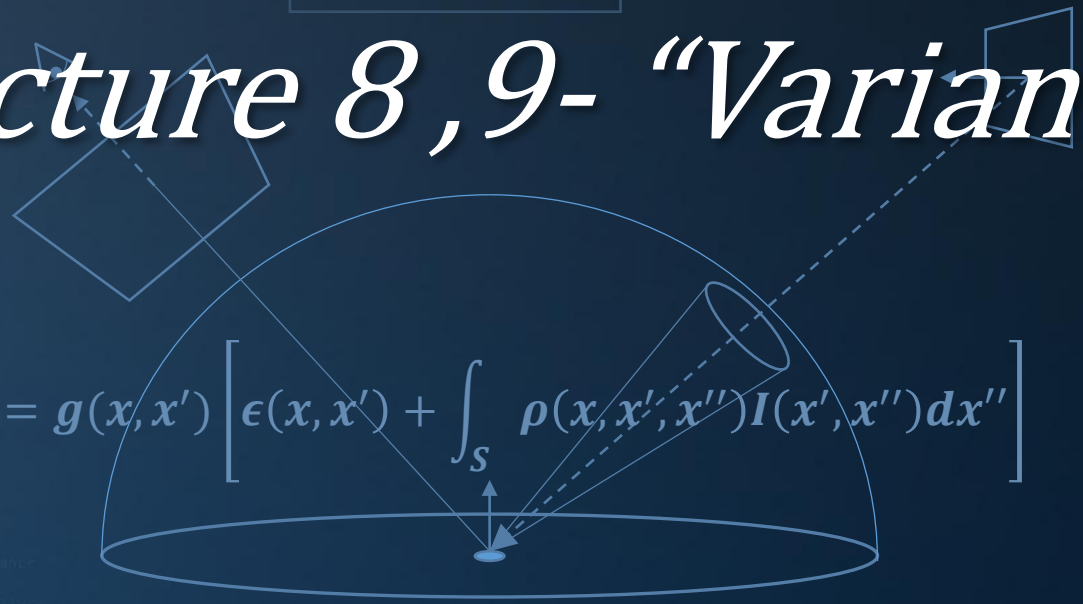
INFOMAGR – Advanced Graphics

Jacco Bikker - November 2022 - February 2023

Lecture 8 ,9- "Variance Reduction"

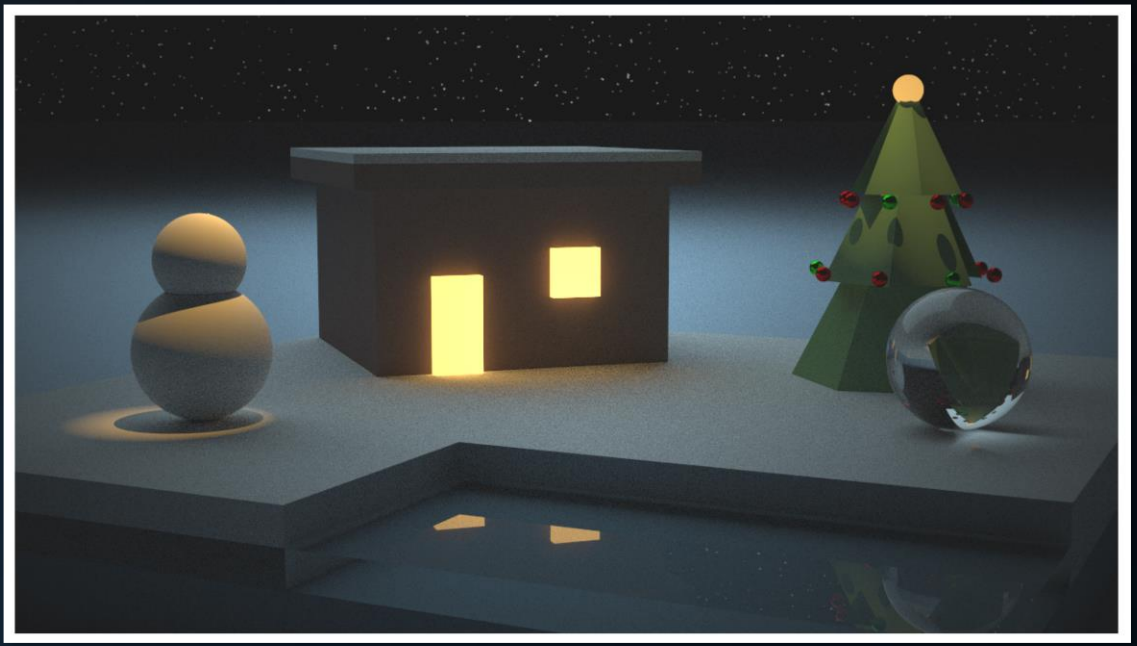


$$I(x,x') = g(x,x')\left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right]$$

Welcome!

