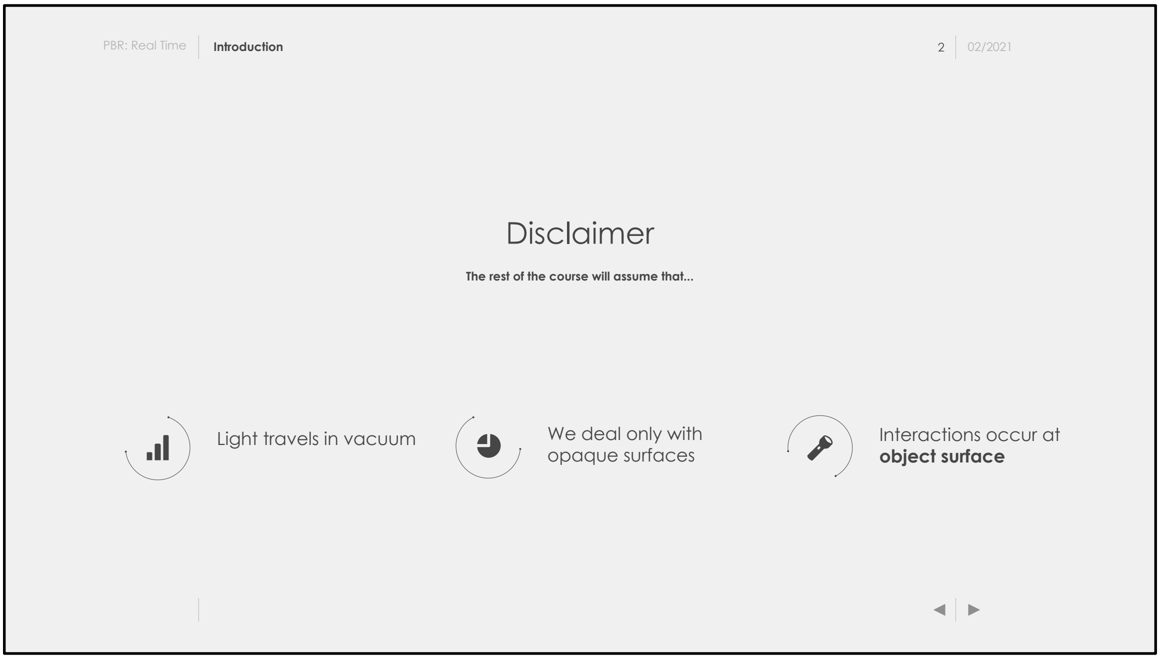
David Peicho

# 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




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 way to generate an image (raytracing, rasterization, etc…)

For each technique, the theory is similar but implementation differs

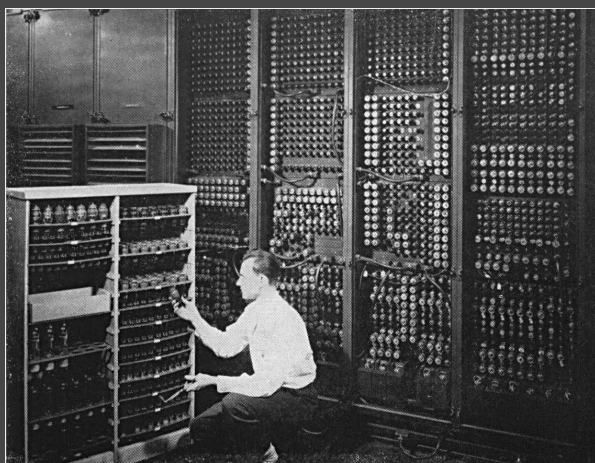
Throughout this course, we will focus on **real-time** with a rasterization pipeline

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.

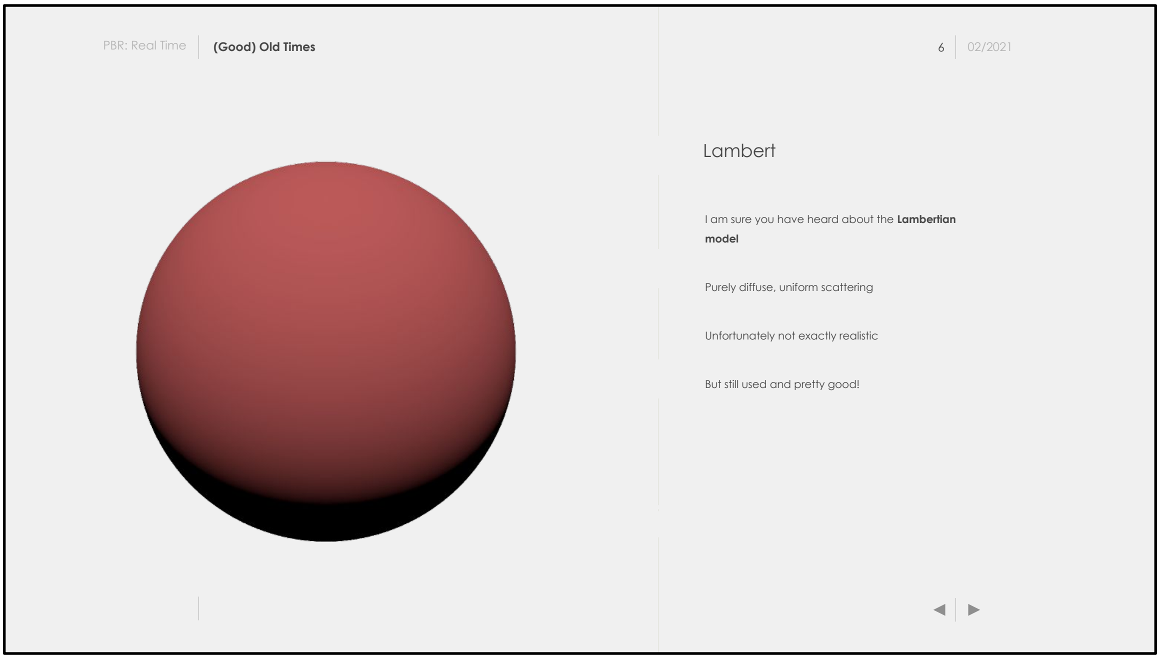
However, because a lot of students are often more interested into video games, we will focus
on the technology behind video games, i.e., real time rendering with a rasterization pipeline.

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

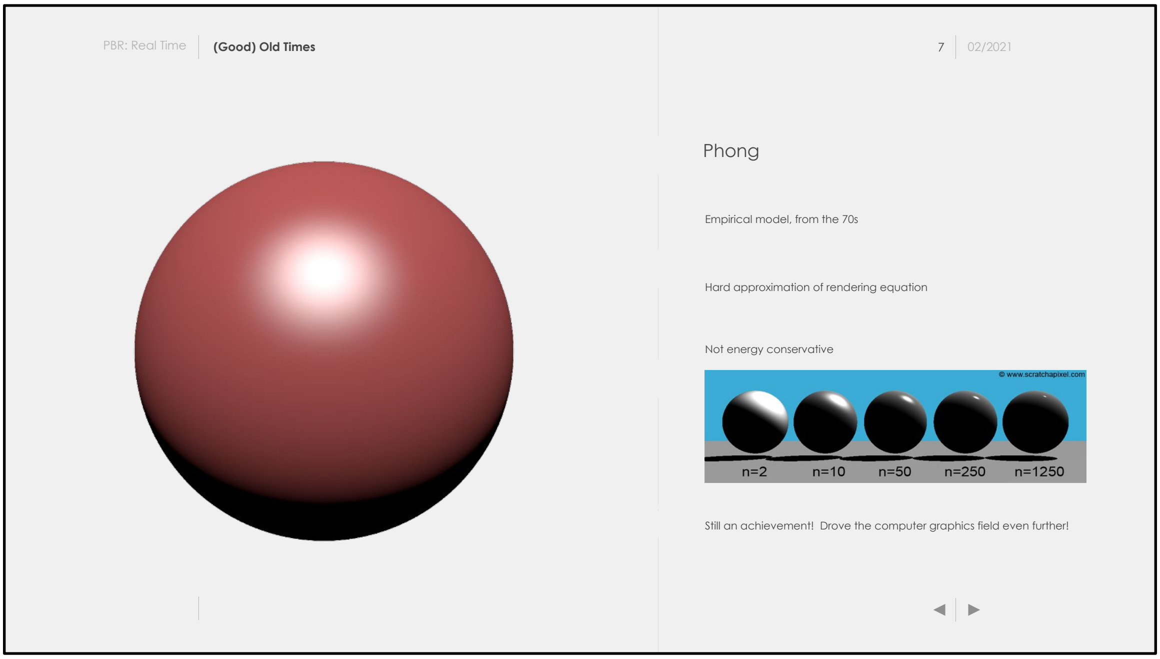



I am sure you have all used a **Lambertian model**, or at least something similar that was
applying a constant to diffuse lighting.

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 describes a perfectly diffuse surface that would reflect light uniformly in every directions.
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 / speed.




I am sure you all have worked on a Phong model implementation at some point. 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.rgba = 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.rgba = vec4(diffuse + specular, 1.0);
}
```

## What & Why

# Introduction

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

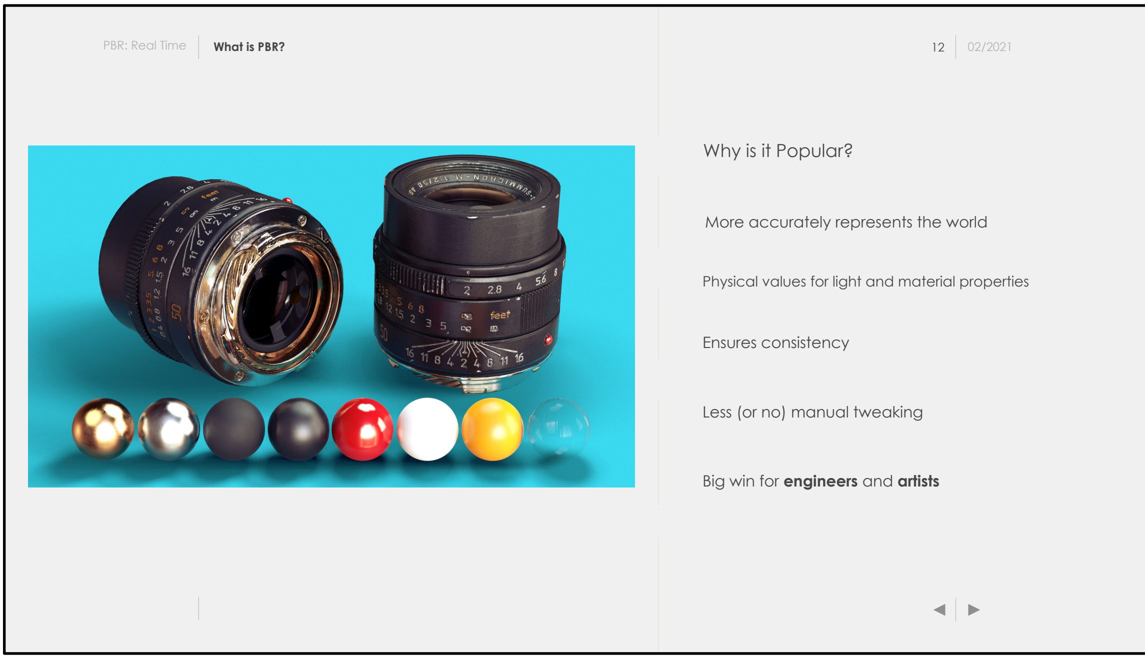
No standard was out there to also help **sharing content**.

