
Ray tracing in one hour
DANIEL MARTINGER
THE CARPENTER'S CELLAR

set up

Goal

- show how ray tracing is implemented in renderers (offline and real time ones)
- get to know the math behind it (not understand it all)
- implement simple path tracer
- provide some more resources

Disclaimer

- this workshop will contain quite a lot of math mainly because it is the core of the ray tracing and without math it would be simply impossible to implement
- don`t worry, all the math is implemented for you in the small framework i made for this workshop
- math we will be implementing here is division, multiplication and addition

GPU Programming in nutshell

SIMD - single instruction multiple data

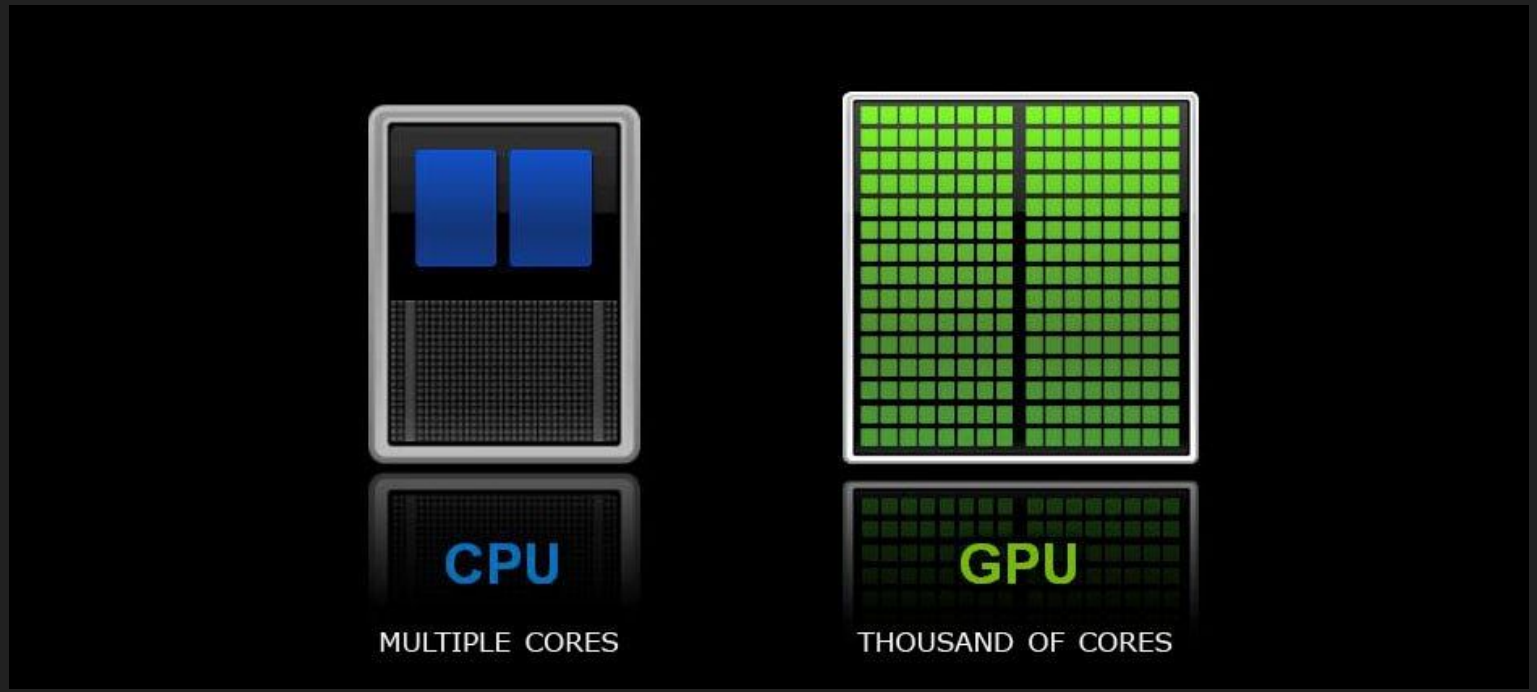


Image after https://ori-cohen.medium.com/real-life-cuda-programming-part-1-a-gentle-introduction-to-the-gpu-16078f39ede9