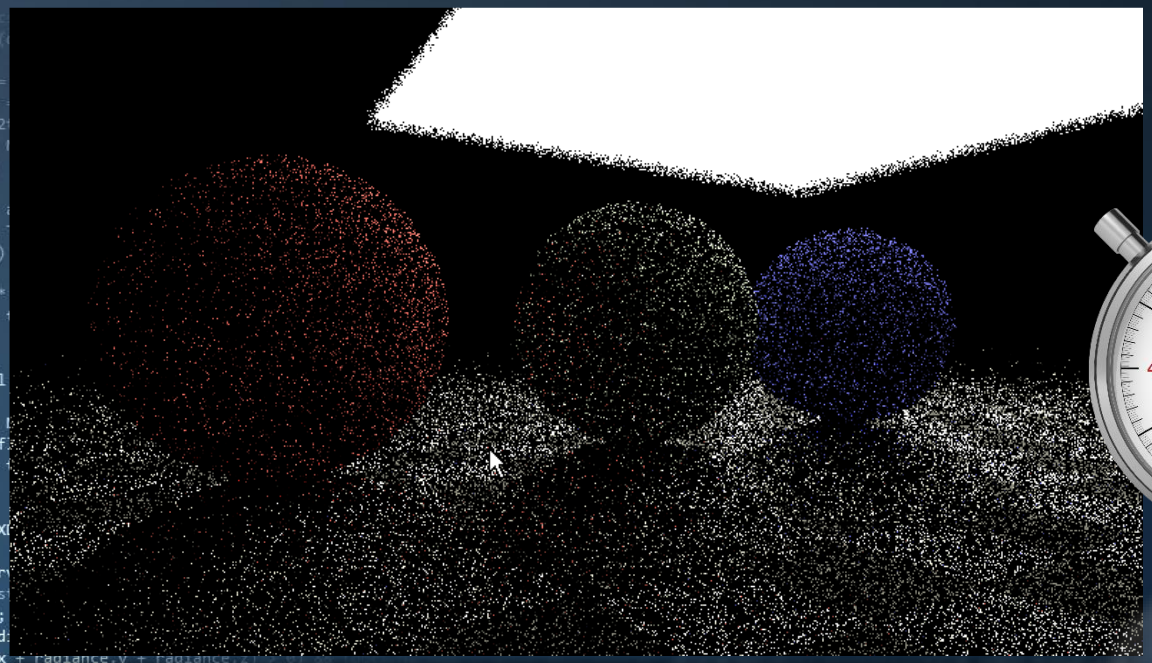
Today's Agenda:

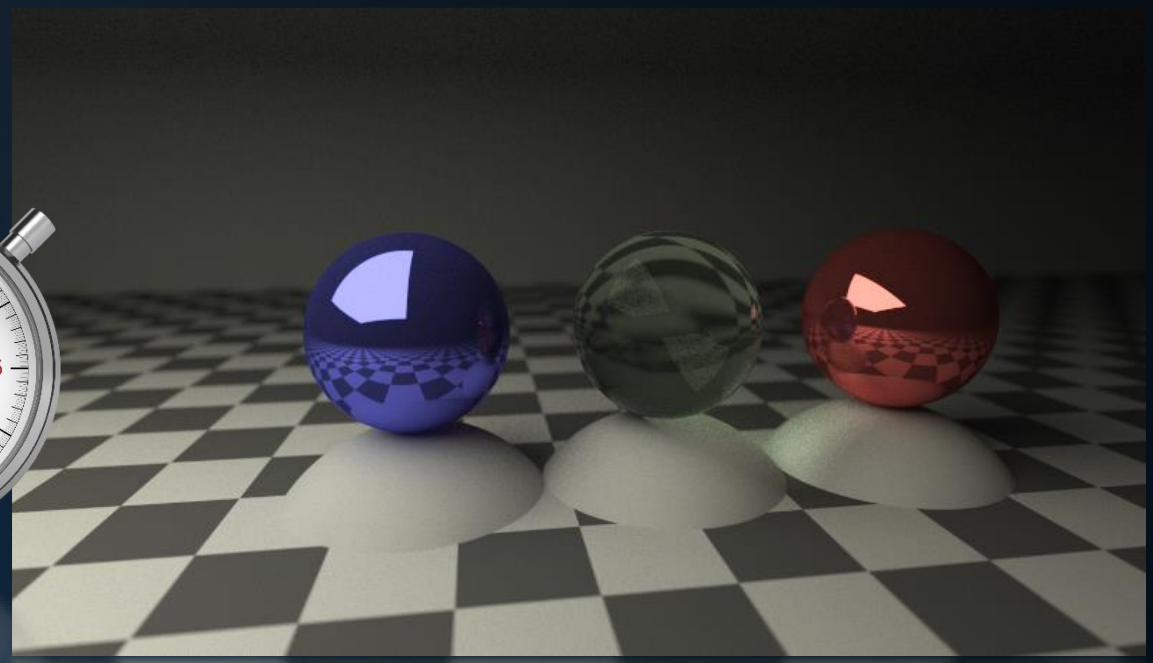
- Introduction
- Random Samples
- Next Event Estimation
- Importance Sampling
- Russian Roulette



