Ray Tracing for Games

Dr. Jacco Bikker - IGAD/BUAS, Breda, February 4

Welcome!

7

Wednesday
13:00 – 17:00

course intro
LH2
template
Whitted
refactoring
RT-centric games

LAB 1

Thursday
09:00 – 14:00

advanced Whitted
audio, AI & physics
faster Whitted
Heaven7

LAB 2

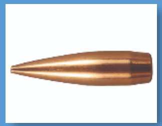
work @ home

End result day 2:

A solid Whitted-style ray tracer, as a basis for subsequent work.

Friday
09:00 – 17:00

optimization
profiling, rules of engagement
threading



LAB 3

SIMD
applied SIMD
SIMD triangle
SIMD AABB

LAB 4

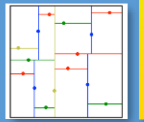
End result day 3:

A 5x faster tracer.

GLOBAL GAME JAM

Monday
09:00 – 17:00

acceleration
grid, BVH, kD-tree
SAH
binning



LAB 5

refitting
top-level BVH
threaded building

LAB 6

End result day 4:

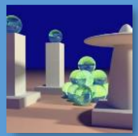
A real-time tracer.

Tuesday
09:00 – 17:00

Monte-Carlo
Cook-style
glossy, AA
area lights, DOF

LAB 7

path tracing



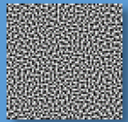
LAB 8

End result day 5:

Cook or Kajiya.

Thursday
09:00 – 17:00

random numbers
stratification
blue noise



LAB 9

importance sampling
next event estimation

LAB 10

End result day 6:

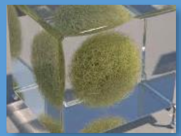
Efficiency.

Friday
09:00 – 17:00

future work

LAB 11

path guiding



LAB 10

FINISH

End result day 6:

Great product.

Agenda:

- Maths
- Monte-Carlo
- The Rendering Equation
- Applied

Math Symbols in Graphics Papers

$f(x) = x^2$: generic function, where ‘x’ is the parameter.

$\sum$ (sigma): ‘for loop’.
E.g.:

$$v = \sum_{i=1}^{4} f(x_i) = f(x_1) + f(x_2) + f(x_3) + f(x_4)$$

$\prod$ (‘product’): also a for loop.
E.g.:

$$v = \prod_{i=1}^{4} f(x_i) = f(x_1)\, f(x_2)\, f(x_3)\, f(x_4)$$

$\int$ (‘integral’).
E.g.:

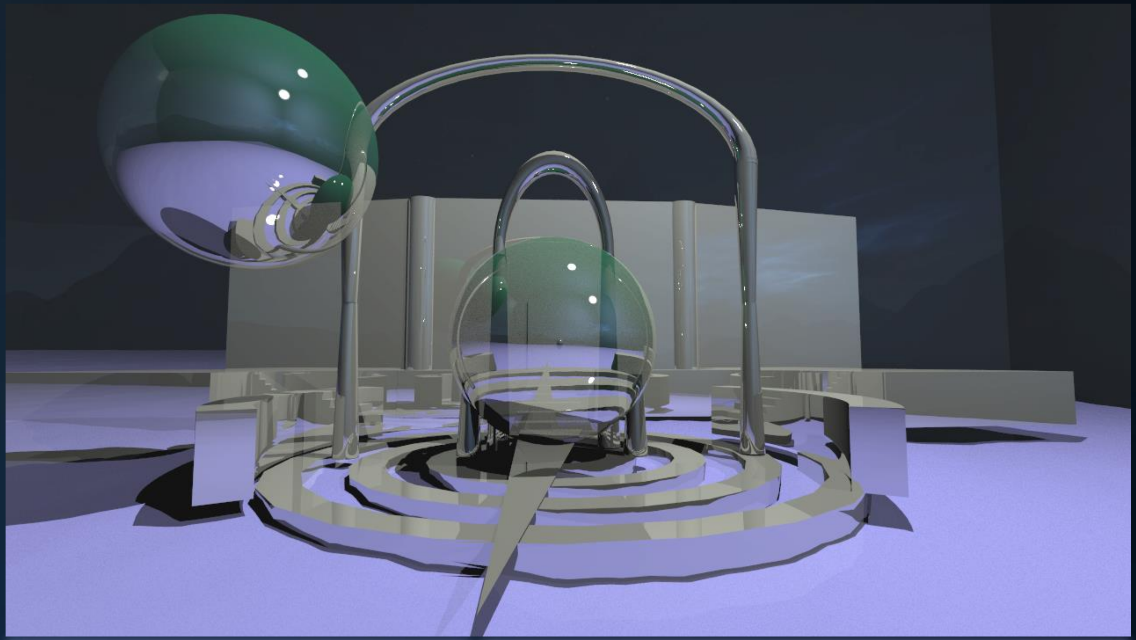
$$a = \int_0^{\pi} \sin x \; dx$$

$\mathbb{R}^2, \mathbb{R}^3$: 2D and 3-dimensional space.

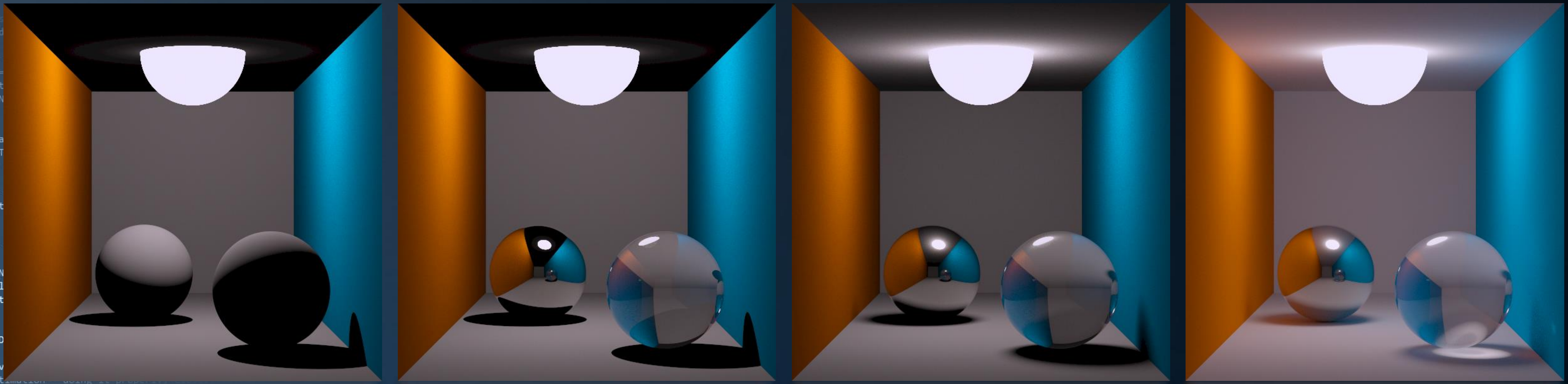
ω, Ω: ‘omega’.
Θ: ‘theta’, e.g. $\cos\theta = N \cdot L$

Whitted


xy: (1216, 692)
rgba: (0.475, 0.485, 0.506, 1.000)
before gamma correction



Whitted



Anti-aliasing

Adding anti-aliasing to a Whitted-style ray tracer:

Send multiple primary rays through each pixel and average their result.