CPU

GPU

HLSL

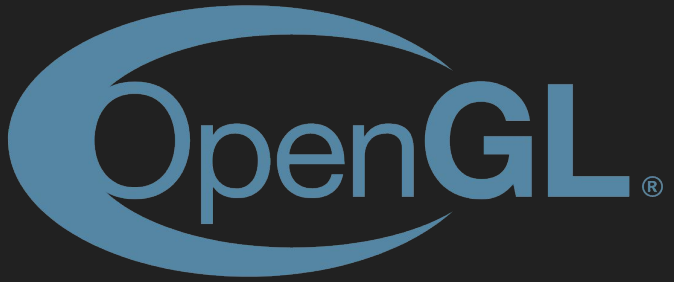
GLSL

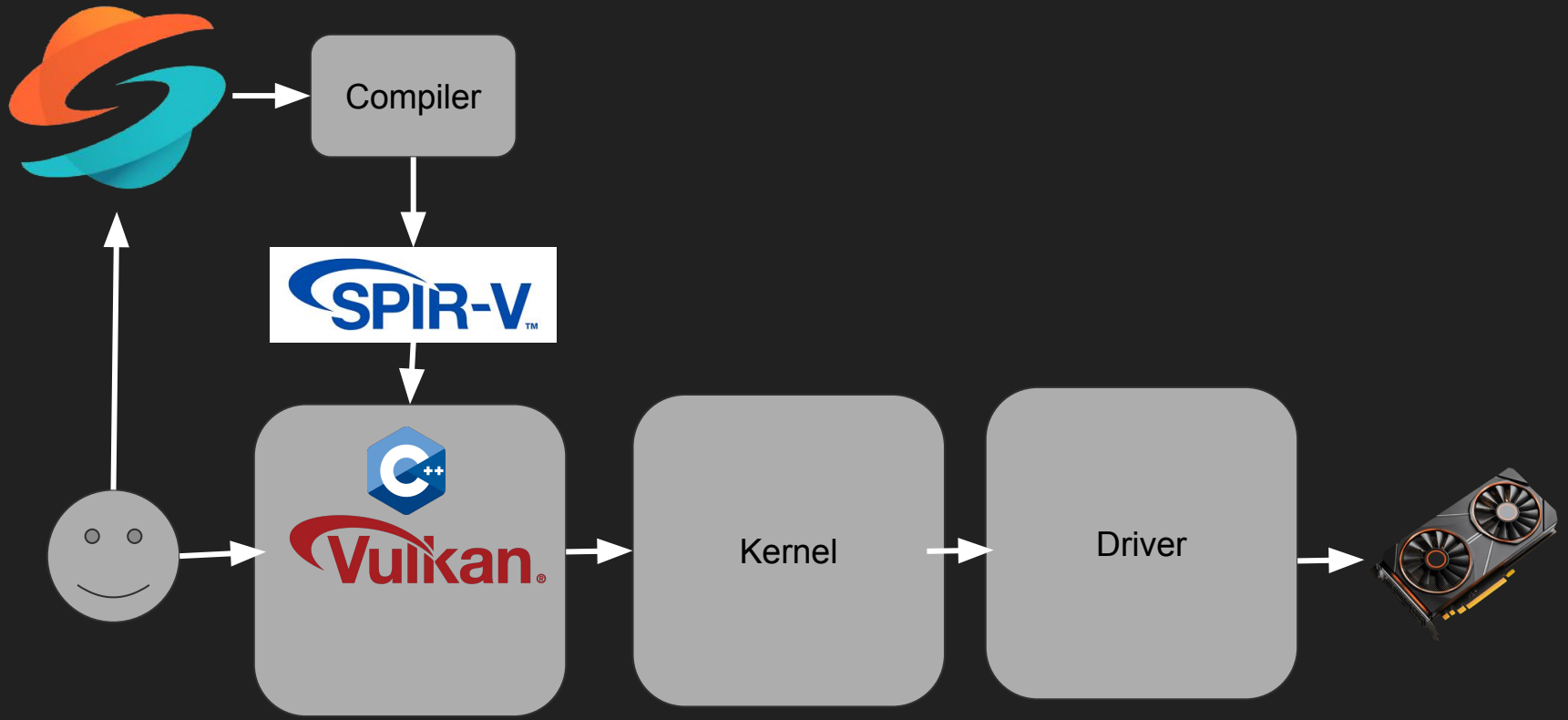
Slang™

- old shading languages were very similar to C and did not provide lot of modern functionality
- developed by Nvidia
- modernises how we write GPU programs (interfaces, generics, etc.)
- used by Valve in CS2 being slowly but surely adapted by the industry
- it is compiled language

GPU Programming basics: Graphics APIs

- languages on its own are useless
- we need a way to communicate with GPU




Compiler
SPIR-V
Vulkan
Kernel
Driver

Slang specific data types

float3 - component of 3 floating point numbers

- colour [r,g,b],
- direction [x,y,z],
- position [x,y,z]

float2, int2, float4, int4 - follow the same pattern

- Optional<T> : might or might not have a value

```
var intersection = scene.GetClosestIntersection(ray);
if(intersection.hasValue){
    var hit = intersection.value;
}else{
    // we dont have any hit
}
```

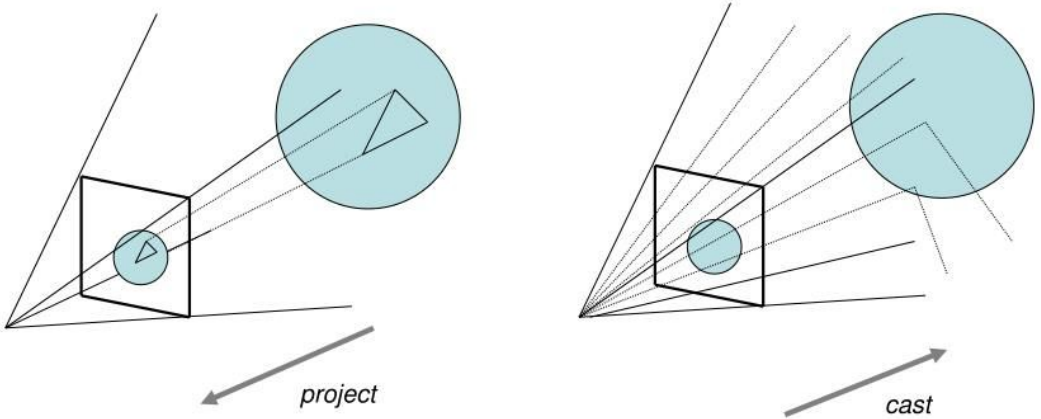
How we write the shader

- we have a vertex shader (not important for us)
- and fragment (a.k.a pixel shader)
  - this runs for every pixel on our window
  - we write our program to the function that returns *float4*
    - here float4 represents colour [x: r, y: g, z: b, w: alpha]
    - colour in GPU programming is between 0 - 1 for every component
    - [0, 0, 0] - black | [1, 1, 1] - white | [0.5, 0.0, 0.5] - light pink
    - whatever colour is returned by this function will be displayed on the screen

Image synthesizing

- projecting 3D world onto the 2D plane a.k.a our display
- 2 most popular approaches
- Ray tracing and rasterization
- Rasterization
  - very fast ~0.8ms
  - very hard to approximate effects like reflection, refraction and other global illumination techniques
  - projecting primitives (eg.: triangles) on screen and colouring pixels
- Ray tracing
  - slow
  - global illumination, reflections and other effects accounted for by default
  - shooting rays through each pixel and gathering radiance

Rasterization vs. Raytracing



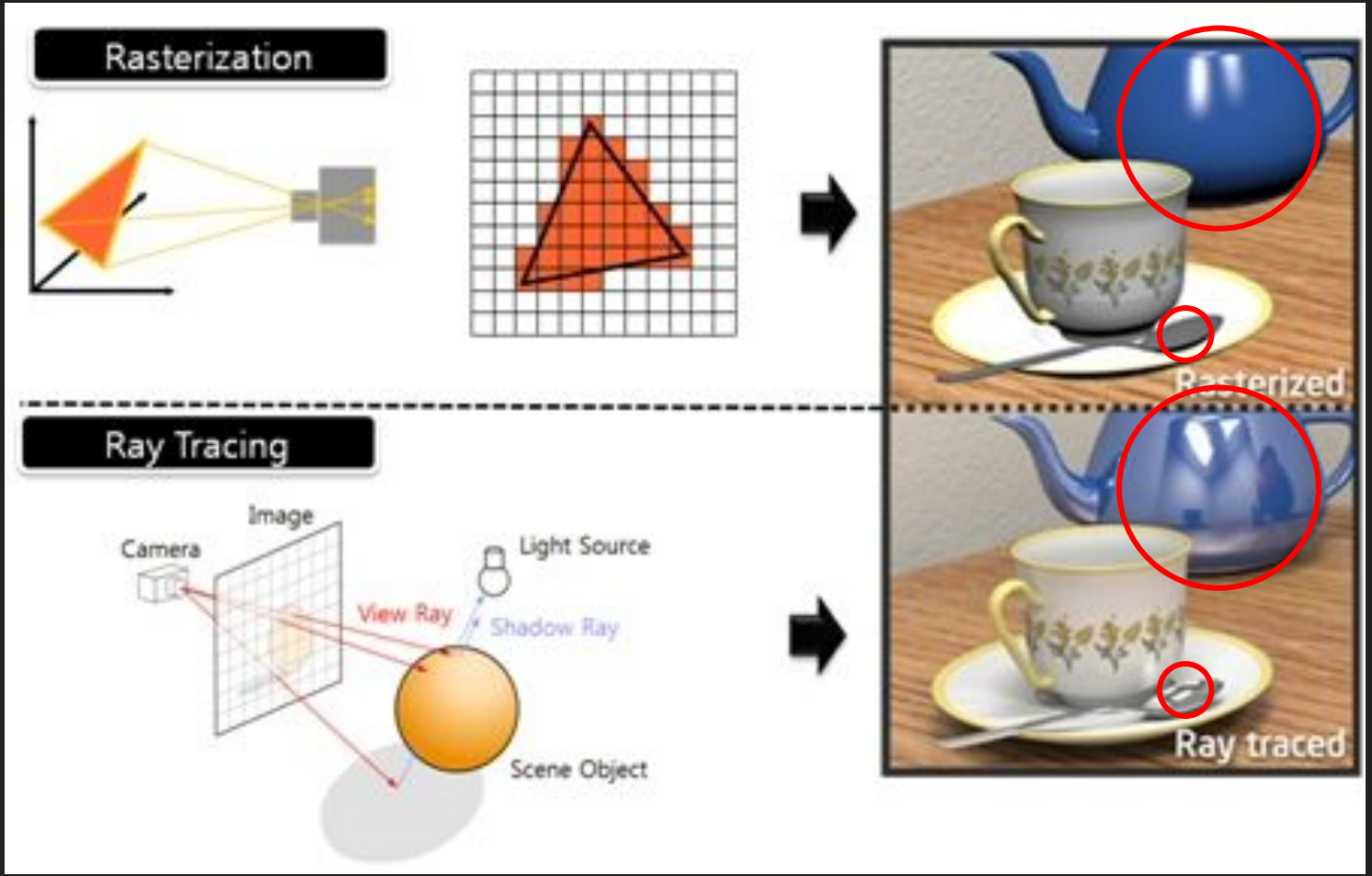
Rasterize:

