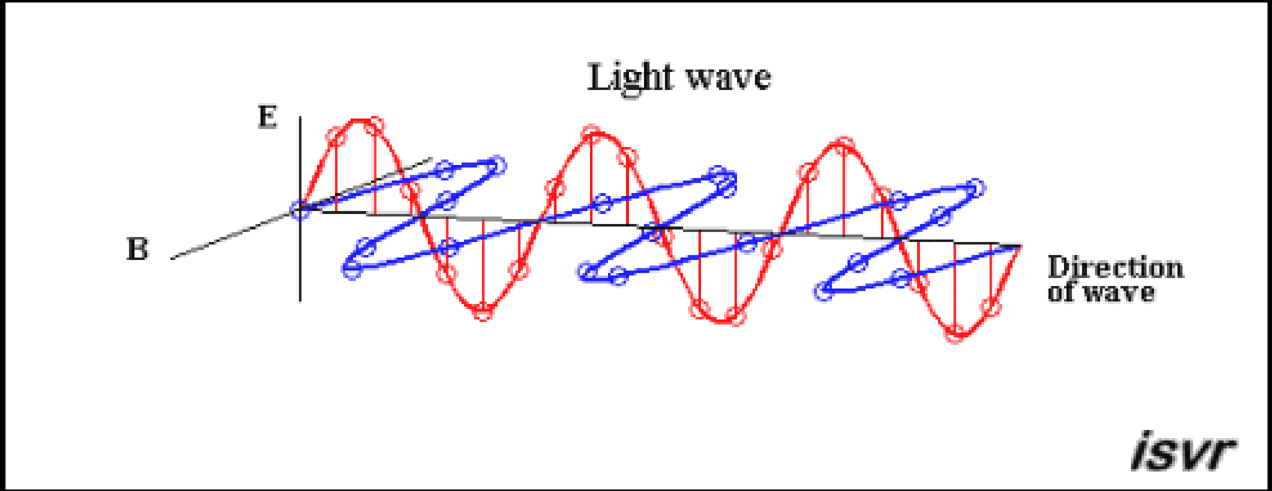
GRAPHICS PROGRAMMING I

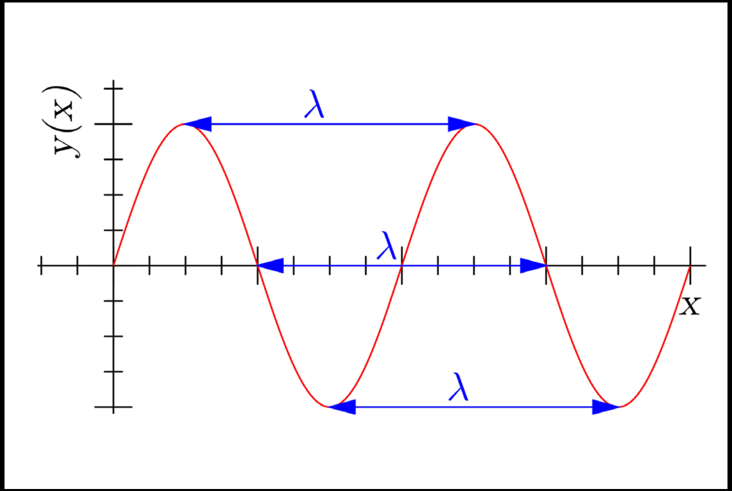
LIGHTING

Ray Tracing| Physics of Light

- Before we go into how we are going to model light in our 3D ray tracer, we have to know what light is and how we can formalize it!
- In physical optics, light is modeled as an **electromagnetic wave**. It consists of both electric and magnetic fields, that are **perpendicular** to the direction of the **propagation**.
  It has a **(wave)length (λ)**, that can be measured between any two points with the same phase. The wavelength is strongly related to how we perceive the color of light.



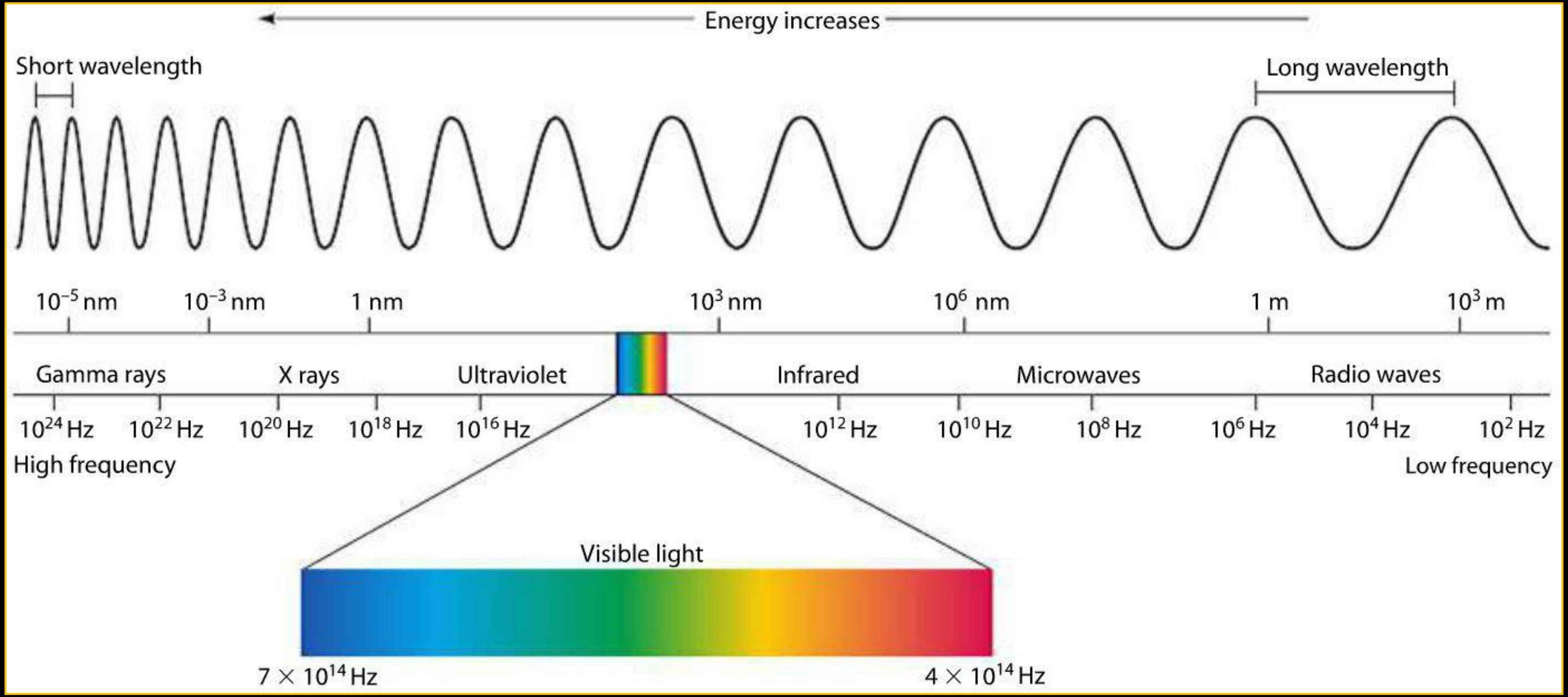
1



1. University of Southampton

Ray Tracing| Physics of Light

- The human eye can only see a small portion of the **spectrum**, which is determined by the wavelength.
- Most light waves have multiple wavelengths to create a variety of colors. There are called: **polychromatic**. Light waves with one wavelength are called: **monochromatic**.

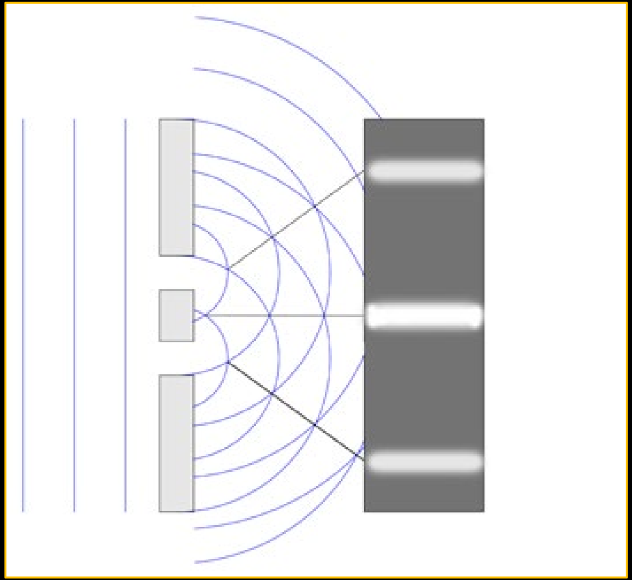
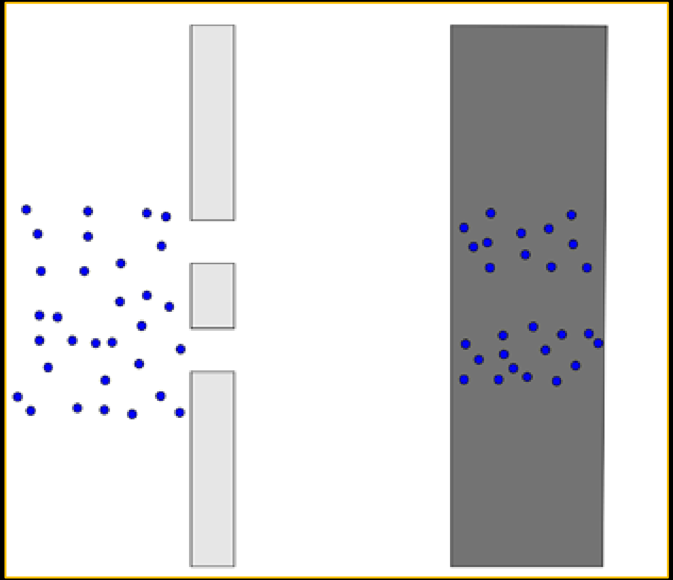
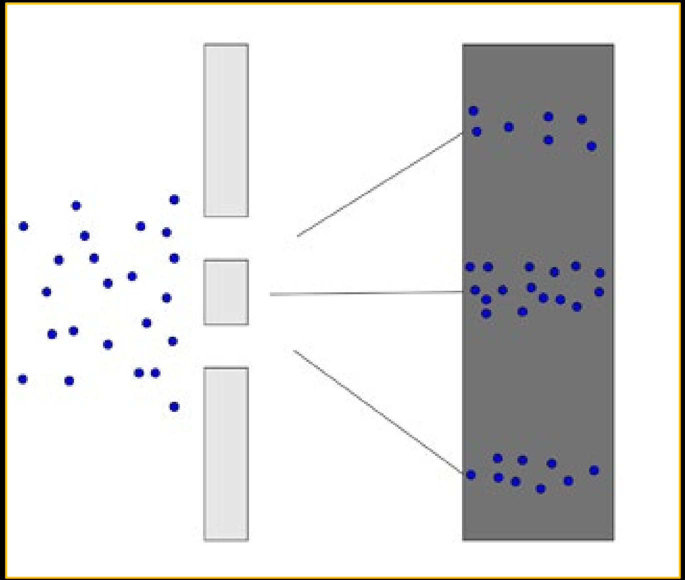


1

1. Mini Physics

Ray Tracing| Physics of Light

- Let's dive a bit deeper, into the quantum realm, and take a look how light behaves. Behold the double slit experiment and its weird result!

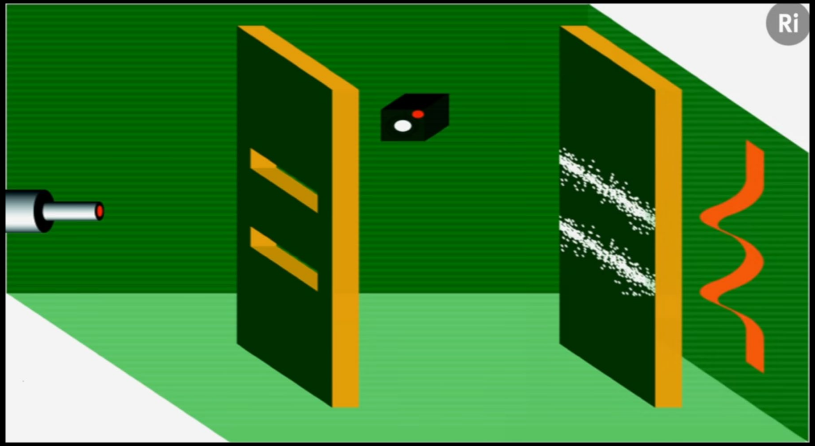
1 



- Photons are the fundamental particles of light. They have the unique property of behaving like both a wave and a particle, depending how you observe them ☺
  - Particle that behaves like a wave? Wave that behaves like a particle?

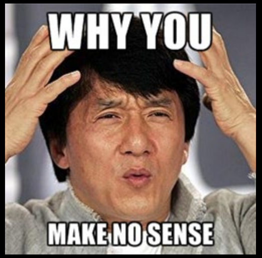
1. https://plus.maths.org/content/physics-minute-double-slit-experiment-0

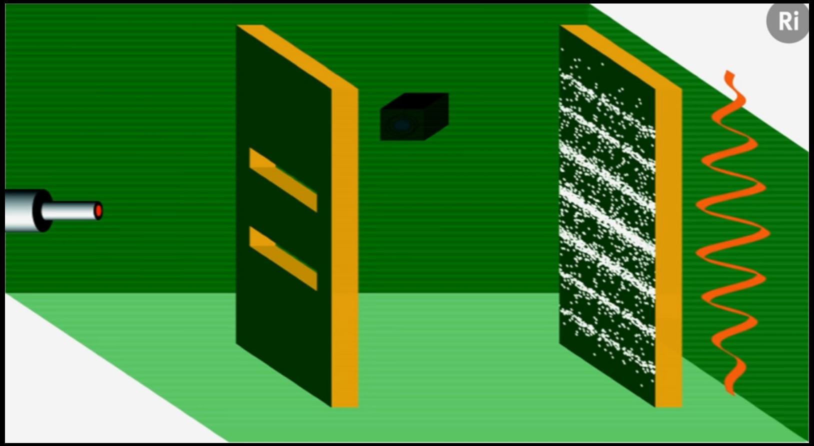
Ray Tracing| Physics of Light

- Let's dive a bit deeper, into the quantum realm, and take a look how light behaves. Behold the double slit experiment and its weird result!



Single Shot Photons/Atoms
Observed > Behaves as particles





Single Shot Photons/Atoms
NOT Observed > Behaves as wave

https://www.youtube.com/watch?v=A9tKncAdlHQ

Ray Tracing| Modeling Light

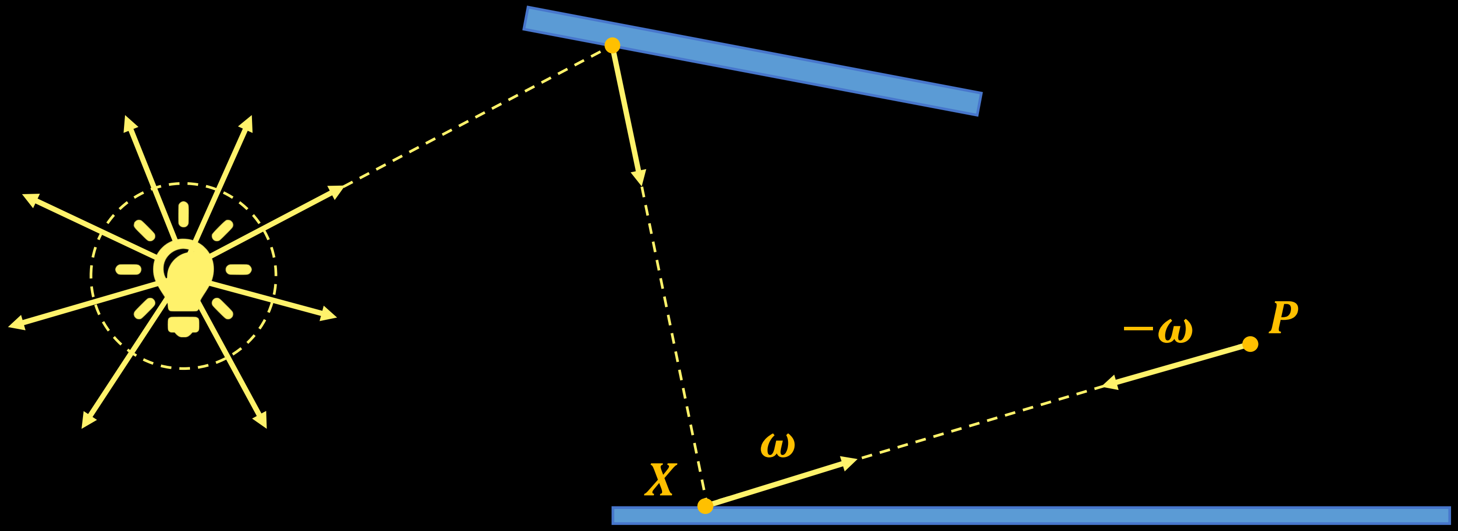
- This is all too complicated for us… Let's just think of light as a bunch of particles propagating in a certain direction. We'll call this a light field $\boldsymbol{L(P, \omega)}$, where $\boldsymbol{P}$ is the origin/position of the light and $\boldsymbol{\omega}$ is the direction of propagation.
- When light propagates in a medium (air, fog, water, etc.) it can get absorbed or scattered. But for now, let's neglect absorption or scattering, and let's think of light traveling in a vacuum. So, we'll think of air as a vacuum from now on (the absorption in "clean" air is negligible).

  For your information, only electromagnetic waves can travel through vacuum, mechanical waves, like sound, can't.

- As we've mentioned above, in a vacuum, light travels in a certain direction. In other words, it travels along a ray. In a vacuum, light will not loose any energy. So, the amount of energy is conserved along the ray.

Ray Tracing| Modeling Light

- This preservation means that all light that leaves $\boldsymbol{X}$ in direction $\boldsymbol{\omega}$ must reach point $\boldsymbol{P}$.

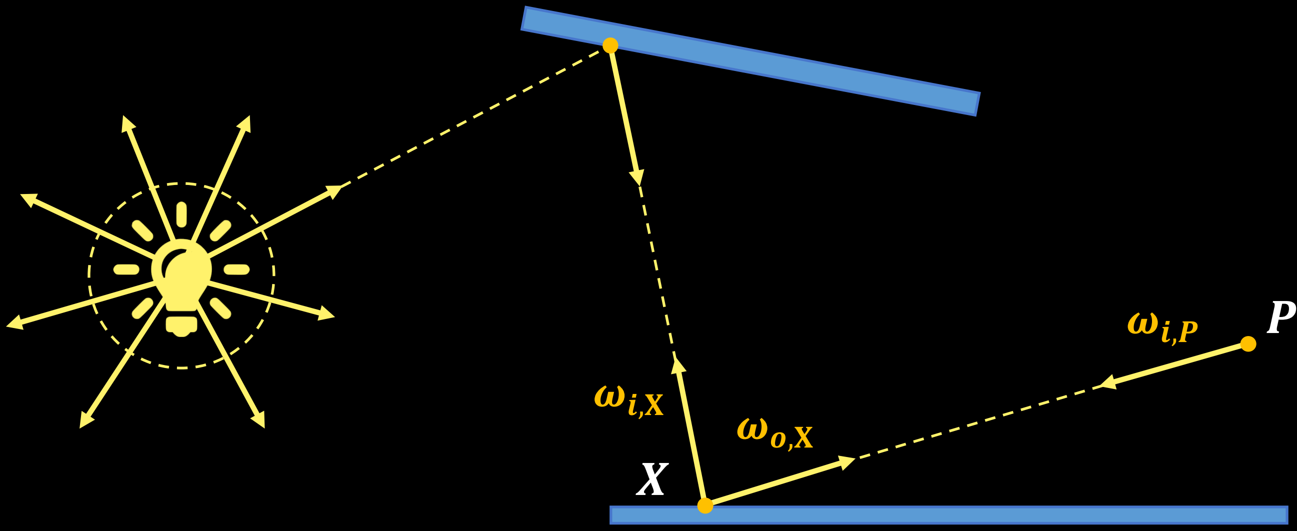


$$L(P - \omega, \omega) = L(X + \omega, \omega) \rightarrow \omega = P - X$$

Ray Tracing| Modeling Light

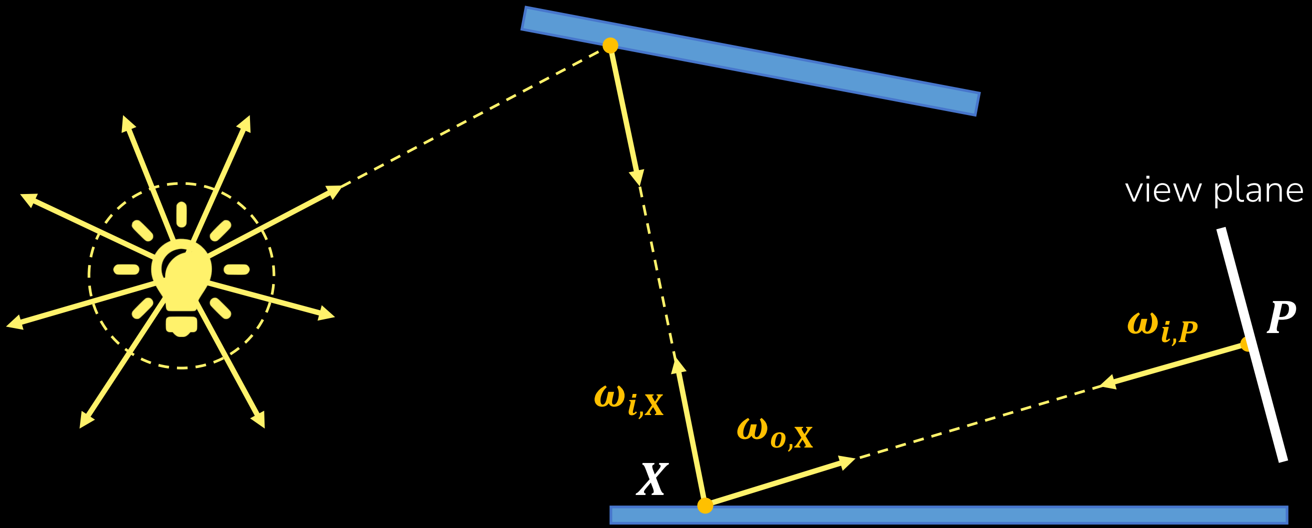
- This notation can be a bit annoying, so usually we will denote it as:
  - the outgoing light from point X in direction ω: $\boldsymbol{L_o}(X, \omega) = L(X + \omega, \boldsymbol{\omega})$
  - the incoming light to point X from direction $-\omega$: $\boldsymbol{L_i}(X, \omega) = L(X - \omega, -\boldsymbol{\omega})$

- So, we can write the preservation as: $\boldsymbol{L_i}(\boldsymbol{P}, \boldsymbol{\omega_{i,P}}) = \boldsymbol{L_o}(\boldsymbol{X}, \boldsymbol{\omega_{o,X}})$



Ray Tracing| Modeling Light

- Point P can be a point on our view plane we want to calculate. In other words, we want to calculate the incoming light of that pixel that holds point P.
- With our current raytracer, we know point X is visible. It's this point we want to render. So, we are interested in the amount of light point X receives and how much is visible from point P. With the knowledge of conservation, we can start building an equation.