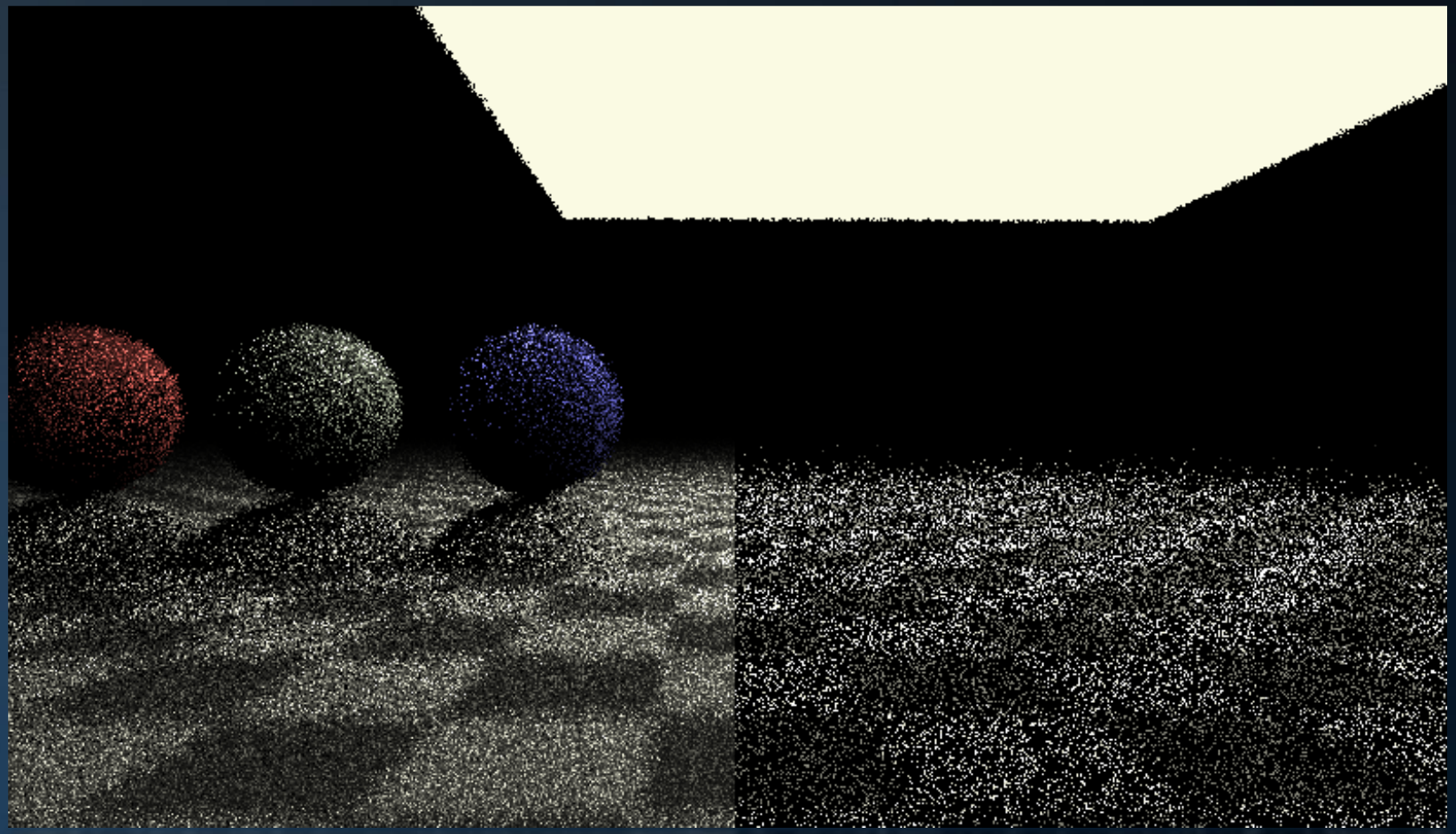
Introduction

Previously in Advanced Graphics





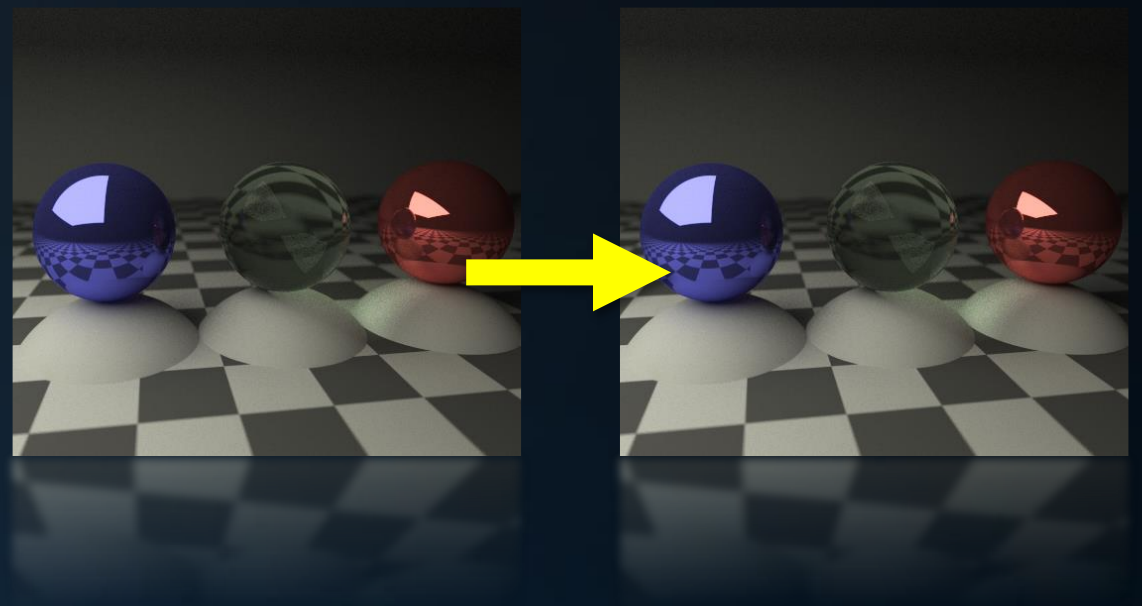
Introduction



Introduction

Today in Advanced Graphics:

- Stratification
- Blue Noise
- Next Event Estimation
- Importance Sampling