But finally, Physically-Based Rendering became the industry standard

# Introduction

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



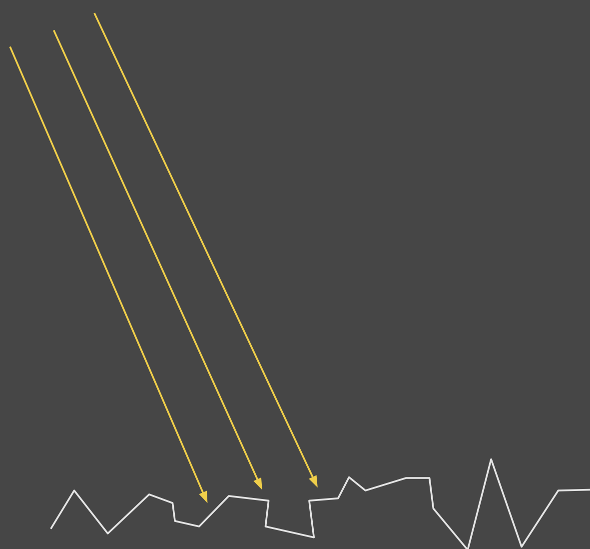

**Physically-Based Rendering** emerged because of all the advantages it brings.

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

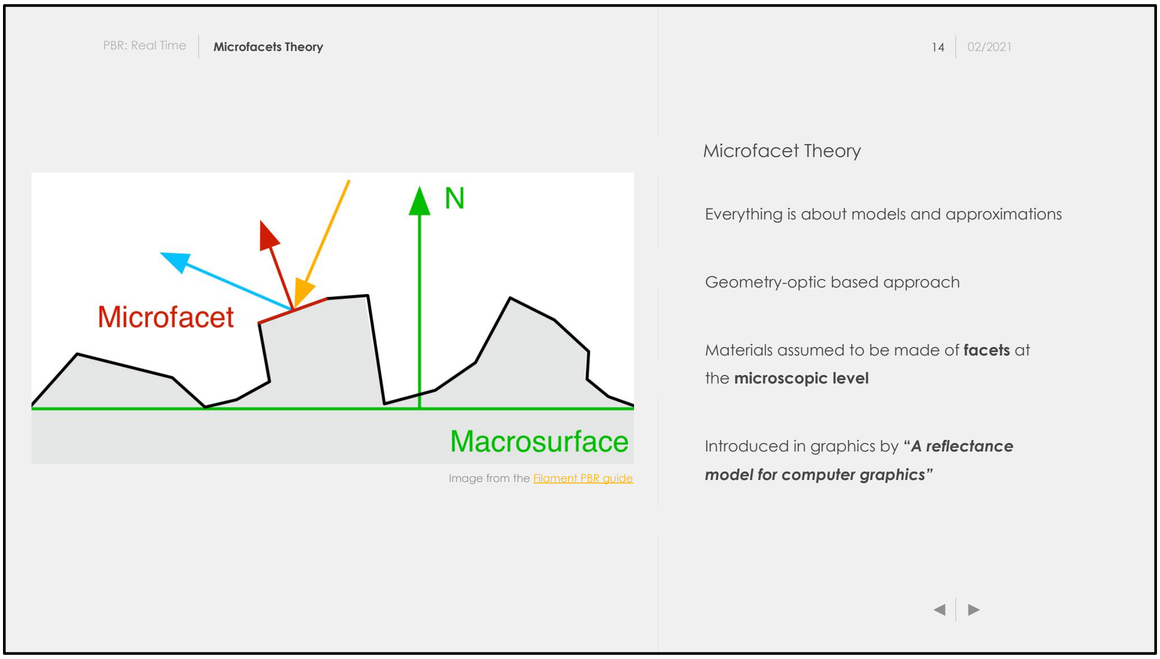
- We can setup scene realistically. We know the temperature of the sun in / out the atmosphere, etc…
- No need to modify the materials in a scene where the lighting would drastically change

All those advantages bring **consistency**.

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.



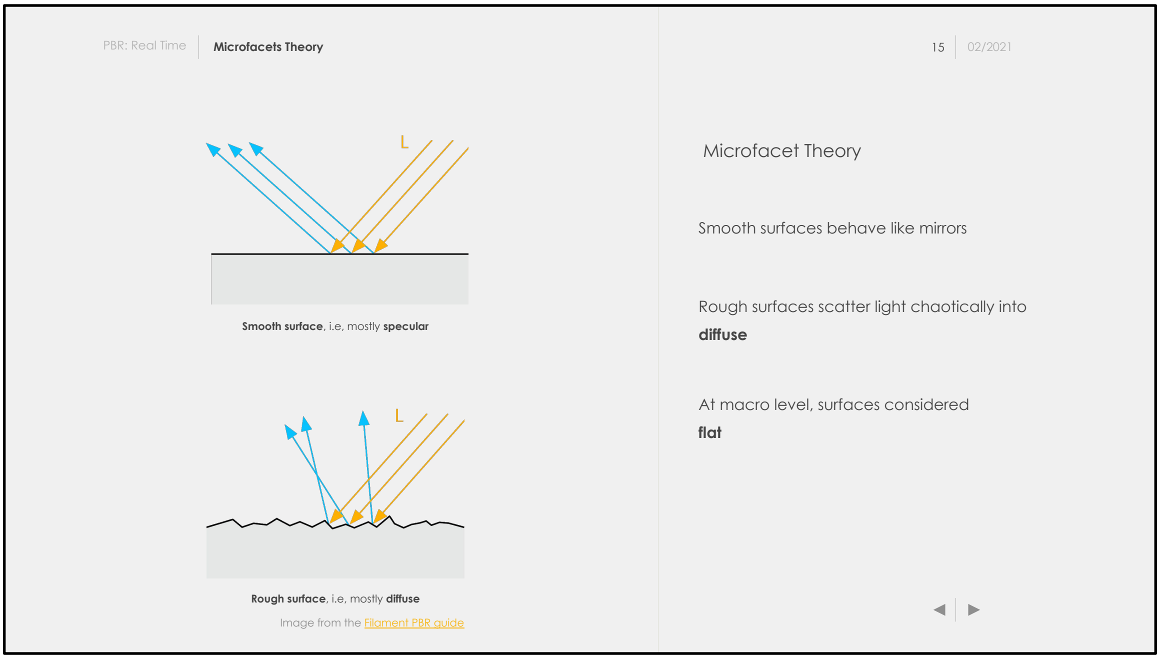
# Microfacets Theory




Our goal is to render something that looks close enough to reality. This is where the **Microfacet Theory** comes in.
As far as I know, the theory has been introduced in graphics by: [Cook82]. The idea goes back even beyond to [Torrance67]

Microfacets 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).




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.

# Dielectrics vs Conductors


Image from UE4


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!.

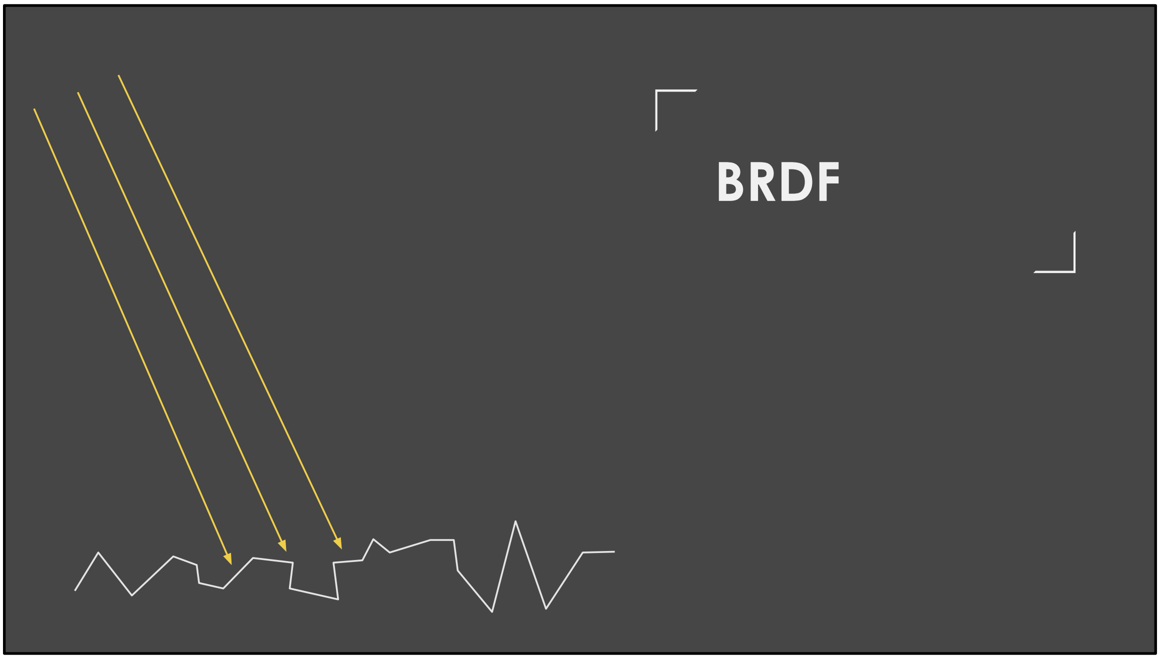
There is a catch to that: some metal however absorbs at specific wavelength, it's the case of gold. Because it absorbs only
some wavelength, the material appears tainted differently than the incoming light.

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.




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.

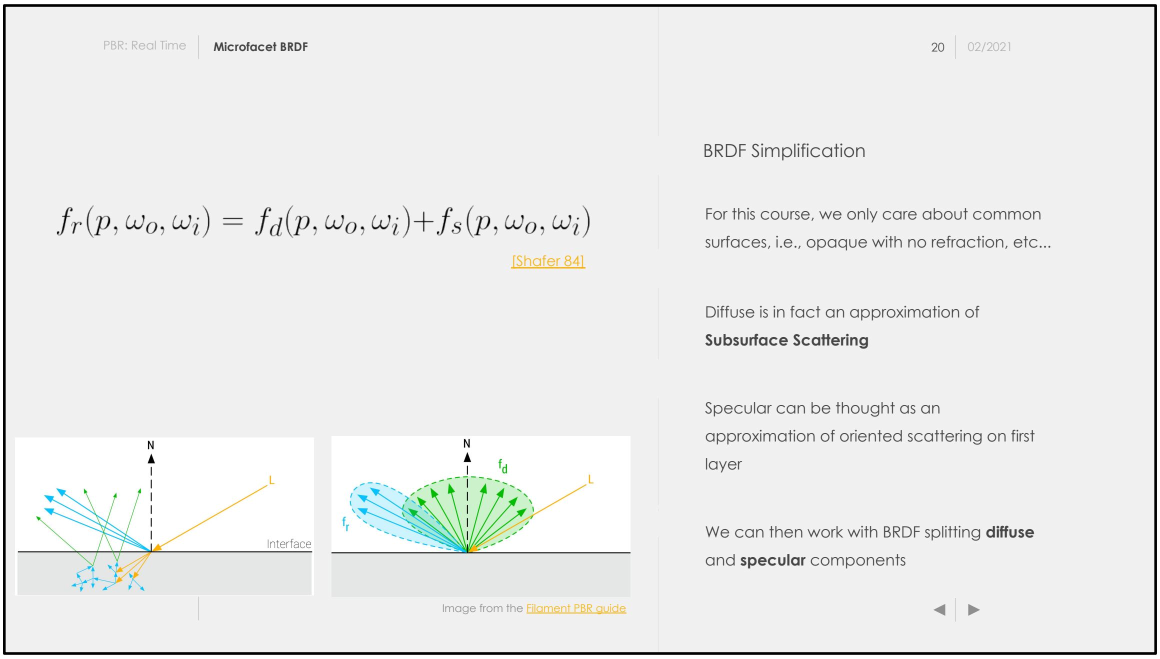
# Before Starting

Remember we said that a physical **BRDF** needs to respect 3 conditions.