- Project polygons onto picture plane
- Efficient hardware. OpenGL, DirectX

Raytrace:

- Cast light rays into scene through picture plane.

image after https://www.pixelsham.com/2019/10/24/whats-the-difference-between-ray-tracing-and-rasterization/r

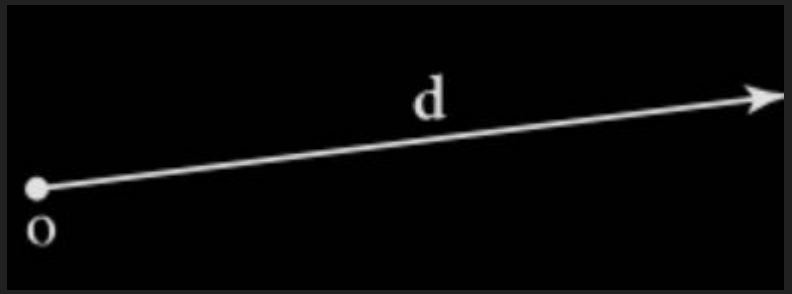

Rasterization
Ray Tracing
Image
Camera
Light Source
View Ray
Shadow Ray
Scene Object
Rasterized
Ray traced

Ray

Ray, semi-infinite line specified by direction (*d*) and origin (*o)*

Mathematical parametric representation given by function of t :

$$r(t) = o + d * t; 0 < t < \infty$$



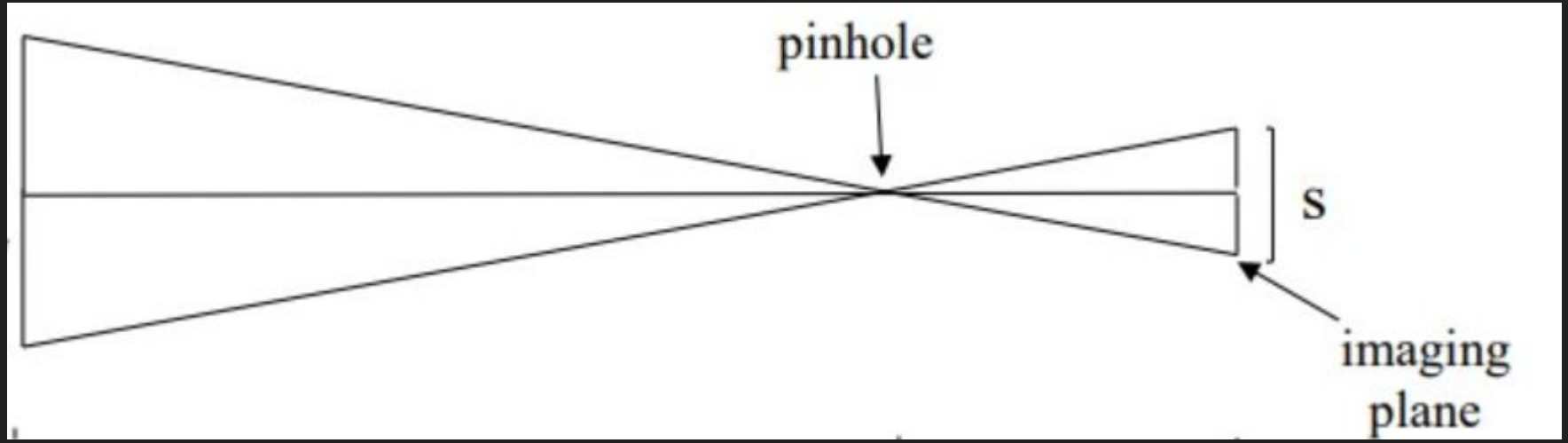
- In ray tracing Rays usually have min (t_min) and max (t_max)
- t_max = to preventing tracing ray to infinity
- t_min = hit surface position or camera origin, usually 0.0 or very small number

In code, ray is given by the struct *RayDesc* provided by Slang it looks like this.

```
struct RayDesc{
    float3 origin, direction;
    float tMin, tMax;
}
```

Camera and image plane

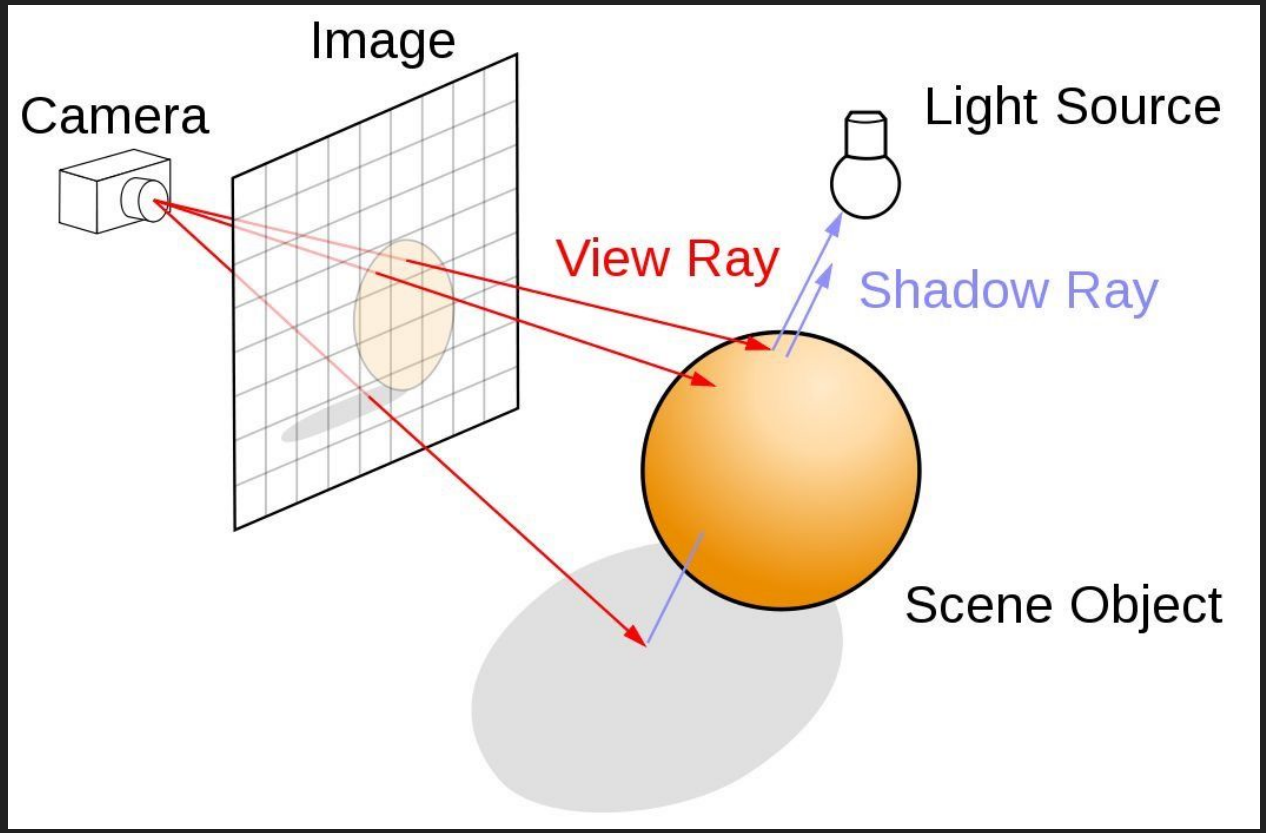
- to shoot a rays to scene we have to define a camera
- simplest camera representation is a pinhole
- not realistic, simplest to model