- Multiple Importance Sampling
- Resampled Importance Sampling



Aim:

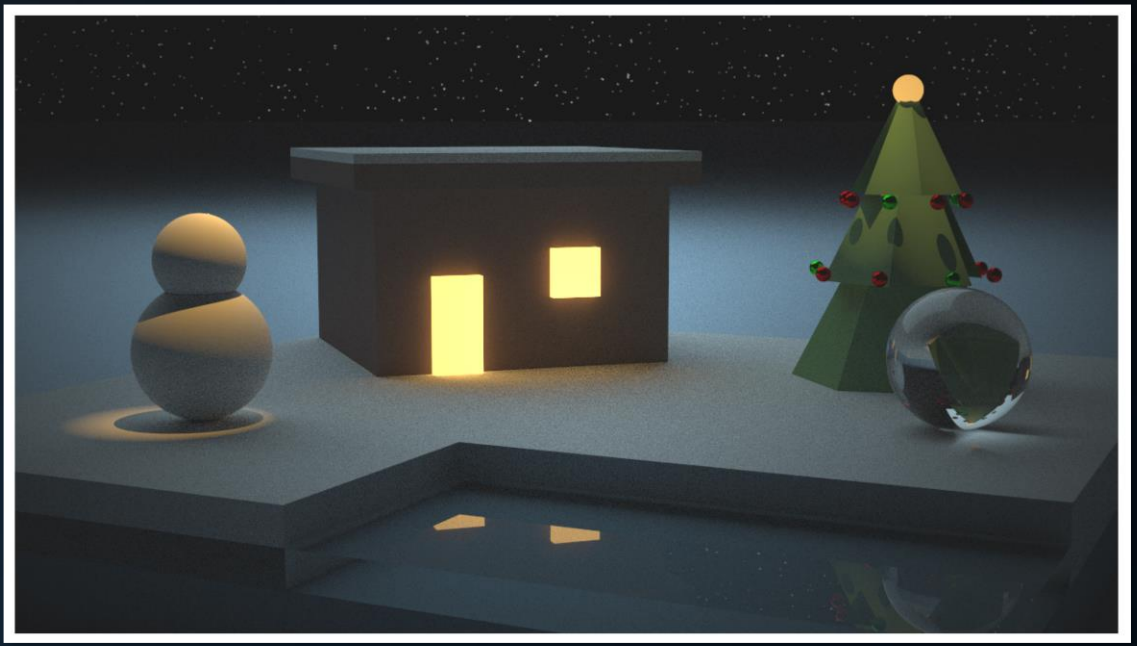
- to get a better image with the same number of samples
- to increase the efficiency of a path tracer
- to reduce variance in the estimate