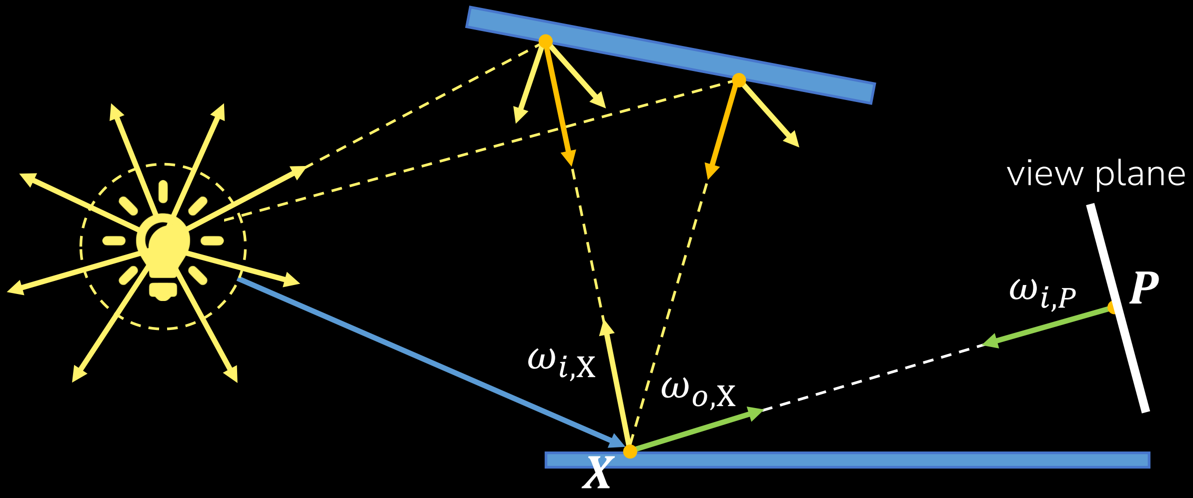


Ray Tracing| Modeling Light

- Point P is the sum of the light scattered (by incoming lights, $\boldsymbol{L_i}$) at X in direction $\omega_{o,\mathrm{X}}$ and the light emitted ($\boldsymbol{L_e}$) from X in the same direction.

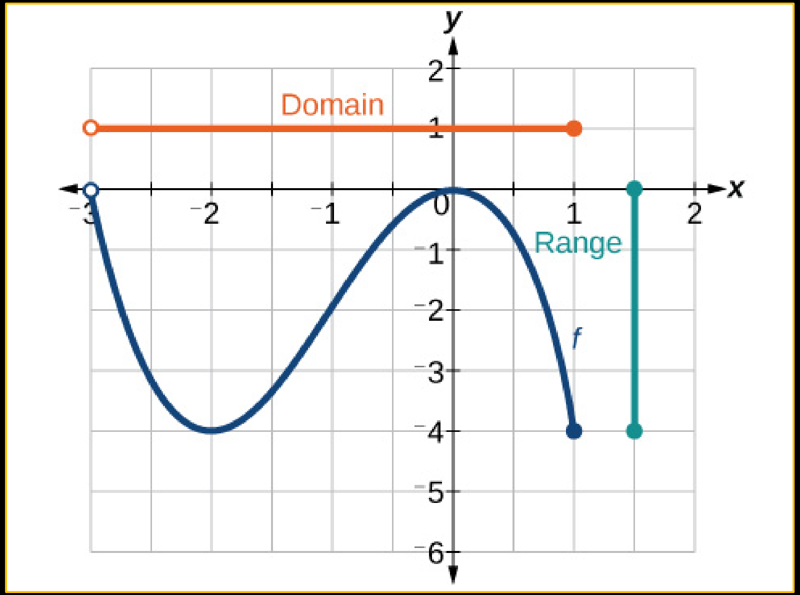
$$L_i(P, \omega_{i,P}) = L_o(X, \omega_{o,X}) \quad \rightarrow \quad \boldsymbol{L_o}(X, \omega_{\mathrm{o,x}}) = \boldsymbol{L_e}(X, \omega_{\mathrm{o,x}}) + \int \boldsymbol{L_i}(X, \omega_{\mathrm{i,x}}) \text{...scattering...}\, \mathrm{d}\omega_{i,x}$$

- In real life, light comes from many different places, both surfaces and light fields. Because of our limited amount of time, we will only focus on direct illumination from light fields. But to get a physically correct image, we need to consider all possible points from which light might reach point X.



Ray Tracing| Radiometry

- Up until now we didn't talk about units. How do we quantify light?
- In Computer Graphics we will largely follow the conventions of optics in physics, called radiometry.
- Radiometric quantities exist of both domain and range units:
  - Domain:
    - Individual = directions, positions, etc.
    - Sets = solid angles, areas, etc.
  - Range:
    - Energy, power, radiance, etc.
- A lot of people fear radiometry and the used units, but there is nothing to be afraid of!



1

1. Lumen Learning

Ray Tracing| Radiometry

- Let's start with some units used in physics you all should know from high school:
  - A mass $\boldsymbol{m}$ is measured in kilograms (kg).
  - Distance $\boldsymbol{x}$ is measured in meters (m).
  - Time $\boldsymbol{t}$ is measured in seconds (s).
  - Acceleration is expressed as $\boldsymbol{a = x/t^2}$ (m/s^2).
  - Force is expressed as $\boldsymbol{F = ma}$ and is measured in newtons (N = kg * m/s^2).
  - Work or Energy is expressed as $\boldsymbol{E = Fx}$ and is measured in Joules (J = N * m).

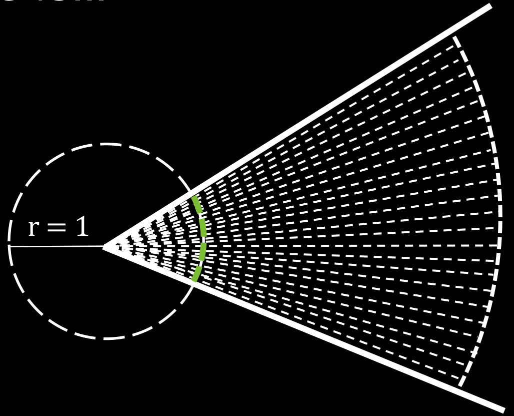
- Based on what we already discussed, light can be seen as an infinite beam of photons that are moving extremely fast (at the speed of light ☺). These photons have energy. The energy in light is measured, as any other form of energy, in Joules.
- We are interested in light (energy) hitting a point. As you can imagine, the energy of the light flows through the point, as a stream of photons hit the point over time. To make our equation time independent we will instead consider the rate at which energy flows through the point. This rate is called Radiant Power or Radiant Flux, and is measured in Watt ($\boldsymbol{W = J/s}$).

Ray Tracing| Radiometry

- Radiant Flux is also denoted as ϕ. Also, for clarification, Radiant Flux is energy in a restriction of time.
- When we are interested in the incident power measured on an entire (oriented) surface, instead of a single point, we denote it as a function: $\phi(\mathbb{A})$. This is called Radiant Flux Density, which is Radiant Flux in a restriction of space, or Energy in a restriction of time and space. It's measured in W/m^2.
- The Radiant Flux Density that arrives at a surface (point) from all directions is often called Irradiance(E), and is also measured in W/m^2. It can be calculated as $E = \frac{d\phi}{d\mathbb{A}}$.
- Radiosity is similar to Irradiance, but instead it measures the spatial rate at which Radiant Flux leaves the surface in all directions.
- If we want to measure light along a ray, we will need a way to restrict the direction along which it is measured. So, we are interested in the Radiosity from or Irradiance to one direction instead of all directions.

Ray Tracing| Radiometry

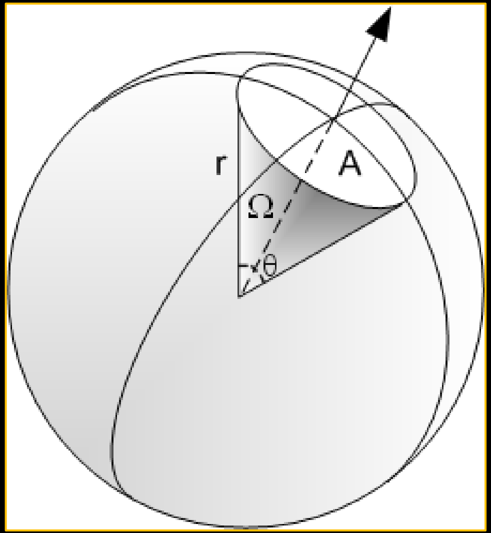
- When we measure light with a restriction to time (Radiant Flux), we divided the energy (J) by time (s), giving us a unit measured in $\boldsymbol{W = J/s}$. To then restrict it to a single point we divided the unit by the area measure $\boldsymbol{m^2}$, giving us irradiance in $\boldsymbol{W/\,m^2}$. So what should we divide by to restrict our measurement to a single direction?
- Let's take a look what an angle is…



- Quoting Real-Time Rendering: "An angle can be thought of as a measure of the size of a continuous set of directions in a plane, with a value in radians equal to the length of the arc this set of directions intersects on an enclosing circle with radius 1",

Ray Tracing| Radiometry

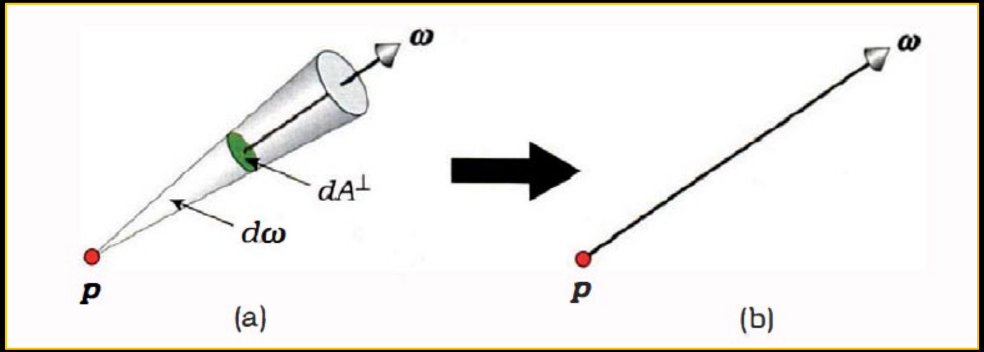
- Similarly, a **solid angle** measures the size of a continuous set of directions in 3D, and it is measured in **steradians (sr)**.
- Instead of representing the arc, it represents the intersection **patch** on an enclosing sphere with radius 1.
- In 2D, an angle of 2π radians covers a whole unit circle.
  In 3D, a solid angle of $\mathbf{4\pi}$ **steradians** cover a whole area of a unit sphere. This means, $\mathbf{2\pi}$ **steradians** covers half of a unit sphere, also called a **hemisphere**.
- **Radiant Intensity(I)** is the Radiant Flux (Watts) with respect to a direction, more precisely, a solid angle. Or Energy in a restriction of direction. It can be written as $I = \frac{d\phi}{d\omega}$ and has units $W/\,sr$.



- Finally, **Radiance(L)**, is the measure of the irradiance over a solid angle with respect to the area. In other words, the Radiant Flux Density with restriction to direction and space. It can be written as $\mathbf{L} = \frac{d^2\phi}{d\mathbb{A}\,d\omega}$ with units $W/\,m^2\,sr$. (Combination of Radiant Intensity [$I = \frac{d\phi}{d\omega}$] and Irradiance [$E = \frac{d\phi}{d\mathbb{A}}$])

Ray Tracing| Radiometry

- It's this Radiance that is important to us! We want to measure the amount of radiant energy with respect to time, space and direction for every pixel!
- This solid angle is still measured as a cone-like shape. Didn't we say we want to measure it along a ray?



1

- Also, interesting to know. The solid angle($\mathbf{d\omega}$) can be calculated by $\frac{\mathbb{A}}{\mathbf{R}^2}$, where R is distance. This means we can convert from irradiance and radiance using the following formula:

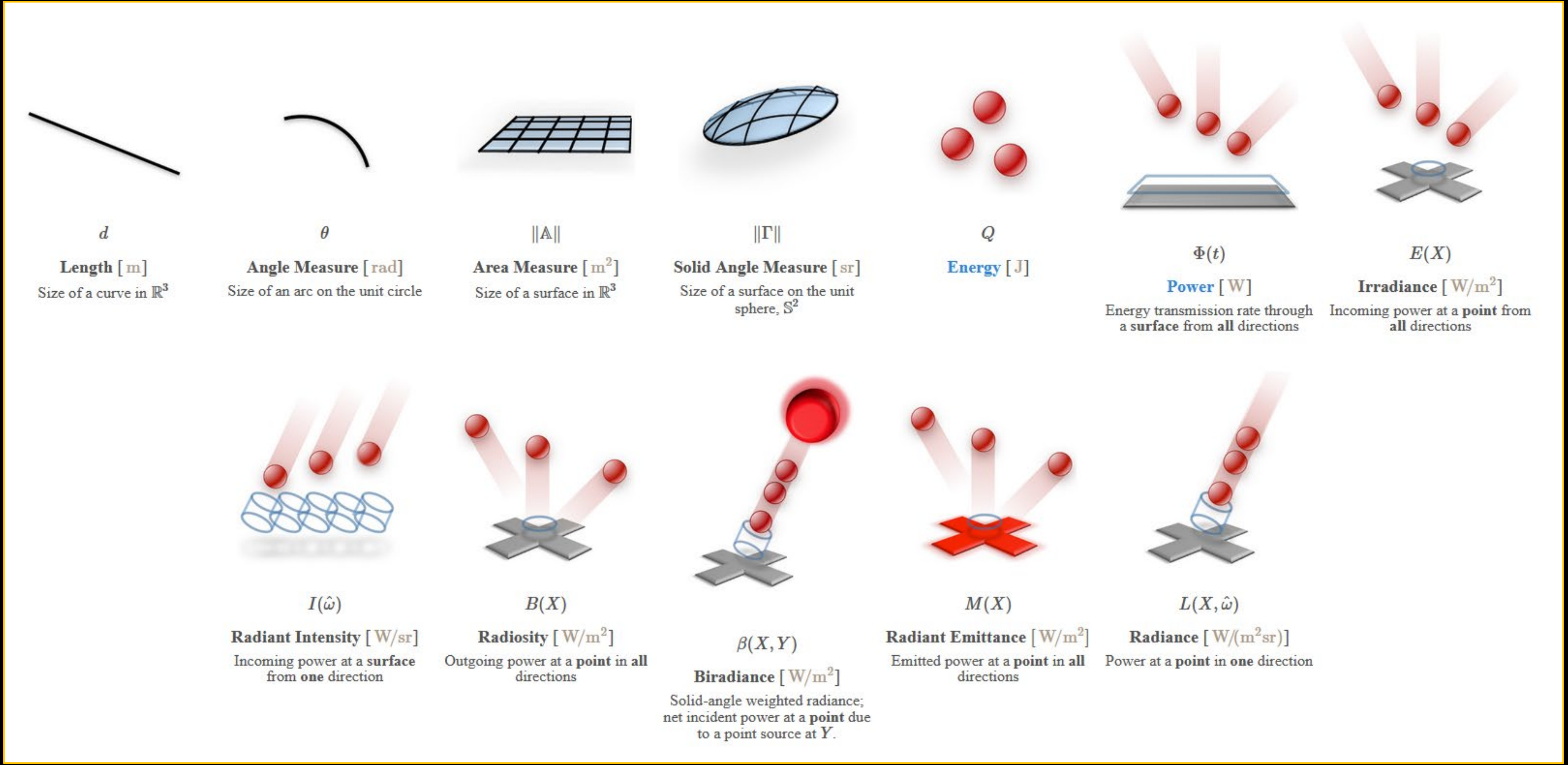
$$\mathbf{L} = \frac{E}{\mathbf{d\omega}} \quad \text{and} \quad E = \mathbf{L} * \mathbf{d\omega}$$

1. Ray Tracing From The Ground Up – Kevin Suffern

Ray Tracing| Radiometry

- As a reminder:

Graphics Codex – Morgan McGuire



d
Length [m]
Size of a curve in $\mathbb{R}^3$

θ
Angle Measure [rad]
Size of an arc on the unit circle

$\|\mathbb{A}\|$
Area Measure [m^2]
Size of a surface in $\mathbb{R}^3$

$\|\Gamma\|$
Solid Angle Measure [sr]
Size of a surface on the unit sphere, $\mathbb{S}^2$

Q
Energy [J]

$\Phi(t)$
Power [W]
Energy transmission rate through a **surface** from **all** directions

$E(X)$
Irradiance [$\mathrm{W/m^2}$]
Incoming power at a **point** from **all** directions

$I(\hat{\omega})$
Radiant Intensity [W/sr]
Incoming power at a **surface** from **one** direction

$B(X)$
Radiosity [$\mathrm{W/m^2}$]
Outgoing power at a **point** in **all** directions

$\beta(X, Y)$
Biradiance [$\mathrm{W/m^2}$]
Solid-angle weighted radiance; net incident power at a **point** due to a point source at Y.

$M(X)$
Radiant Emittance [$\mathrm{W/m^2}$]
Emitted power at a **point** in **all** directions

$L(X, \hat{\omega})$
Radiance [$\mathrm{W/(m^2 sr)}$]
Power at a **point** in **one** direction

ARE YOU OK? STILL WITH US?

Ray Tracing| Modeling Light

- But why? Well, you need to know your units! Using wrong units will screw up your image!
- Back to our equation:

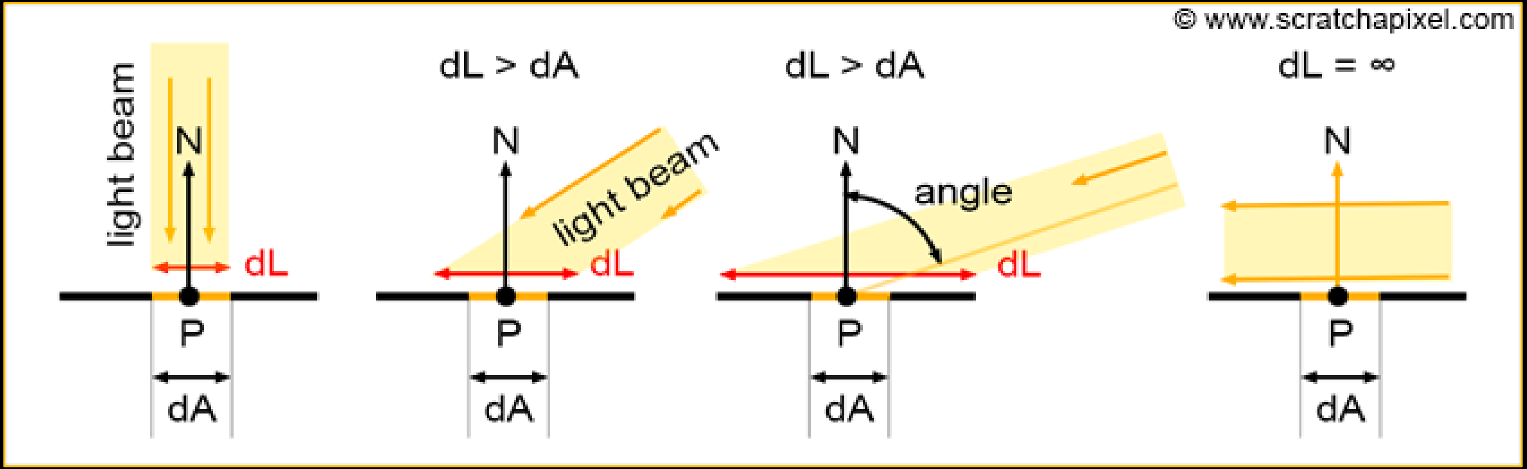
$$\underbrace{\boxed{L_o(X,\omega_{o,x})} = \boxed{L_e(X,\omega_{o,x})} + \int \boxed{L_i(X,\omega_{i,x})}}_{\text{Radiance : } W/\,m^2\,sr} \text{...scattering...} \underbrace{\boxed{d\omega_{i,x}}}_{sr}$$

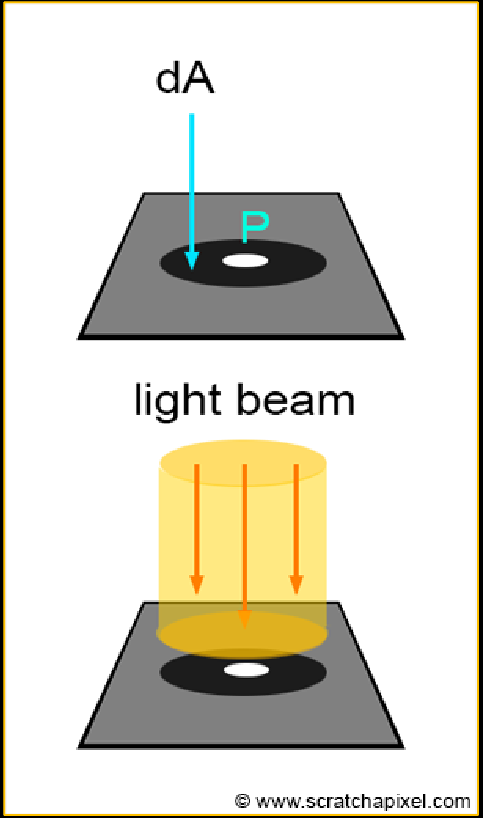
- We are not done yet… There are two important pieces missing! ☺

Ray Tracing| Lambert Cosine Law (Observed Area)

- When we observe a surface and we measure the incident radiance from a direction parallel with the normal of the surface, everything is fine. But as soon as light hits the surface from a certain angle, it appears that the incident radiance covers a bigger area. This is due to the fact that we cannot just take the similar area, instead, we need to **project** the area of our light onto the surface.