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

$$\int_\Omega f_(p, \omega_o, \omega_i) = 1$$

$$\int_\Omega f_(p, \omega_o, \omega_i) = \int_\Omega f_(p, \omega_i, \omega_o)$$



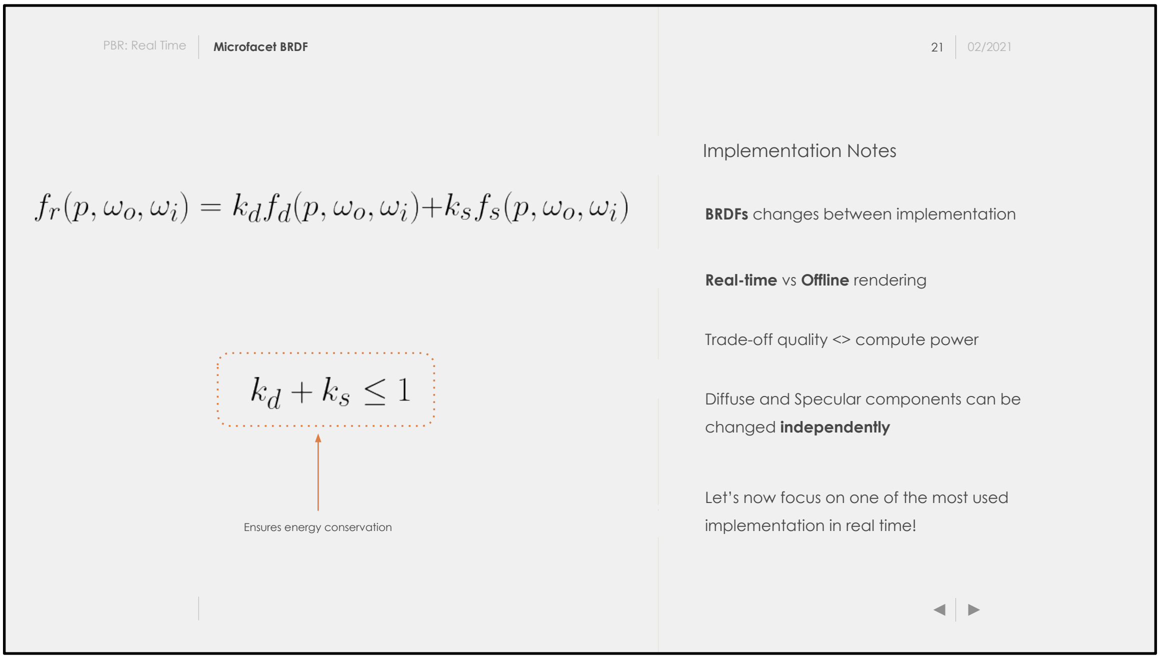

Remember that this course only cares about:

- 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.

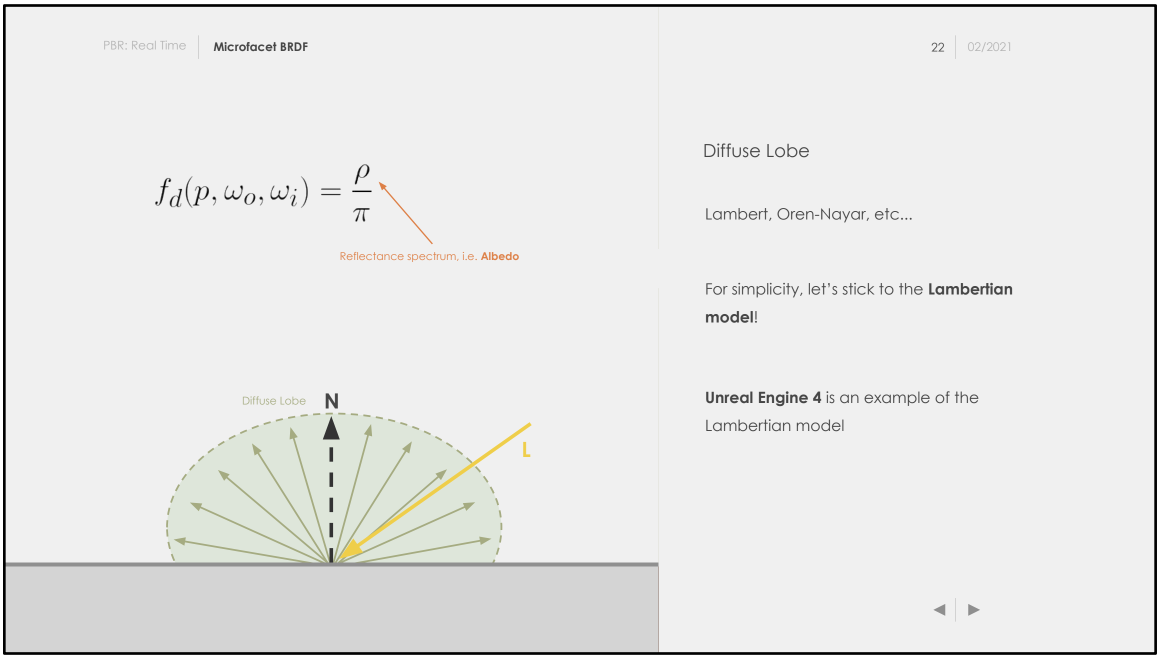



Before diving in concrete equations for the diffuse / specular parts of the BRDFs, I want to clarify something.
Online, you will find a **lot** of different names for microfacets **BRDFs** (GGX, Oren-Nayar, etc…). Each of those **BRDFs**
are made for either the diffuse or the specular components.

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.

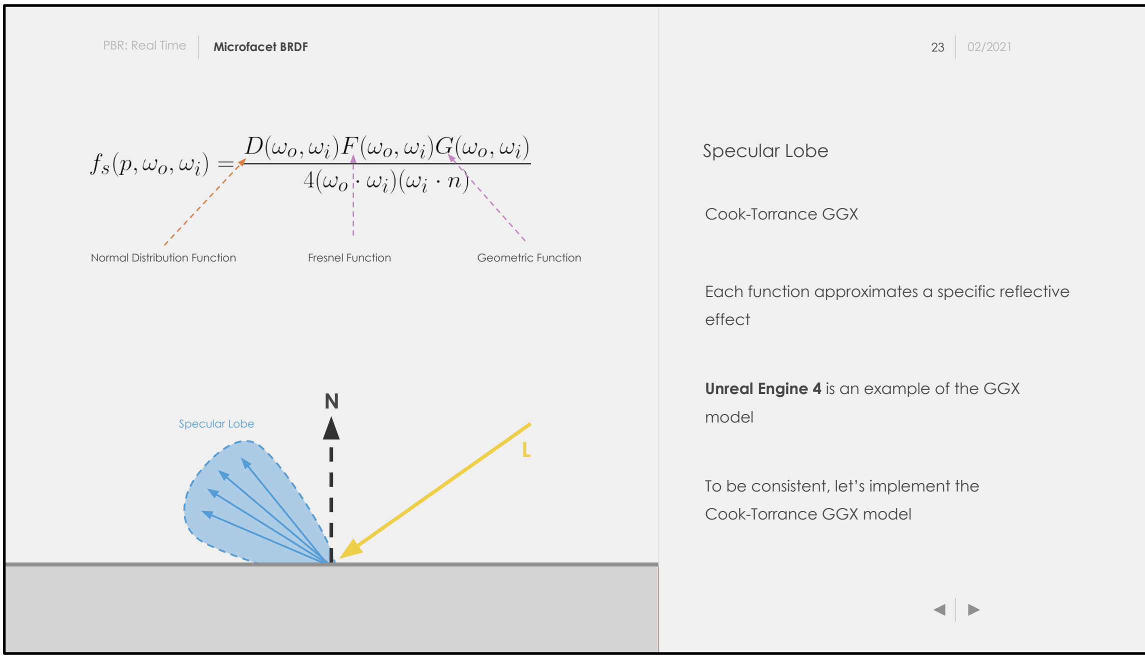
We will use the reflectivity of the material to ensure **energy conservation**. The *kd* and *ks* terms will be
used to weight both the diffuse and specular components. If we know the reflectivity of the material (*ks* term), we can
then find the *kd* term and weight the diffuse lobe accordingly.




**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.

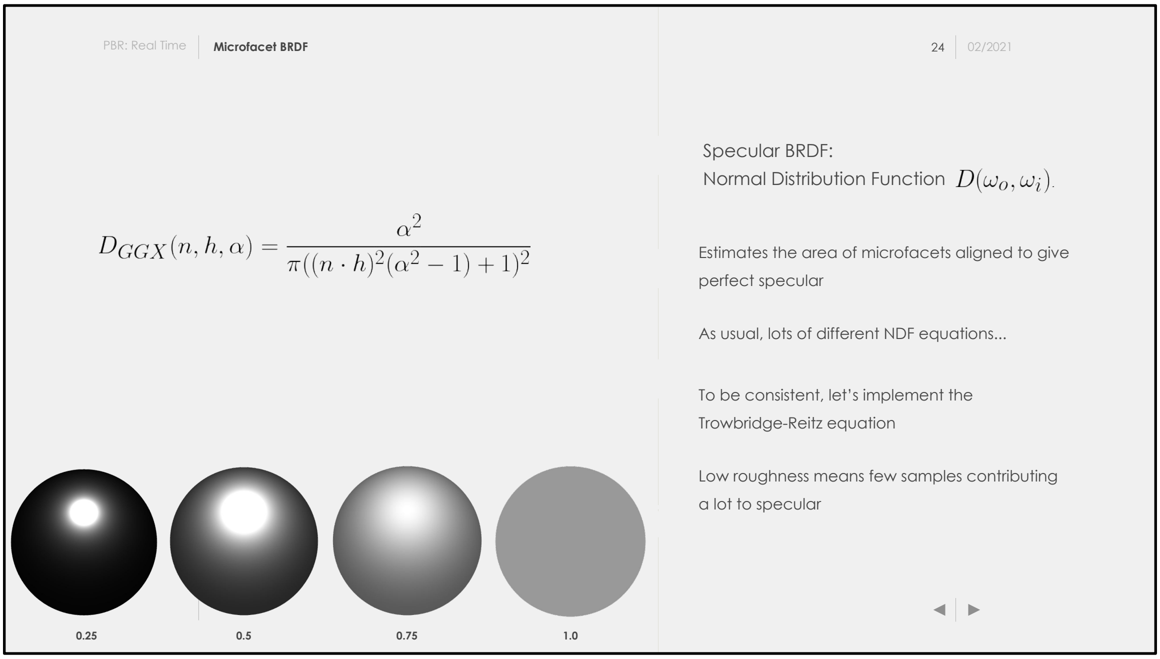
You may already have worked on some material on which you applied a texture to get more color information.