Anti-aliasing – Sampling Patterns

Adding anti-aliasing to a Whitted-style ray tracer:

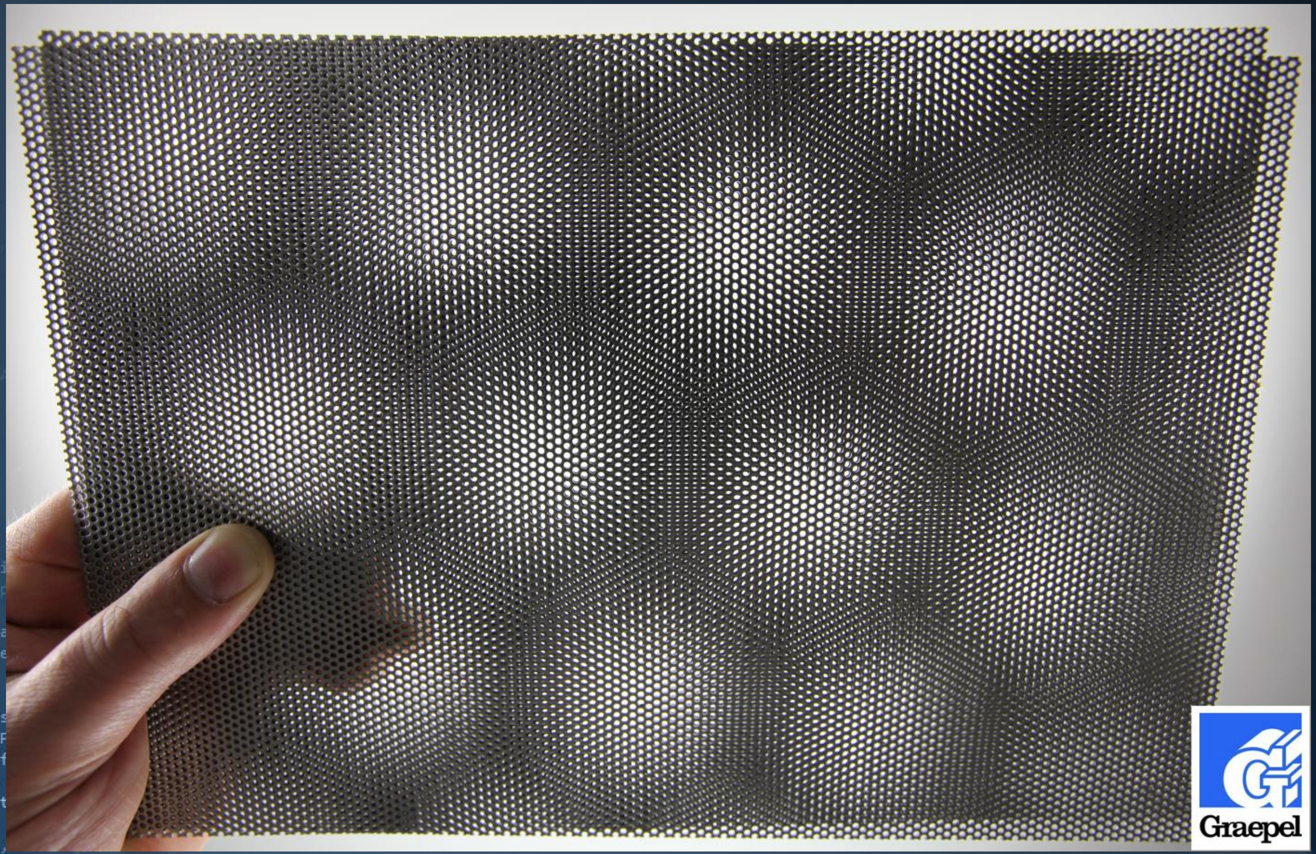
Send multiple primary rays through each pixel, and average their result.

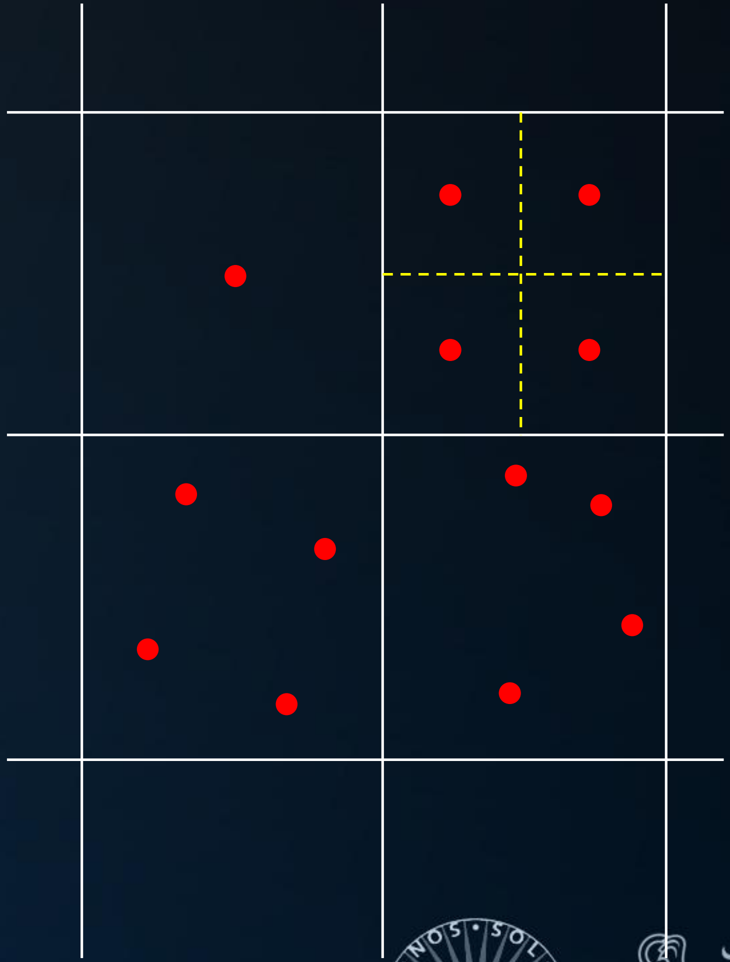
Problem:

- How do we aim those rays?
- What if all rays return the same color?