  Lambert's cosine law states that the observed area measure of a surface falls with the cosine of the measure of the angle (from the normal) at which it is observed.[1]





Ray Tracing| Lambert Cosine Law

- So, we get the following relation between the projected area and the surface are:

$$\mathbf{d}\mathbb{A}^{\text{proj}} = \boldsymbol{cos\,\theta}\,\mathbf{d}\mathbb{A}$$

- When we substitute it in relation with the incident radiance, we get:

$$\boldsymbol{L} = \frac{\mathbf{d}^2\boldsymbol{\phi}}{\mathbf{d}\mathbb{A}\,\mathbf{d}\boldsymbol{\omega}} \rightarrow \boldsymbol{L} = \frac{\mathbf{d}^2\boldsymbol{\phi}}{\mathbf{d}\mathbb{A}^{\text{proj}}\,\mathbf{d}\boldsymbol{\omega}} \rightarrow \boldsymbol{L} = \frac{\mathbf{d}^2\boldsymbol{\phi}}{\boldsymbol{cos\,\theta}\,\mathbf{d}\mathbb{A}\,\mathbf{d}\boldsymbol{\omega}}$$

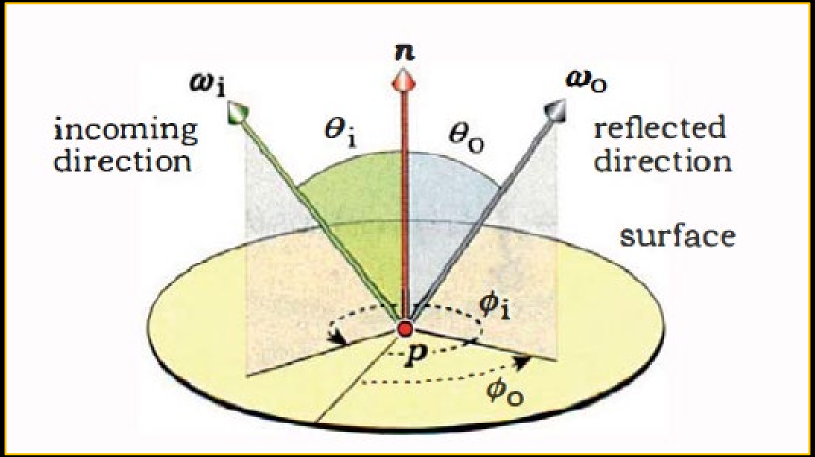
- We now can take this into account in our equation:

$$L_{\mathrm{o}}(X, \omega_{\mathrm{o,x}}) = L_e(X, \omega_{o,x}) + \int L_i\,(X, \omega_{\mathrm{i},x})\ \text{...scattering...}\ \boldsymbol{cos\,\theta_i}\ \mathrm{d}\omega_{i,x}$$

- Is there another way to calculate/represent this $cos\,\theta_i$?
  - Dot product between $\omega_{i,x}$ (incident radiance) and n (normal of surface)!
- All we are missing now is the scattering part.

Ray Tracing| BRDF

- How light scatters depends on the material. There are a lot of different models. All of them try to simulate the light scattering with a mathematical function.
- This function is called the Bidirectional Reflectance Distribution Function (BRDF) or Bidirectional Scattering Distribution Function (BSDF).
- It's often denoted as: $f_{X,n}(\omega_{i,x}, \omega_{o,x})$



1

- It has units of inverse steradians (sr^{-1}), because it's the amount of light that scatters in each direction over a small set of angles near that direction.

1. Ray Tracing From The Ground Up – Kevin Suffern

Ray Tracing| BRDF

- A BRDF has a few useful properties:
  - Reciprocity: when we swap the directions, the BRDF stays the same, thus the reflected radiance stays the same! → Bidirectional!
  - Linearity: most good materials need multiple BRDF's to model the correct reflectivity. The total amount of reflected radiance of a point is the sum of each BRDF.
  - Conservation of Energy: reflected radiance can never exceed the incoming radiant flux (excluding the emitted light from emissive objects).

- Thus, the BRDF mostly determines how physically correct a material is! Warning, everything we do in rendering approximates reality, so we will never have an 100% physically accurate image.
- Ok, let's go back to our equation and plug in this BRDF…

Ray Tracing| Rendering Equation!

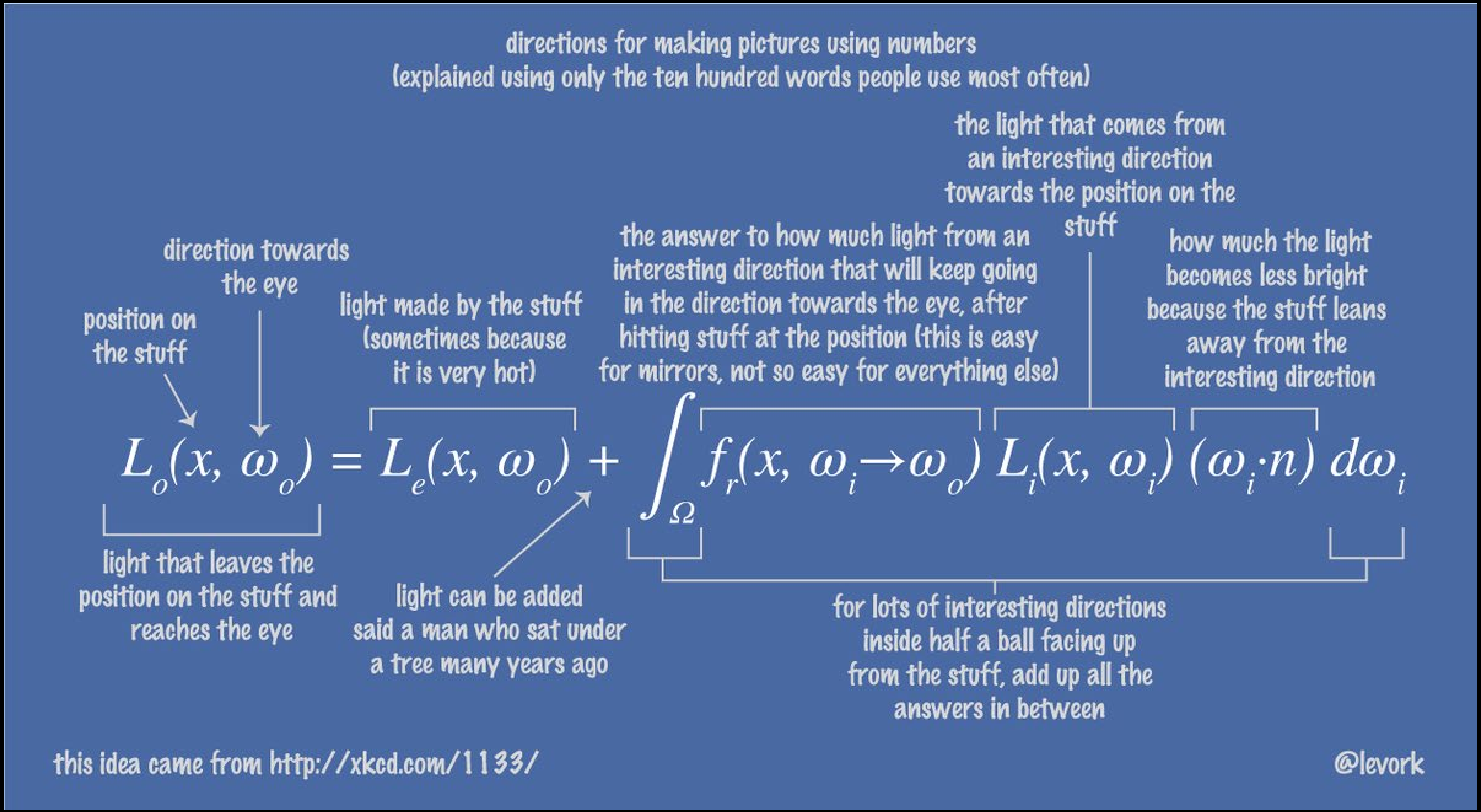
$$L_o(X,\omega_{o,x}) = L_e(X,\omega_{o,x}) + \int L_i(X,\omega_{i,x}) \underbrace{\boxed{f_{X,n}(\omega_{i,x},\omega_{o,x})}}_{sr^{-1}} \cos\theta_i \, d\omega_{i,x}$$

- This is called the Rendering Equation! This equation encapsulates how light interacts in an 3D environment. How you solve this equation determines how accurate your final image is. And as mentioned before, we still use approximations! Solving this is hard! In fact, it's impossible.
- To help you remember what every part means…

$$L_o(X,\omega_{o,x}) = \underbrace{L_e(X,\omega_{o,x})}_{\text{Object}} + \int \underbrace{L_i(X,\omega_{i,x})}_{\text{Lights}} \underbrace{f_{X,n}(\omega_{i,x},\omega_{o,x})}_{\text{Materials}} \underbrace{\cos\theta_i}_{\text{View Angle}} \, d\omega_{i,x}$$

- Outgoing radiance from point X with direction to the viewer $\omega_{o,x}$ is equal to the sum of the incident radiance of all the lights from any direction and the light scattered from any direction by the material to the viewer, within a hemisphere, in the projected area based on the incident radiance and the normal, plus all the radiance emitted by the object.

Ray Tracing| Rendering Equation!



directions for making pictures using numbers
(explained using only the ten hundred words people use most often)

$$\underbrace{L_o(x, \omega_o)}_{\text{light that leaves the position on the stuff and reaches the eye}} = \overbrace{L_e(x, \omega_o)}^{\text{light made by the stuff (sometimes because it is very hot)}} + \int_\Omega \overbrace{f_r(x, \omega_i \rightarrow \omega_o)}^{\text{the answer to how much light from an interesting direction that will keep going in the direction towards the eye, after hitting stuff at the position (this is easy for mirrors, not so easy for everything else)}} \overbrace{L_i(x, \omega_i)}^{\text{the light that comes from an interesting direction towards the position on the stuff}} \overbrace{(\omega_i \cdot n)}^{\text{how much the light becomes less bright because the stuff leans away from the interesting direction}} d\omega_i$$

position on the stuff

direction towards the eye

light can be added said a man who sat under a tree many years ago

for lots of interesting directions inside half a ball facing up from the stuff, add up all the answers in between

this idea came from http://xkcd.com/1133/

@levork

LET'S DO THIS!

Ray Tracing | Lights

- Let's start by looking how we can implement this render equation, using some easy approximations.
- We'll first look at the following part of the equation: $L_i(X, \omega_{\mathrm{i},x})$
- As mentioned before, we'll implement direct illumination. In other words, only light from light sources that are directly visible from the point we want to shade. We won't take into account light scattered from other surface (indirect/global illumination).
- There are different types of lights in game engines:
  - Ambient: cheap indirect illumination – it lets the object receive a default amount of radiance.
  - Directional: simulates light coming from the sun, all incident light rays are parallel.
  - Point: simulates light radiating from a single point in space.
  - Spot: similar as a point light, but it restricts the radiation of light in a cone shape.
  - Area: simulates light being emitted from a surface.
- Most light types discussed are not physically accurate. All light sources have a surface, so in theory, only area lights are physically correct. The other light types are used because they are easy to implement and to control by artists. Area lights also makes our equation harder to solve, so we'll avoid them for now.

Ray Tracing | Lights

- Whenever we determine a point is hit by a light, we just return the incident Radiance of the light and add it to value we currently have. Remember, we are summing up the influence of all visible lights!
- So, our current render loop will, at this moment, change to:

```
for each pixel hit by our ray
    for each light in our scene
            calculate the incident radiance of the light
            (based on its properties).
```

- At this point we don't care if a light source is visible from the point or not. We'll get to this later!
- For now, we'll just determine the Irradiance of each light, based on its properties. These properties are different for different types of light. Let's discuss two types of light: directional and point light.

Ray Tracing | Lights (Point Light)

- Point Light:
  - When creating a point light, the user determines a few parameters:
    - Position [Vector3] – position of the light source
    - Color[RGBColor] – color the light emits (fakes the wavelengths)
    - Light Intensity [float] – total amount of energy the light emits (~ Radiant Intensity)
  - We must pay attention to our units! We want radiance ($W/\,m^2\,sr$) and only have Radiant Intensity to start with. So, how do we do this?

  - Reminder, for now we are going to focus on the incident part.

$$L_{\mathrm{o}}(X, \omega_{\mathrm{o,x}}) = L_e(X, \omega_{o,x}) + \int \boxed{L_i\,(X, \omega_{\mathrm{i},x})}\, f_{X,n}(\omega_{i,x}, \omega_{o,x})\, cos\,\theta_i\, \mathrm{d}\omega_{i,x}$$

Ray Tracing | Lights (Point Light)

- Point Light:
  - Remember:
    - Radiant Flux (or Power), denoted as $\boldsymbol{\phi}$, is in $\boldsymbol{W = J/s}$.
    - When we know the Intensity($\boldsymbol{I}$) we can rearrange the terms to find our Radiant Power:

$$I = \frac{d\phi}{d\omega} \quad \rightarrow \quad d\phi = I\, d\omega$$

    - A point light emits in all directions. It has all directions from its position to the surface of a unit sphere.
    - So for a point light, we can determine the radiant flux by integrating the Intensity, over the surface of a unit sphere (steradians).

$$\phi = \int_{4\pi} I\, d\omega = 4\pi I$$

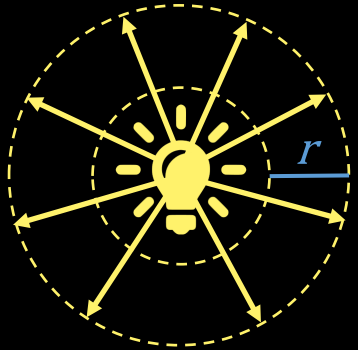
Ray Tracing | Lights (Point Light)

- Point Light:
  - We know the Radiant Flux. Next step is the Irradiance. For that we need to consider the area.

$$E = \frac{\mathrm{d}\phi}{\mathrm{d}\mathbb{A}}$$

  - Light is getting scattered in all direction from a sphere-like shape. If we draw another circle/sphere, we can actually see that it scales based on the radius. If we plug in the surface area of our sphere into the previous formula, we get:

$$E = \frac{\mathrm{d}\phi}{4\pi r^2}$$



  - We can then simplify this because:

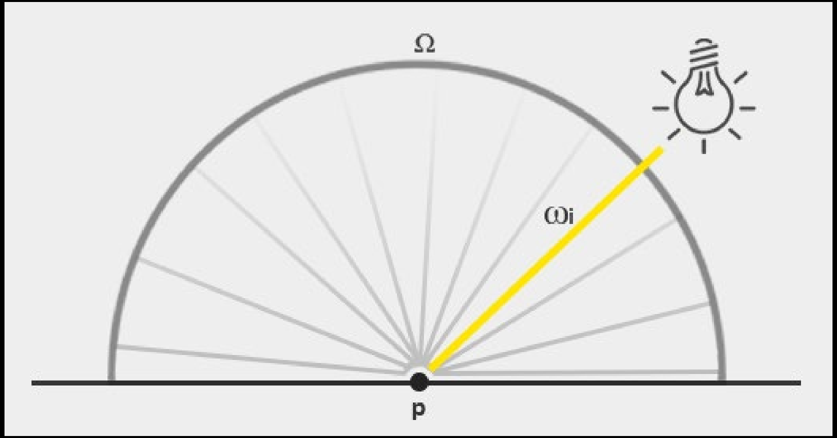
$$E = \frac{\mathrm{d}\phi}{4\pi r^2} \rightarrow E = \frac{4\pi I}{4\pi r^2} \rightarrow E = \frac{I}{r^2}$$

  - To finish, we just multiply our light color with this irradiance value. For the radius, you can use the square distance between the position of the light and the point you are shading.

$$E_{rgb} = lightColor * \frac{I}{(LightPosition - PointToShade)^2}$$

Ray Tracing | Lights (Point Light)

- Point Light:
  - But we wanted Radiance right? Well, it turns out we are a bit lucky…
  - Our lights are not area lights, which means they have only influence from one direction. All other possible incoming light directions have zero radiance observed over the intersection point.



  - So, in this case, we have our solution as the we only have one direction, which we sample directly anyway!
  - And to quote: "This brings us back to the integral part. As we know beforehand the single locations of all the contributing light sources while shading a single surface point, it is not required to try and solve the integral. We can directly take the (known) number of light sources and calculate their total irradiance, given that each light source has only a single light direction that influences the surface's radiance."

https://learnopengl.com/

Ray Tracing | Lights (Directional Light)

- Directional Light:
  - When creating a directional light, the user determines some other parameters:
    - Direction [Vector3] – Directional Light only has one direction (all are parallel), no position
    - Color[RGBColor] – color the light emits (fakes the wavelengths)
    - Light Intensity [float] – total amount of energy the light emits (~ Radiant Intensity)
  - Calculating the Irradiance of a directional light is even easier! It has no area, so we can just get rid of the everything that contributes to an area. We end up with just:

$$E_{rgb} = lightColor \ast I$$

  - Fun fact, a point light almost acts like a directional light, except the direction changes. In other words, it can be viewed as a point light without any attenuation or distance falloff. So, if you don’t want falloff, but still using the direction vectors based on the location, use the above formula for the Irradiance calculation.

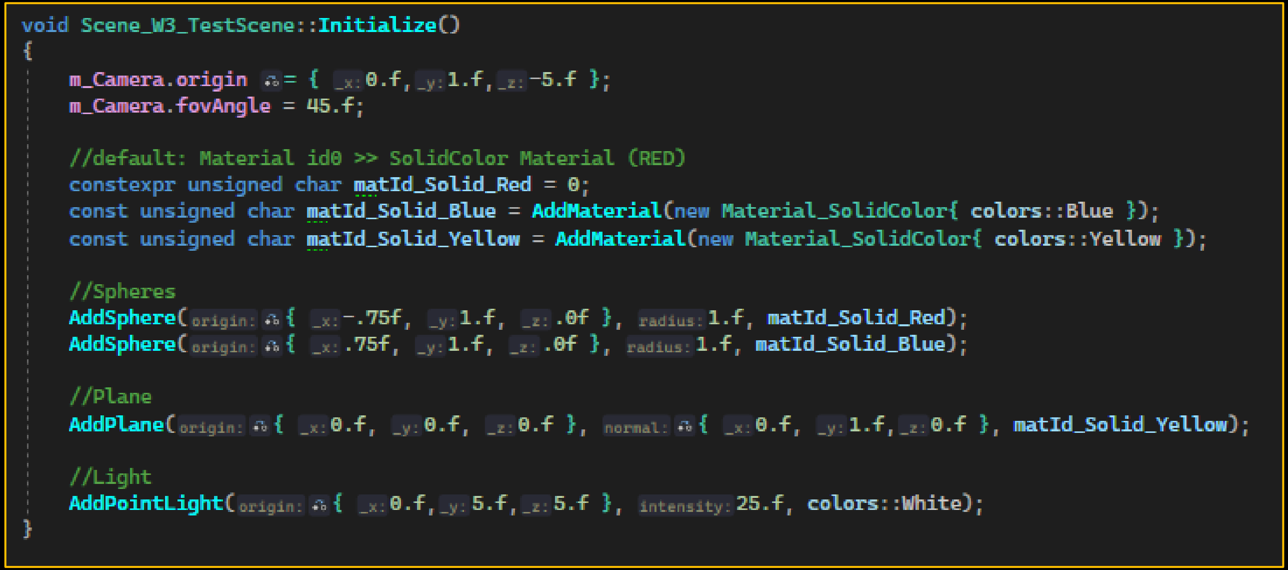
Ray Tracing | Lights (TestScene)

- Let's test this out! Change our render loop to go over all the lights and calculate the final color (disable shadow code for now!) :

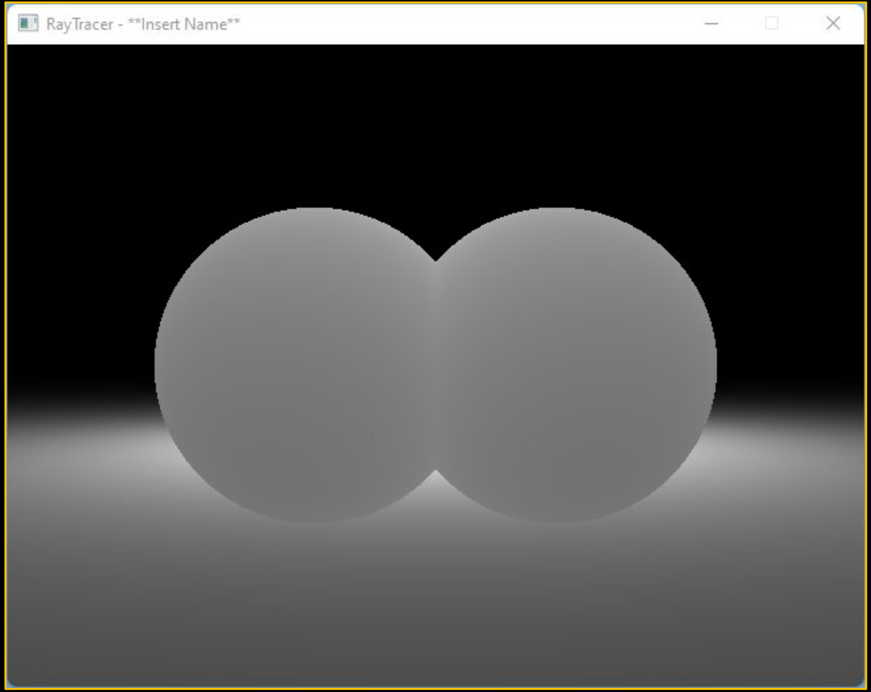
$$finalColorPixel += E_{rgb}$$

Note: E_{rgb} is also what LightUtils::GetRadiance should return

Use the following 'temporary' test scene, again, disable the hard-shadow code (!)