Requirement:

- *produce the correct image*

Today's Agenda:

- Introduction
- Random Samples
- Next Event Estimation
- Importance Sampling
- Russian Roulette



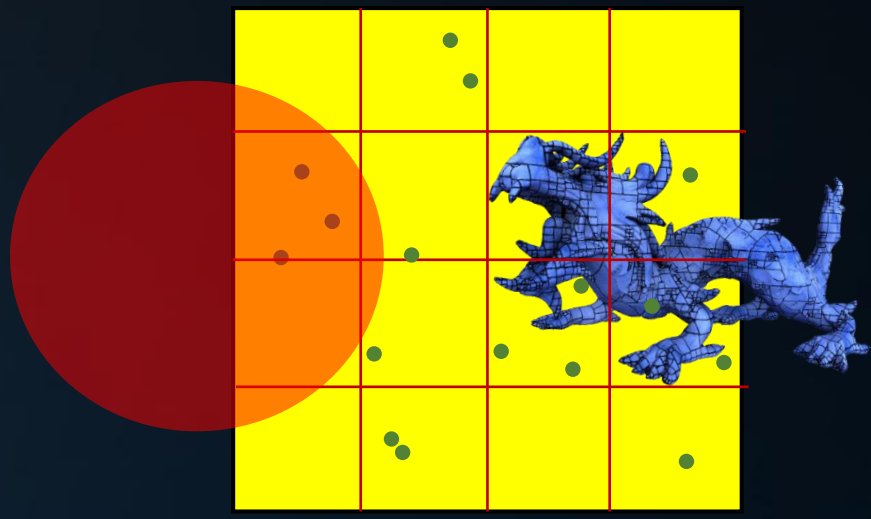
Stratification

Uniform Random Sampling

To sample a light source, we draw two random values in the range 0..1.

The resulting 2D positions are not uniformly distributed over the area.

We can improve uniformity using *stratification*: one sample is placed in each stratum.

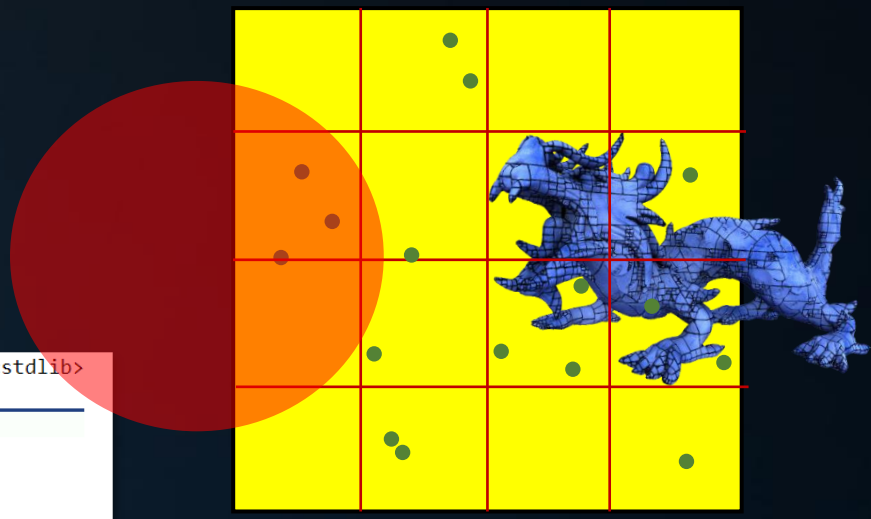


Stratification

Randomness

By the way…

What is a good random number?



rand

<cstdlib>

```
int rand (void);
```

Generate random number

Returns a pseudo-random integral number in the range between 0 and RAND_MAX.