Camera in code

- located in *modules/camera.slang* and contains struct that represents the pinhole camera
- main purpose
  - define rays
  - store radiance collected through the scene which is arriving at the image plane (film)

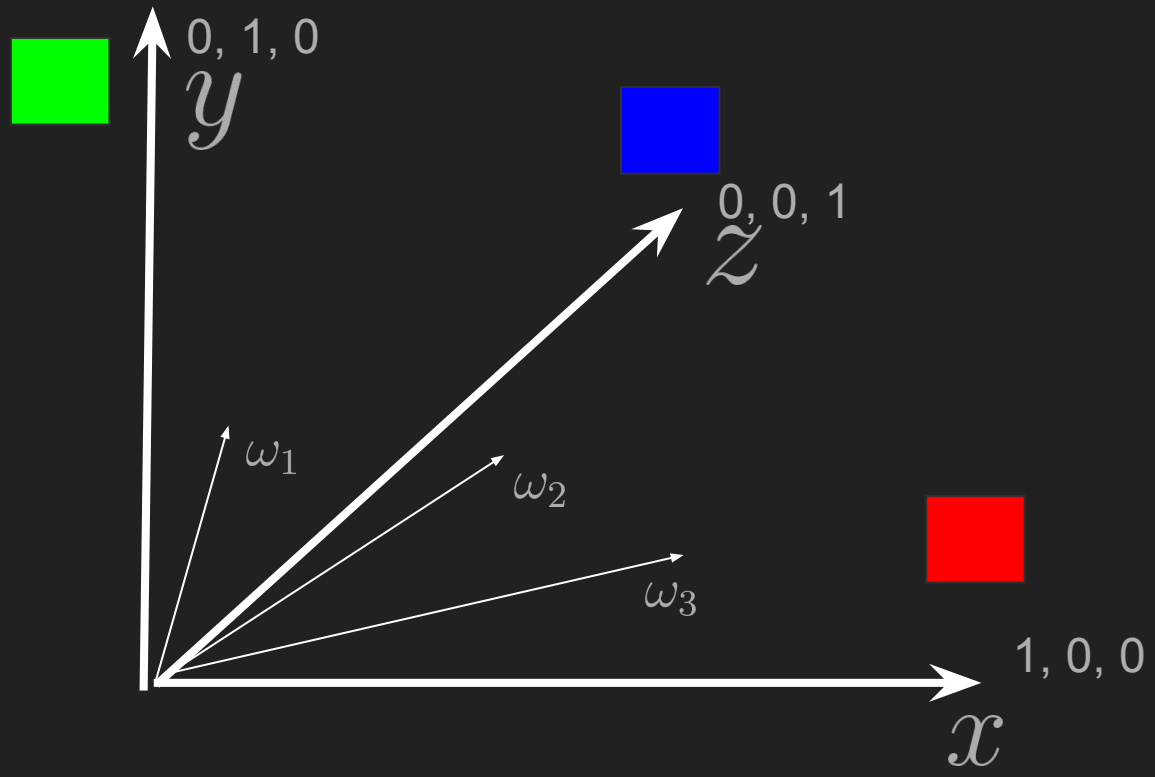
Lets shoot rays



Let`s implement this

Coordinate system

$||\omega|| = 1$



Intersection

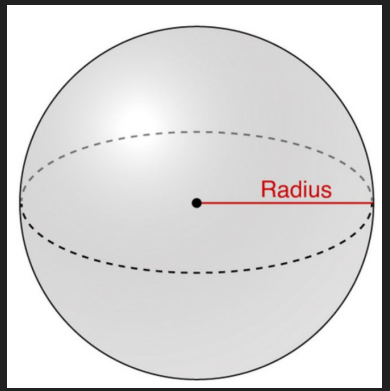
- we have rays being shot to the scene
- we have to define scene somehow, it will be done by intersecting the ray with the geometry in the scene and retrieving the information about the intersection
- we have two options
  - geometrical surfaces (by describing points and edges, usually as triangles)
  - implicit surfaces (pure mathematical representation)
- we will go with implicit surfaces as their are simple to implement and will run much faster then geometric surfaces (mainly due to number of shapes)

Spheres

$$x^2 + y^2 + z^2 = R^2$$

where x, y, z is position of the sphere in the world and R is its radius

if we consider $x^2 + y^2 + z^2$ as coordinates of point P we get:



$$P^2 - R^2 = 0$$

to calculate the intersection we have to substitute the equation of the ray

$$r(t) = o + d * t$$

to the equation of the sphere

$$P^2 - R^2 = 0$$

which gives

$$(o + d * t)^2 - R^2 = 0$$

$$d^2t + 2odt + (o^2 - R^2) = 0 \qquad ax^2 + 2bx + c$$

this is a quadratic equation and we will try to solve for the t

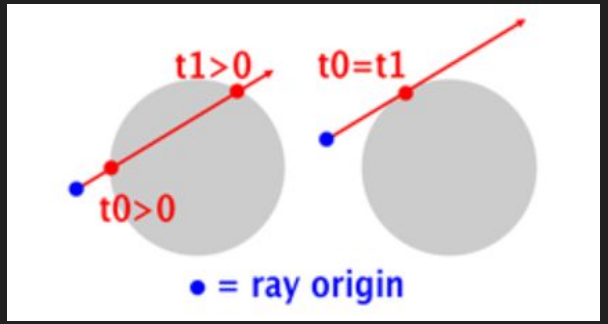
based on the discriminant we will solve the equation and get the value for *t*

if the discriminant = 0 the ray intersects sphere exactly in one place

if 0 > discriminant then ray intersected with the sphere 2 times

if discriminant < 0 we do have missed our sphere

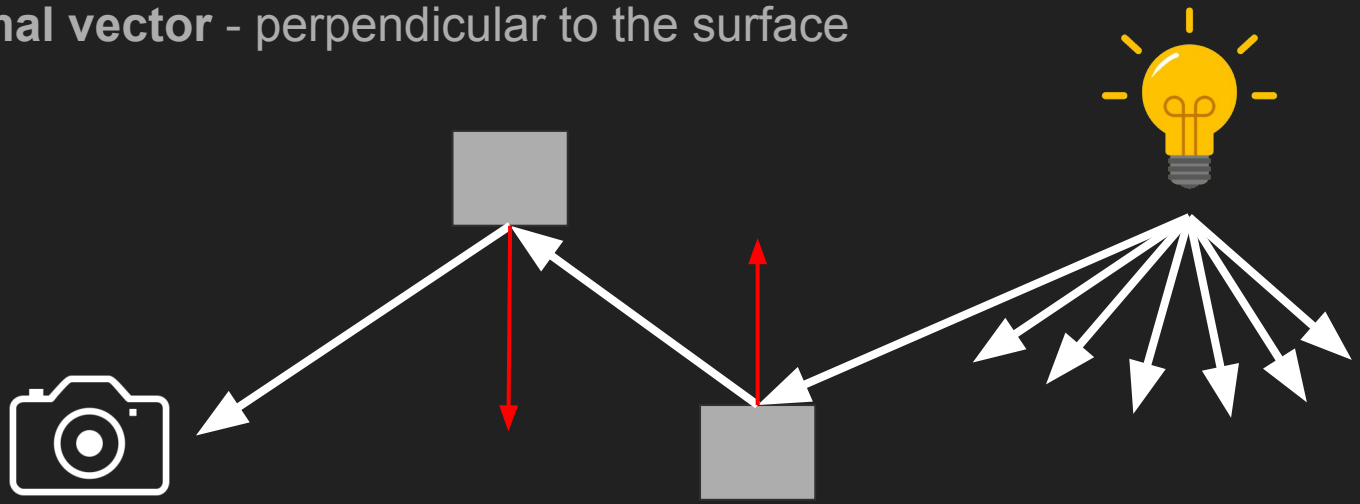
t - is the distance along the ray where the sphere was intersected



