coherency**

No access to **view** direction / **reflected** direction

Hard approximation: $V=R=N$ ([Karis14](#))

In the UE paper, the view V is our ω_o

In the UE paper, the normal is denoted N



The technique to pre-integrate the environment is similar to the diffuse one. The difference here though is that we need to take into account the **roughness** as well. The higher the roughness, the blurrier (wider) the specular lobe should be. We can't simply integrate for a constant roughness.

The solution is to integrate for several level of roughness (e.g: 0.25, 0.5, 0.75, 1.0), and store the results in different images. In order to reduce the memory footprint, it's common to store higher roughness level in higher level of mipmaps. Because the convolution acts as a low-pass filter, we can take advantage of mipmapping.

If you don't know what mipmap is, it's basically a downsampled version of the original texture. Mipmap pyramids are built by dividing by 2 the previous level. If the first layer has size 512x512, the second will have 256x256, and so on.

One other thing to note: we assume the view / surface angle is 0, so $V=R=N$. This obviously affects the result negatively. If you look at the image on the right, we lose a ton of information and crisp edges at grazing angles.

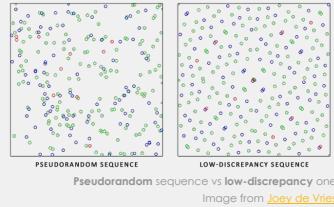
$$\int_{\Omega} L_i(p, \omega_i) d\omega_i$$

```

vec3 acc = vec3(0.0);
vec3 w0 = normalize(fragmentPosition); // w0 is often called "view"
vec3 normal = w0;
for(int i = 0; i < N; ++i)
{
    // Direction must be updated using phi and theta.
    vec3 direction = generateImportanceSampledDirection(roughness, normal, w0);
    vec3 radiance = texture(environment, direction).rgb;
    acc += radiance * brdf(roughness, w0, normal) / pdf(N, V, roughness);
}
acc /= N;

```

Pseudo code demonstrating how to compute the convolution for the specular component



Pseudorandom sequence vs low-discrepancy one

Image from [Joey de Vries](#)

Specular

Convolution algorithm

Much harder than for the diffuse

Some information are left out for interested students

Use **Quasi Monte Carlo** method

Generating **low-discrepancy** sequence of random numbers

Generating biased direction using the BRDF

Do not worry if you find that difficult

It's a really complex topic

A starting point would be to just try to get the ideas right

More information by reading [\[Karis14\]](#)



There is one major difference with the diffuse component. The specular component isn't spread uniformly in the hemisphere.

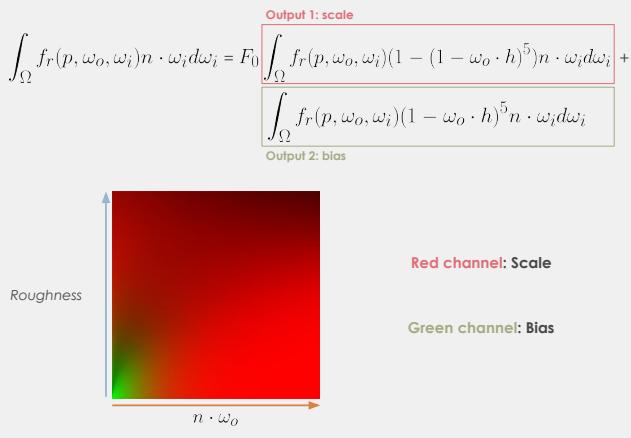
Quite the opposite, low roughness values means narrow specular lobe. Using uniformly distributed samples, we would end up with many samples not contributing at all to our final result, which is wasteful.

The function `generateImportanceSampledDirections` hides a lot of detail.

The first important thing in a **Monte Carlo** is the random number generator. Using a pseudorandom generator might give a lot of collisions and convergence might take a longer time to reach.