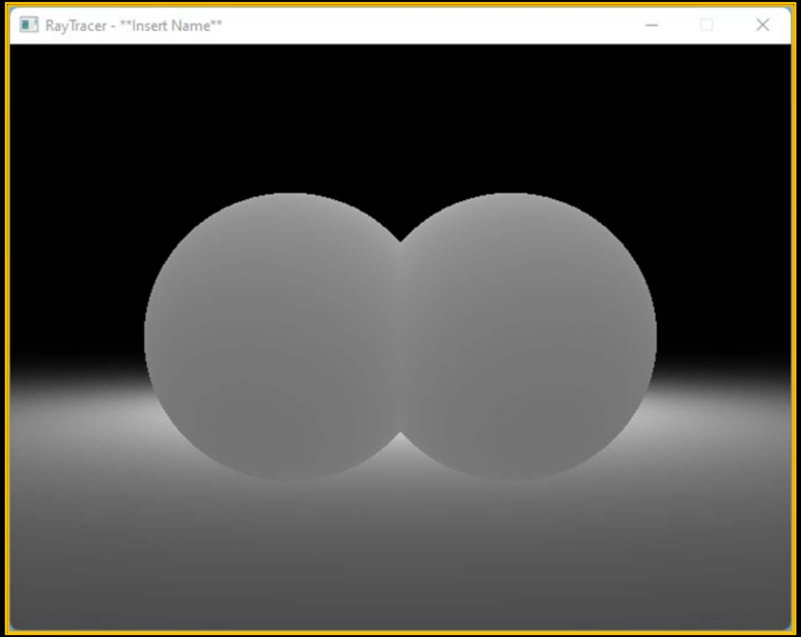


```
void Scene_W3_TestScene::Initialize()
{
    m_Camera.origin = { _x:0.f, _y:1.f, _z:-5.f };
    m_Camera.fovAngle = 45.f;

    //default: Material id0 >> SolidColor Material (RED)
    constexpr unsigned char matId_Solid_Red = 0;
    const unsigned char matId_Solid_Blue = AddMaterial(new Material_SolidColor{ colors::Blue });
    const unsigned char matId_Solid_Yellow = AddMaterial(new Material_SolidColor{ colors::Yellow });

    //Spheres
    AddSphere(origin: { _x:-.75f, _y:1.f, _z:.0f }, radius:1.f, matId_Solid_Red);
    AddSphere(origin: { _x:.75f, _y:1.f, _z:.0f }, radius:1.f, matId_Solid_Blue);

    //Plane
    AddPlane(origin: { _x:0.f, _y:0.f, _z:0.f }, normal: { _x:0.f, _y:1.f, _z:0.f }, matId_Solid_Yellow);

    //Light
    AddPointLight(origin: { _x:0.f, _y:5.f, _z:5.f }, intensity:25.f, colors::White);
}
```

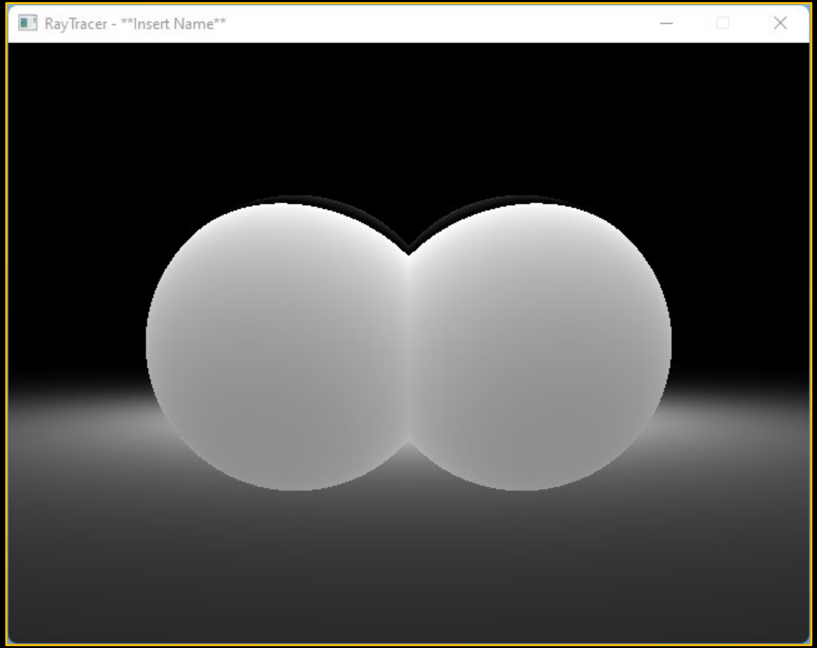


Ray Tracing | Lights (TestScene)

- Ok that doesn't look right! What's wrong?
- Oh, we forgot to take the Lambert's Cosine Law (~Observed Area) into account ! Let's fix this by multiplying our radiance with the cosine θ between our normal and the light direction (hint: this is different for different types of light!).
- So now we have: $\boldsymbol{finalColorPixel += E_{rgb} * Dot(normal, lightDirection)}$

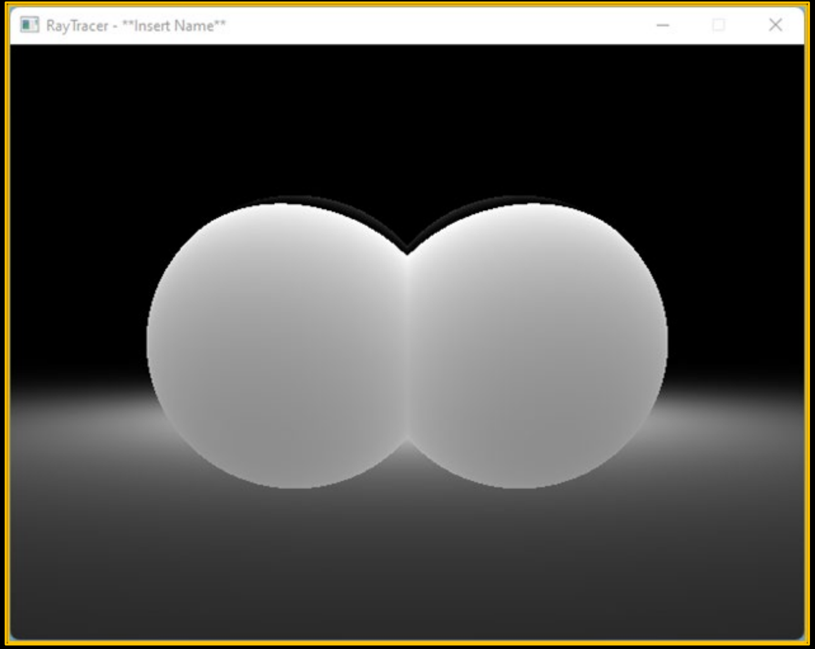


→

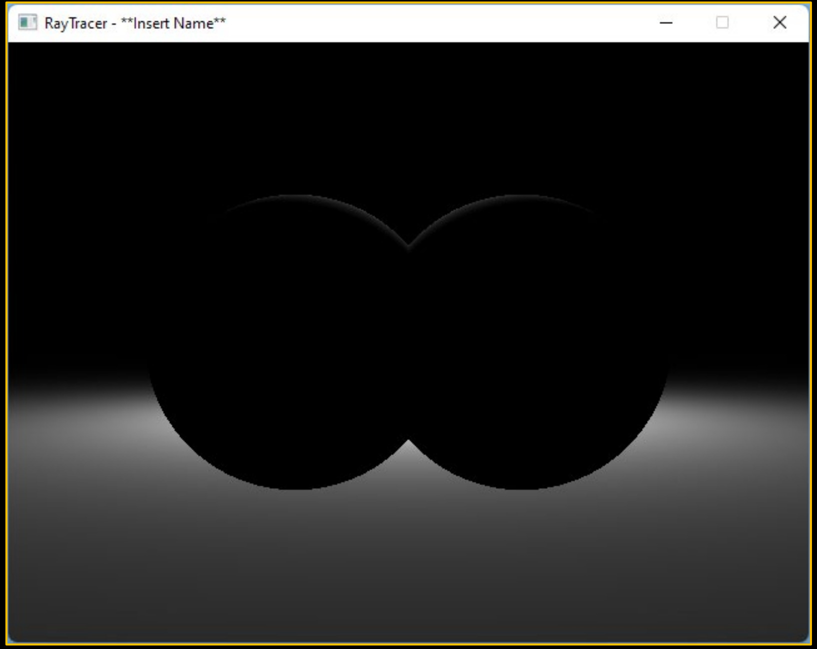


Ray Tracing | Lights (TestScene)

- That is even worse! What is happening?
- Our Dot Product will give us a value ranging from [-1,1]. If a value is below 0, we are not interested in the value, because the point on the surface points away from the light!
- We can easily fix this by skipping the contribution of this light when that happens.

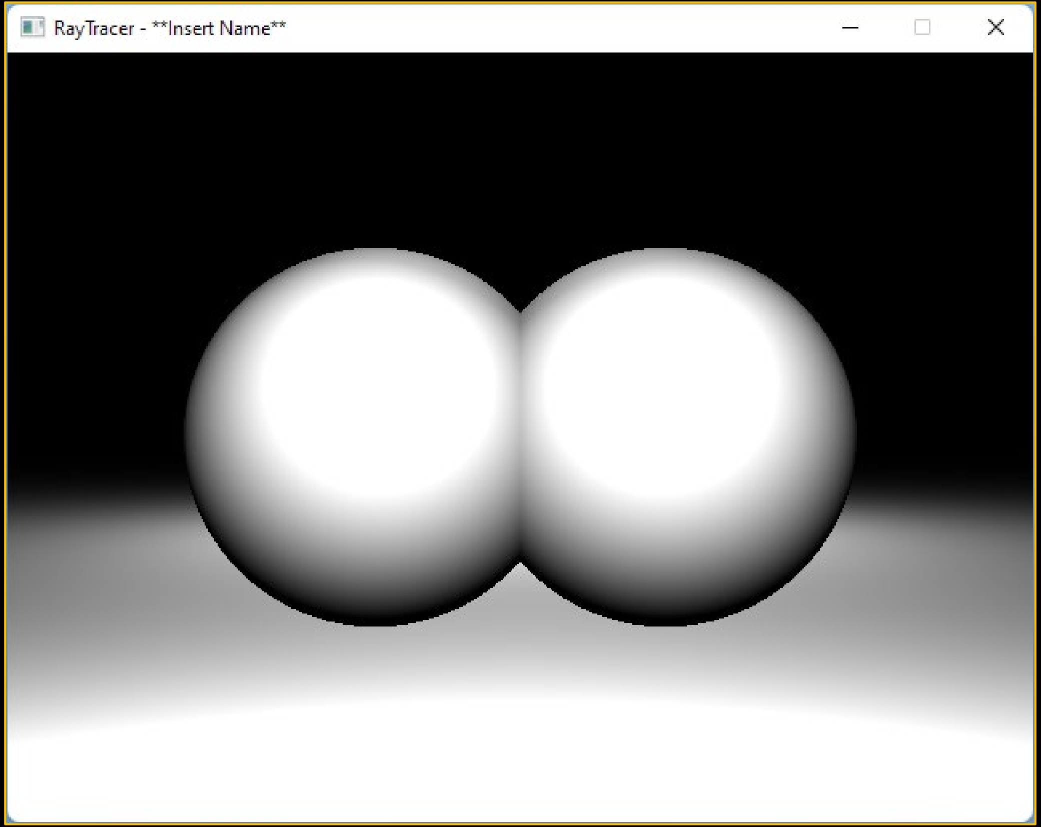


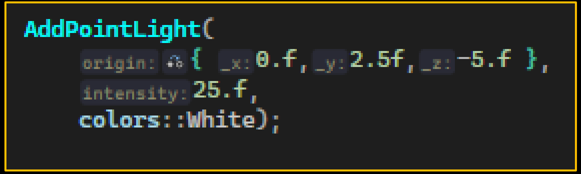
→



Ray Tracing | Lights (TestScene)

- Let's add another point light



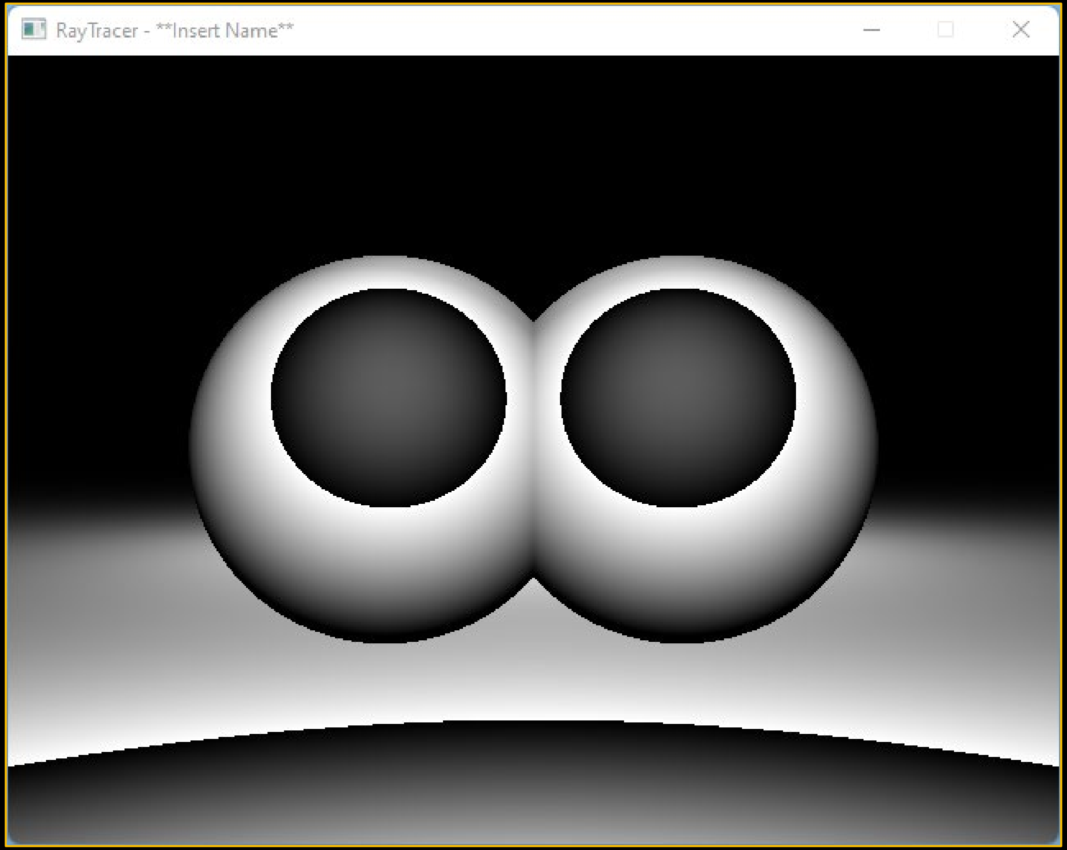


```
AddPointLight(
    origin: { _x: 0.f, _y: 2.5f, _z: -5.f },
    intensity: 25.f,
    colors::White);
```

Ray Tracing | Lights (TestScene)

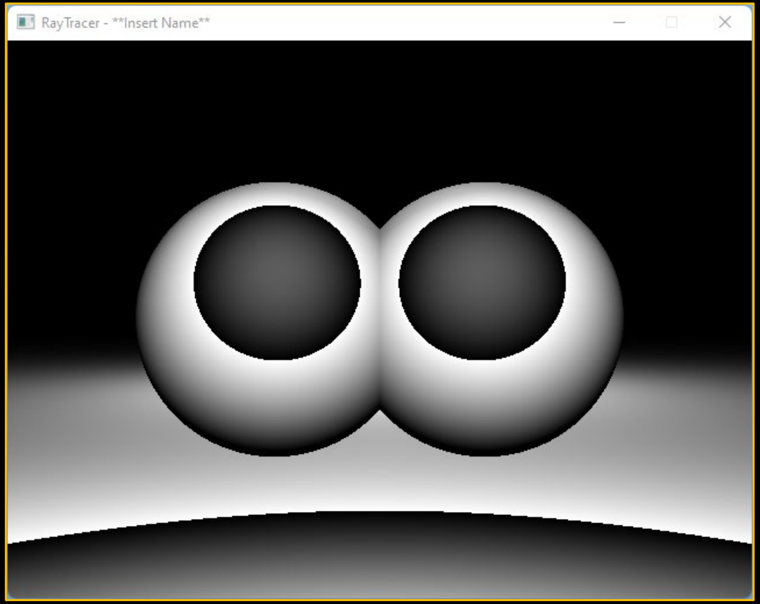
- Comment out the MaxToOne() function call at the end of the render loop, observe what happens

```
//Update Color in Buffer
//finalColor.MaxToOne();
```

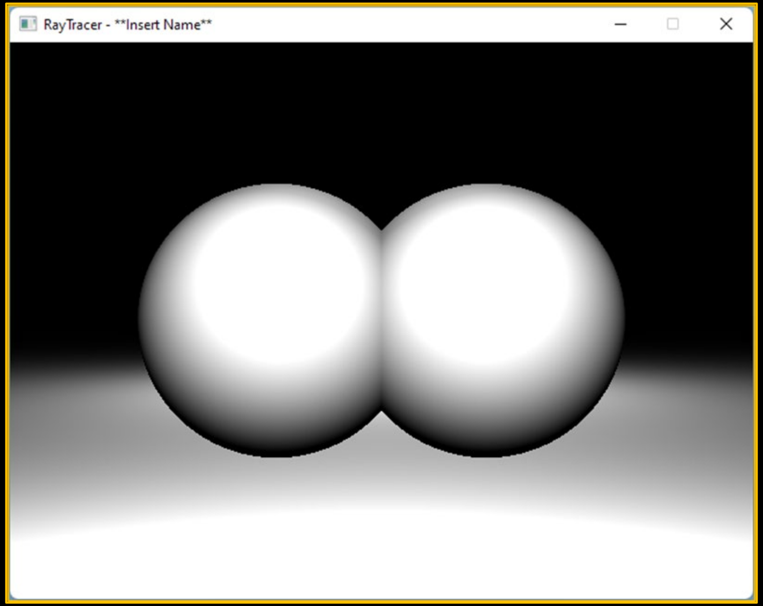


Ray Tracing | Lights (TestScene)

- We exceed the range of our colors and get an overflow. We can easily fix this by "normalizing" the range after we've done all the color calculations.
- One method is: MaxToOne(), where we take the biggest value and divide all components with this value. The biggest value will become 1 and all the other components will scale accordingly. There are other, better ways though! (e.g. Tone Mapping) ☺

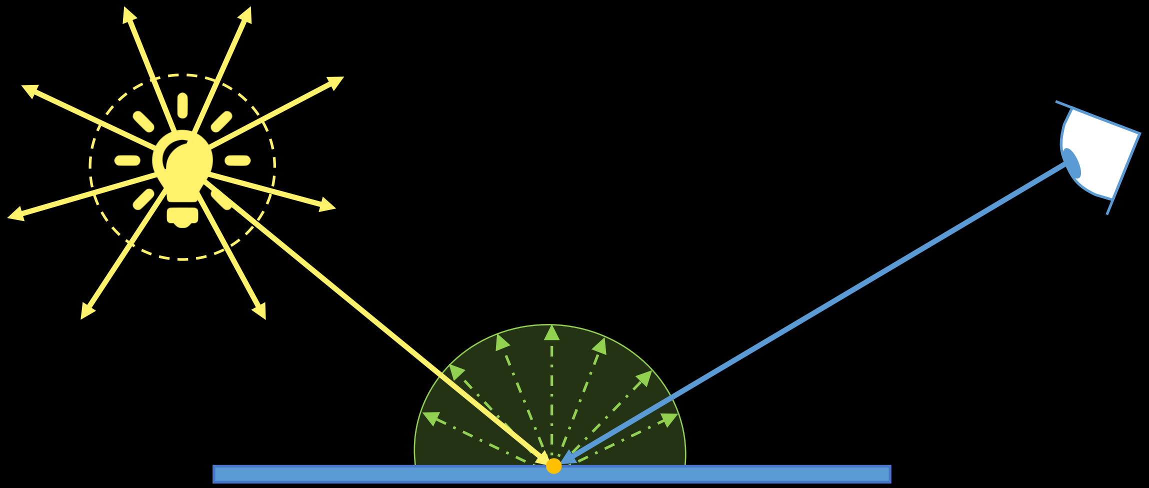


→



Ray Tracing | BRDF (Lambert Diffuse)

- Let's now look at our BRDFs. We are going to implement one of the simplest, yet not psychically not correct, BRDFs, called the Lambert BRDF.
- This Lambert BRDF has nothing to do with the Lambert Cosine Law!
- The Lambert (Reflection) BRDF gives us a perfect diffuse reflection. In other words, all incident radiance is scattered equally in all directions over a hemisphere. That is, it is also independent from the view direction!

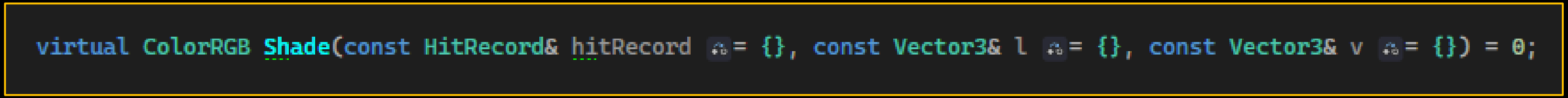


Ray Tracing | BRDF (Lambert Diffuse)

- Because of the equal scattering and independence of the view direction, the equation is very simple (it’s a constant). This material is ideal for basic matte objects.

$$\textit{Lambert}:\ f_{X,n}(\omega_{i,x}, \omega_{o,x}) = \frac{\rho}{\pi}$$

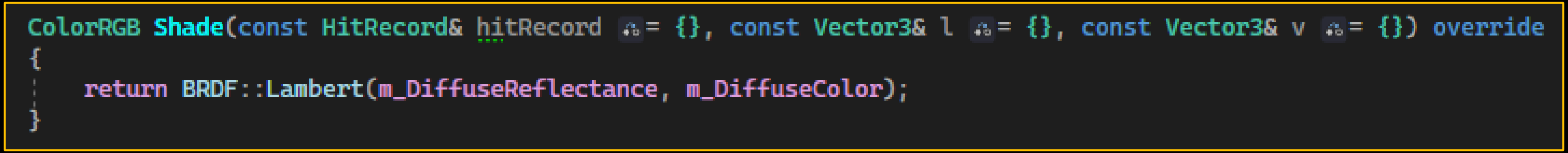
- The value ρ (rho) is called the reflectivity of the surface, or to be more accurate, the perfect diffuse reflectance. You can calculate ρ in terms of a RGBColor value by multiplying the diffuse color (cd) of the object with the diffuse reflection coefficient (kd).
- These parameters are the ones you’ll often see in a Lambert Material:
  - (kd) Diffuse Reflectance [float] – How reflective is this matte material, should range [0, 1].
  - (cd) Diffuse Color [RGBColor] – Color of the material.
- So, you can now create a Material class that has a function that calculates the outgoing radiance (e.g.: Shade(…)) that takes the usual parameters that most BRDF’s need, incoming/incident light direction, view direction and your HitRecord (normal, point, etc.).



```cpp
virtual ColorRGB Shade(const HitRecord& hitRecord = {}, const Vector3& l = {}, const Vector3& v = {}) = 0;
```

Ray Tracing | BRDF (Lambert Diffuse)

- Make sure you can easily change which BRDF you use in a material. We will add more later! So, you should have:
  - A class Material, with a pure virtual Shade(…) function. From this base class there are already some derived classes defined.
    - For example: Material_Lambert, this one should use the BRDF::Lambert (BRDFs.h) function in the Shade(…) function. The Shade(…) function returns a color!