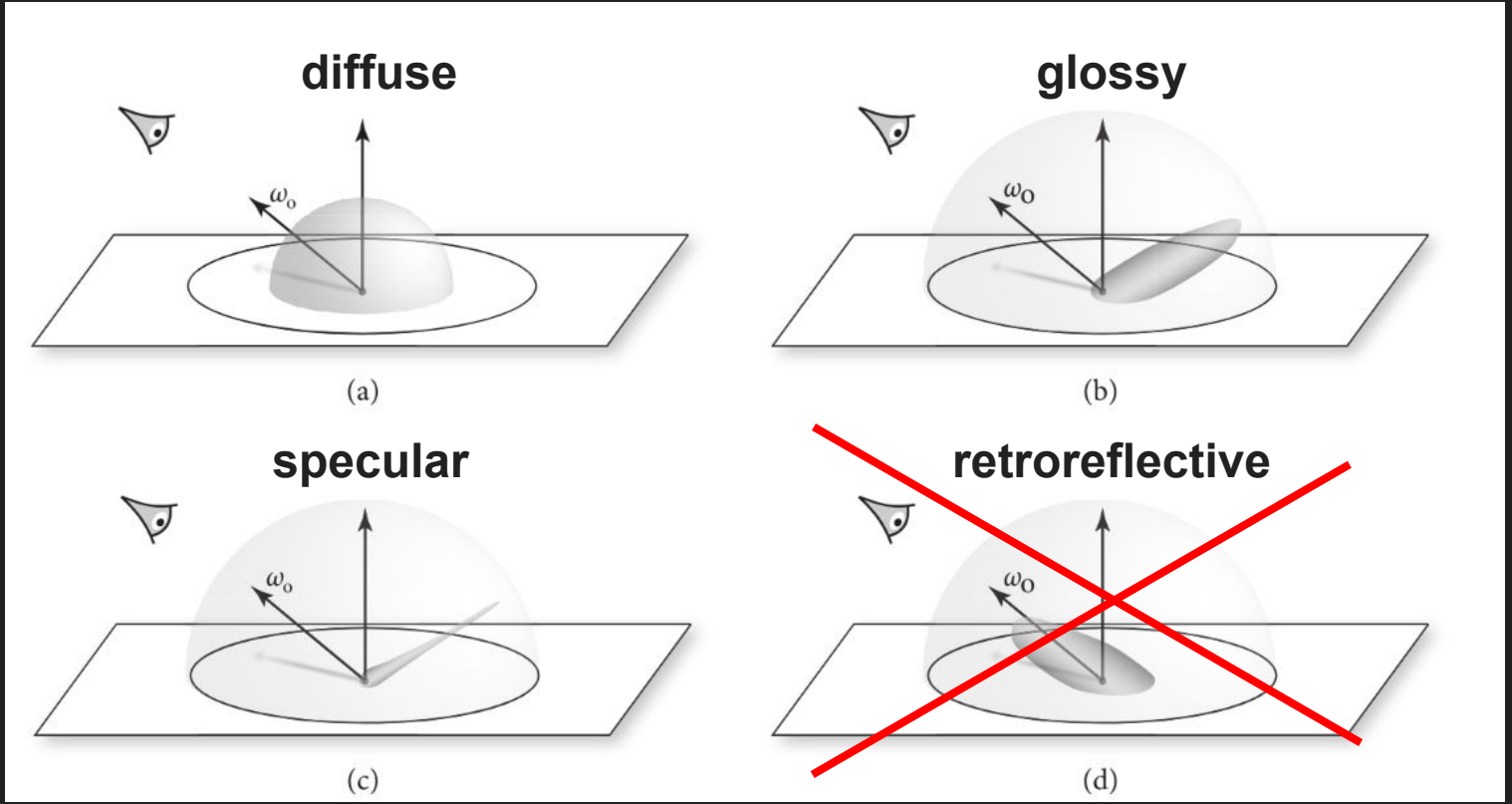
Lets implement this

Path tracing

- tracing light paths from the light sources and how they scatter around in the scene
- **normal vector** - perpendicular to the surface



Light surface interaction



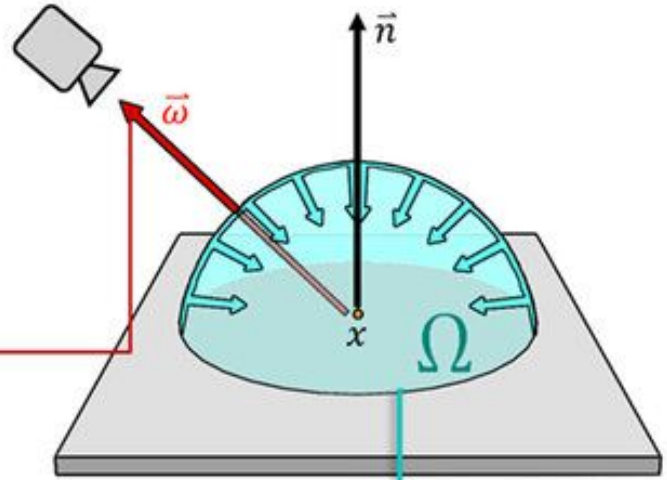
https://pbr-book.org/4ed/Reflection_Models

LTE - Light transport equation

- describes how light is being transported in the scene
- core of the ray tracing algorithm