The albedo is basically the base color of the object. Just remember that it shouldn't contain **any extra** lighting information, i.e.,
no ambient occlusion, no shadowing, etc...

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



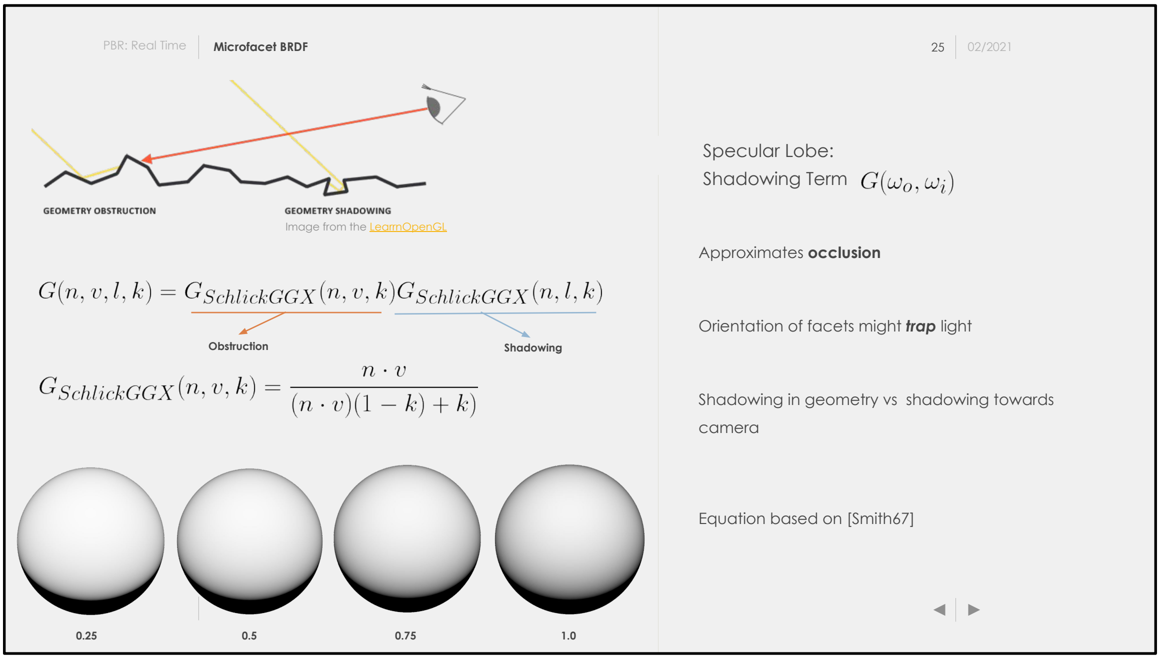

One of the most used specular BRDF is the Cook-Torrance. It's made out of three swappable terms.
Just remember: the *D, F,* and *G* functions **must be** normalized as well. No energy should be created by those functions.

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



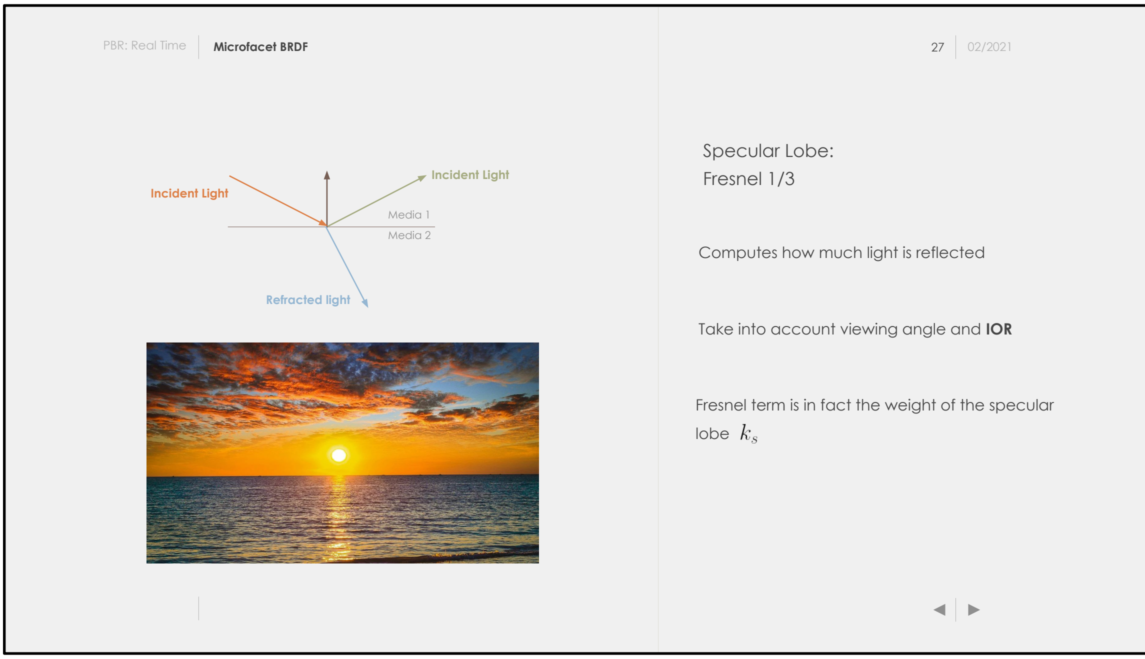

The **Normal Distrubution term** computes *"how much"* of the microfacets are aligned to the normal, maximizing specularity when the viewing angle is a reflection of the incoming light direction.

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.




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.




The **Fresnel effect** is the last piece of the equation.

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.

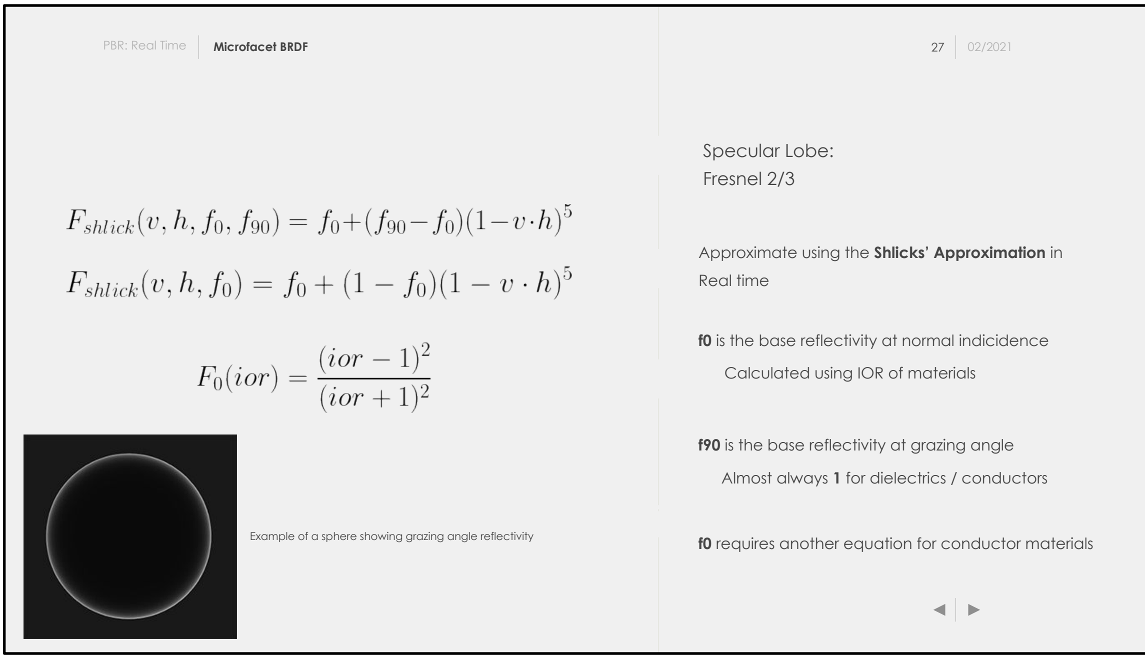
But why is that?

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).

It might not appear clearly, but 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.

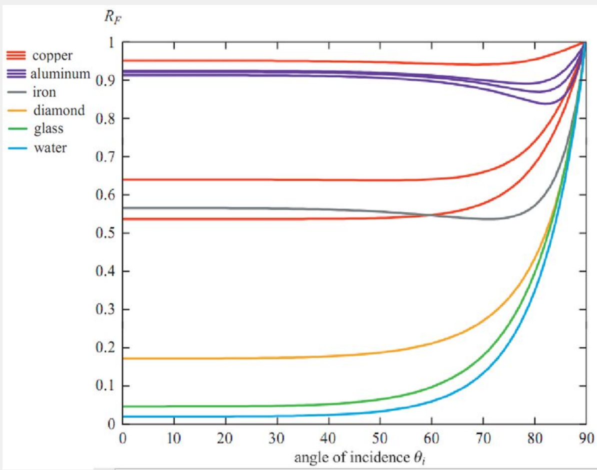



The fresnel effect is normally computed using the Fresnel equation.

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.




Specular Lobe:
Fresnel 3/3

Fresnel **reflectance** for common materials

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

Some materials **f0** ae tainted (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.