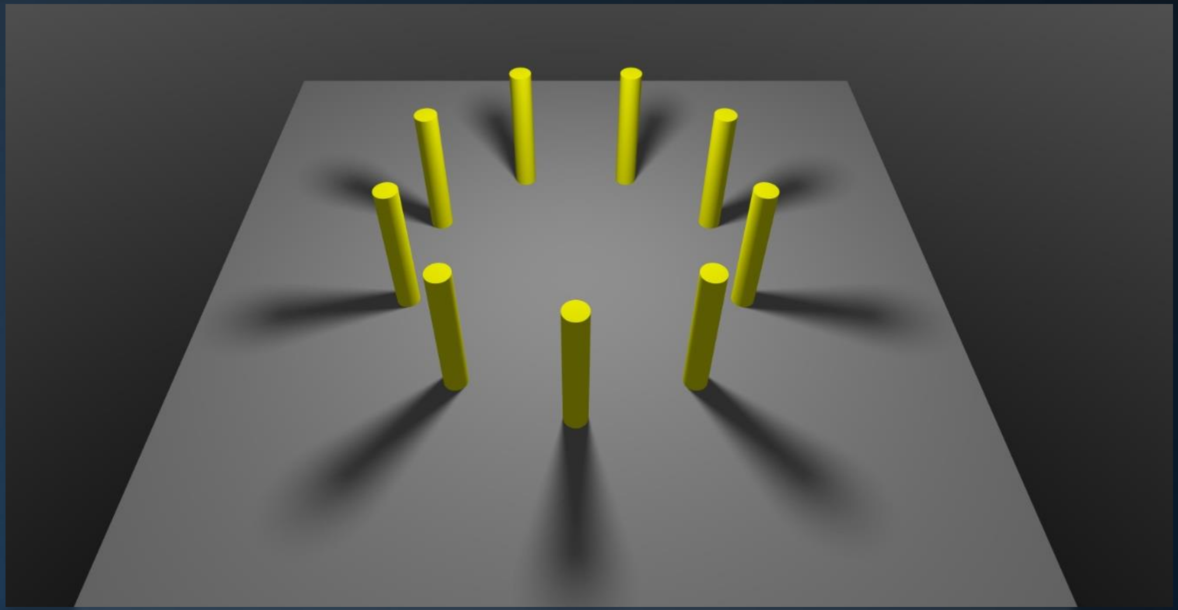
Anti-aliasing – Sampling Patterns

Anti-aliasing – Sampling Patterns

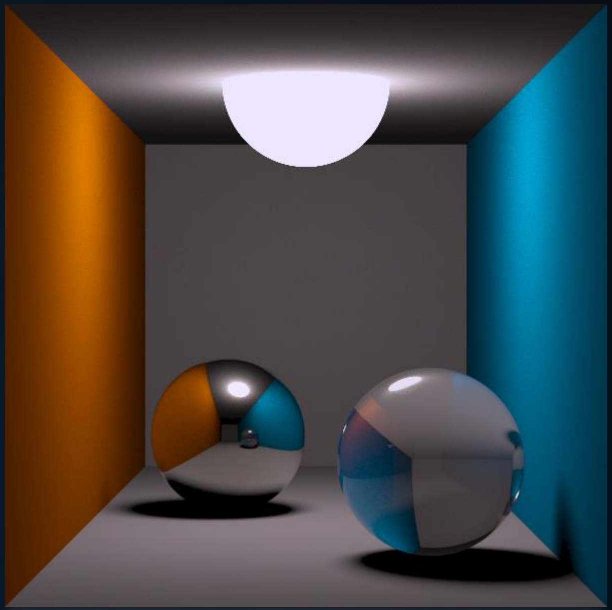




Distribution Ray Tracing*



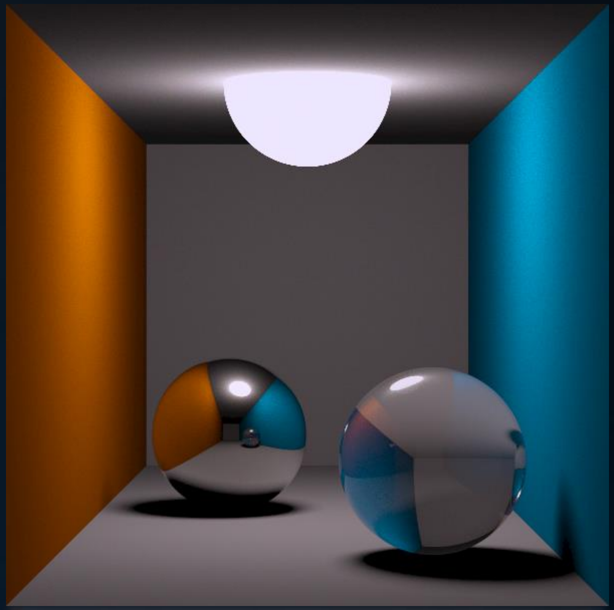
Soft shadows



*: Distributed Ray Tracing, Cook et al., 1984

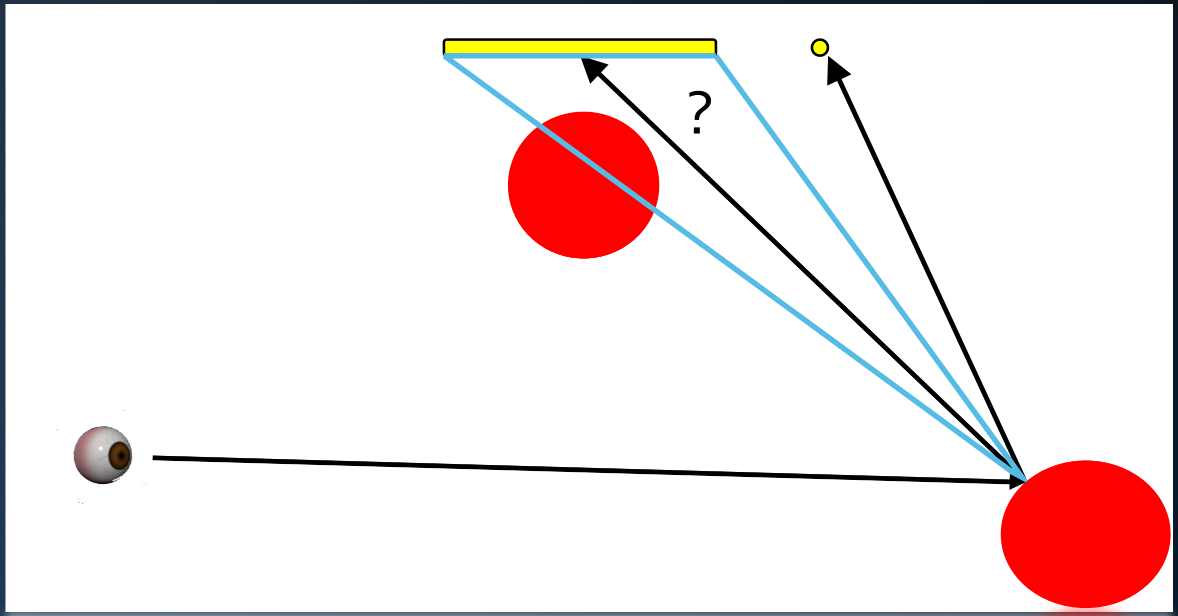
Distribution Ray Tracing*

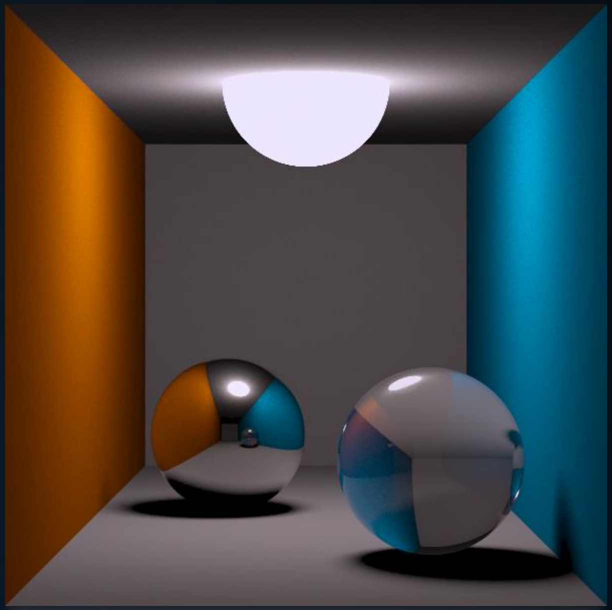
Glossy reflections



*: Distributed Ray Tracing, Cook et al., 1984

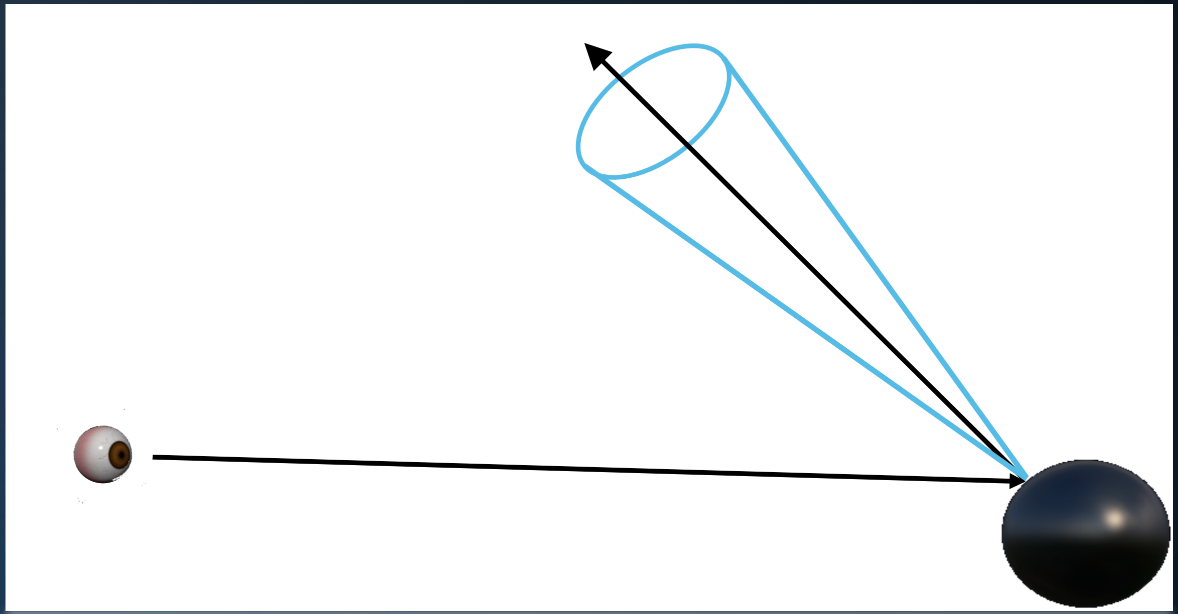
Distribution Ray Tracing*

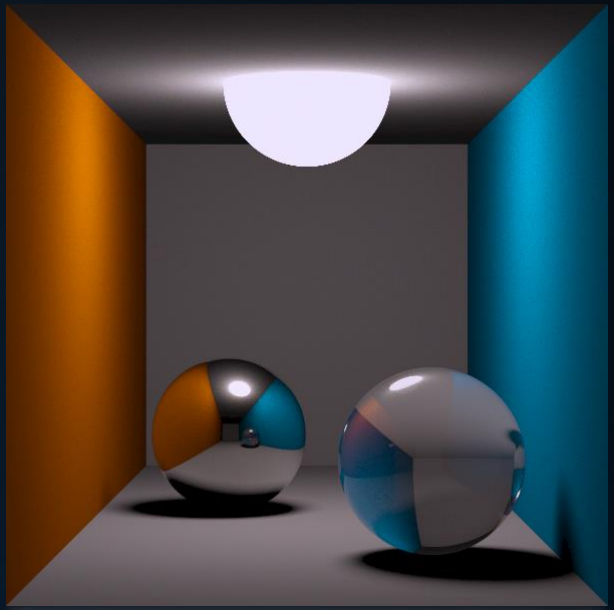