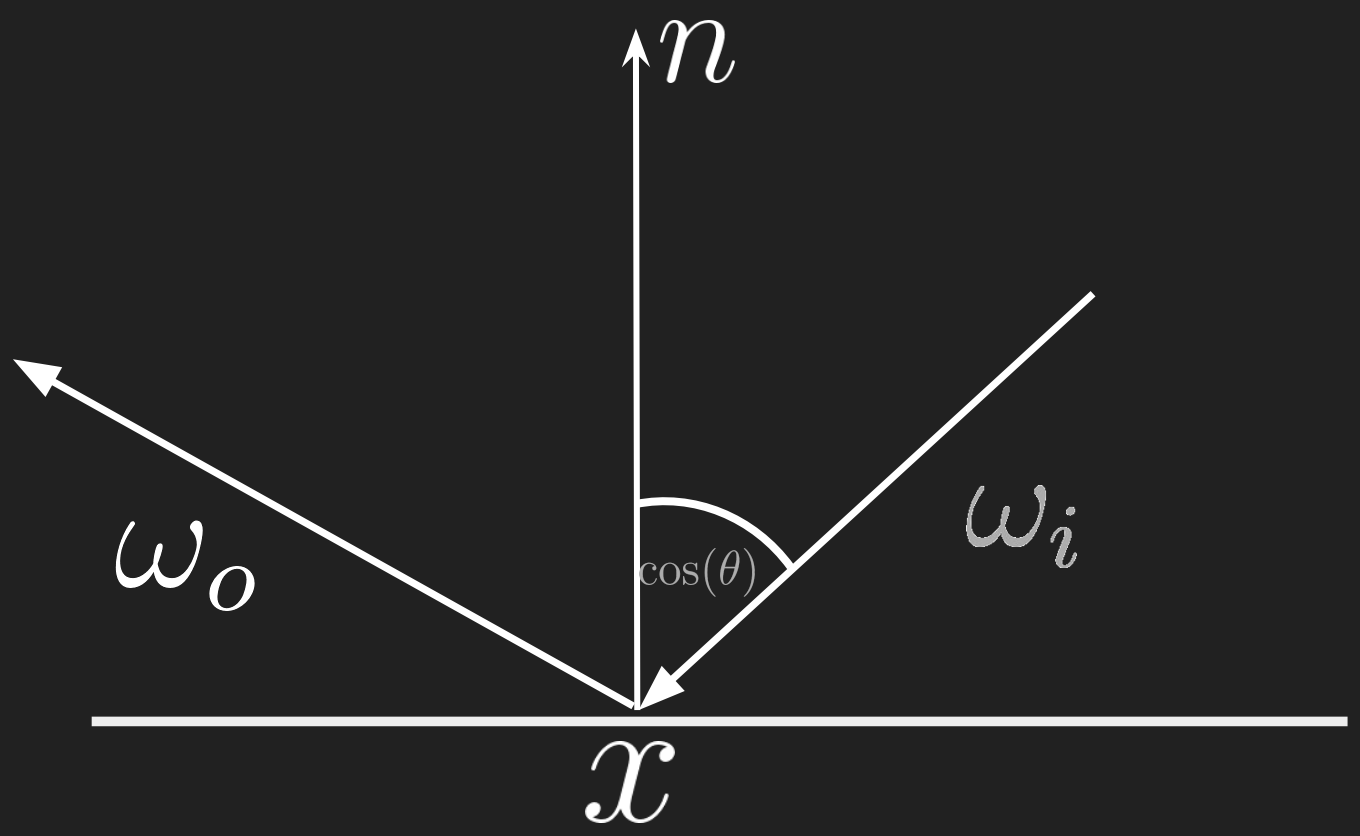
$$L_o(\boldsymbol{x},\omega_o) = L_e(\boldsymbol{x},\omega_o) + \int_{\Omega} f_r(\boldsymbol{x},\omega_i,\omega_o)\, L_i(\boldsymbol{x},\omega_i)\cos(\theta) d\omega$$

Le - emitted light
fr - BRDF (discussed later)
Li - incoming light to our surface
cos(theta) - cosine of the angle between normal vector and incoming light direction



$$L(x,\vec{\omega}) = L_{e(x,\vec{\omega})} + \int_{\Omega} L_i(x,\omega') f_r(\vec{\omega}' \rightarrow \vec{\omega}) \cos\theta' \, d\omega'$$

Hardware Acceleration of Progressive Refinement Radiosity using Nvidia RTX - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/The-fundamentals-of-the-rendering-equation-L-is-a-weighted-integral-over-all-incident_fig4_369557201 [accessed 12 Nov 2025]



$$L_o(\boldsymbol{x},\omega_o)=L_e(\boldsymbol{x},\omega_o)+\int_{\Omega} f_r(\boldsymbol{x},\omega_i,\omega_o)\, L_i(\boldsymbol{x},\omega_i)\cos(\theta)d\omega$$

Monte carlo estimation

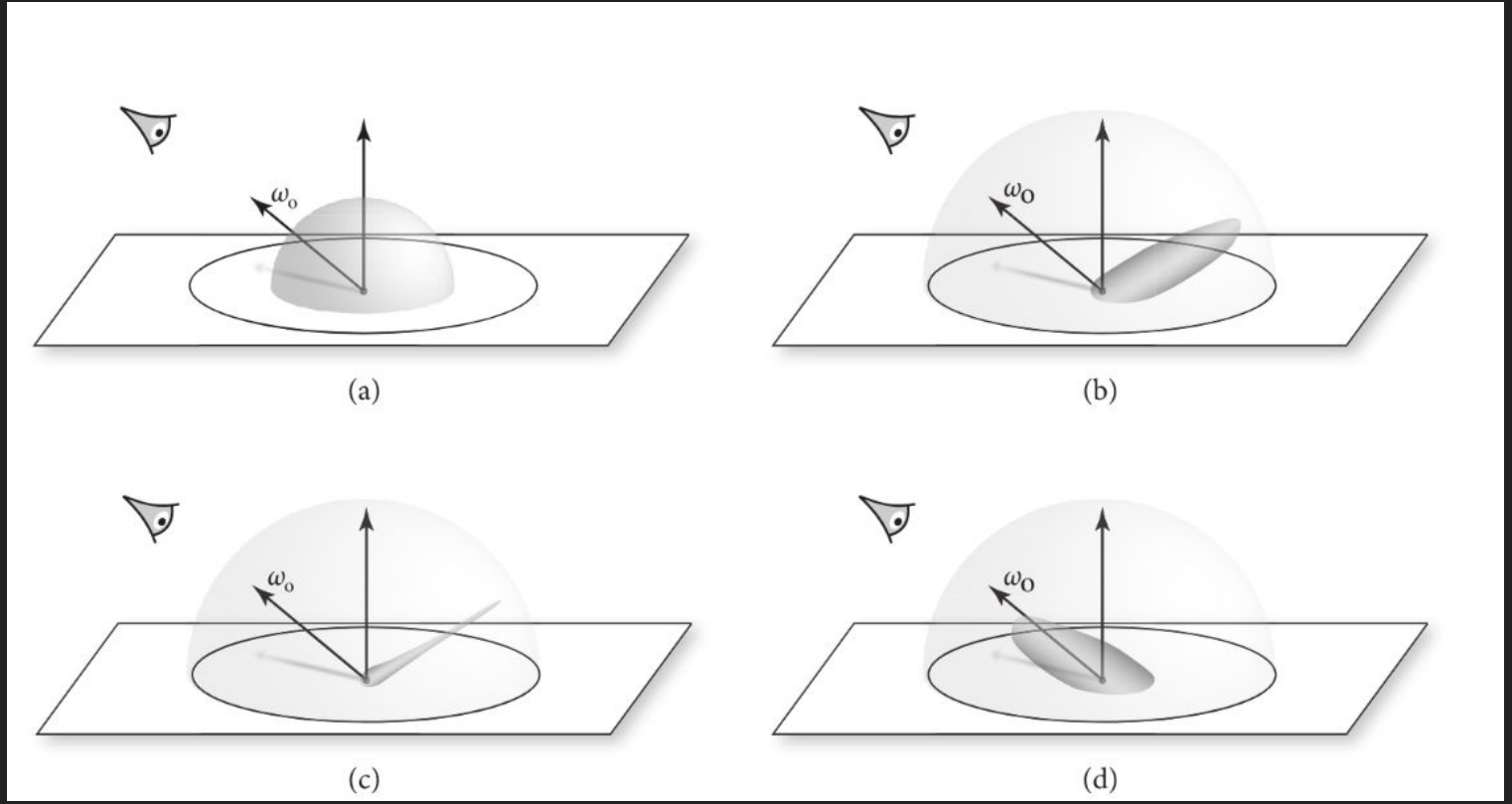
- it is impossible to solve the integral analytically, therefore we have to approximate it
- we do this by selecting random directions where we will trace next rays once we encounter the surface
- we then sum up and avere the result

$$L_o(\boldsymbol{x},\omega_o) \approx L_e(\boldsymbol{x},\omega_o) + \frac{1}{N}\sum_{k=1}^{N} \frac{f_r(\boldsymbol{x},\omega_i^{(k)},\omega_o)\, L_i(\boldsymbol{x},\omega_i^{(k)})\, (\boldsymbol{n}\cdot\omega_i^{(k)})}{p(\omega_i^{(k)})}$$