## Parameters Demo

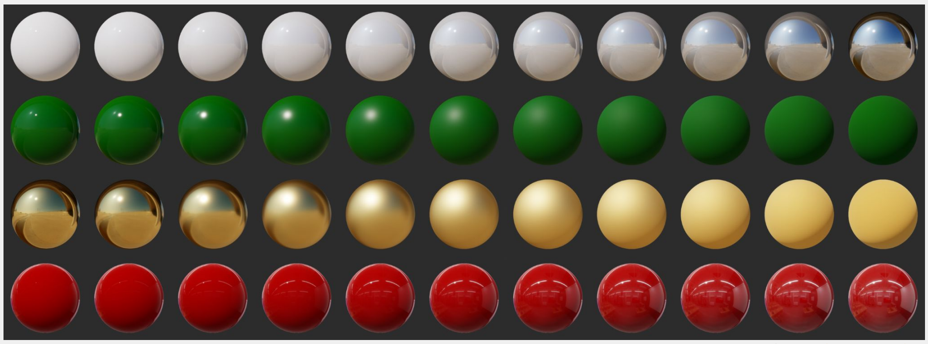
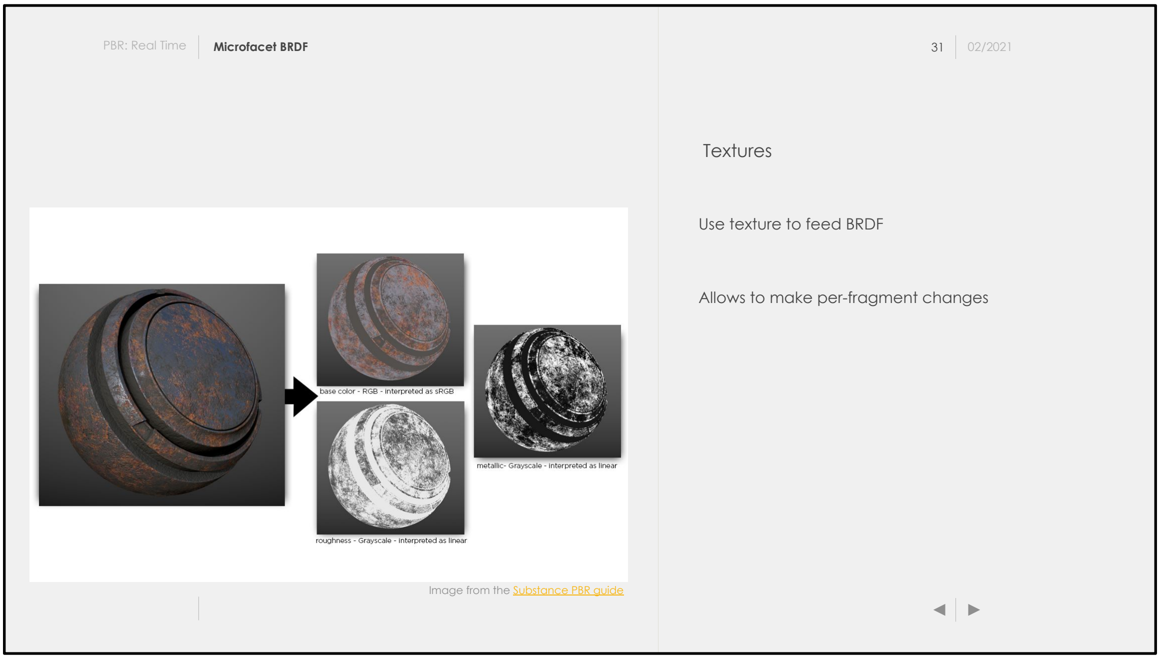


Image from the Filament PBR guide

# Direct-Lighting Pseudocode

```
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);

    radiance += (diffuseBRDFEval + specularBRDFEval) * sampleLight(lights[i], p, w_i) * dot(normal, w_i);
}
```




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**.

## To Remember!

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

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

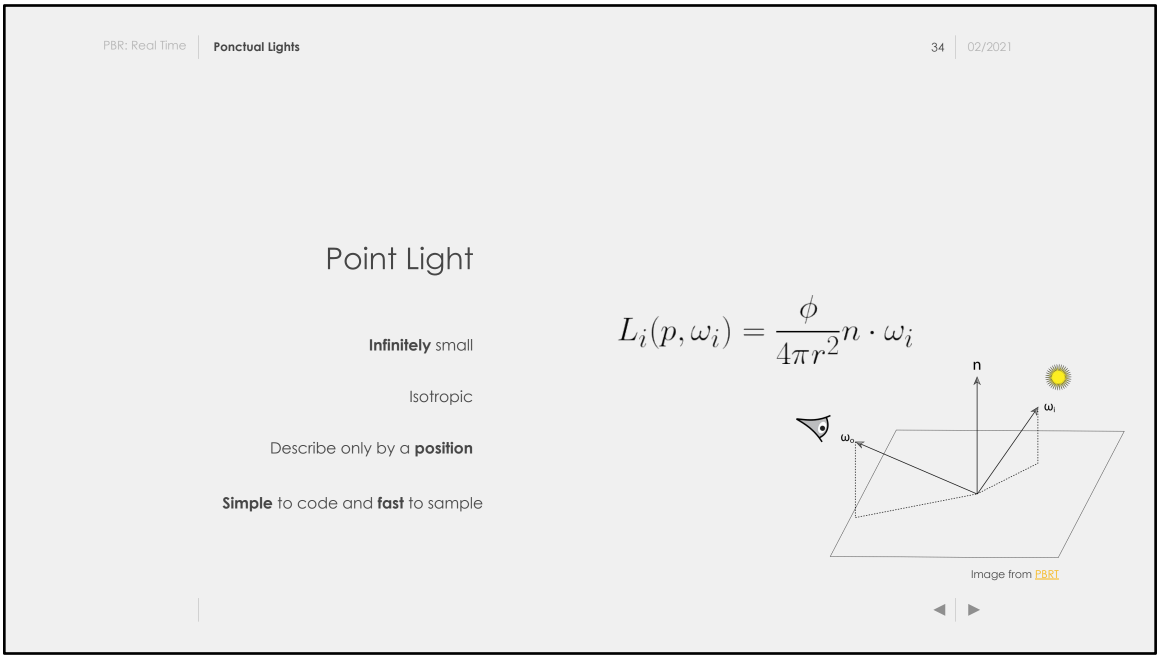
PBR implementations often (always?!) make use of this distinction

Simplify artist workflow and simplify the process of creating meaningful materials

## Ponctual Lights

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

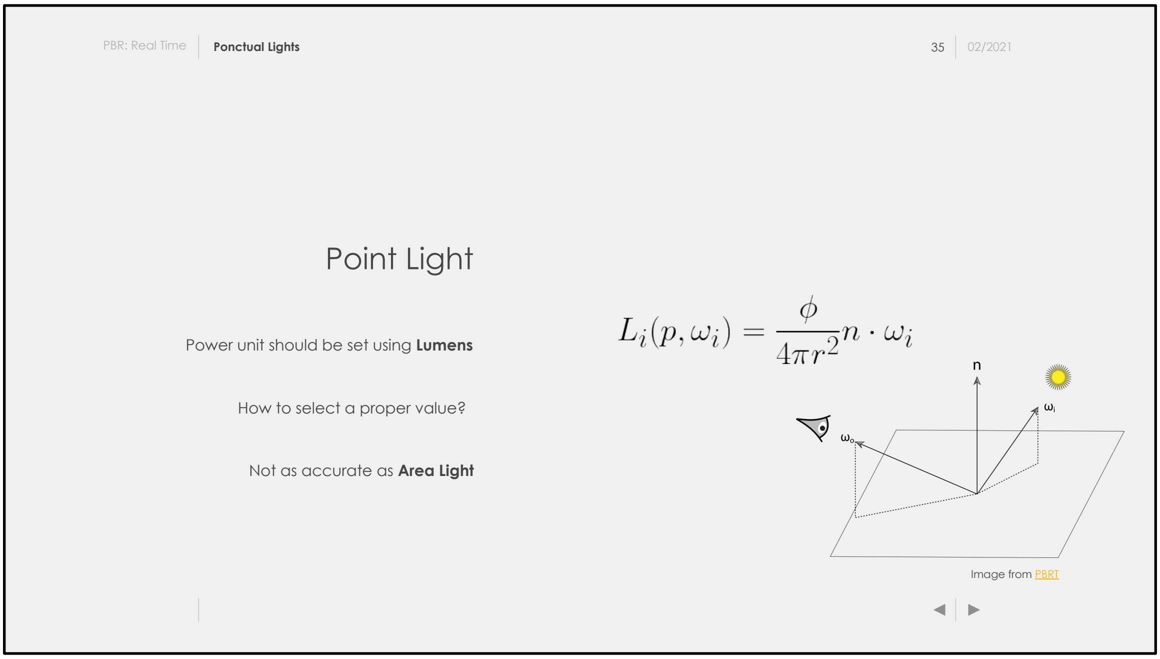
Implementing a microfacet model in an offline renderer works similarly. However, real-time constraints force us to either perform pre-computation ahead of rendering, or to approximate our equations more coarsely.




Point lights are approximations and don’t exist in the real world.

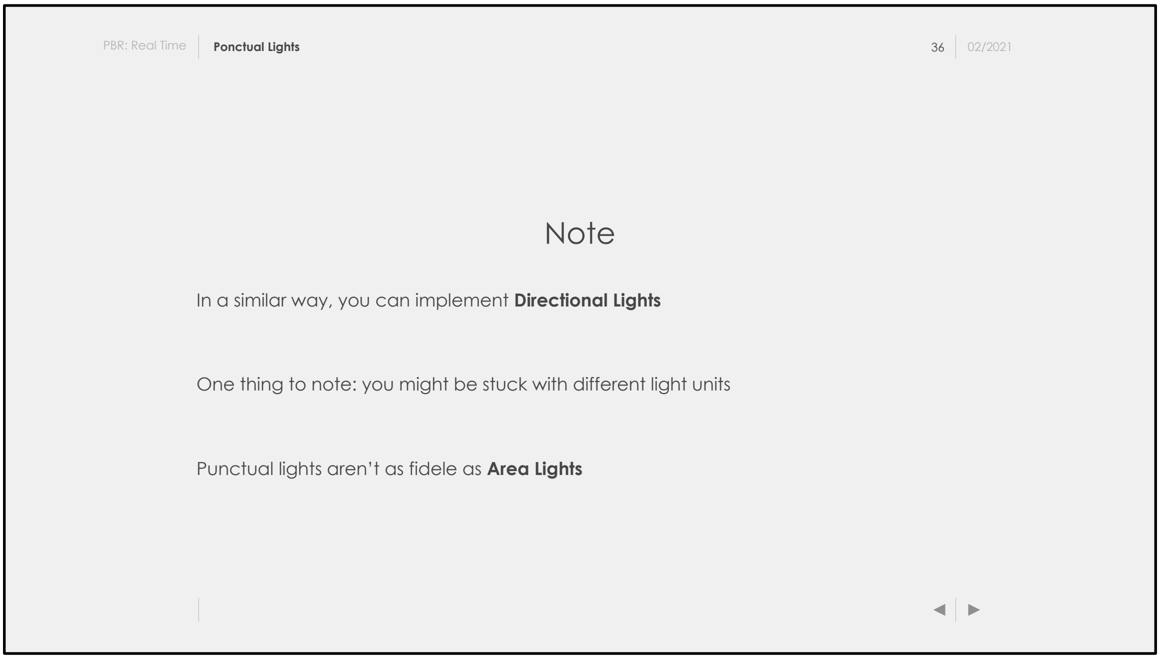
To get better results, it’s common to use Area Lights, which aren’t infinitesimal and represent
better the type of lights we use in the daily life.

However, due to the complexity of implementation and the computational heavy aspect that goes with it,
we will stick to punctual lights for now as they are enough to get pleasant results.




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.