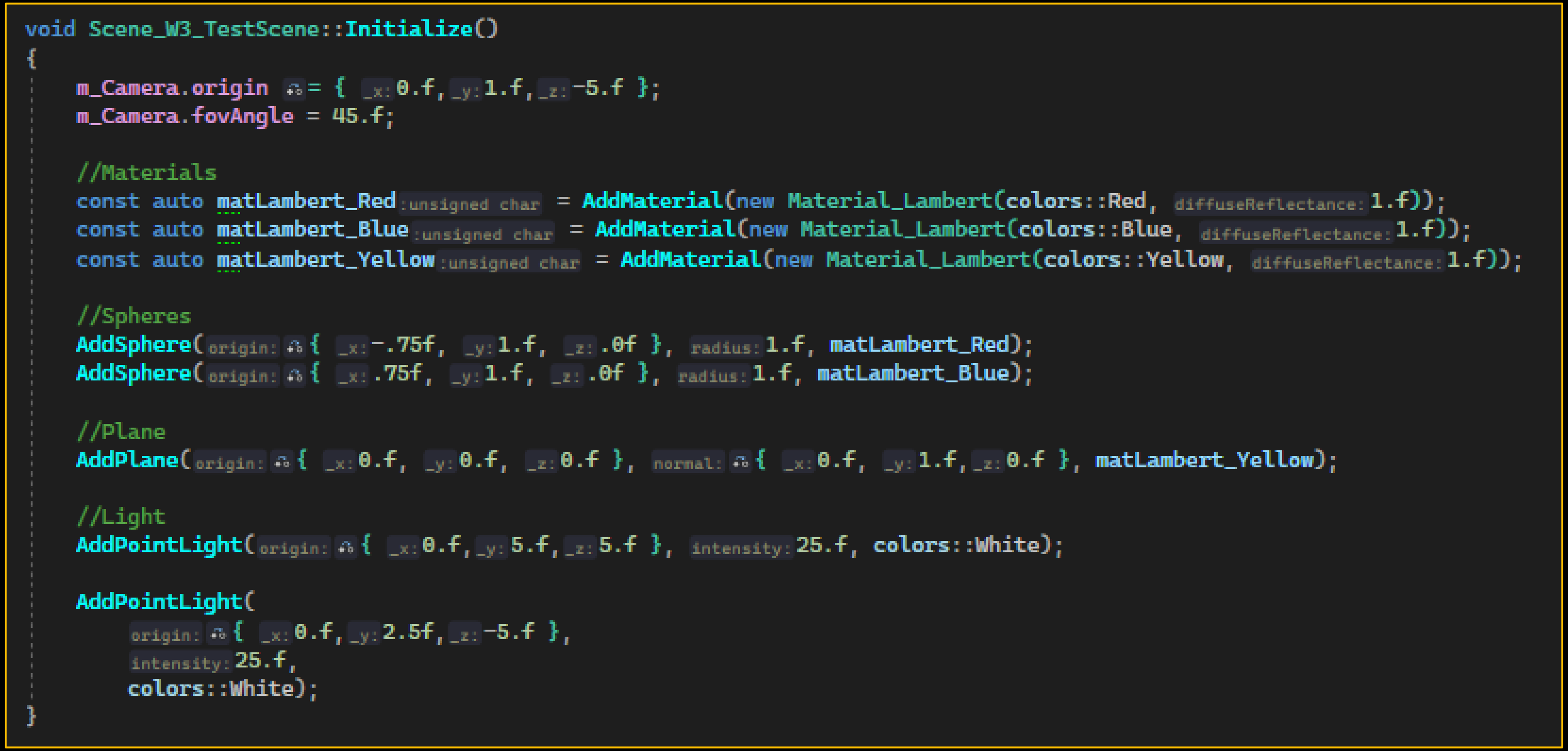
```
ColorRGB Shade(const HitRecord& hitRecord = {}, const Vector3& l = {}, const Vector3& v = {}) override
{
    return BRDF::Lambert(m_DiffuseReflectance, m_DiffuseColor);
}
```

- Now plug this into our render equation. For every light also calculate the BRDF for the material (you must keep track of the material used by the object you are rendering and store the color of the object in the material itself).

$$finalColorPixel += E_{rgb} * BRDFrgb * Dot(normal, lightDirection)$$

Ray Tracing | Lambert Diffuse (TestScene)

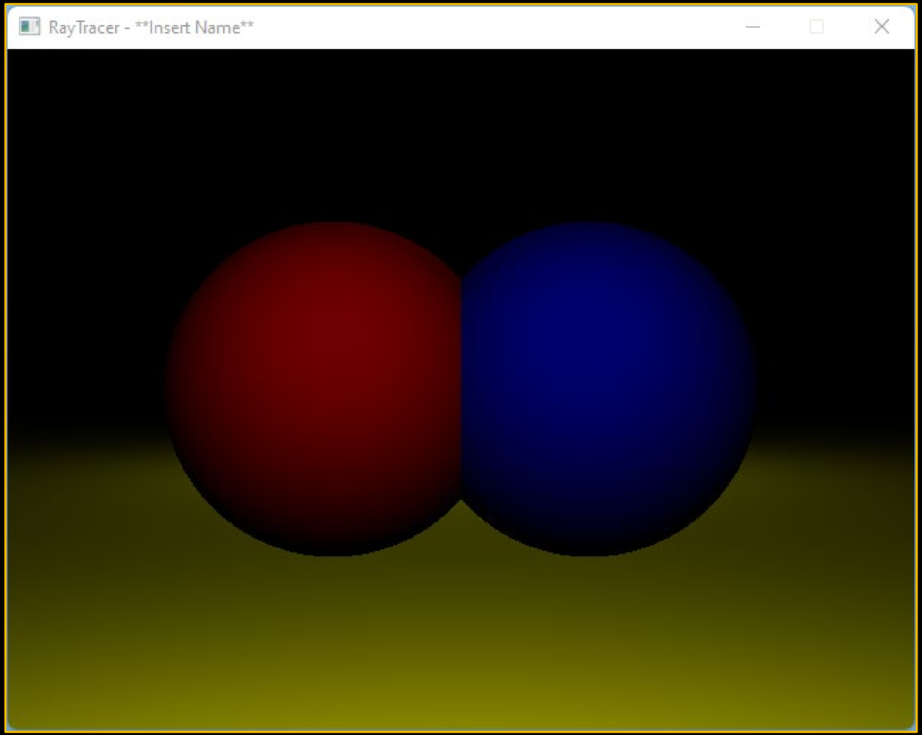
- Adapt the TestScene so it uses Lambert Materials instead of SolidColor Materials
- Implement the Material_Lambert::Shade(…) & BRDF::Lambert(…) functions



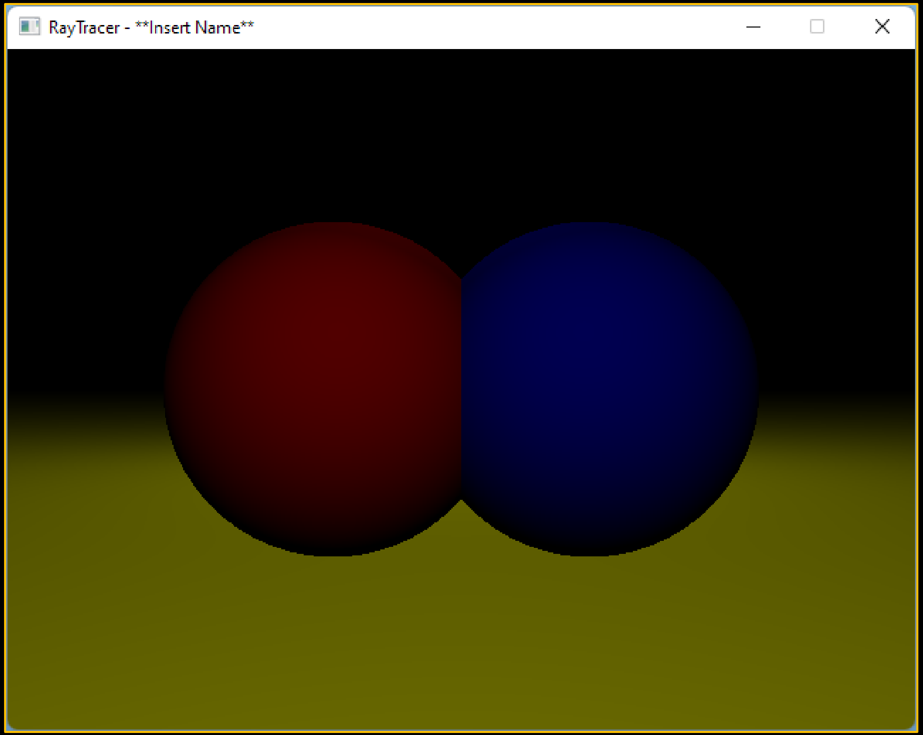
```
void Scene_W3_TestScene::Initialize()
{
	m_Camera.origin = { _x:0.f, _y:1.f, _z:-5.f };
	m_Camera.fovAngle = 45.f;

	//Materials
	const auto matLambert_Red :unsigned char = AddMaterial(new Material_Lambert(colors::Red, diffuseReflectance:1.f));
	const auto matLambert_Blue :unsigned char = AddMaterial(new Material_Lambert(colors::Blue, diffuseReflectance:1.f));
	const auto matLambert_Yellow :unsigned char = AddMaterial(new Material_Lambert(colors::Yellow, diffuseReflectance:1.f));

	//Spheres
	AddSphere(origin:{ _x:-.75f, _y:1.f, _z:.0f }, radius:1.f, matLambert_Red);
	AddSphere(origin:{ _x:.75f, _y:1.f, _z:.0f }, radius:1.f, matLambert_Blue);

	//Plane
	AddPlane(origin:{ _x:0.f, _y:0.f, _z:0.f }, normal:{ _x:0.f, _y:1.f, _z:0.f }, matLambert_Yellow);

	//Light
	AddPointLight(origin:{ _x:0.f, _y:5.f, _z:5.f }, intensity:25.f, colors::White);

	AddPointLight(
		origin:{ _x:0.f, _y:2.5f, _z:-5.f },
		intensity:25.f,
		colors::White);
}
```

Ray Tracing | Lambert Diffuse (TestScene)

- If done correctly, you should get the following result:



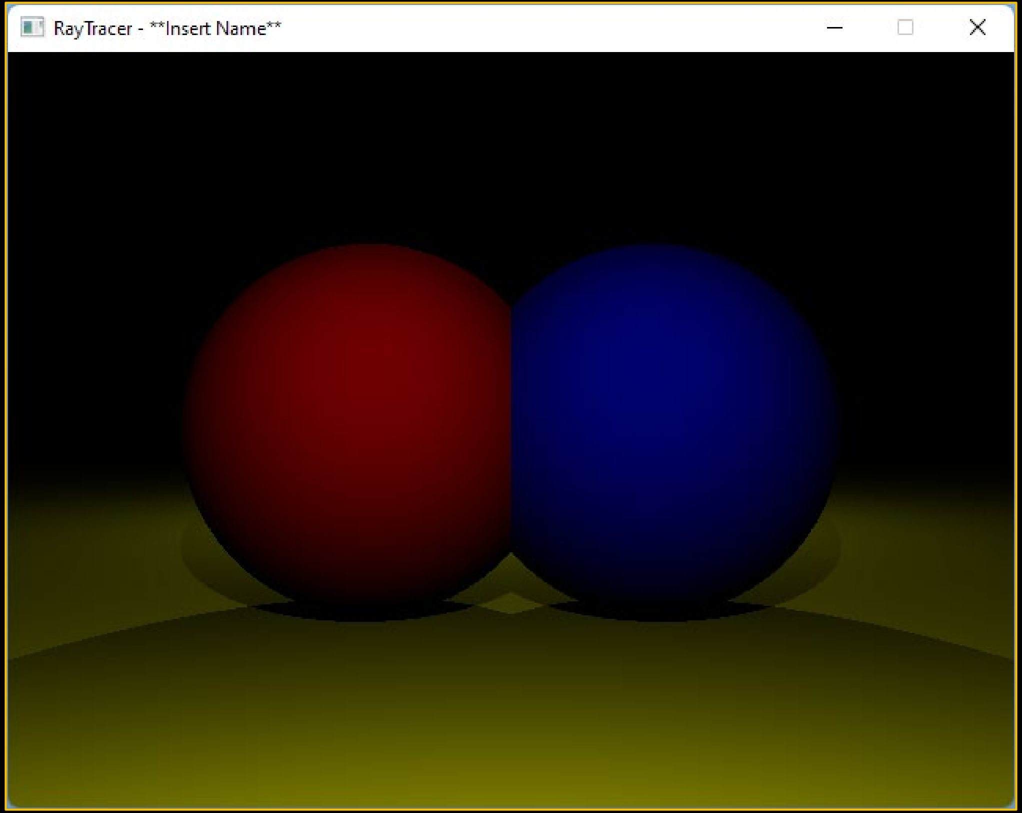
Final Color with BRDF



BRDF with Lambert Cosine Law – no Light

Ray Tracing | Lambert Diffuse (TestScene)

- Final Result (with shadows)
  - Light Radiance * BRDF * Observed Area



Ray Tracing | Current Flow

```
for each pixel hit by our ray
    for each light in our scene
            check if point we hit can see light
                    if not, go to next light and skip light calculation

            calculate observed area (lambert cosine law)
            if observed area < 0
                    skip to the next light (continue)

            if shadowed
                    skip to the next light (continue)

            final color is the sum of: (lighting equation)
                    radiance of the light (based on light properties)
                    * brdf of the material * observed area
    clamp your final color to prevent overflow
else
    the color of nothing
```

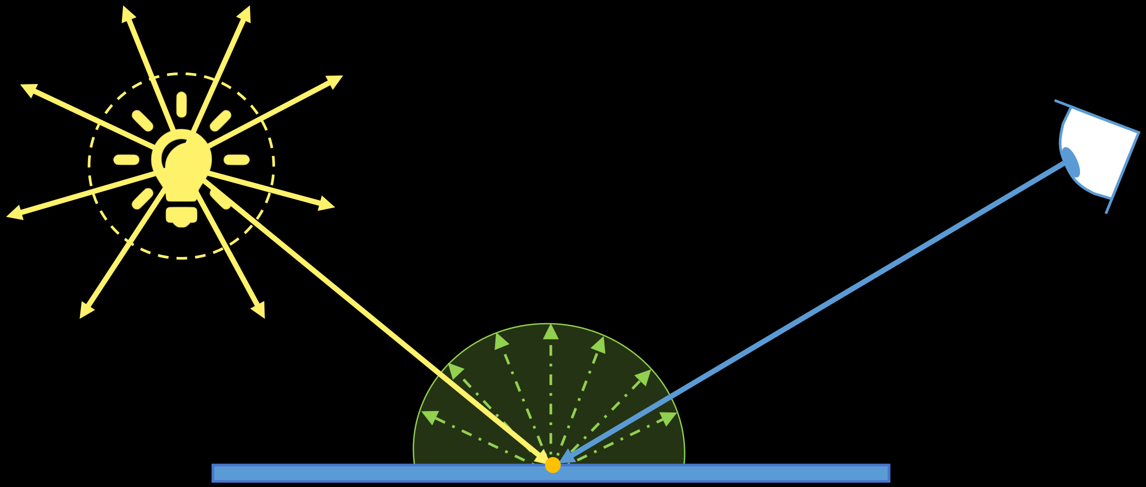
But there is more!

Ray Tracing | BRDF – Lambert Diffuse

- We just discussed the Rendering Equation:

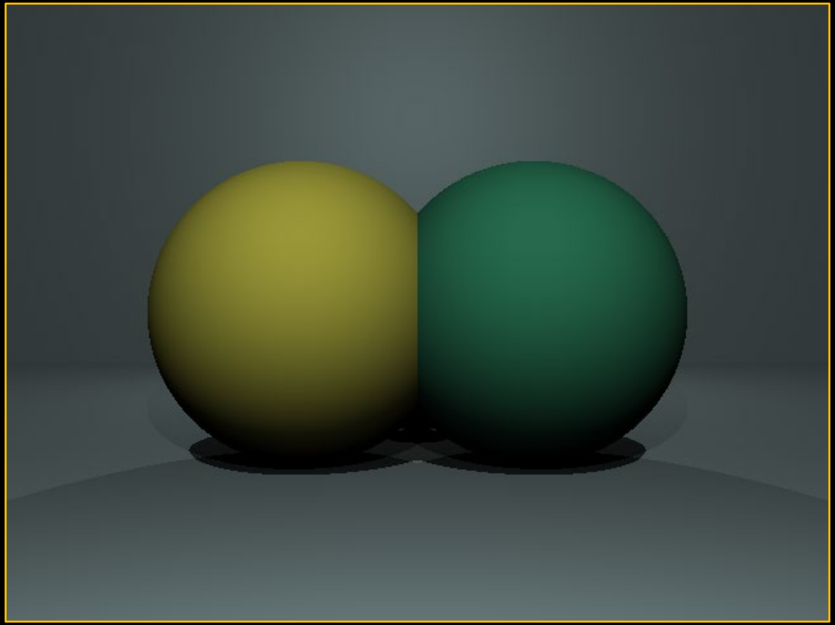
$$L_\mathrm{o}(X, \omega_{\mathrm{o},\mathrm{x}}) = L_e(X, \omega_{o,x}) + \int L_i\,(X, \omega_{\mathrm{i},x})\, f_{X,n}(\omega_{i,x}, \omega_{o,x})\, cos\,\theta_i\, \mathrm{d}\omega_{i,x}$$

- A part of the equation (***f***) is the BRDF. We just 'implemented' the Lambert (Reflection) BRDF, which gives us a perfect diffuse reflection. In other words, all incident radiance is scattered equally in all directions over a hemisphere.



Ray Tracing | BRDF – Lambert Diffuse

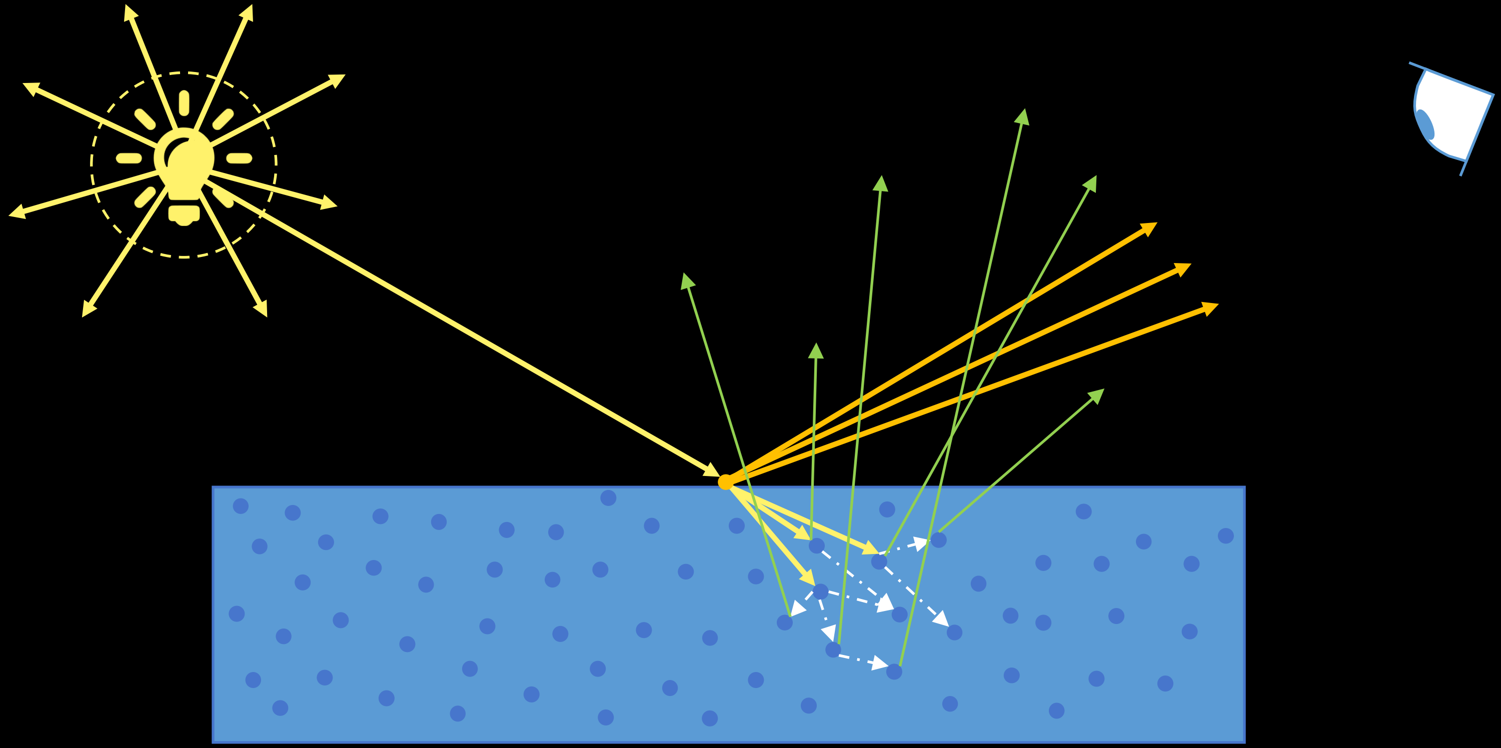
- When implementing the Lambert (Reflection) BRDF, you should get a result similar to the image below.



- This is only one part of the BRDF though! To be accurate, this is the Lambert Diffuse Reflection BRDF and it only models the light that is being scattered in the material, back to the viewer.

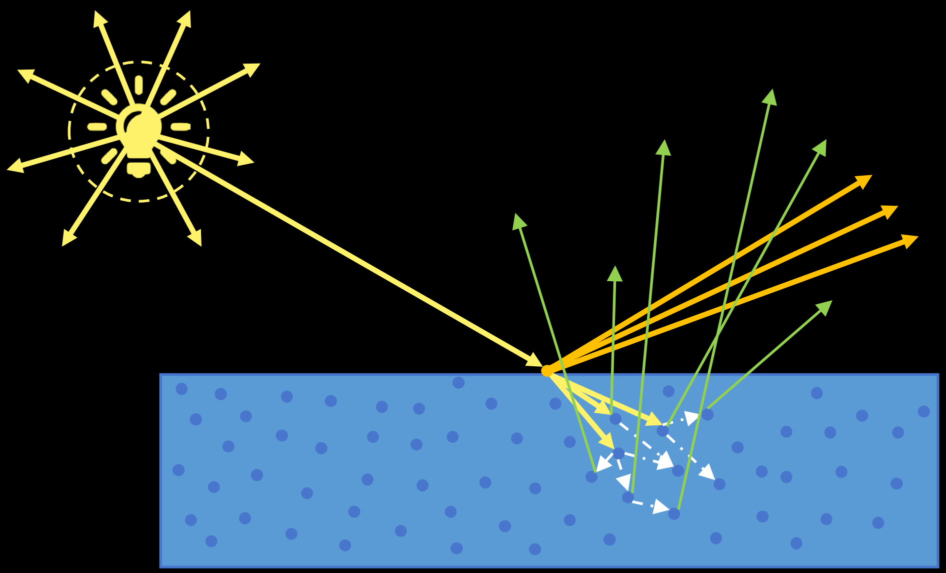
Ray Tracing | BRDF – Lambert Diffuse

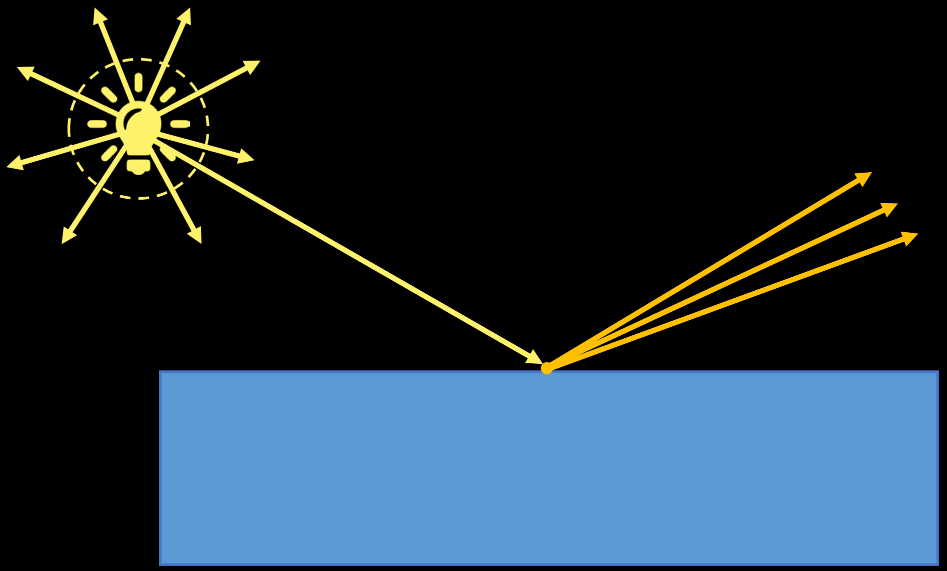
- Let’s have a look what happens when light hits an object.