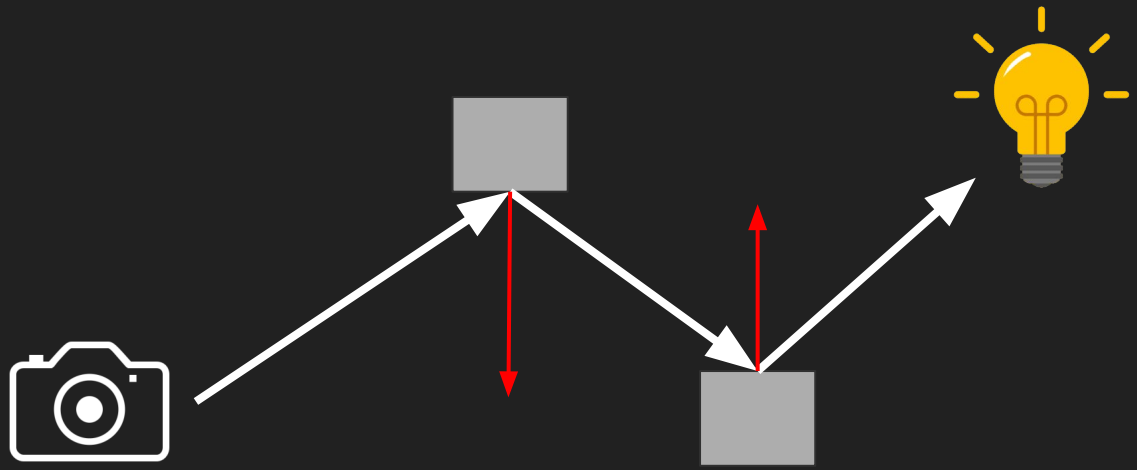
BRDF

- How light scatters from the surface is expressed by the *BRDF* (bidirectional reflectance distribution function)
- In essence it calculates the colour of the surface
- Each scattering event has different brdf based on the material of the object
- We have 2 brdf
  - specular = perfect mirror + glossy (recall, that we don`t do physically correct path tracer)
  - diffuse = equal probability to all directions
- expressed as function of 2 vectors and position (x, w_i, w_o)

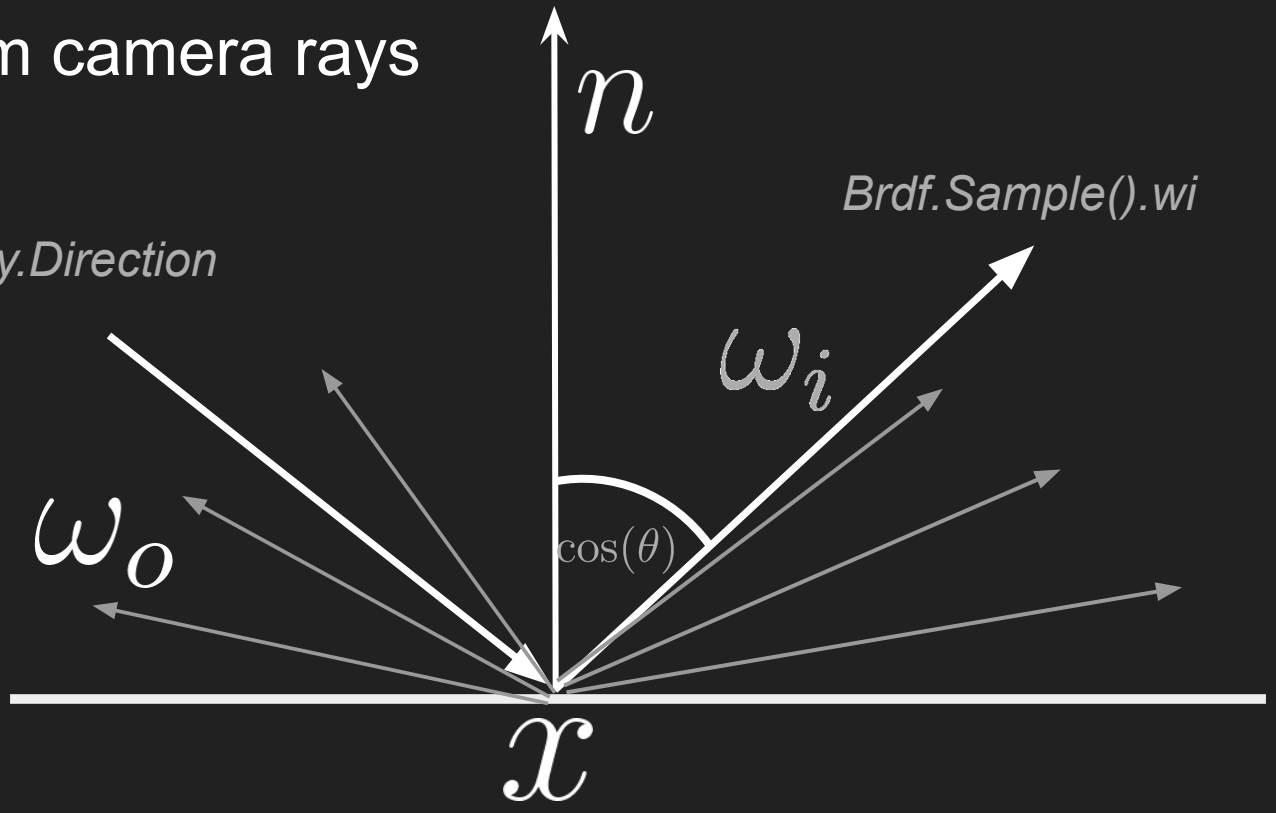
rayColour (rgb) *= $f_r\left(x_p, \omega_i, \omega_o\right)$


ω_o
(a)
ω_o
(b)
ω_o
(c)
ω_o
(d)

- this is inefficient since not lot of light forum source actually scatters in our direction
- we will do it other way around - from camera to scene and hope we will hit the light