A sequence that looks like the right one on the above image is called quasi-random, this is why the process of using quasi-random sequences with **Monte Carlo** is called a **Quasi Monte Carlo** method. Using low-discrepancy sequences isn't mandatory but will help us reach convergence faster. The second thing is to generate biased samples in the direction of interest, using the **roughness**.

$$\int_{\Omega} \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

**Specular**

Pre-computed BRDF

Equation obtained substituting **Fresnel Shlick**: [\[Karis14\]](#)

Pre-compute this equation using two variables:

 ω_o i.e., the view direction The **roughness**Any **normal** works here $(0, 0, -1)$ The pre-computation is done in the hemisphere space,
not in **world space****Agnostic** from input image

Once again, this is left for students with extra free time!



The second part of the split sum approximation is also quite hard to work with. Thanks to the amazing work that has been done, we know how to derive a simpler form to integrate [\[Karis14\]](#).

This is convoluted exactly like the irradiance and specular environment. For every permutation of roughness/cos theta, compute both integral using Monte Carlo, and save the result at the current texel.

This pre-computed texture is agnostic from the input image as you can see. It's simply a pre-computation of the **BRDF** itself. In a production pipeline, you could have this image pre-compute for any **BRDF** you might want to use.

PBR; Real Time

$$\int_{\Omega} \left(k_d \frac{\rho}{\pi} + k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} \right) L_i(p, \omega_i) (n \cdot \omega_i) d\omega_i$$



Composition: Specular + Diffuse

```
// Environment are convoluted around the normal, that's our w_i
vec3 kS = FresnelShlick(f0, normal, w_o);
vec3 kD = (1.0 - kS) * (1.0 - mat.metallic) * albedo;

vec3 diffuseBRDFEval = kD * texture(prefilteredDiffuse, normal).rgb;

// Specular is fetched using reflected direction
vec3 reflected = reflect(w_0, normal);
vec3 prefilteredSpec = fetchPrefilteredSpec(roughness, reflected);

vec2 brdf = texture(brdfPreInt, max(dot(normal, w_0), roughness).xy;
vec3 specularBRDFEval = prefilteredSpec * (kS * brdf.x + brdf.y);

vec3 gi = diffuseBRDFEval + specularBRDFEval;
```

Pseudo code demonstrating how to apply image based lighting contribution

Runtime

Apply all the pre-computed data

Pre-filtered diffuse is fetched using the **geometry normal** and applied as **ambient** lighting at **runtime**

Pre-filtered specular fetched with **reflected** ray

We shifted an **expensive** computation ahead of time

Runtime performance: a single **texture fetch**

Always be careful with your ω_o and other **directions**

Ensure you are using the good sign



Most of the work we talked about was offline. We have seen how to pre-compute the irradiance and specular maps, and we now need to use this cache during shading time. This is demonstrated by the above code and you can see how simple it is: we only pay the price of a **3** texel fetch and a bit of ALU.

To Remember

Real-Time Image Based Lighting consists of **pre-filtering / caching** as much as possible

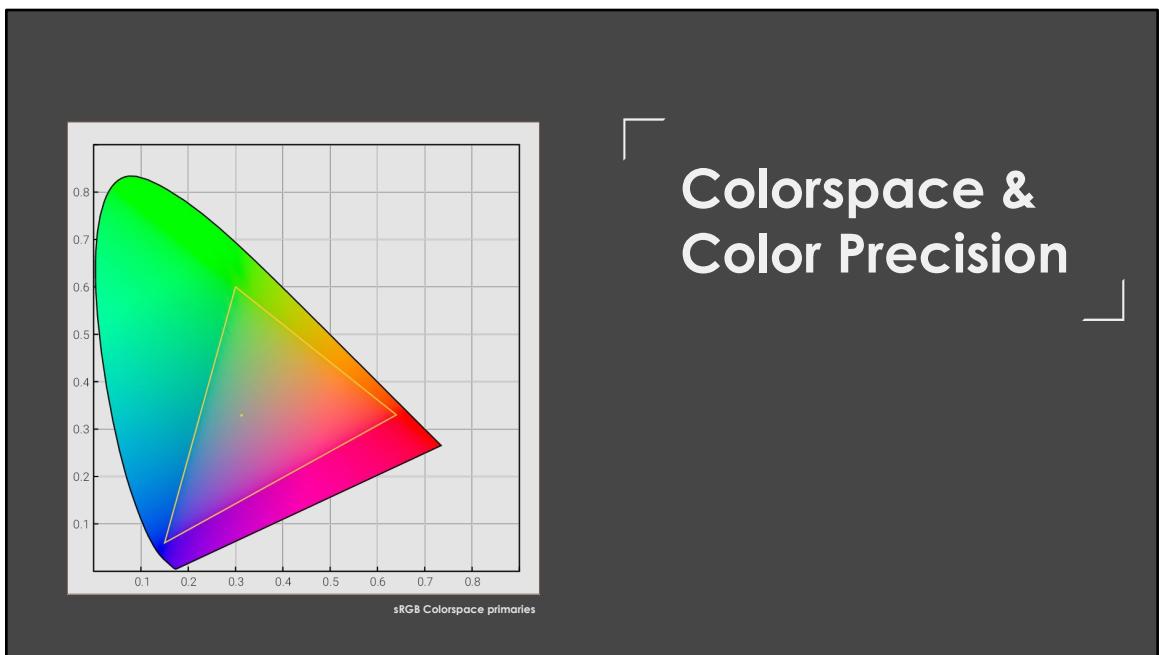
Specular pre-filtered environment is stored in mipmaps per roughness level