This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using function srand.

RAND_MAX is a constant defined in <cstdlib>.

A typical way to generate trivial pseudo-random numbers in a determined range using rand is to use the modulo of the returned value by the range span and add the initial value of the range:

```
v1 = rand() % 100;          // v1 in the range 0 to 99
v2 = rand() % 100 + 1;      // v2 in the range 1 to 100
v3 = rand() % 30 + 1985;    // v3 in the range 1985-2014
```

Notice though that this modulo operation does not generate uniformly distributed random numbers in the span (since in most cases this operation makes lower numbers slightly more likely).

C++ supports a wide range of powerful tools to generate random and pseudo-random numbers (see <random> for more info).

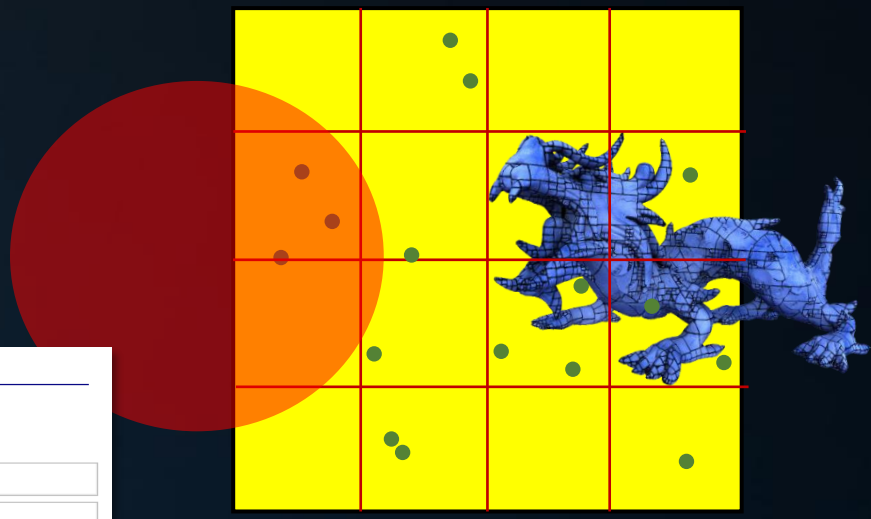
BAD. What if RAND_MAX is 65535, and we want a number in the range 0..50000? The range 50000..65535 will overlap 0..15535…

Stratification

Randomness

By the way…

What is a good random number?



Generators

Pseudo-random number engines (templates)

Generators that use an algorithm to generate pseudo-random numbers based on an initial seed:

linear_congruential_engine	Linear congruential random number engine (class template)
mersenne_twister_engine	Mersenne twister random number engine (class template)
subtract_with_carry_engine	Subtract-with-carry random number engine (class template)

Engine adaptors

They adapt an engine, modifying the way numbers are generated with it:

discard_block_engine	Discard-block random number engine adaptor (class template)
independent_bits_engine	Independent-bits random number engine adaptor (class template)
shuffle_order_engine	Shuffle-order random number engine adaptor (class template)

Pseudo-random number engines (instantiations)

Particular instantiations of generator engines and adaptors:

default_random_engine	Default random engine (class)
minstd_rand	Minimal Standard minstd_rand generator (class)
minstd_rand0	Minimal Standard minstd_rand0 generator (class)
mt19937	Mersenne Twister 19937 generator (class)

Stratification

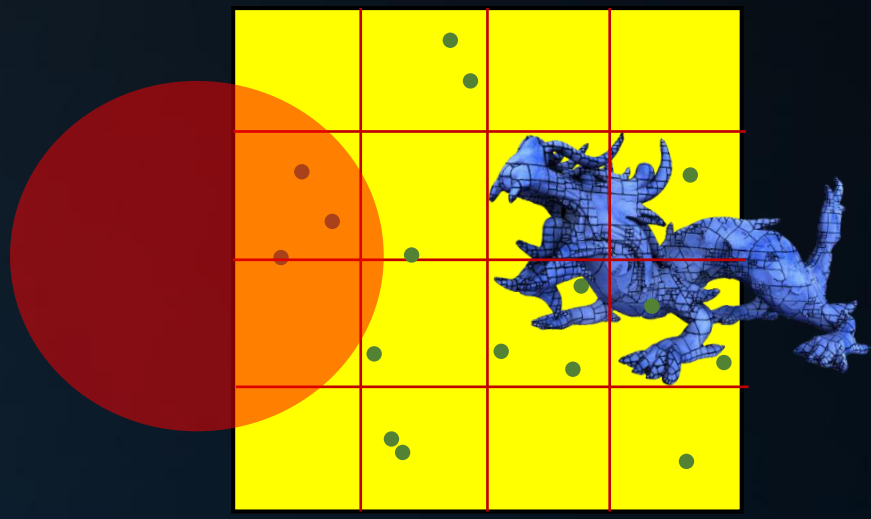
Randomness

By the way…

What is a good random number?