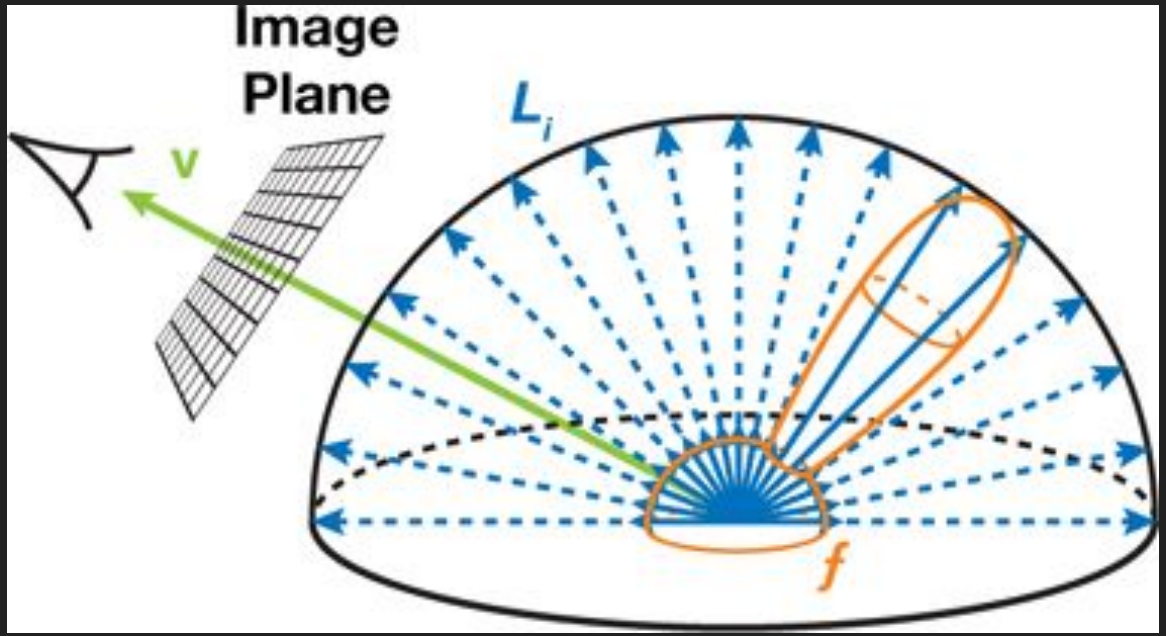
LTE from camera rays



$$L_o(\boldsymbol{x},\omega_o) \approx L_e(\boldsymbol{x},\omega_o) + \frac{f_r(\boldsymbol{x},\omega_i,\omega_o)\, L_i(\boldsymbol{x},\omega_i)\, (\boldsymbol{n}\cdot\omega_i)}{p(\omega_i)}$$

Importance sampling

- how do we pick the most important direction ? we use brdfs



https://developer.nvidia.com/gpugems/gpugems3/part-iii-rendering/chapter-20-gpu-based-importance-sampling

Recap

- to evaluate BRDF(colour of the surface) we need 2 directions *wo*, and *wi*
- we know *wo* (it is inverted ray direction)
- we have to randomly select *wi* based on the BRDF of the surface
- the *wi* will be next direction of our ray

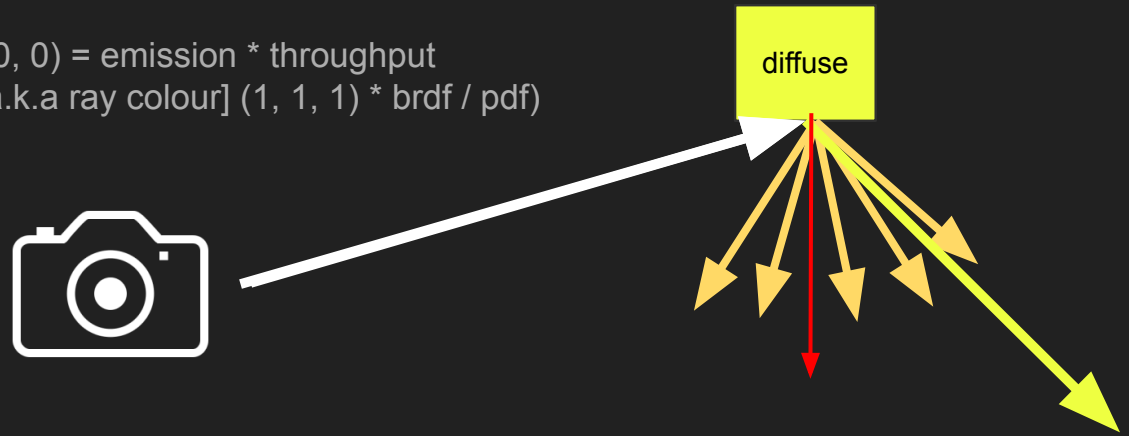
```
var brdf = hit.material.GetBrdf(u);
var brdfSample = brdf.Sample(-ray.Direction, hit.normal, u);
brdfSample.wi;
brdfSample.PDF;
```

Path tracing overview

ray payload - tracks ray through the scene

- throughput (1, 1, 1)
- radiance (0, 0, 0)

ray payload
- radiance (0, 0, 0) = emission * throughput
- throughput [a.k.a ray colour] (1, 1, 1) * brdf / pdf)



- account for emission (ray colour * Le)
- select next direction we will go to (sample brdf and get wi and pdf)
- use the sampled direction to evaluate LTE (Light transport equation)
  - for this we need wi, wo, and normal vector

ray payload

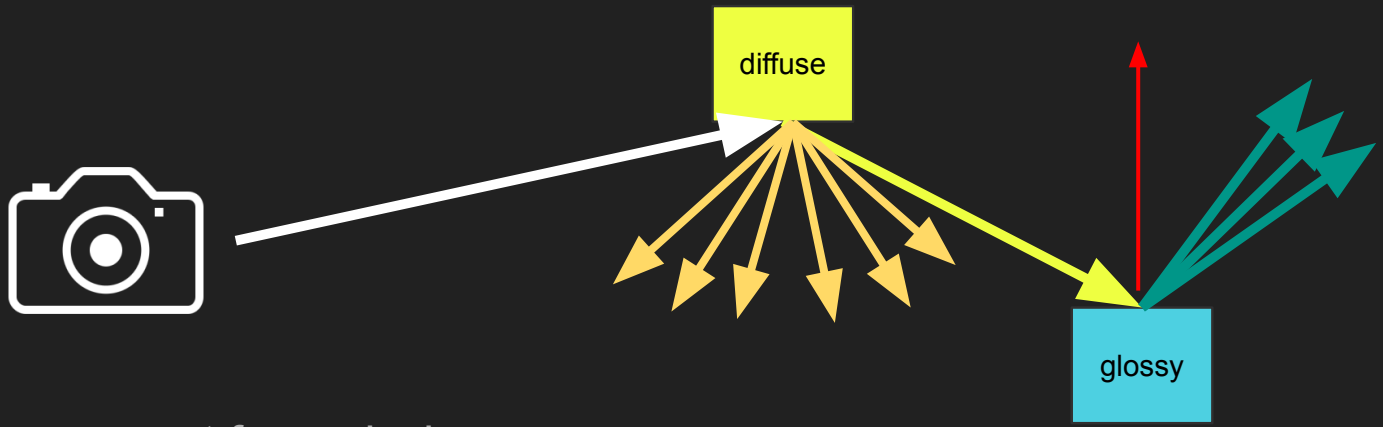
- radiance (0, 0, 0) = emission * throughput
- throughput [a.k.a ray colour] (1, 1, 1) * brdf / pdf)

- account for emission
- select next direction we will go to
- use the sampled direction to evaluate LTE

Overview of path tracing

ray payload

- radiance (0, 0, 0) = emission * throughput
- throughput [a.k.a ray colour] (1, 1, 1) * brdf / pdf)



- account for emission
- select next direction we will go to
- use the sampled direction to evaluate LTE

Averaging the result

Final colour of our pixel is average radiance collected over the multiple bounces

$$L_o = \frac{1}{N} \sum_{n=1}^{n=b_m} L_r$$

Lo - pixel colour
N - number of samples
Lr - radiance on the ray
bm - maximum bounce count

Lets implement this

PS. no worries that you don't fully understand it it is much easier to grasp in the code

```
ray = generatePrimaryRay();
throughput = 1.0;
radiance = 0.0;
for bounce ∈ {1 . . . MAX_BOUNCES} do
    Trace(ray);
    if hit surface then
        brdf, brdfPdf, ray = SampleBrdf();
        throughput *= brdf / brdfPdf;
    else
        radiance += throughput * skyColor;
        break;
return radiance;
```

Additional resources

pbrt - https://pbr-book.org/4ed/contents

ray tracing in one weekend - https://raytracing.github.io/books/RayTracingInOneWeekend.html

ray tracing gems - https://www.realtimerendering.com/raytracinggems/