Ray Tracing| BRDF – Diffuse & Specular

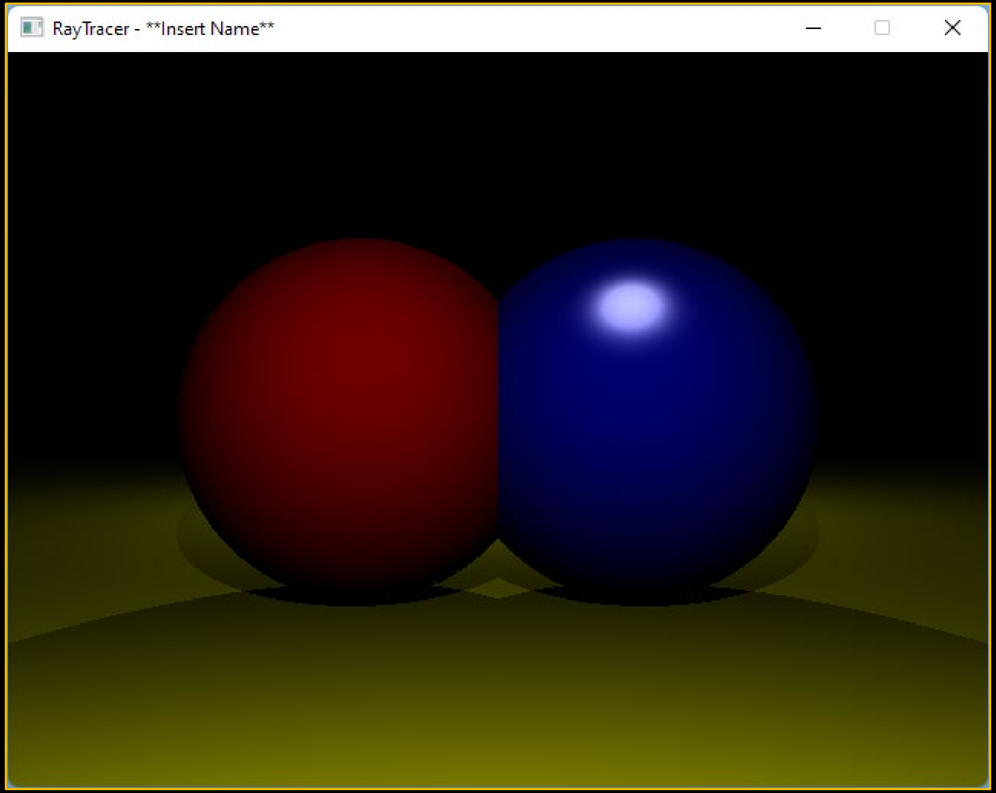
- Thus, we can split the BRDF into two parts:
  - Diffuse Reflection
  - Specular Reflection
- This is true for non-metals. Metals do not absorb and scatter light, they just reflect light!


Non-Metals


Metals

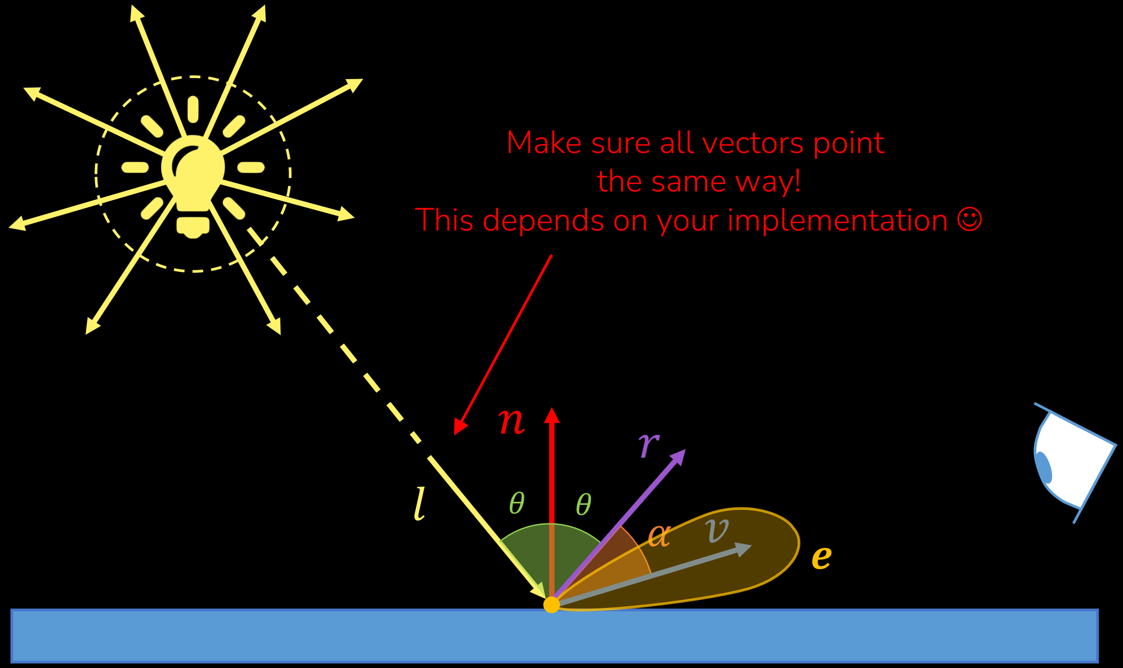
Ray Tracing| BRDF – Specular: Phong

- With the values of the next slide, and integrating it correctly, you should get something similar. (Blue Sphere is now using the Material_LambertPhong)



Ray Tracing| BRDF – Specular: Phong

- We already have the diffuse term through the Lambert Diffuse BRDF. Let's take a look how we can mimic the specular term.
- There are different models out there (Phong, Blinn, Gaussian, Beckmann, etc.), but we are going to focus on the most used, and easy to understand, one for now, Phong.


Make sure all vectors point
the same way!
This depends on your implementation ☺
n
r
l
θ
θ
α
v
e

$$\boldsymbol{reflect} = \boldsymbol{l} - 2(\boldsymbol{n} \cdot \boldsymbol{l})\boldsymbol{n}$$

$$\boldsymbol{cos}(\boldsymbol{\alpha}) = \boldsymbol{r} \cdot \boldsymbol{v}$$

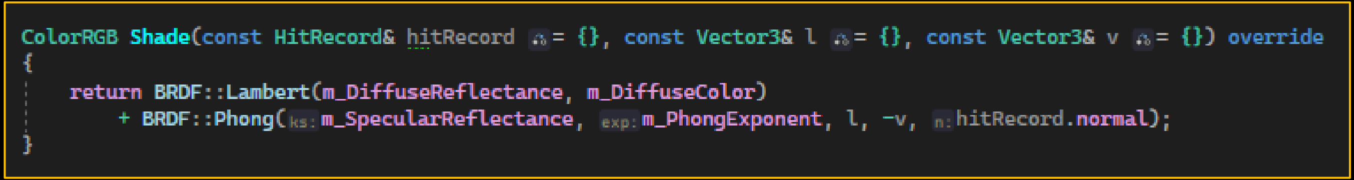
$$\textbf{Phong Specular Reflection} = \boldsymbol{k_s}(\boldsymbol{\cos}(\boldsymbol{\alpha}))^{\mathbf{e}}$$

$$\downarrow$$

$\boldsymbol{k_s}$ = Specular Reflectance Factor
$\boldsymbol{e}$ = Phong Exponent

Ray Tracing| BRDF – Specular: Phong

- Using this formula, you can program the Phong Specular BRDF.
- How to plug it in our existing code?
  - Remember the linearity rule of BRDF's? You can just add the result of the Phong BRDF to the result of the Lambert Diffuse BRDF, which will give you the correct result.

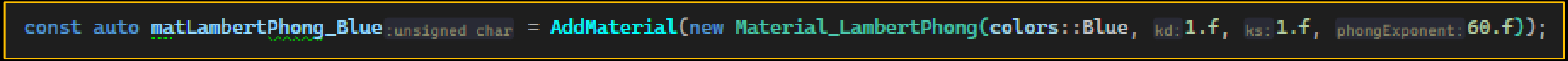


```
ColorRGB Shade(const HitRecord& hitRecord = {}, const Vector3& l = {}, const Vector3& v = {}) override
{
    return BRDF::Lambert(m_DiffuseReflectance, m_DiffuseColor)
        + BRDF::Phong(ks: m_SpecularReflectance, exp: m_PhongExponent, l, -v, n: hitRecord.normal);
}
```

Material_LambertPhong::Shade

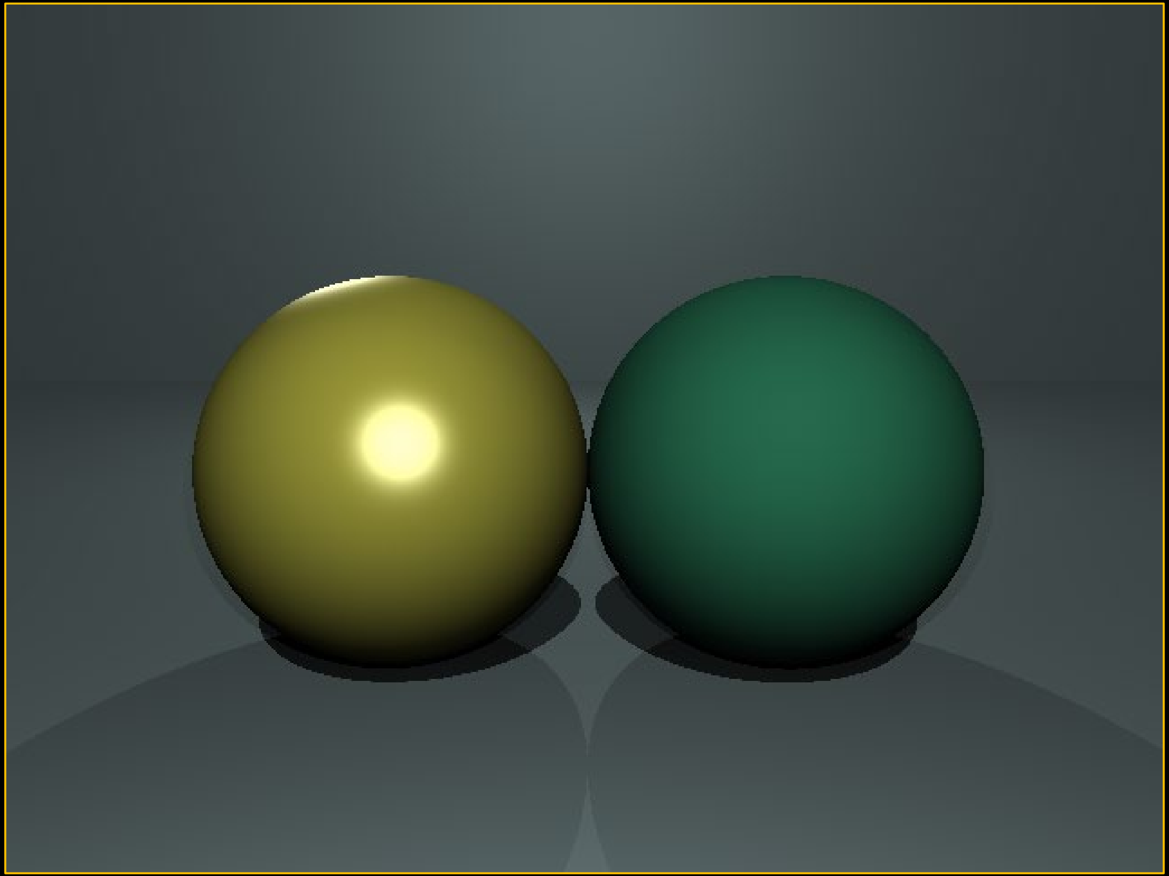
  - Internally we use: **BRDF =** $\boldsymbol{k_s\ Specular} + \boldsymbol{k_d\ Diffuse}$

  - You can control the contribution of both using the factors $\boldsymbol{k_s}$ (specular reflectance) and $\boldsymbol{k_d}$ (diffuse reflectance). See what happens when you play with these values



```
const auto matLambertPhong_Blue :unsigned char = AddMaterial(new Material_LambertPhong(colors::Blue, kd: 1.f, ks: 1.f, phongExponent: 60.f));
```

Ray Tracing| Try different setups & values!



LET'S GET
PHYSICAL
METRIX
METRIX
FITNESS BY FORCE

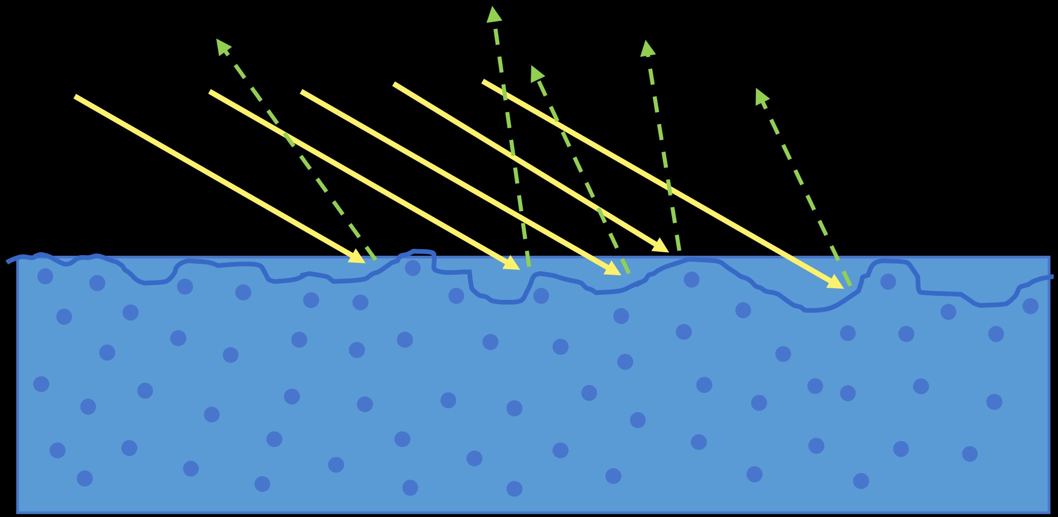
Ray Tracing| Physically Based Rendering

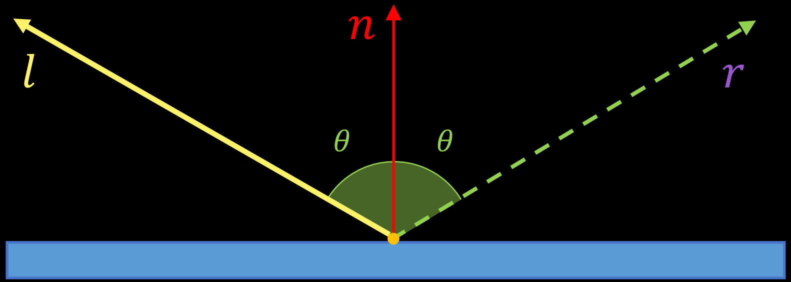
- None of the BRDFs are physically accurate. As mentioned before, all of them are approximations. Though, some are more accurate than others. The most accurate ones are Physically Based. Hence the hot term, Physically Based Rendering (PBR).
- For a BRDF to be physically based, it needs to satisfy the following conditions:
  - Have the BRDF properties:
    - Reciprocity
    - Linearity
    - Energy Conservation
  - Be based on the microfacet theory.
  - Use physically based values & be based on those values.

- So, before we can continue with the implementation, we must know what the microfacet theory is.

Ray Tracing| Microfacet Theory

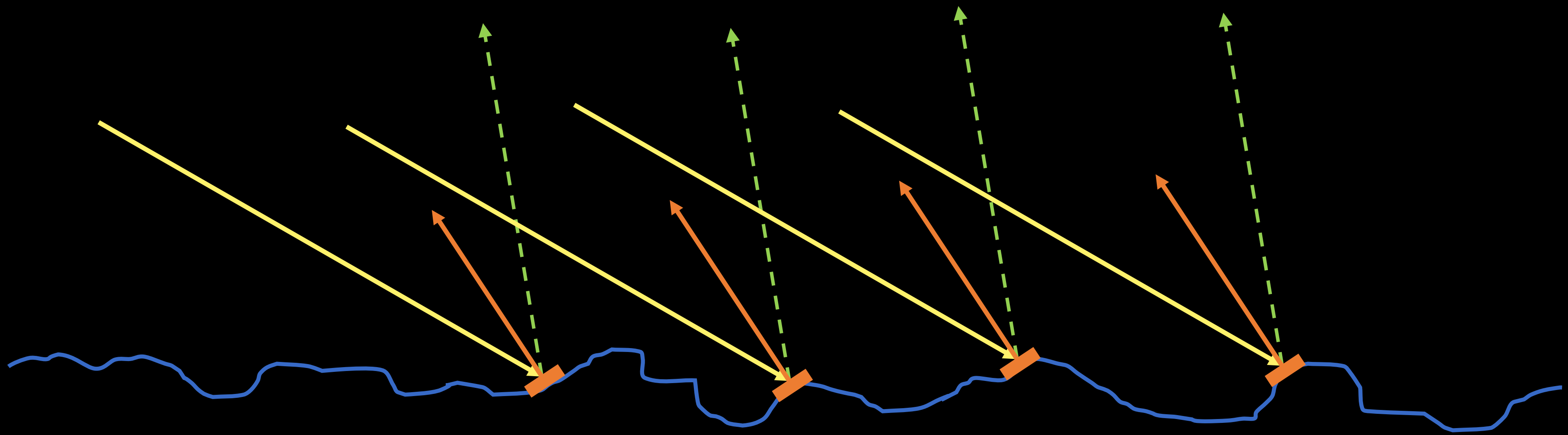
- Not a single surface is perfectly smooth, not even metal. On a molecular level they have bumps and dents. Modeling those imperfections on a nano level (nanogeometry) is impractical, so we try to analytically model the imperfections on a microfacet level (microgeometry).
- Physically Based BRDF's are all about approximating the light behavior on this microgeometry, using parameters we can control.





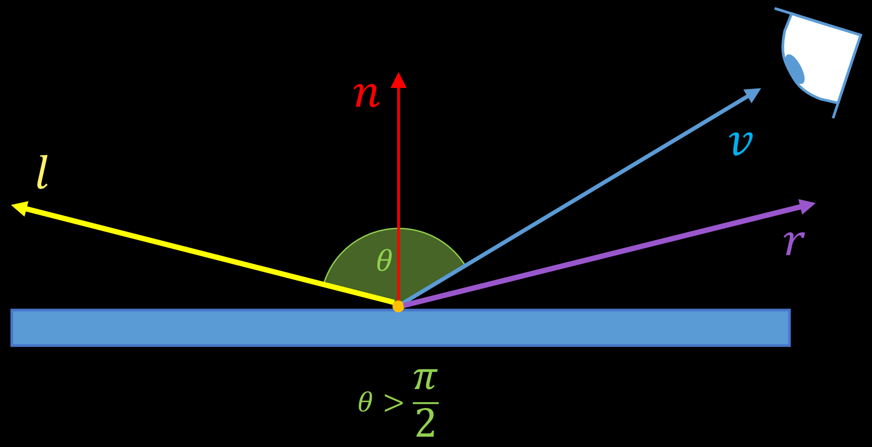
Ray Tracing| Normal Distribution

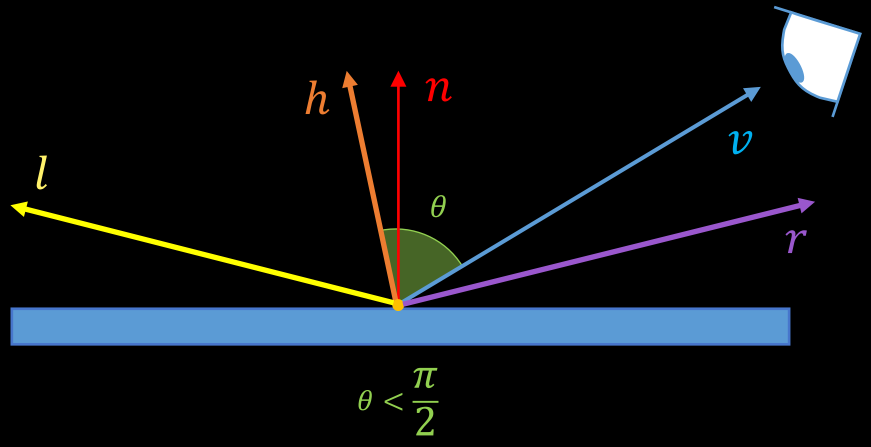
- First, we're going to model how rough the material is. In other words, modeling how many of the microfacets, over a surface area, are aligned with the **half vector**.
- Why do we use this half vector and not the normal of the surface?