Consider Marsaglia's xor32*:

```
uint xorshift32( uint& state )
{
    state ^= state << 13;
    state ^= state >> 17;
    state ^= state << 5;
    return state;
}
```



Xor32, plus:

```
float RandomFloat( uint& s )
{ return xorshift32(s) * 2.3283064365387e-10f; }
```

Seeding:
Try the 'WangHash'.

*: Marsaglia, 2003. "Xorshift RNGs". Journal of Statistical Software.

Stratification

Uniform Random Sampling

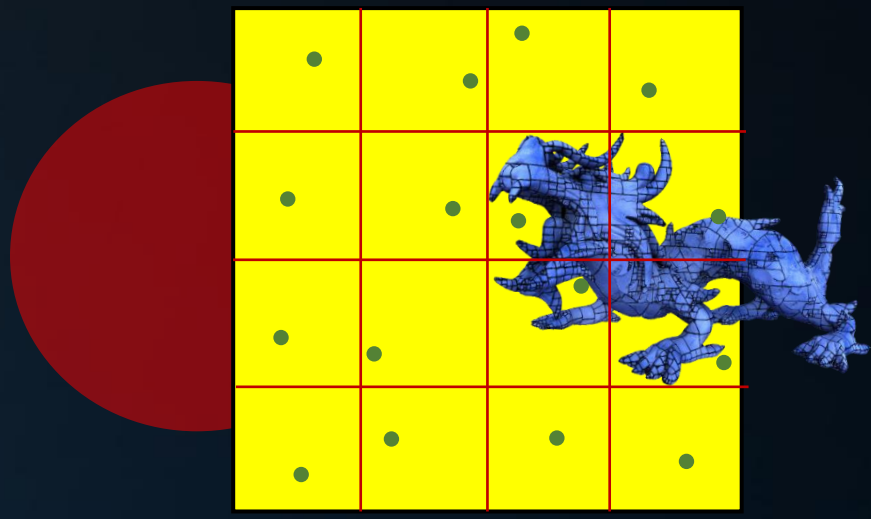
To sample a light source, we draw two random values in the range 0..1.

The resulting 2D positions are not uniformly distributed over the area.