*: Distributed Ray Tracing, Cook et al., 1984

Distribution Ray Tracing*





*: Distributed Ray Tracing, Cook et al., 1984

Area Lights

Visibility of an area light source:

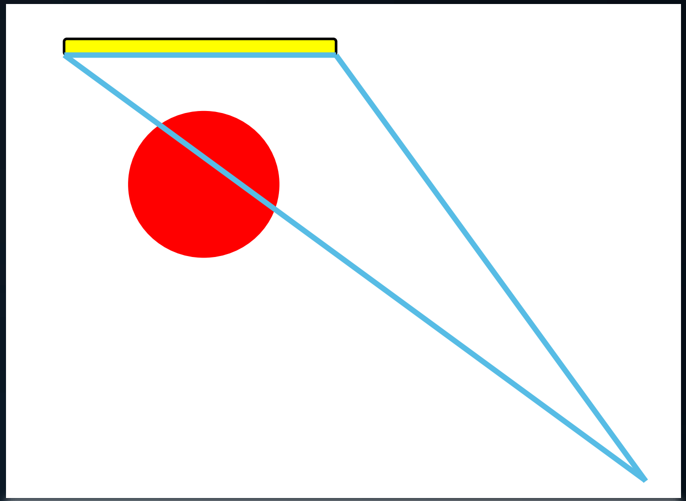
$$V_A = \int_A V(x)\, dx$$

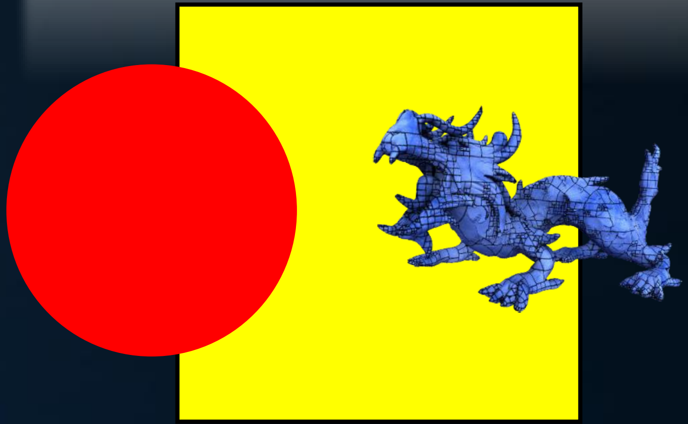
Perfect ('analytical') solution case 1:

$$V_A = A_{light} - A_{light \cap sphere}$$

Analytical solution case 2:

$$V_A = \ ?$$





Approximating Integrals

An integral can be approximated as a Riemann sum:

$$V_A = \int_A^B f(x)\,dx \approx \sum_{i=1}^{N} f(t_i)\,\Delta_i\,,\text{where} \sum_{i=1}^{N} \Delta_i = B - A$$