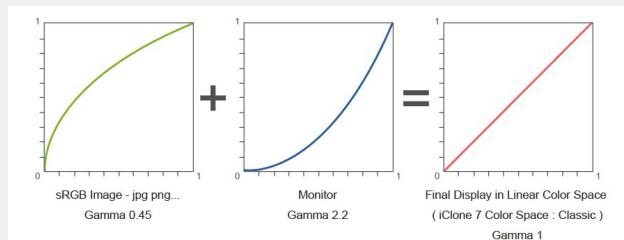
Remember how **Monte Carlo** is used and how **Importance Sampling** can speed up **convergence**



IBL generation isn't an easy process. Do not worry if you think the topic is complex. For the assignment, you will not be asked to pre-filter your environments.

However, sampling the pre-filtered environment and pre-computed BRDF at runtime is relatively easy and will help you achieve much better rendering.





sRGB vs Linear

Monitors apply **pow** function to luminance

CRT luminance was proportional to input voltage raised to power of **gamma**

Images are stored in **sRGB** to compensate monitor transfer function

You should always compute in **Linear**, and output **sRGB**

Hardware can do some automatic conversion



At the time of Cathode-ray Tube (**CRT**), the relationship between input voltage and luminance wasn't linear, i.e., changing the input voltage by a factor of n didn't end up modifying the luminance by factor of n . The process of correcting the CRT image is called **Gamma Correction**, it consists in applying the inverse transform of the gamma function. If your monitor has a gamma of '*gamma*', the gamma correction function will be: $\text{pow}(x, 1/\text{gamma})$

Nowadays, we don't use CRTs anymore (at least I don't :)). However, **Gamma Correction is everywhere**. Your movies / images / anything are most likely gamma corrected. Because of that, monitors nowadays **still** apply a **Gamma function**.

Textures you edit in software like Photoshop, GIMP, will be in the sRGB colorspace, i.e., a **Gamma Correction Function** will be applied to the texture before it's saved on disk. Obviously, you are free to change your software settings, and you could save your textures in linear if the option is available.

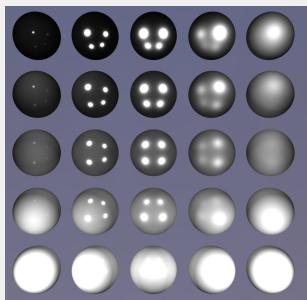
It's important to be consistent and to work in the appropriate color space or you might end up with colors that are too saturated or just too dark. Stay consistent, i.e., always work in the same color space.

This is the workflow I prefer, but it will be different depending on the codebase you work on:

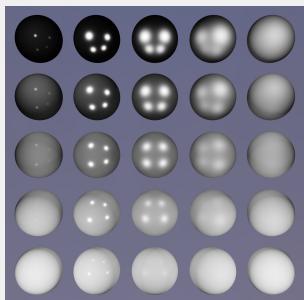
1. Convert all textures to linear before uploading to GPU
2. Do all lighting calculation in **Linear**
3. Convert the final color to **sRGB** in the fragment shader

For the assignment, you can perform the conversion **sRGB** -> **Linear** directly in the shader for simplicity. In a more advanced codebase, the conversion most likely occurs beforehand to avoid unnecessary operations.