Ray Tracing| Normal Distribution

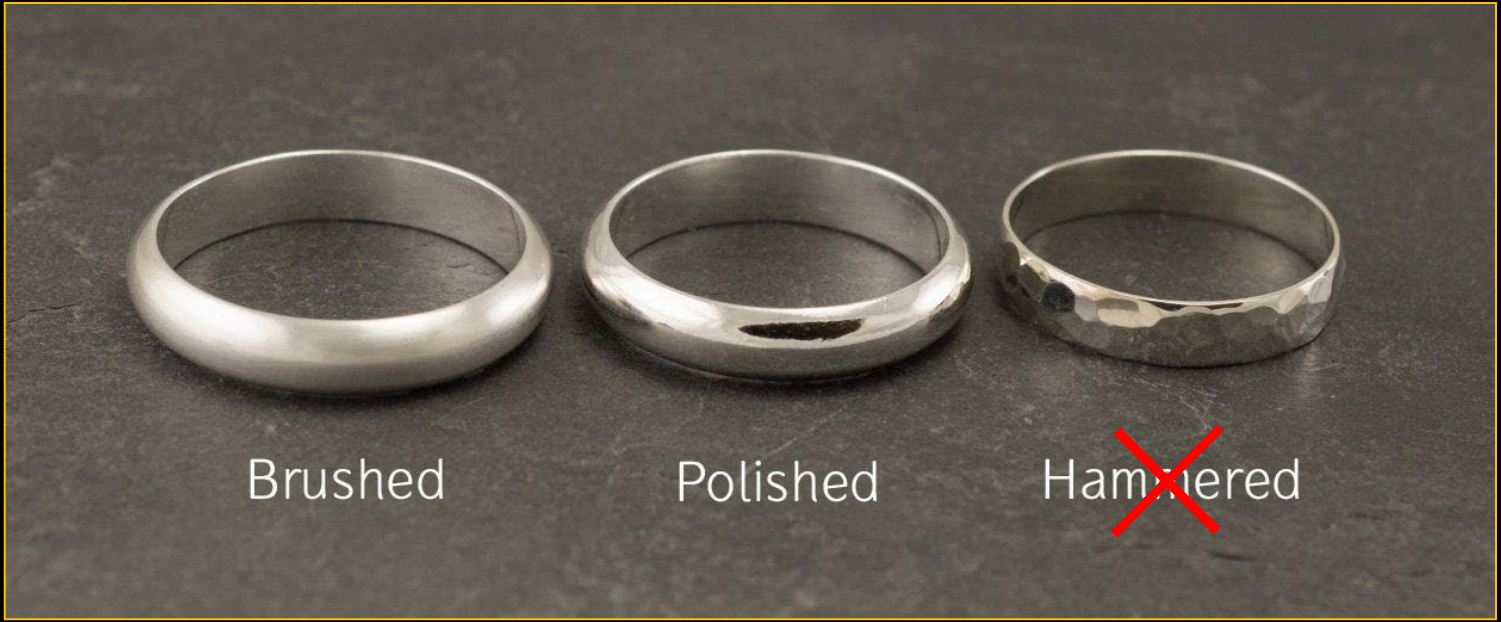
- We are comparing the light vector and the view vector, when calculating the specular reflection. Because of this the angle can become bigger than $\frac{\pi}{2}$. When using the half vector, we make sure this doesn't happen.
- The half vector can be constructed by: $h = \frac{v + l}{\|v + l\|}$





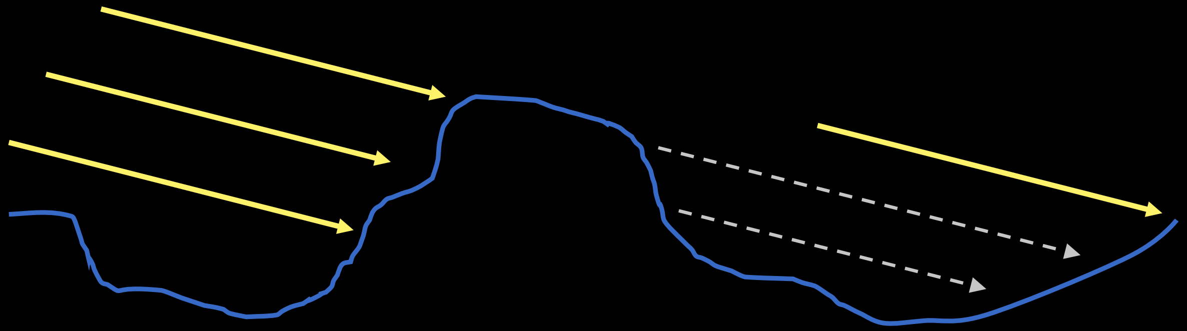
Ray Tracing| Normal Distribution

- To then determine how rough the material is, we can compare how many half vectors of the microfacets match the normal of the surface (instead of the view direction, because now we are interested in the ratio with the surface).
- The function that determines how many microfacet face the viewer is called the Normal Distribution Function.

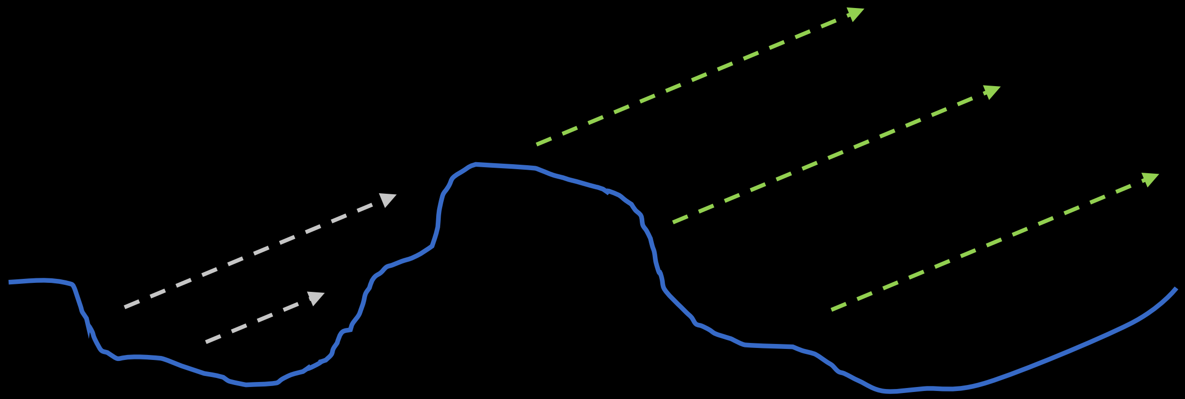


Ray Tracing| Geometry Shadowing - Masking

- Another property of rough surfaces is microfacets obstructing each other. Some incoming light will not influence microfacets because other microfacets block them. Same goes for scattered light that gets blocked by microfacets.
- The blocking of incoming light is called **shadowing**, while the blocking of scattered light is called **masking**.
- We again try to approximate this, over a surface area, through a function called the **Geometry Function**.



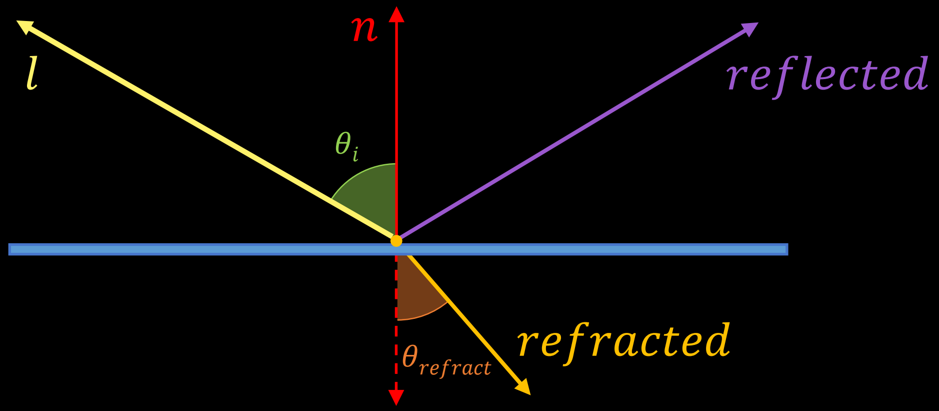
Shadowing



Masking

Ray Tracing| Fresnel

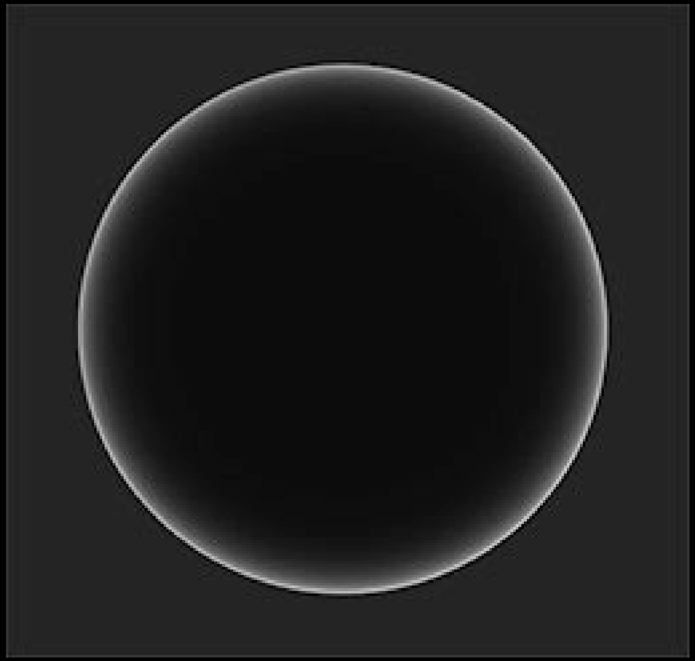
- There is one last piece missing in our puzzle. We talked about the amount of microfacets that face us, based on the roughness of the material, and how much of the incoming and outgoing light is blocked by the microfacets.
- We didn't talk about how much of the light is getting reflected versus how much light is getting refracted.
- As we've discussed, when light hits a surface, some of the light gets reflected and some of the light gets refracted. The Fresnel equation gives us the percentage of light that gets reflected, based on the view angle. Based on the ratio of reflection, and the conservation of energy, we can obtain the amount of refracted light.

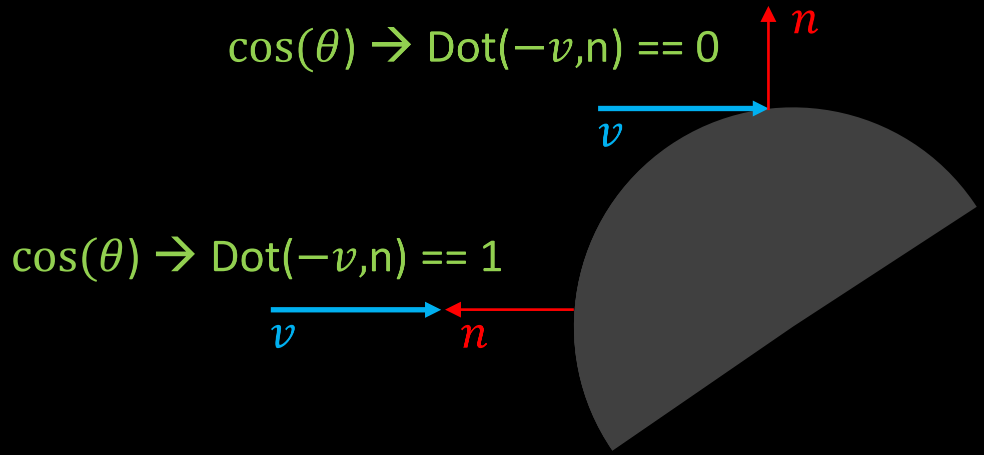


$$ratio_{snell's\ law} = \frac{\sin(\theta_i)}{\sin(\theta_{refract})}$$

Ray Tracing| Fresnel

- To makes things easy, let's look at how Fresnel looks like when we take into account the view direction and the normal of the surface. We see it has an interesting behavior…




$\cos(\theta) \rightarrow \text{Dot}(-v,\text{n}) == 0$
n
v
$\cos(\theta) \rightarrow \text{Dot}(-v,\text{n}) == 1$
v
n

Ray Tracing| Fresnel

- When the dot product between the view and the normal is (close to) 0, so they are perpendicular, we can see that the reflectance color really changes. In matter of fact, we lose a lot of the **"specular color"** of the surface, and we see more **reflections** of the environment.

Ray Tracing| Fresnel

- Some smart people captured this behavior for different types of materials and put it in a graph, which looks like this.

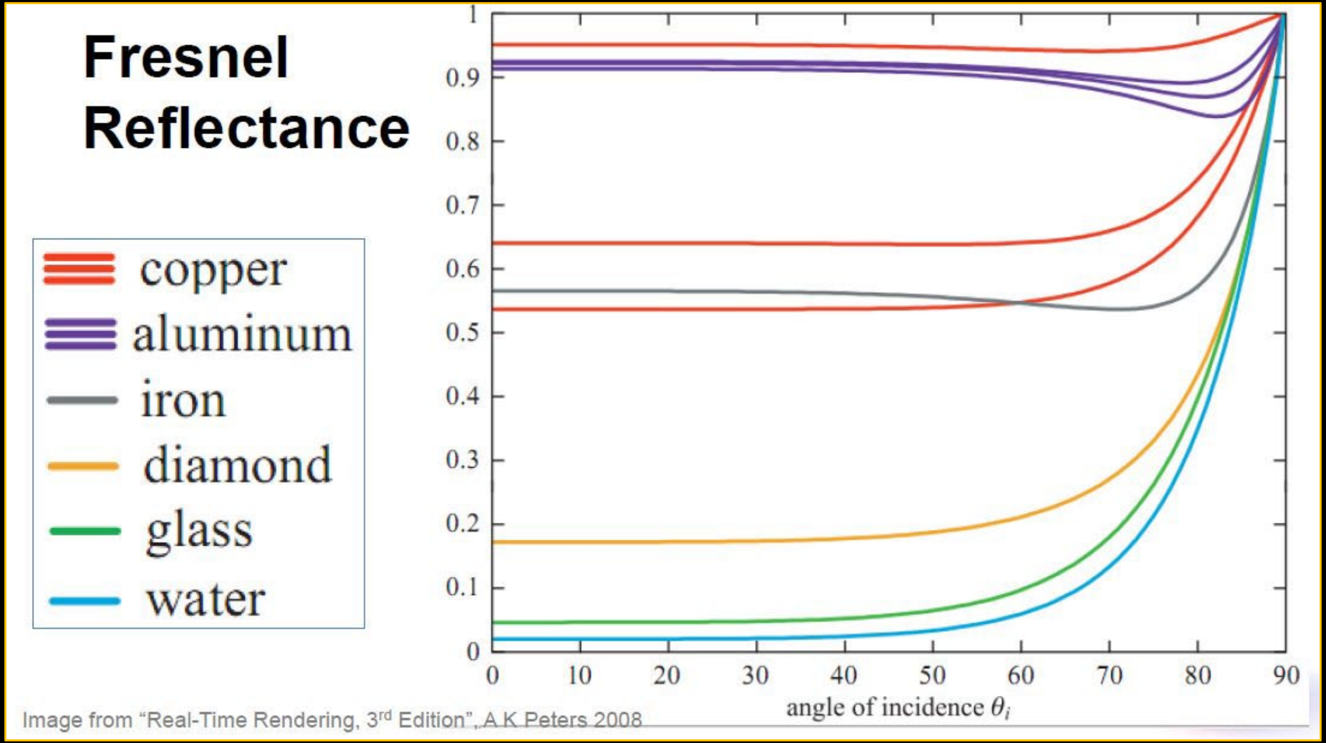


Image from "Real-Time Rendering, 3rd Edition", A.K.Peters 2008

- As we can see, when the view angle (angle of incidence) becomes more perpendicular, we see an abrupt change in "color".

Ray Tracing| Fresnel

- What do I mean with "color"? When we directly look at an object, so with a view angle of 0 degrees, we can see the surface's characteristic specular color, often called $\boldsymbol{F_0}$.

$\boldsymbol{F_0}$

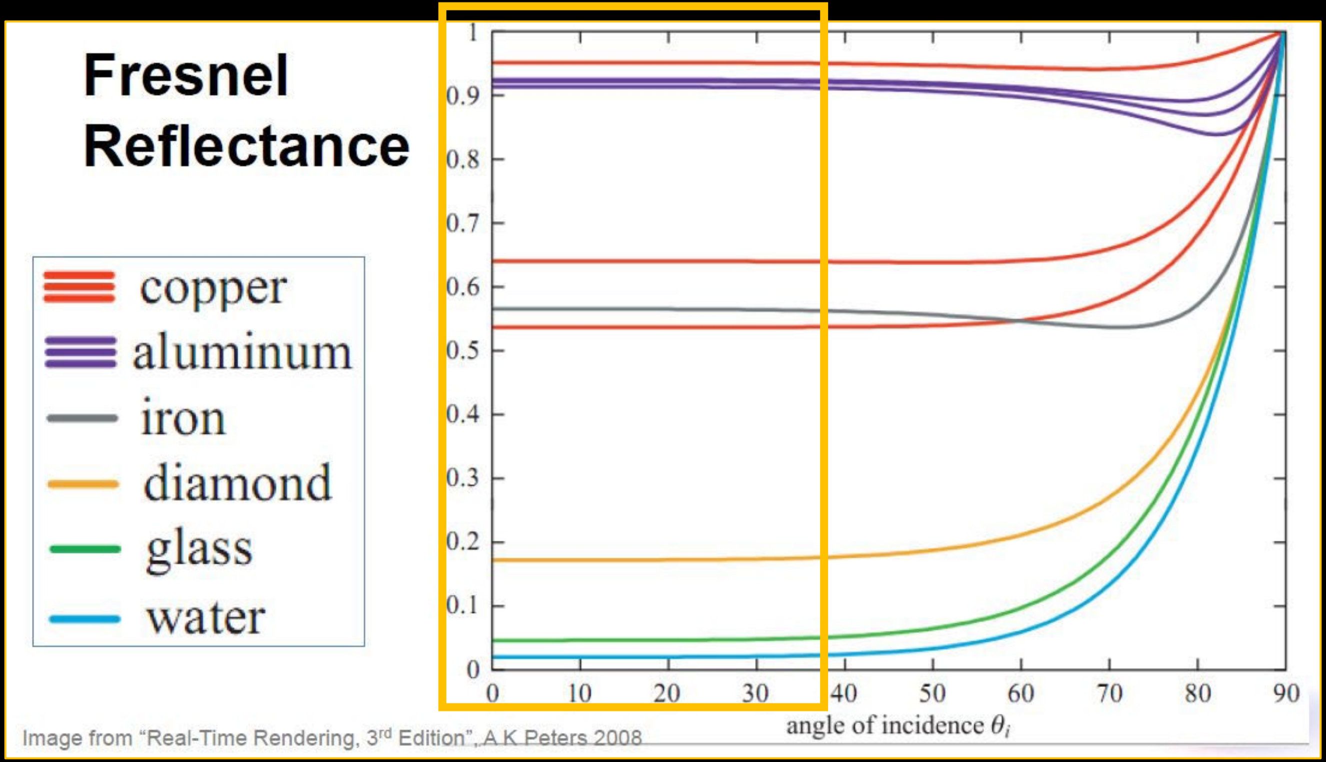
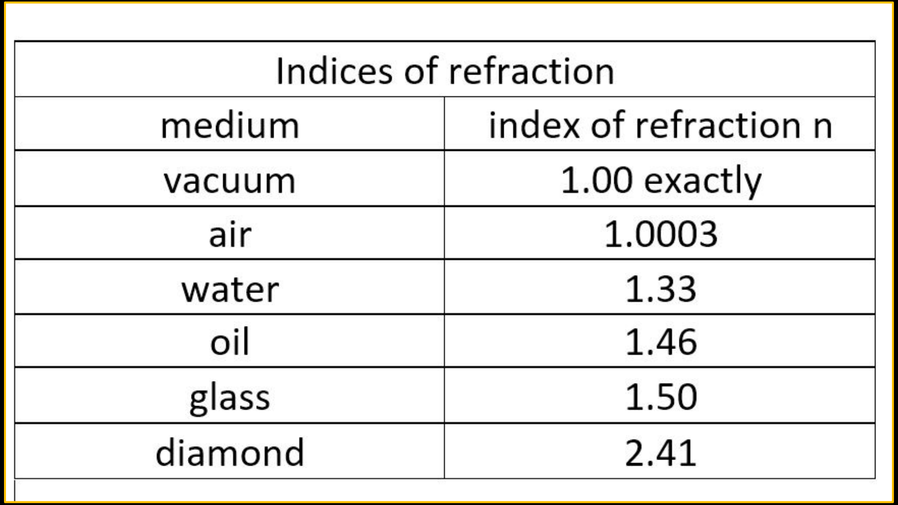


Image from "Real-Time Rendering, 3rd Edition", A.K Peters 2008

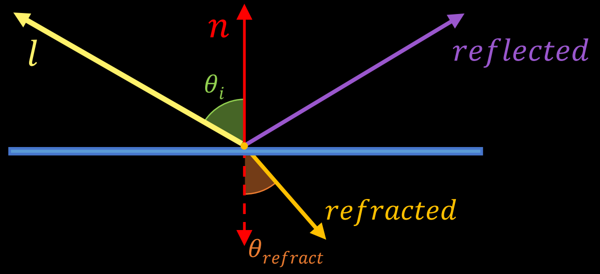
Ray Tracing| Fresnel

- Snell's Law described the ratio of reflection and refraction for light traveling from one medium into another medium.
- These values previously mentioned are captured from light traveling in air into a material.
- In our ray tracer, we are only interested in light that travels in air/vacuum to another medium, or the other way around. So, we can luckily use these values! Using these values make our lives easier. If you want to simulate light traveling through different mediums (underwater scenes), we will have to take into account different refraction indices.