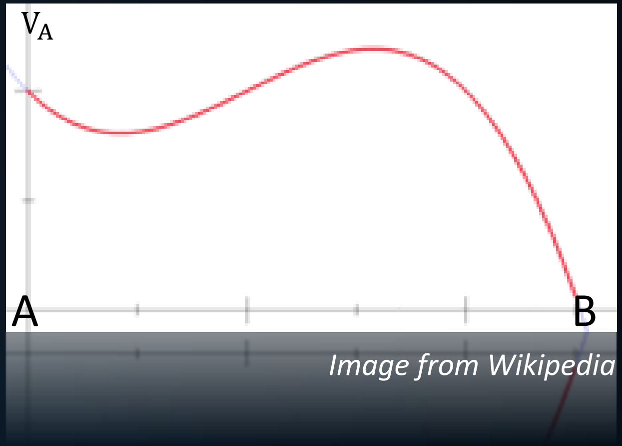
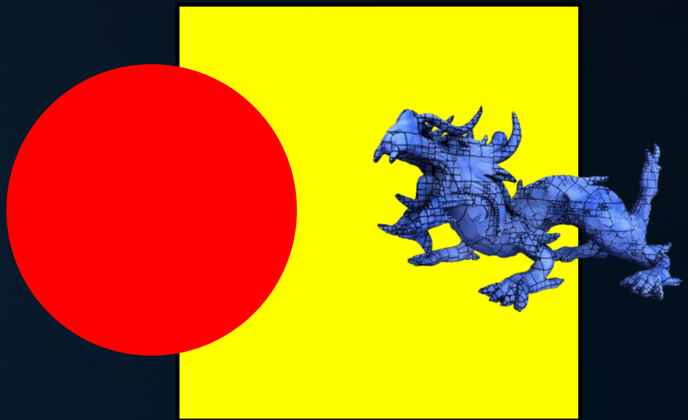


Image from Wikipedia

Alternatively, we can approximate an integral by taking *random* samples:

$$V_A = \int_A^B f(x)\,dx \approx \frac{B-A}{N} \sum_{i=1}^{N} f(X_i)$$



Monte Carlo Integration of Area Light Visibility

To estimate the visibility of an area light source, we take N random point samples.

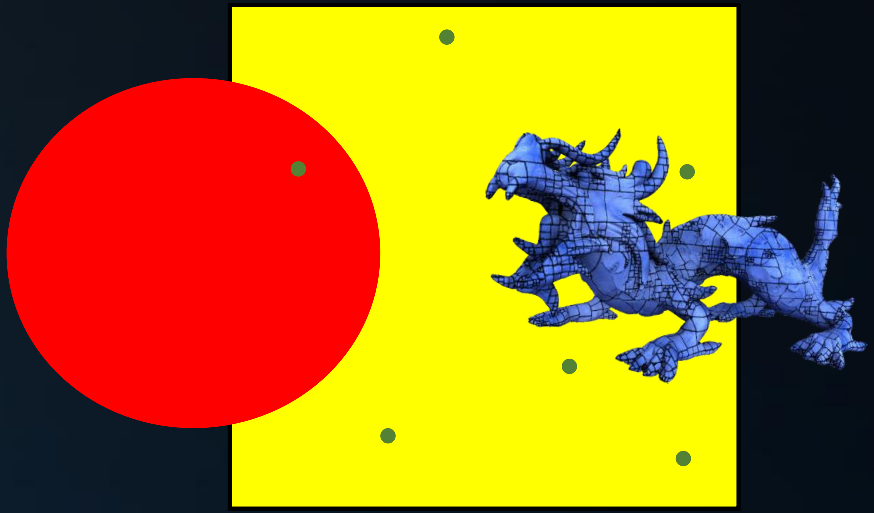
In this case, 5 out of 6 samples are unoccluded:

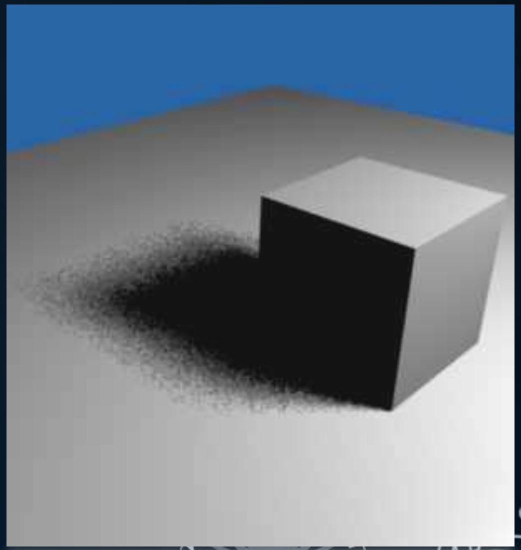
$$V \approx \frac{1}{6}(1+1+1+0+1+1) = \frac{5}{6}$$

Properly formulated using a MC integrator:

$$V = \int_{S^2} V(p)\, dp \approx \frac{1}{N} \sum_{i=1}^{N} V(P)$$

With a small number of samples, the *variance* in the estimate shows up as noise in the image.