Color space issues / conversion aren't only occurring when doing PBR. It's a general rendering topic that every graphics programmer needs to be aware of.



LDR



HDR

Reinhard Tonemapping

$$color_{final} = \frac{c}{c + 1}$$

Famous Tonemapping: [Reinhard](#), [ACES](#), [Uncharted 2](#)

HDR vs LDR

HDR has larger range of values

Units will create radiance color outside the **0..1** range

Perform computation in **HDR**, tonemap to **LDR** if required

HDR is required to get correct PBR result

Especially important for **IBL**, otherwise relative lighting will be messed up!



High Dynamic Range (**HDR**) encodes more values than Low Dynamic Range (**LDR**). For instance, your monitor might use the red, green, and blue channel with a bit depth of 8. Thus, you have **256** possible values per channel. With **HDR**, more bits would be allocated which allows to store more color values.

With OpenGL / WebGL, the bit depth depends on the framebuffer's texture attachment we are rendering to.

Because we now work with real physical quantities, the final pixel color after running all computation will likely be out of the range **0...1**. Light bulbs for instance are already with thousands of Lumens. If we render a scene just like that, we will end up with most of the fragments saturated.

In reality, the human visual system performs a mapping and uses adaptive exposure to generate the final image. Technically, instead of using radiometric quantities, we should have been using photometric quantities, adjusted for the human perception system.

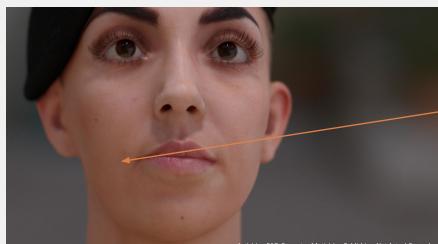
Even though we don't use photometric quantities, we still need a way to capture data out of the **0..1**.

When we have a single shader, we simply need to perform all the calculations in float or double, and map the values at the end of the shader back to the range **0...1**. Mapping the value from HDR to LDR is called tonemapping.

Going Further



Advanced Materials



Light units are **super important**

Lot of materials exhibit more complex light-matter interactions

More complex materials:

- Multiple **specular lobes**
- More complex diffusion profile
- Examples: hair, skin, clouds, etc...

Have a look at **BSDF** and **BSSRDF**



This course was all about making you familiar with simple PBR. We have seen how to render simple (not so simple!) opaque materials by using clever approximations and models.

However, our models will only hold for a variety of materials, and fail for others. Many materials exhibits transmission that give them their particular look. Among those materials we can list **skin**, **marble**, and much more!

Further Reading

Beginner

- [The PBR Guide by Allegorithmic](#)
- [Basic Theory of Physically Based Rendering](#)

Advanced

- [Filament PBR Guide](#)
- [PBR: From Theory to Implementation](#)
- [LearnOpenGL_PBR](#)



References

References

- [Karis14], B. Karis, Real Shading in Unreal Engine 4
- [Torrance67], K.E Torrance, E.M Sparrow, Theory for off-specular reflection from roughened surfaces
- [Cook82], R.L Cook, K.E Torrance, A Reflectance Model For Computer Graphics
- [Shafer84], S.A Shafer, Using Color to Separate Reflection Components. *Color Research & Application*
- [Heitz14], E Heitz, Understanding the Masking-Shading Function in Microfacet-Based BRDFs . *Journal of Computer Graphics Techniques Vol. 3, No. 2*
- [Smith67], B Smith, Geometrical shadowing of a random rough surface
- [Lagarde14], S Lagarde, C de Roussiers, Moving Frostbite to Physically Based Rendering 3.0
- [Hoffman10], N. Hoffman, Physics and Math of Shading
- [Shlick94], C. Shlick, An Inexpensive BRDF Model for Physically-Based Rendering



David Peicho

Thanks

Found an error? Please contact me at david.peicho@gmail.com