Indices of refraction	
medium	index of refraction n
vacuum	1.00 exactly
air	1.0003
water	1.33
oil	1.46
glass	1.50
diamond	2.41

$$\boxed{F_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2}$$



$$F_0 = \left(\frac{1 - 1.33}{1 + 1.33}\right)^2$$

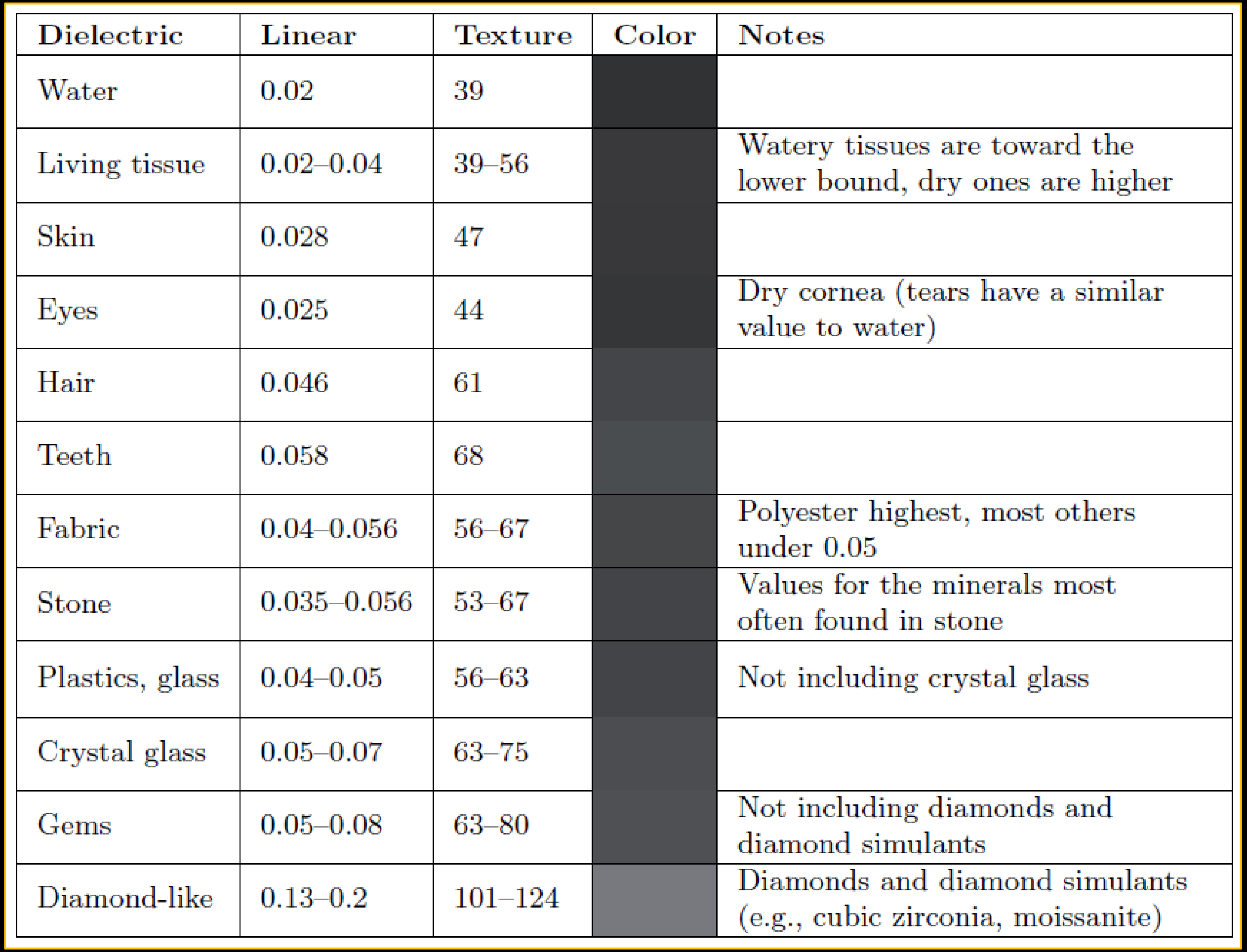
$$\downarrow$$

$$F_0 = (0.14)^2$$

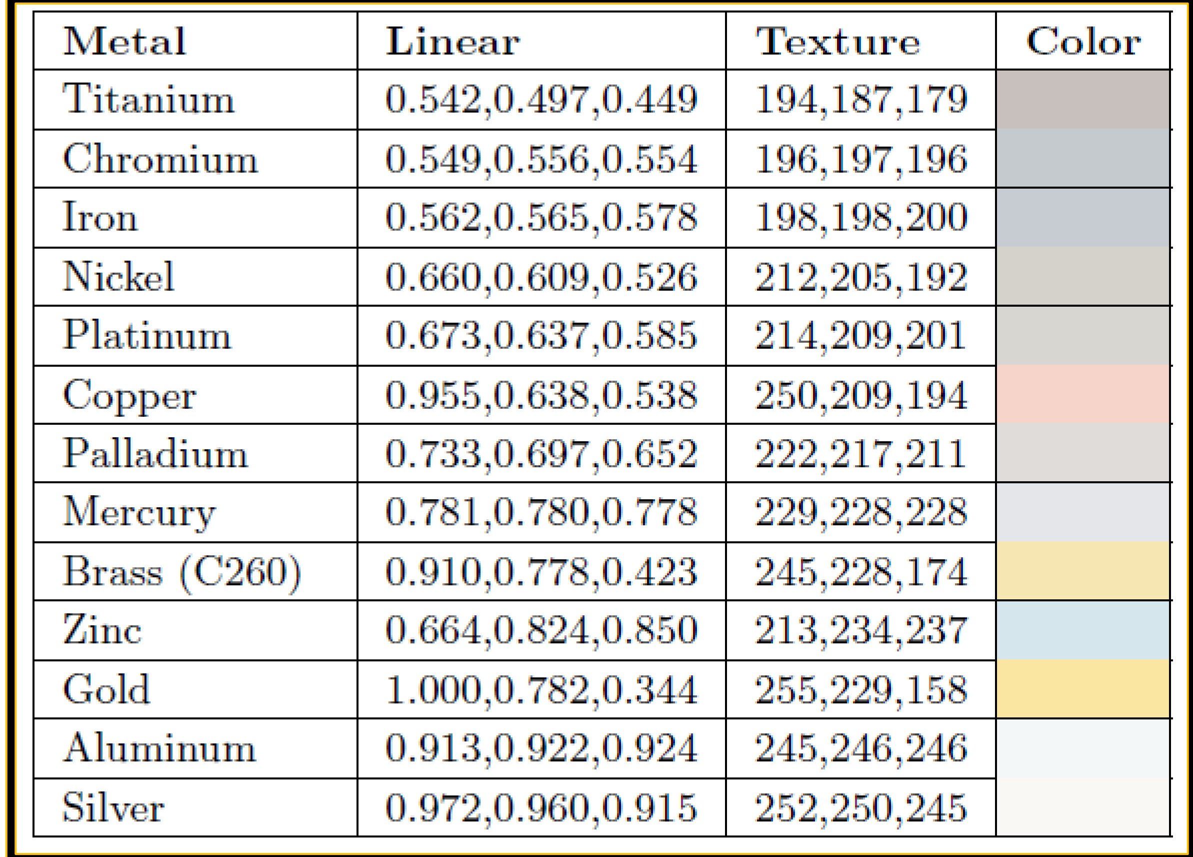
$$\downarrow$$

$$F_0 = 0.02 \rightarrow \text{Water}$$

Ray Tracing: Fresnel



Dielectric	Linear	Texture	Color	Notes
Water	0.02	39	⬛	
Living tissue	0.02–0.04	39–56	⬛	Watery tissues are toward the lower bound, dry ones are higher
Skin	0.028	47	⬛	
Eyes	0.025	44	⬛	Dry cornea (tears have a similar value to water)
Hair	0.046	61	⬛	
Teeth	0.058	68	⬛	
Fabric	0.04–0.056	56–67	⬛	Polyester highest, most others under 0.05
Stone	0.035–0.056	53–67	⬛	Values for the minerals most often found in stone
Plastics, glass	0.04–0.05	56–63	⬛	Not including crystal glass
Crystal glass	0.05–0.07	63–75	⬛	
Gems	0.05–0.08	63–80	⬛	Not including diamonds and diamond simulants
Diamond-like	0.13–0.2	101–124	⬛	Diamonds and diamond simulants (e.g., cubic zirconia, moissanite)

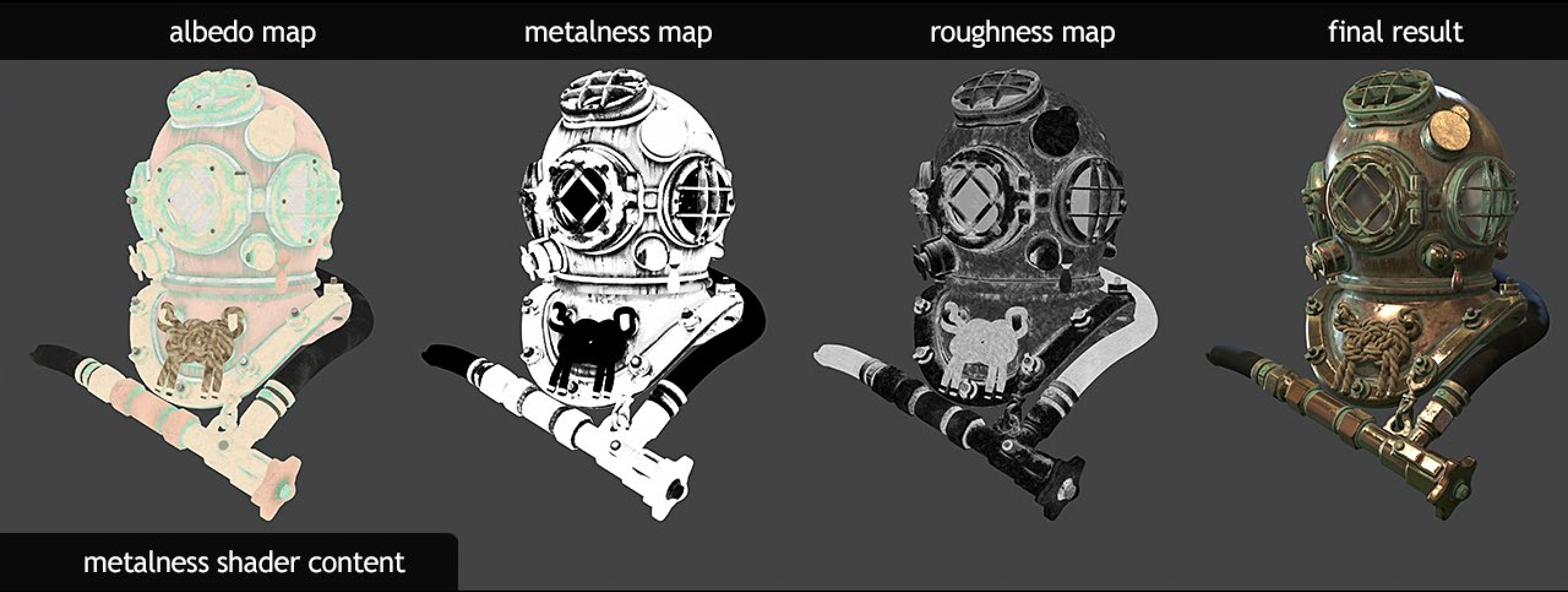


Metal	Linear	Texture	Color
Titanium	0.542,0.497,0.449	194,187,179	⬜
Chromium	0.549,0.556,0.554	196,197,196	⬜
Iron	0.562,0.565,0.578	198,198,200	⬜
Nickel	0.660,0.609,0.526	212,205,192	⬜
Platinum	0.673,0.637,0.585	214,209,201	⬜
Copper	0.955,0.638,0.538	250,209,194	⬜
Palladium	0.733,0.697,0.652	222,217,211	⬜
Mercury	0.781,0.780,0.778	229,228,228	⬜
Brass (C260)	0.910,0.778,0.423	245,228,174	🟨
Zinc	0.664,0.824,0.850	213,234,237	⬜
Gold	1.000,0.782,0.344	255,229,158	🟨
Aluminum	0.913,0.922,0.924	245,246,246	⬜
Silver	0.972,0.960,0.915	252,250,245	⬜

Ray Tracing| Fresnel

- We can also see that the range of non-metals (dielectrics) is way smaller and around the same value (0.04) compared to metals.
- So, besides the fact that metals don't scatter diffuse light, they also have a different range when it comes to specular color.
- In most game engines they will simplify the specular color. They will give all dielectrics a default value of 0.04 (plastic). Metals will sample their specular color from a texture. This is called the METALNESS workflow in a game engine. Unreal Engine 4 uses this technique. In other game engines you can choose (e.g. Unity) between this metalness workflow or the specular workflow.
- We will use the metalness workflow! → Requirement!
- Usually, you define the values using textures. The texture you use to define the diffuse color for dielectrics and the reflectivity for metals is called the albedo map. So yes, it contains two different types of information! Let's have a look…

Ray Tracing| PBR Workflow



Ray Tracing| PBR BRDF

- There are different methods to implement every function. Have a look at the following page to find more methods, and feel free to implement and compare them. But luckily for us there are already smart people who did this for us and picked the "right" methods to get a good result. Do keep in mind that there are still a lot of advancements in this area!
- Different methods: http://graphicrants.blogspot.com/2013/08/specular-brdf-reference.html
- We are going to use the following functions:
  - $\boldsymbol{D}$: Trowbridge-Reitz GGX
  - $\boldsymbol{F}$: Schlick
  - $\boldsymbol{G}$: Schlick-GGX (Schlick-Beckmann with remapped roughness)
- Let's dive into some of the implementations!

Ray Tracing| PBR BRDF

- This brings us to the PBR BRDF, that takes the three functions we've discussed into account. Just as Unreal Engine 4, we are going to use the Cook-Torrance BRDF, which looks like this:

$$f_{cook-torrance} = \frac{DFG}{4(v \cdot n)(l \cdot n)}$$

- D: Normal Distribution Function → How many microfacets point in right direction?
- F: Fresnel Function → How reflective are the microfacets?
- G: Geometry Function → How much shadowing and masking is happening because of the microfacets?
- The denominator is for the reprojection of the factors, don't worry.

Ray Tracing| PBR BRDF – D

- For the Normal Distribution function, we are going to use Trowbridge-Reitz GGX, which looks as follows:

$$N_{float}(n, h, \alpha) = \frac{\alpha^2}{\pi\left((n \cdot h)^2(\alpha^2 - 1) + 1\right)^2}$$

- So, you are going to need to use the half vector, which will be measured against the microfacets, with α being the roughness parameter.
- Input parameters are:
  - n : normal of the surface / hitpoint
  - h : half vector between view direction and light direction
  - α : roughness value squared

- For the α parameter, we are also going to use the definition of Unreal Engine 4, not Disney, which is → α = roughness^2

Ray Tracing| PBR BRDF – F

- For the Fresnel function we are going to use Schlick, which looks as follows:

$$F_{rgb}(h, v, F_0) = F_0 + (1 - F_0)\left(1 - (h \cdot v)\right)^5$$

- As mentioned, before it describes the reflectivity of the microfacets.
- Input parameters are:
  - h : half vector between view direction and light direction
  - v : view direction
  - F_0 : the base reflectivity of the surface → which is (0.04, 0.04, 0.04) for dielectrics or the albedo value for metals. We are not going to implement textures, so you will have to store an albedo color for metals. For example, Silver(0.95, 0.93, 0.88).
- Determining which value to use for F_0 is crucial! You can easily achieve this by using a metalness value (0 or 1) → F_0 = (metalness == 0) ? (0.04, 0.04, 0.04) : Albedo$_{rgb}$

Ray Tracing| PBR BRDF – G

- Finally, for the Geometry function we are going to use Schlick-GGX (Schlick-Beckmann with remapped roughness), which looks as follows:

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

- It describes overshadowing of microfacets.
- Input parameters are:
  - n : normal of the surface / hitpoint
  - v : view direction
  - k : is the α (roughness^2) remapped based on whether you use the function with direct or indirect lighting. We will use direct lightning ☺

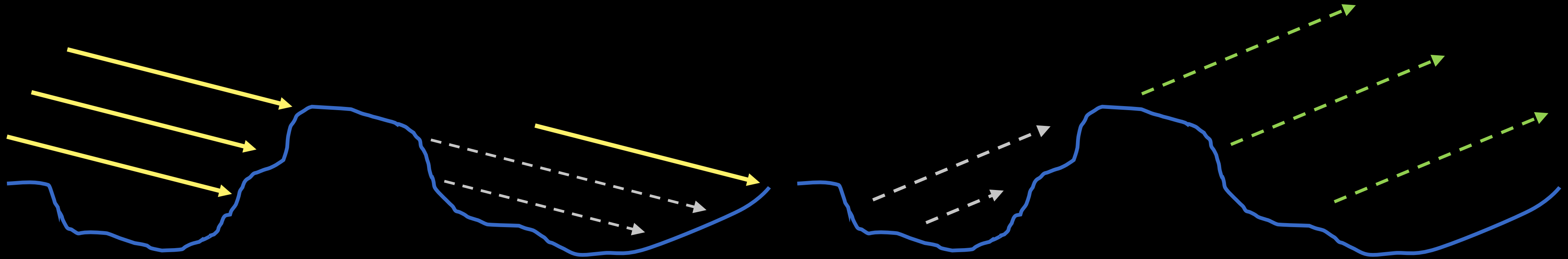
$$k_{direct} = \frac{(\alpha + 1)^2}{8} \qquad k_{indirect} = \frac{\alpha^2}{2}$$

Ray Tracing| PBR BRDF – G

- To get a correct approximation we will have to use this function for both the shadowing and the masking! Both can be combined using the Smith's method:

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

- So, you just use the previous method, calculate it for both the masking (using view direction) and the shadowing (light direction), and you multiply both values.



Shadowing Masking

- That's it… Almost ☺

Ray Tracing| PBR BRDF

- How do you combine all of this?
- This BRDF only described the specular reflectance of the material. We also need the diffuse reflectance.
- Well, it turns out the Lambert Diffuse Reflectance (aka Lambert Diffuse Reflection BRDF) we already have is good enough in most cases. There are better and more expensive once out there, but for games this gives us sufficient results.
- We only need to know how much specular reflectance ($\boldsymbol{ks}$) and how much diffuse reflectance
($\boldsymbol{kd}$) we want!
- This is pretty straightforward. Using Snell's Law we can know how much light gets refracted if we know how much light gets reflected. The Fresnel function describes how much light get reflected, so we can say: $\boldsymbol{kd_{rgb} = 1 - FresnelResult_{rgb}}$,unless it is a metal, then: $\boldsymbol{kd_{rgb} = 0}$
- And because BRDF's have linearity, we can just add them:

$$\text{FinalColor} = \boldsymbol{kd} * \text{BRDF}_{\text{LambertDiffuse}} + \boldsymbol{ks} * \text{BRDF}_{\text{cook-torrance}}$$

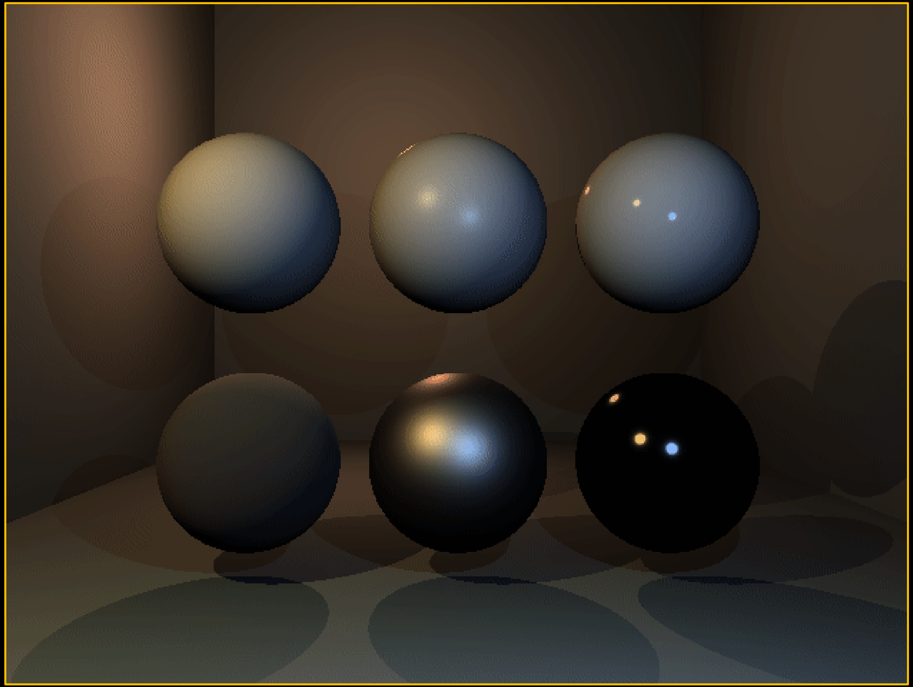
Ray Tracing| PBR BRDF (Material_CookTorrence)

- The material probably has a few parameters you want to be able to tweak:
  - Albedo Color : RGBColor → Only for metals necessary, else use (0.04, 0.04, 0.04)
  - Metalness : bool
  - Roughness : float → 0.01 = smooth, 1.0 = rough. Make sure smooth is never 0, else you cancel!
- In the shading function (Shade(…)), write the logic. Feel free to do this a bit differently (different function) based on your code. Below pseudo code of our function:

```
Determine F0 value → (0.04, 0.04, 0.04) or Albedo based on Metalness
Calculate half vector between view direction and light direction
Calculate Fresnel (F)
Calculate Normal Distribution (D)
Calculate Geometry (G)
Calculate specular => Cook-Torrance → (DFG)/4(dot(v,n)dot(l,n))
Determine kd -> 1 - Fresnel, cancel out if it's a metal (kd = 0)
Calculate Diffuse => BRDF::Lambert using the kd
Return final color -> diffuse + specular
```

Ray Tracing: PBR BRDF

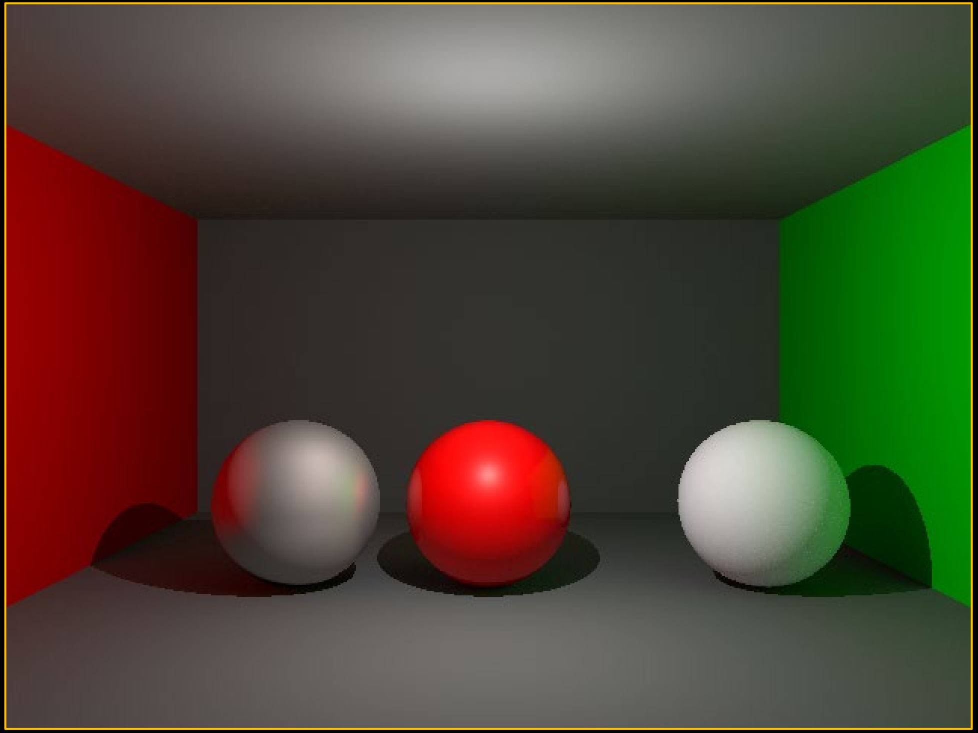
- Follow the instruction in the Implementation Todos
- If you did everything well, make 6 spheres (3 metals + 3 dielectrics) with roughness values (1.0, 0.6, 0.1). Play around with the Fresnel specular color for the metals (We used Silver) and feel free to make your scene pretty.



Ray Tracing| What to do?

- But... Why do the metals look black?
  - There are no environment reflections! When it is perfectly smooth, and there is nothing to reflect, it's black! Adding correct reflections based on the PBR BRDF can be quiet challenging, so you can leave it for now!

    - Uniform Random Sampling Methods
    - Probability Theory (CDF, PDF, etc.)
    - (Monte Carlo) Integration
    - (Multiple) Importance Sampling



GOOD LUCK!