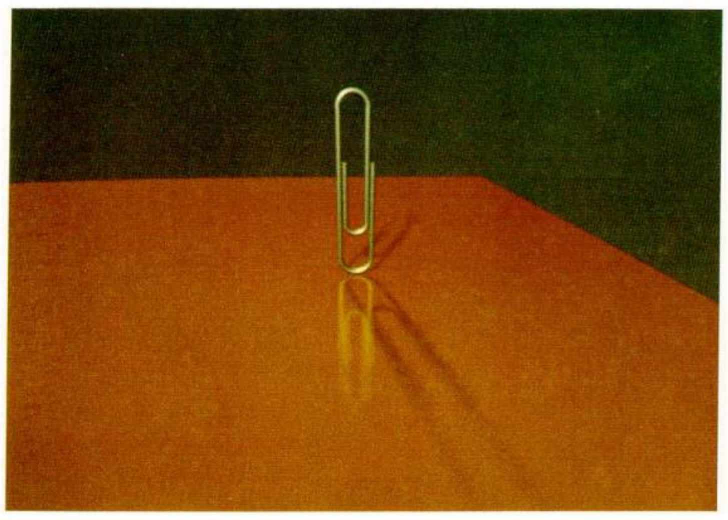
Distribution Ray Tracing

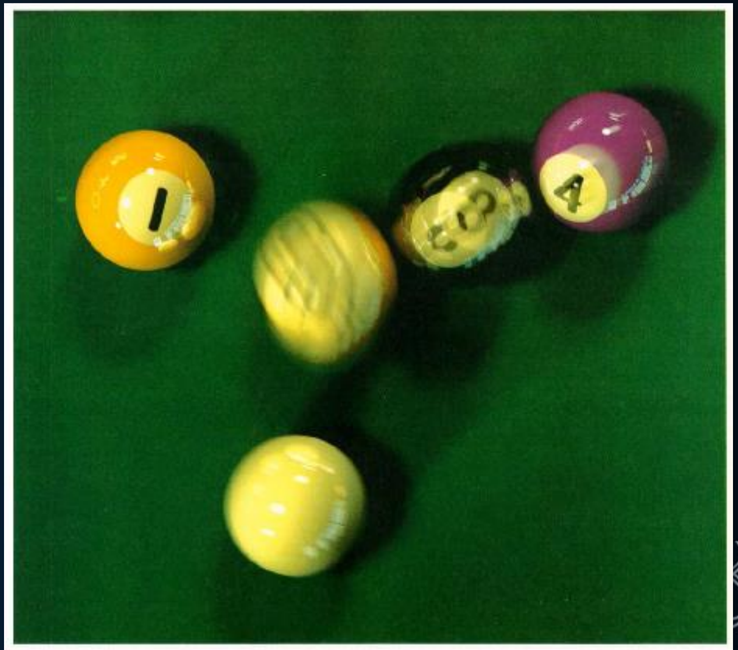
Key concept of distribution ray tracing:

We estimate integrals using Monte Carlo integration.

Integrals in rendering:

- Area of a pixel
- Lens area (aperture)
- Frame time
- Light source area
- Cones for glossy reflections
- ...





Agenda:

- Maths
- Monte-Carlo
- The Rendering Equation
- Applied

God's Algorithm

1 room
1 bulb
100 watts
10^{20} photons per second

Photon behavior:

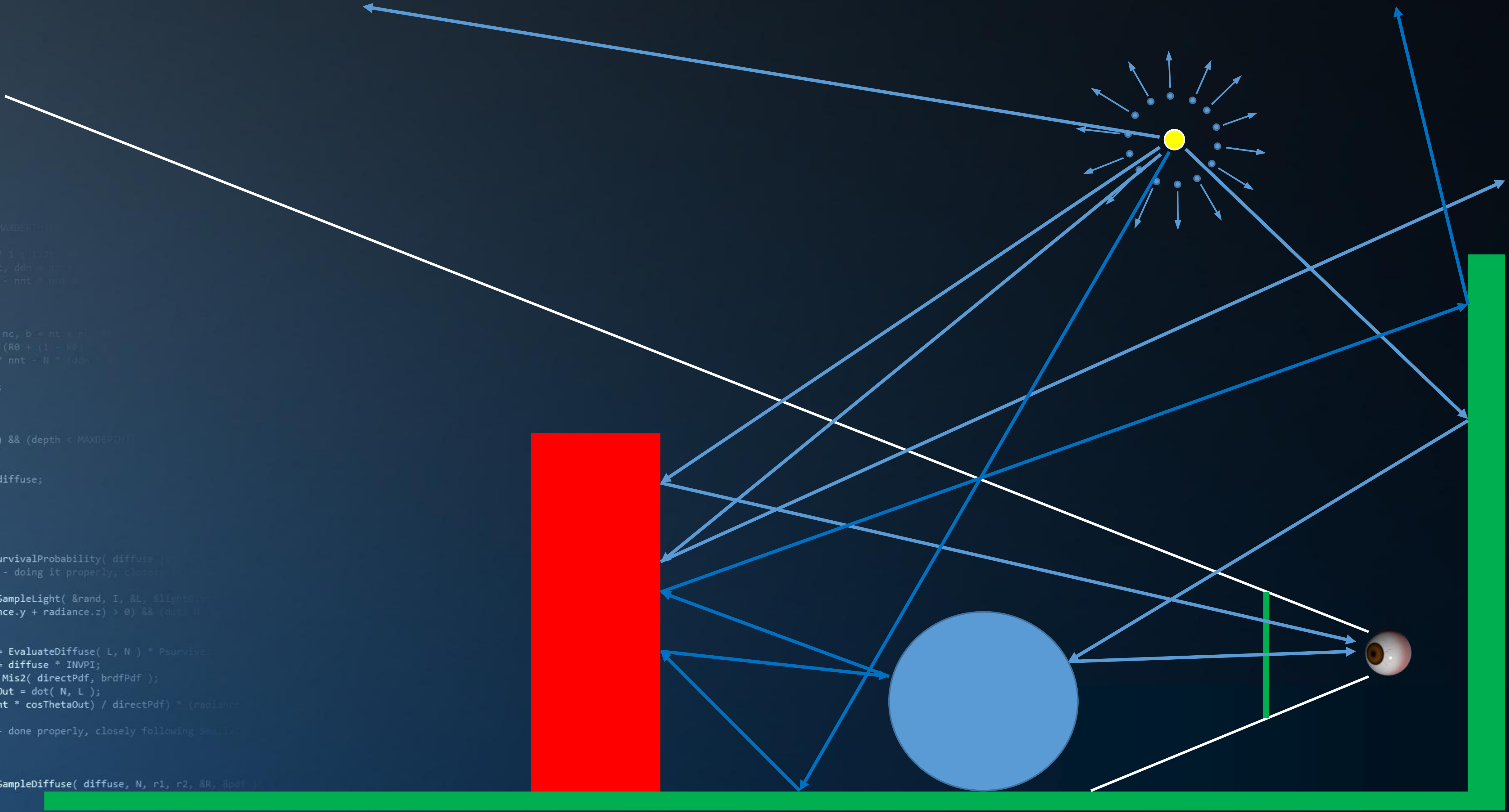
- Travel in straight lines
- Get absorbed, or change direction:
  - Bounce (random / deterministic)
  - Get transmitted
- Leave into the void
- Get detected

Ray Tracing for Games

```
& (depth < MAXDEPTH)
c = inside ? 1
nt = nt / nc, ddn
os2t = 1.0f - nnt
D, N );
0)
at a = nt - nc, b = nt
at Tr = 1 - (R0 + (1
Tr) R = (D * nnt - N
E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)
survive = SurvivalProbability( diffuse
estimation - doing it properly, closely
df;
radiance = SampleLight( &rand, I, &L, &lightD
e.x + radiance.y + radiance.z) > 0) && (dot( N
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
andom walk - done properly, closely following Small
vive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true:
```



God’s Algorithm - Mathematically

A photon may arrive at a sensor after travelling in a straight line from a light source to the sensor:

$$L(s \leftarrow x) = L_E(s \leftarrow x)$$

Or, it may be reflected by a surface towards the sensor:

$$L(s \leftarrow x) = \int_{\Omega} f_r(s \leftarrow x \leftarrow y)\, L(x \leftarrow y)\, G(x \leftrightarrow y)\, dy$$

Those are the options.
Adding direct and indirect illumination together:

$$L(s \leftarrow x) = L_E(s \leftarrow x) + \int_{A} f_r(s \leftarrow x \leftarrow y)\, L(x \leftarrow y)\, G(x \leftrightarrow y)\, dA(y)$$