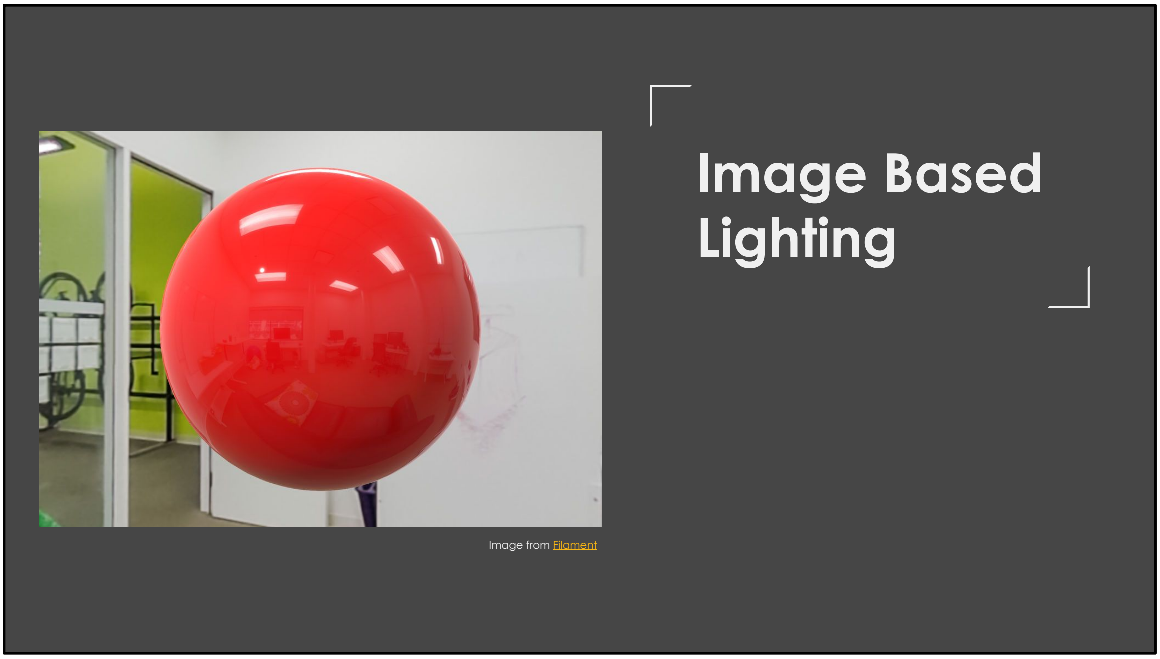



To get better results, it's common to use **Area Lights**, which aren't infinitesimal and represent
better the type of lights we use in the daily life.

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 :)




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.

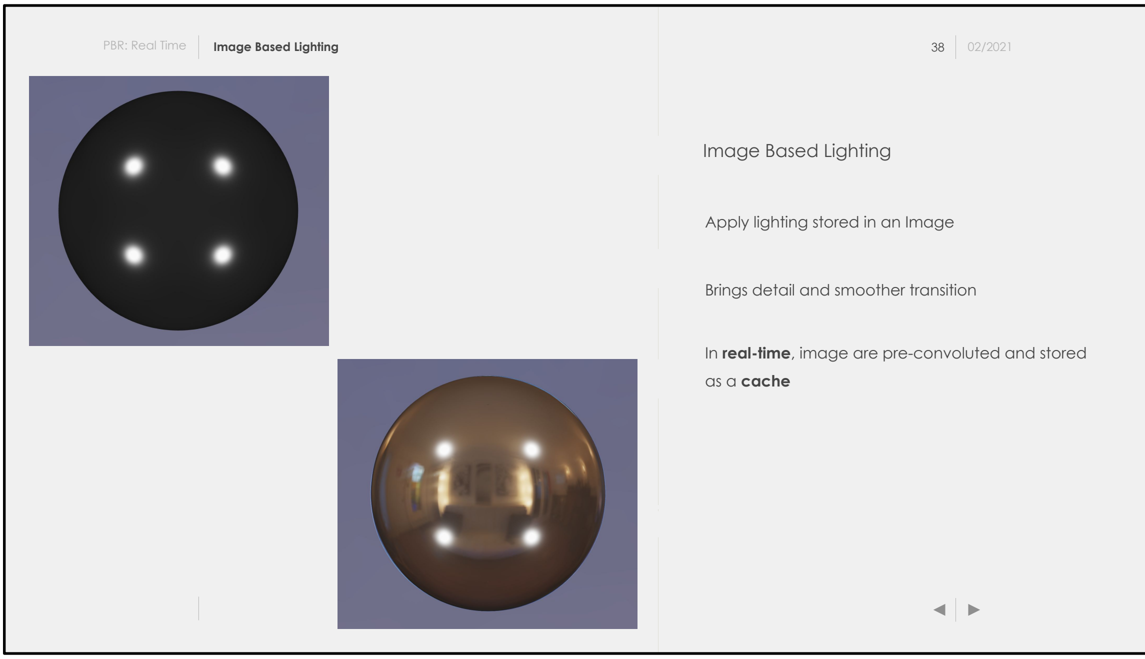
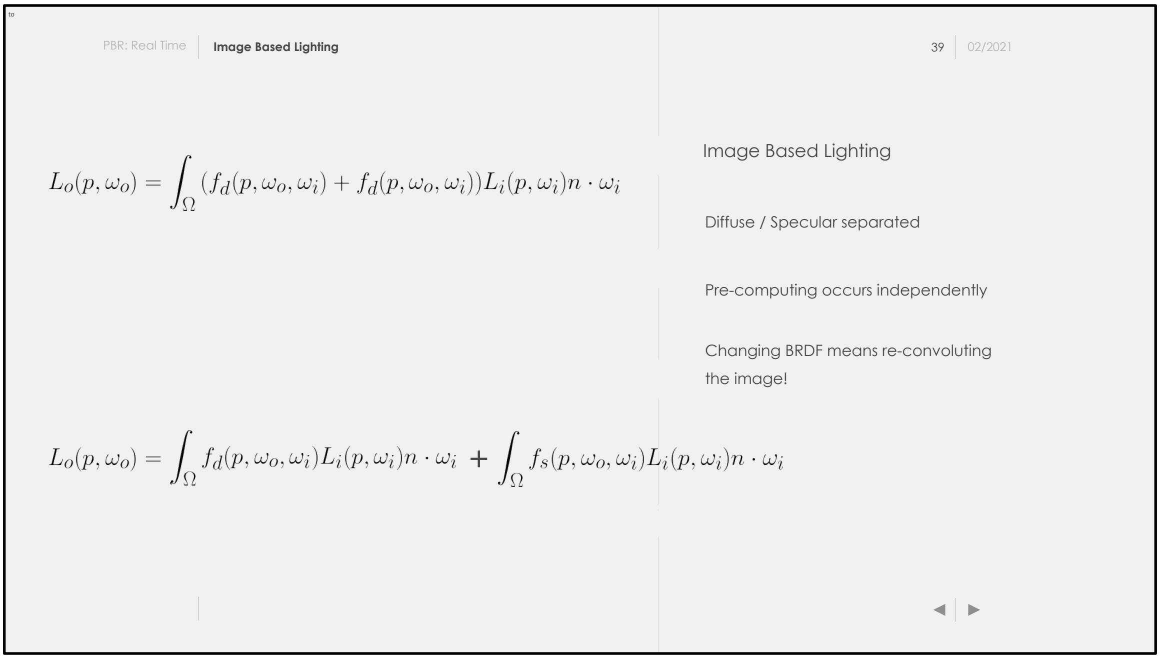



**Image-Based Lighting** is one of the most important type of light you can use to lit your scenes.
Instead of being lit by punctual lights, objects are lit by complex environment encoding a lot of data.
Those environment are simply 360 images.

If we go back to the rendering equation, you will recall that we need to integrate light incoming from all over the
oriented hemisphere. You can imagine that doing such computation is barely possible even on modern hardware.

In order to get **Image-Based Lighting** to work in real-time, the trick is to pre-compute as much as possible part
of the rendering equation, to reduce the problem to a few fetch in the final real-time shader.




Here we simply split the BRDF into the diffuse and specular part to be able to convolute them
separately. The specular needs view direction information, while the diffuse doesn’t.

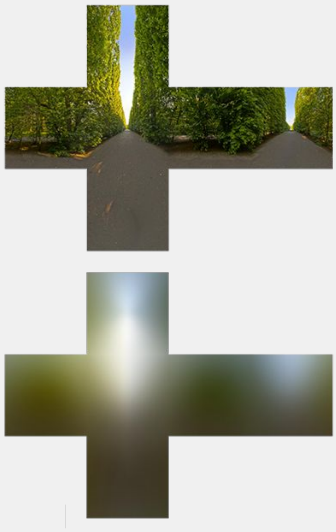
One thing to note: we will pre-compute the equation for **a given** diffuse and specular BRDFs.
This means that you can’t simply used the pre-computed environment with **any** BRDF when rendering.

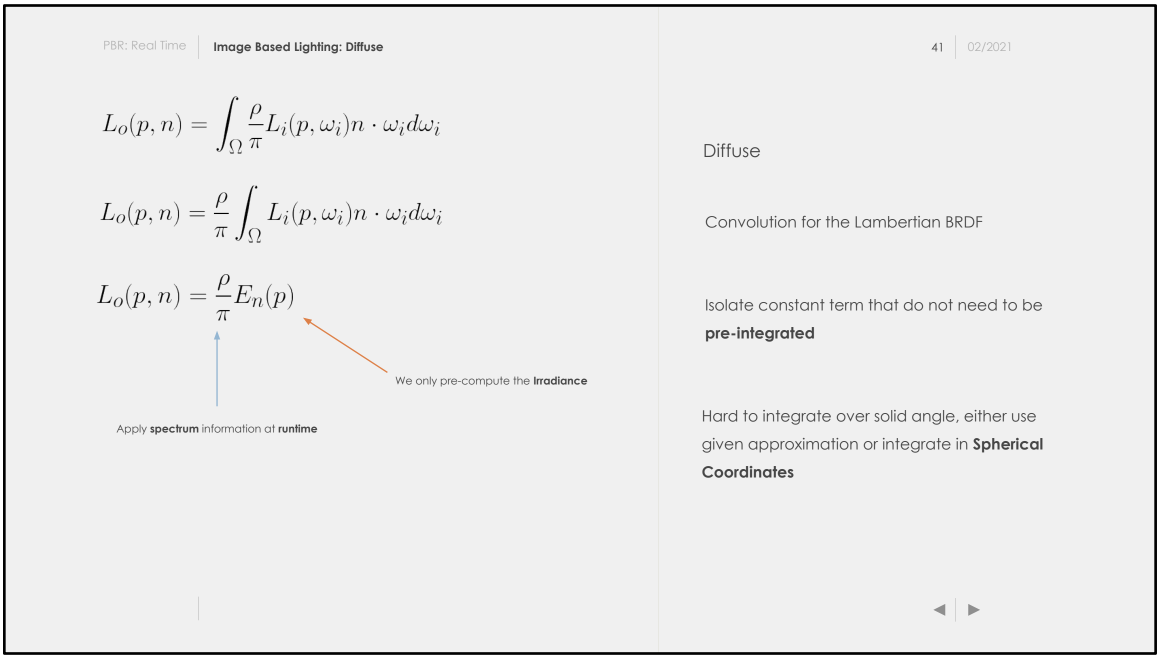
If you find some pre-computed environment online for different BRDFs that you aren’t using, you technically
shouldn’t be using them.