God's Algorithm - Mathematically

$$L(s \leftarrow x) = L_E(s \leftarrow x) + \int_A f_r(s \leftarrow x \leftarrow y)\, L(x \leftarrow y)\, G(x \leftrightarrow y)\, dA(y)$$

Geometry factor

Indirect

Reflection

Hemisphere

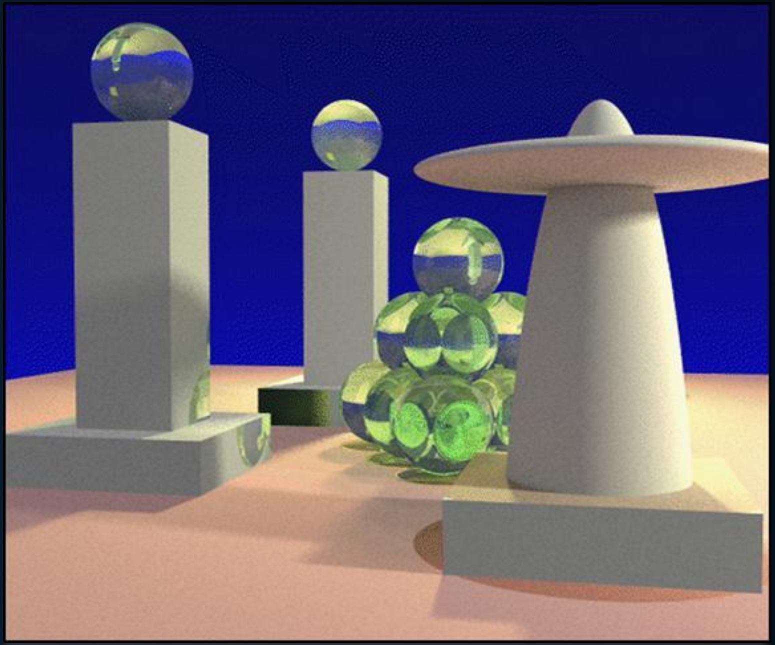
Emission

$$L(s \leftarrow x) = L_E(s \leftarrow x) + \int_A f_r(s \leftarrow x \leftarrow y)\, L(x \leftarrow y)\, G(x \leftrightarrow y)\, dA(y)$$

The Rendering Equation*:

- Light transport from lights to sensor
- Recursive
- Physically based

The equation allows us to determine to which extend rendering algorithms approximate real-world light transport.



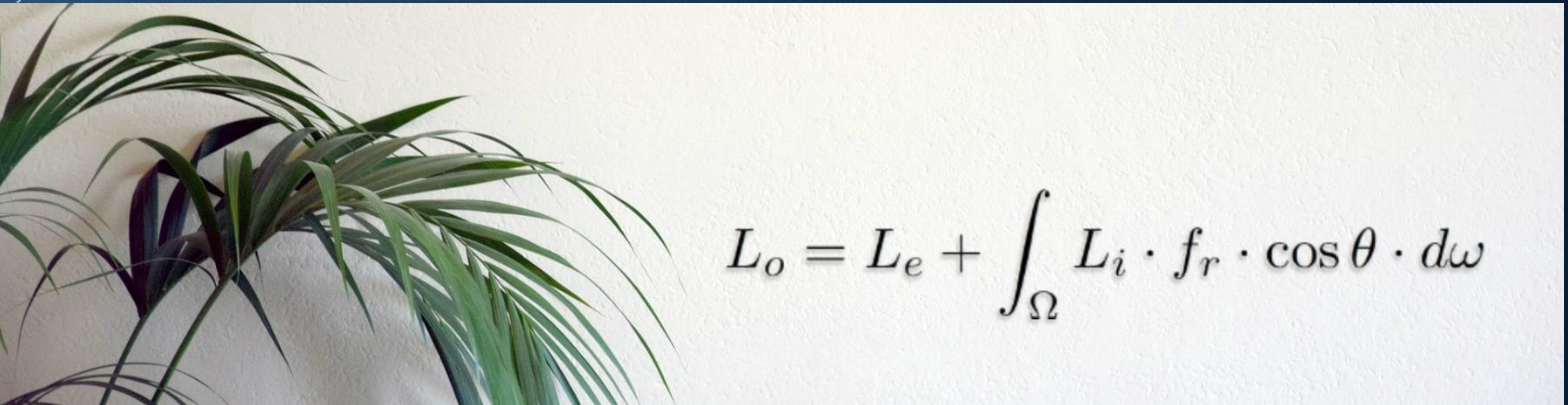
*: The Rendering Equation, Kajiya, 1986

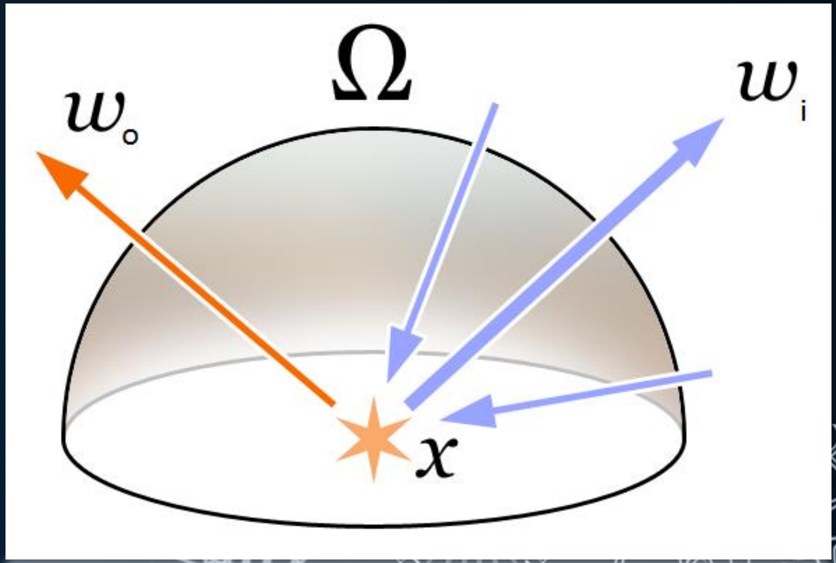
Light Transport

$$L(s \leftarrow x) = L_E(s \leftarrow x) + \int_A f_r(s \leftarrow x \leftarrow y)\, L(x \leftarrow y)\, G(x \leftrightarrow y)\, dA(y)$$

The above formulation integrates over all the points in the scene.
An alternative formulation:

$$L_o(x, \omega_o) = L_E(x, \omega_o) + \int_\Omega f_r(x, \omega_o, \omega_i)\, L_i(x, \omega_i) \cos\theta_i \; d\omega_i$$

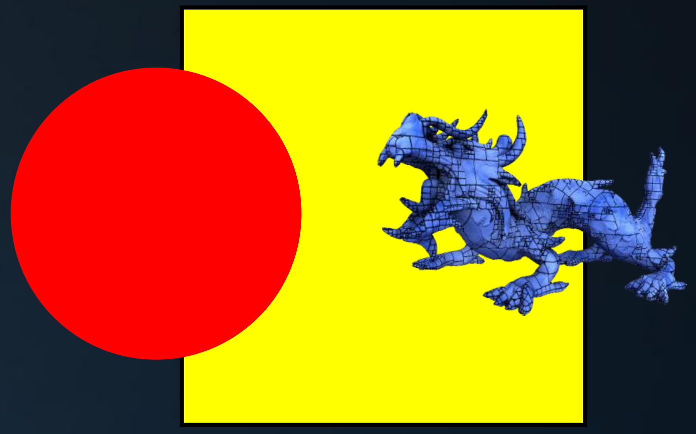




Agenda:

- Maths
- Monte-Carlo
- The Rendering Equation
- Applied

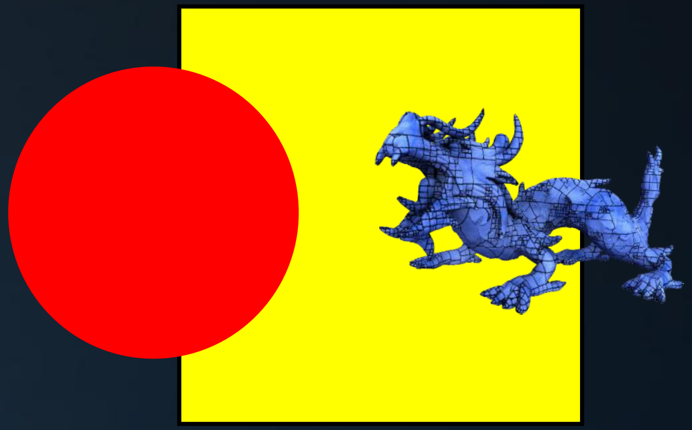
$$V_A = \int_A^B f(x)\,dx \approx \frac{B-A}{N}\sum_{i=1}^{N} f(X_i)$$



```
Color DirectIllumination( I, N )
   Color sum = BLACK
   for each light l
      L = light.pos - I
      dist2 = L.squareLength()
      if (!IsOccluded( I, L, sqrt(dist2) ))
         normalize(L)
         sum += (l.color * dot(N,L))/dist2)
   return sum
```

```
Color DirectIllumination( I, N )
   Color sum = BLACK
   for each light l
      P = l.RandomPointOnLight()
      L = P - I
      dist2 = L.squareLength()
      if (!IsOccluded( I, L, sqrt(dist2) ))
         normalize(L)
         sum += (l.color * dot(N,L))/dist2)
   return sum
```

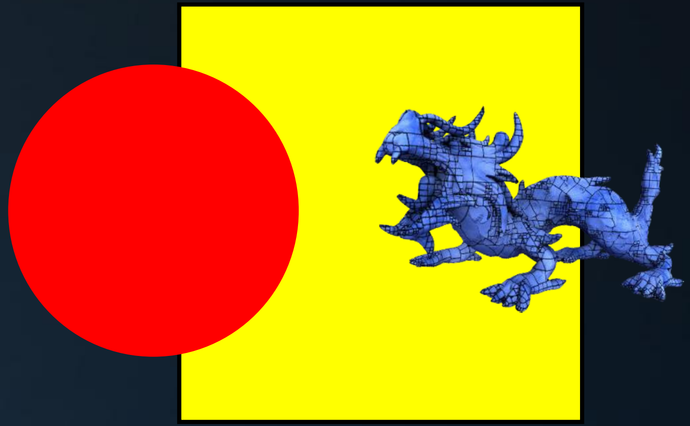
$$V_A = \int_A^B f(x)\,dx \approx \frac{B-A}{N} \sum_{i=1}^{N} f(X_i)$$



```
Color DirectIllumination( I, N )
   Color sum = BLACK
   for each light l
      L = light.pos - I
      dist2 = L.squareLength()
      if (!IsOccluded( I, L, sqrt(dist2) ))
         normalize(L)
         sum += (l.color * dot(N,L))/dist2)
   return sum
```

```
Color DirectIllumination( I, N )
   l = SelectRandomLight()
   P = l.RandomPointOnLight()
   L = P - I
   dist2 = L.squareLength()
   if (!IsOccluded( I, L, sqrt(dist2) ))
      normalize(L)
      return l.color * lightCount * dot(N,L) / dist2
   return BLACK
```

$$V_A = \int_A^B f(x)\,dx \approx \frac{B-A}{N}\sum_{i=1}^{N} f(X_i)$$

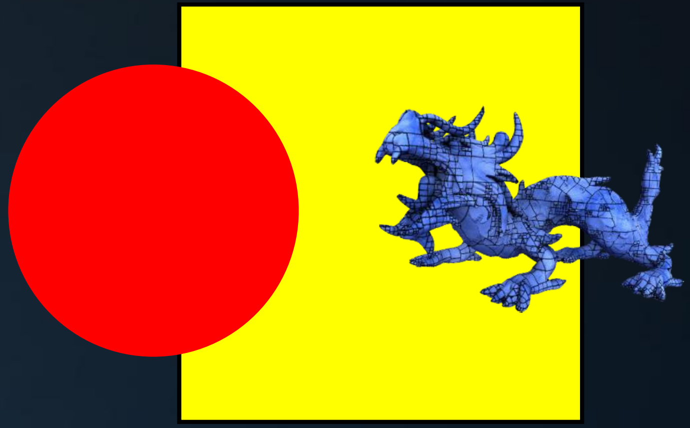


```
float3 accumulator =
   new float3[512 * 512];
memset( accumulator, 0, 512 * 512 * 4 );

scale = 1.0f / spp;
for( int y = 0; y < 512; y++ )
   for( int x = 0; x < 512; x++ )
      float3 p = accumulator[x + y * 512];
      p *= scale;
      int red   = min( 1.f, p.x ) * 255;
      int green = min( 1.f, p.y ) * 255;
      int blue  = min( 1.f, p.z ) * 255;
```

```
Color DirectIllumination( I, N )
  l = SelectRandomLight()
  P = l.RandomPointOnLight()
  L = P - I
  dist2 = L.squareLength()
  if (!IsOccluded( I, L, sqrt(dist2) ))
    normalize(L)
    return l.color * lightCount * dot(N,L) / dist2
  return BLACK
```

$$V_A = \int_A^B f(x)\,dx \approx \frac{B-A}{N}\sum_{i=1}^{N} f(X_i)$$



```
float3 accumulator =
   new float3[512 * 512];
memset( accumulator, 0, 512 * 512 * 4 );

scale = 1.0f / spp;
for( int y = 0; y < 512; y++ )
   for( int x = 0; x < 512; x++ )
      float3 p = accumulator[x + y * 512];
      p *= scale;
      int red   = sqrtf( min( 1.f, p.x ) ) * 255;
      int green = sqrtf( min( 1.f, p.y ) ) * 255;
      int blue  = sqrtf( min( 1.f, p.z ) ) * 255;
```

Also take a look at tonemapping, e.g.
ACES filmic tonemapping.

Other Uses for Monte-Carlo:

- Dielectrics: sample reflection *or* refraction.
- Glossy reflections: jitter the reflected ray.
- Diffuse interreflections: …
- Full Rendering Equation: …

That noise tho…

End of PART 7.