PBR: Real Time

$$\int_{\Omega} f_d(p, \omega_o, \omega_i) L_i(p, \omega_i) n \cdot \omega_i$$

# IBL Diffuse

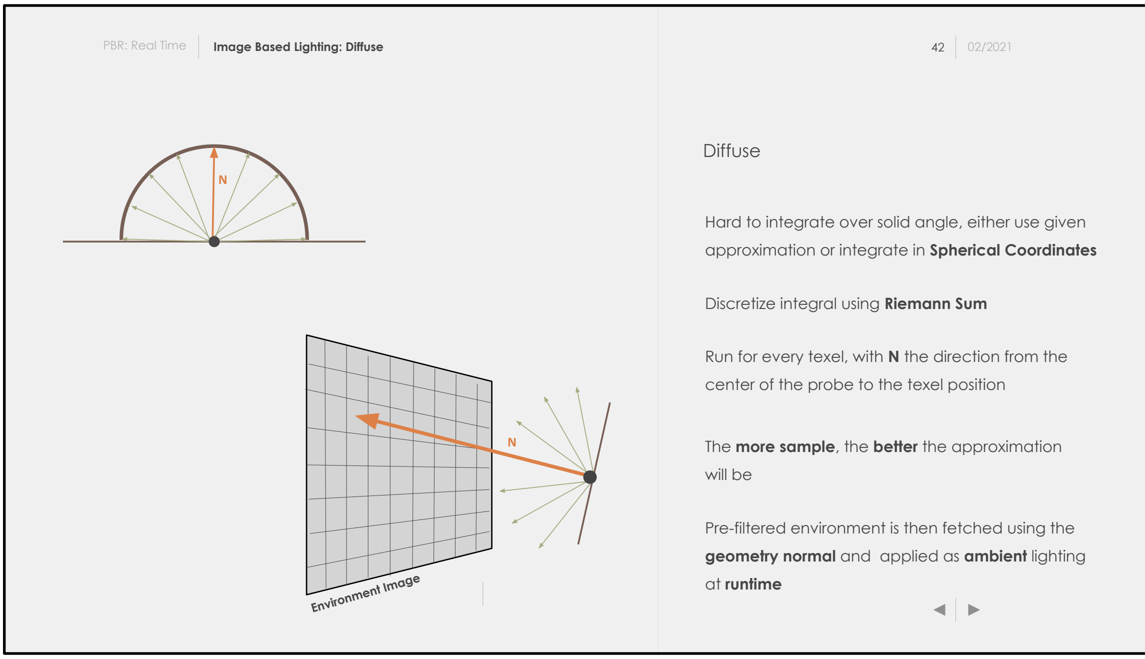





In order to convolute the diffuse lobe, we first isolate constants that can be applied at **runtime**.
Because it’s not easy to integrate the irradiance as-is based on the solid angle, we can either:

- 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.



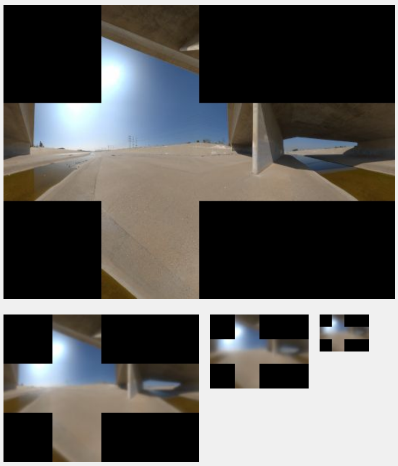

We can then use a Riemann Sum in order to compute an approximation of the integral. The idea
is to use discrete weighted samples. For the case of the irradiance map, it's enough to select uniformly distributed samples.

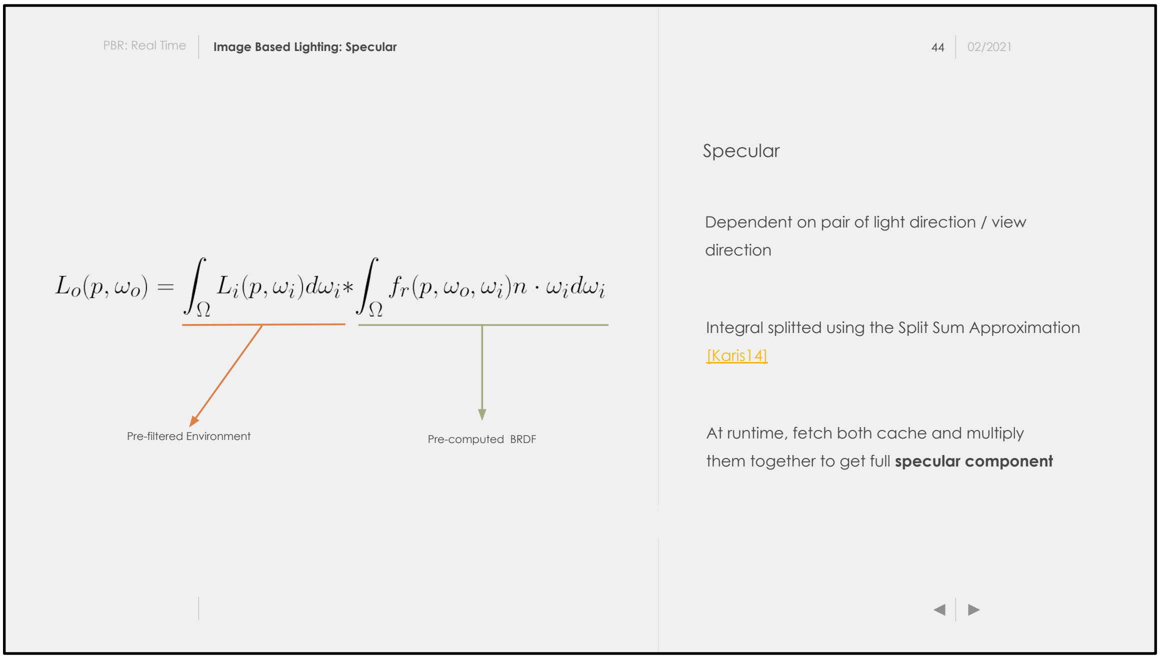
The two drawings on this slide explains how the algorithm work. For every texel of the environment map (cubemap, …),
you should compute the oriented hemisphere with normal the direction to the currently processed texel.
The rendering equation is then applied to neighboring texels that will contribute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

$$\int_{\Omega} f_s(p, \omega_o, \omega_i) L_i(p, \omega_i) n \cdot \omega_i$$

# IBL Specular

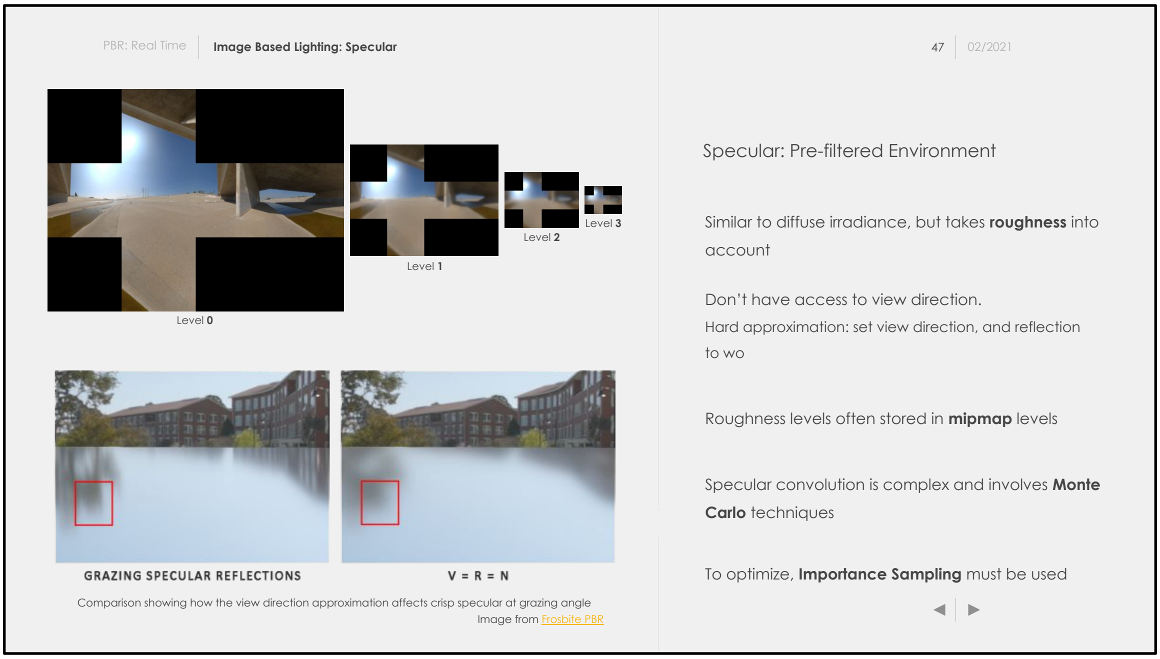





Pre-computing the specular component is much harder because it depends on more variables.
Trying to generate all combination isn't feasible and wouldn't make sense in a real-time application.

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 present 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.




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 blurier (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.

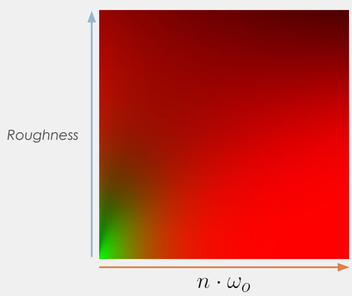
However, There is one major difference with the diffuse component. The specular component isn’t spread uniformly in the hemisphere.
Quite the opposite, low roughness values means narrow specular lobe. Using uniformly distributed samples, we would end up
with many samples not contributing at all to our final result, which is wasteful.

We unfortunately don’t have time in this course to go through it, but the idea is to

use **Monte Carlo** Integration with non-uniform **Importance Sampling.**
Basically, we would transform the integral to solve into a statistical problem, where samples would be taken randomly but biased toward the specular reflection.

## Specular: Pre-computed BRDF

$$\int_\Omega f_r(p,\omega_o,\omega_i) n\cdot\omega_i d\omega_i = F_0 \underline{\int_\Omega f_r(p,\omega_o,\omega_i)(1-(1-\omega_o\cdot h)^5) n\cdot\omega_i d\omega_i} + \underline{\int_\Omega f_r(p,\omega_o,\omega_i)(1-\omega_o\cdot h)^5 n\cdot\omega_i d\omega_i}$$




This image contains the pre-integrated BRDF data. You can re-use it as-is, as long as you use the same BRDF

The above equation is obtained by substituting **Fresnel Shlick**: [Karis14]

Only two inputs left: **roughness**, **viewing angle**

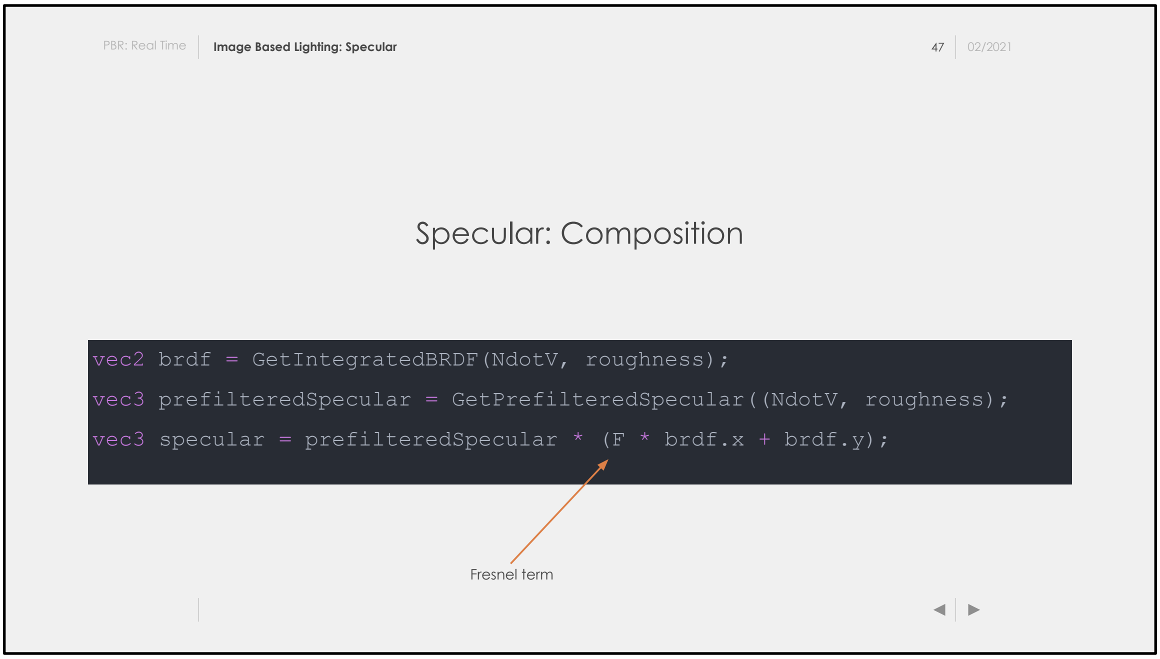
At runtime:

1. Fetch **pre-integrated** BRDF texture
2. Fetch convoluted environment
3. Apply the above equation to get the full specular component

◀ | ▶

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.




All the work described on the previous slides is used at **runtime** with those three lines of code.
This code simply applies the equations we have seen on the two previous slides.

## To Remember

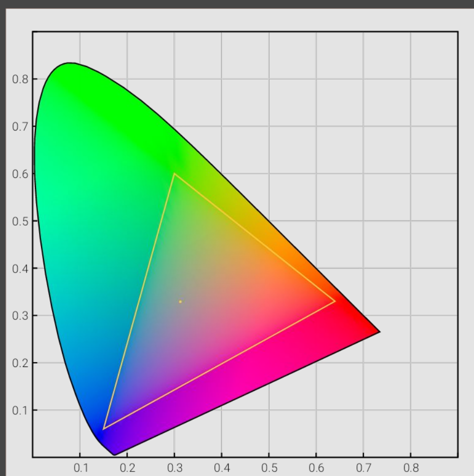
Real-Time Image Based Lighting consists in **pre-filtering** / **caching** as much as possible

Specular pre-filtered environment is stored in mipmaps depending on the roughness level

Always use HDR environment to get proper lighting intensity

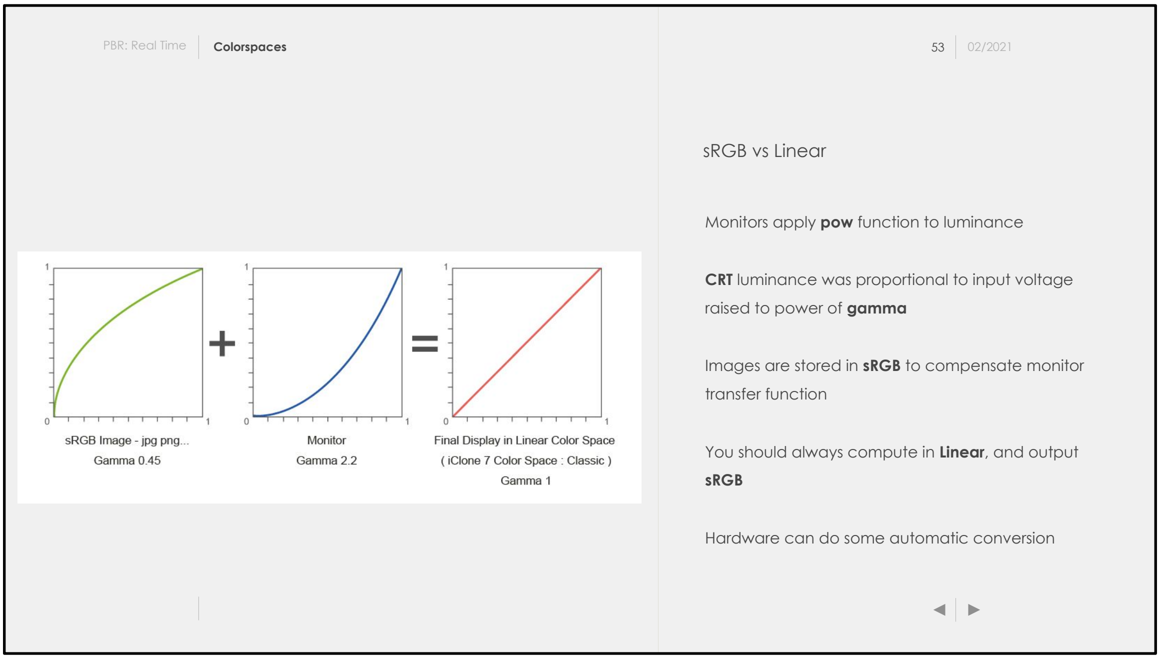
IBL generation isn’t an easy process. Do not worry if you think the topic is complex. For the assignment, you will not be asked to pre-filter your environments.

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




sRGB Colorspace primaries

# Colorspace & Color Precision




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:
*pow(x, 1/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.

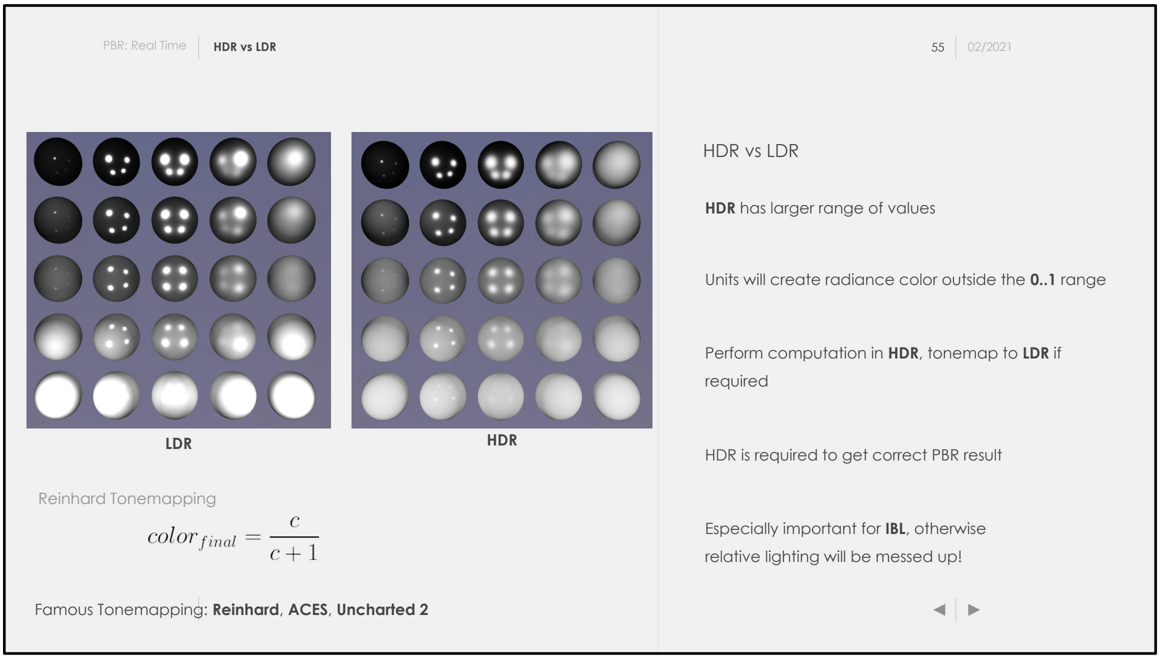
Stay consistent, i.e, always work in the same color space.

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 occuring when doing PBR. It’s a general rendering topic that every graphics programmer
needs to be aware of.




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.

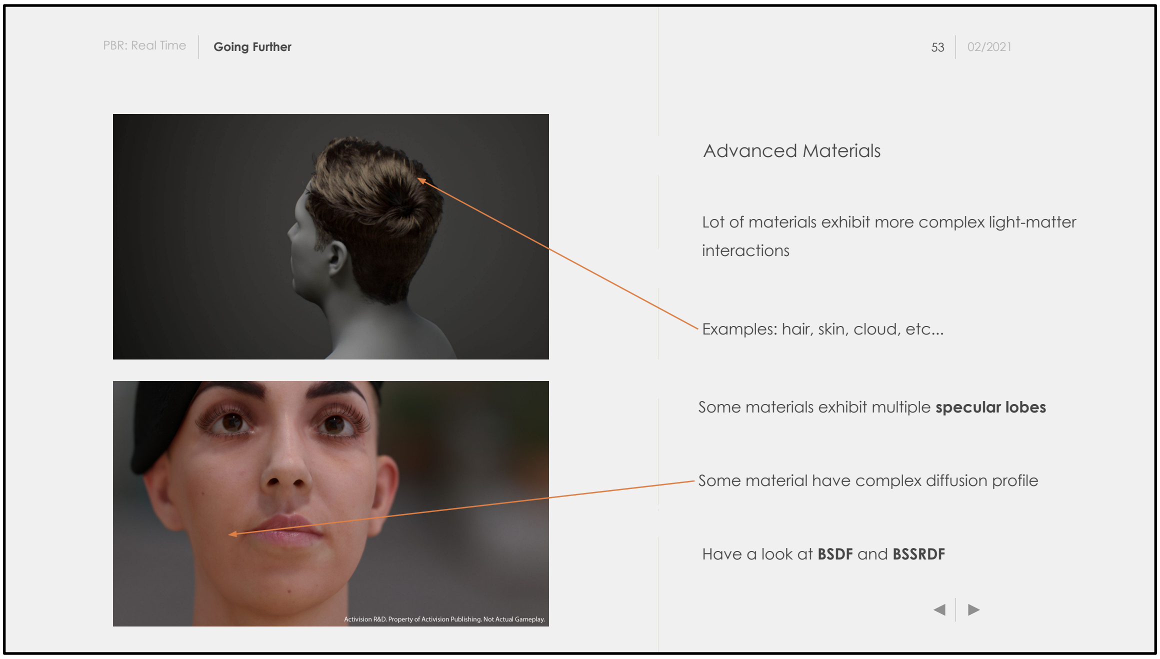
Because we now work with real physical quantities, the final pixel color after running all computation will likely be out of the range **0...1**
Light bulbs for instance are already with thousands of Lumens. If we render a scene just like that, we will end up with most of the fragments
saturated.

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




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
- PBRT: From Theory to Implementation

◀ ▶

# 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-Shadowing 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 Frosbite to Physically Based Rendering 3.0
- [Hoffman10], N. Hoffman, Physics and Math of Shading

David Peicho

# Thanks

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