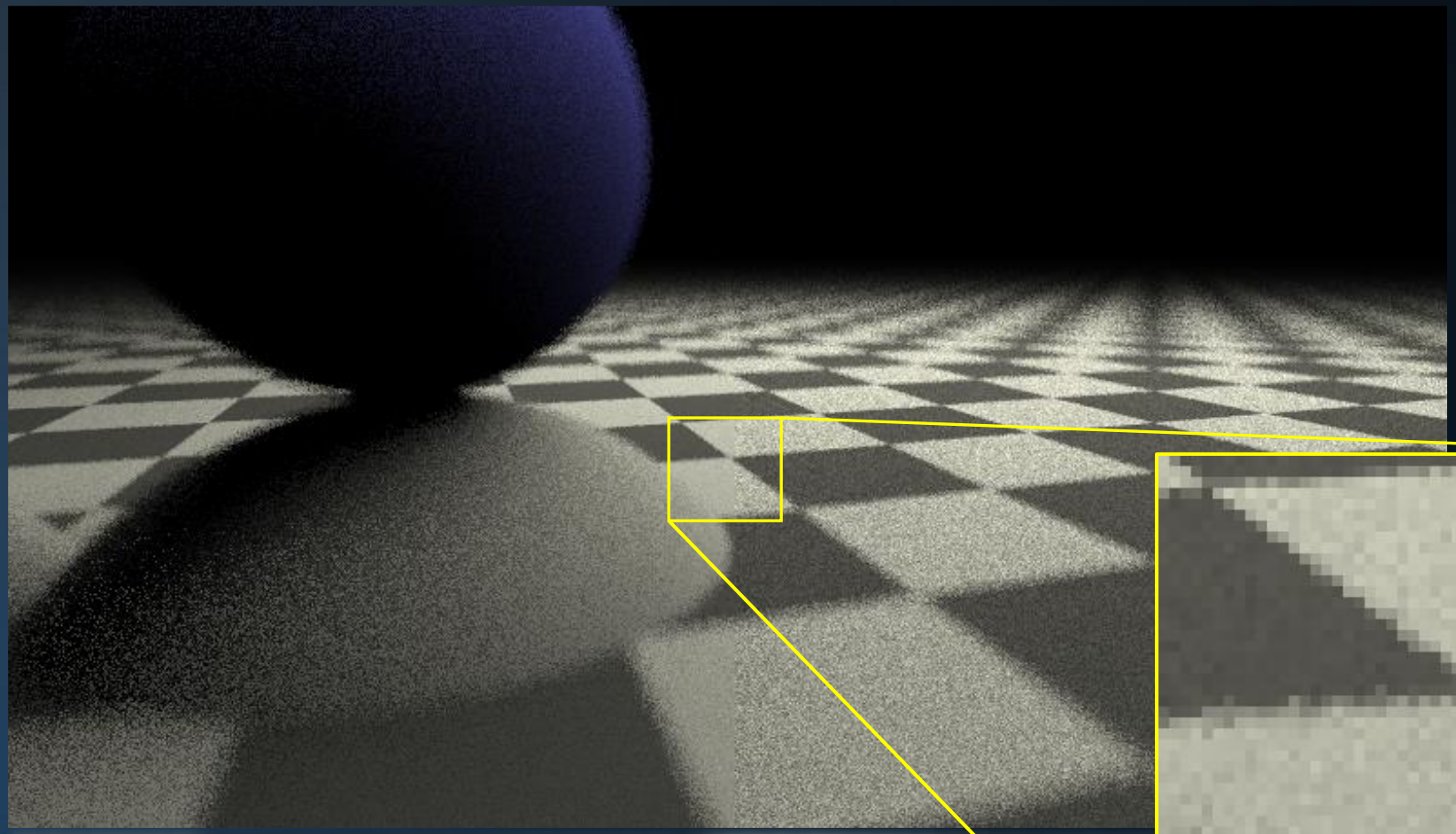
We can improve uniformity using *stratification*: one sample is placed in each stratum.

For 4x4 strata:

```
stratum_x = (idx % 4) * 0.25 // idx = 0..15
stratum_y = (idx / 4) * 0.25
r0 = Rand() * 0.25
r1 = Rand() * 0.25
P = vec2( stratum_x + r0, stratum_y + r1 )
```



Stratification

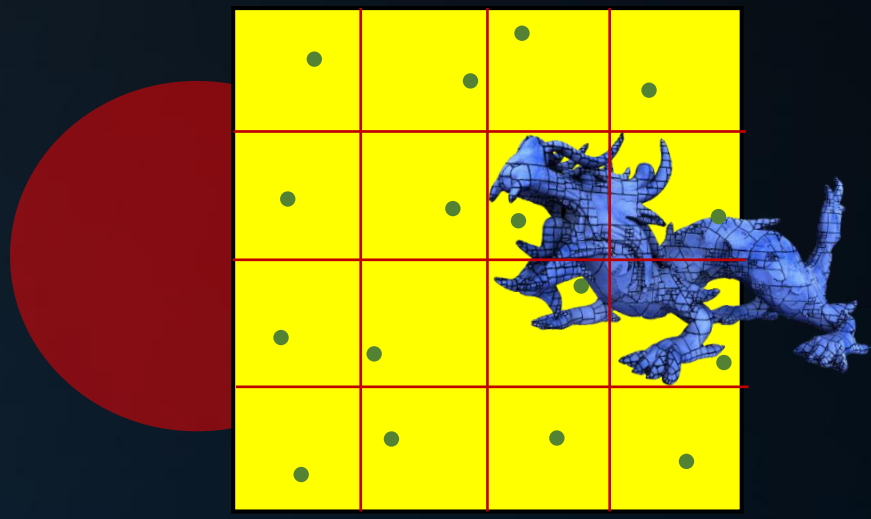


Stratification

Use Cases

Stratification can be applied to any Monte Carlo process:

- Anti-aliasing (sampling the pixel)
- Depth of field (sampling the lens)
- Motion blur (sampling time)
- Soft shadows (sampling area lights)
- Diffuse reflections (sampling the hemisphere)



However, there are problems:

- We need to take one sample per stratum
- Stratum count: higher is better, but with diminishing returns
- Combining stratification for e.g. depth of field and soft shadows leads to *correlation* of the samples, unless we stratify the 4D space - which leads to a very large number of strata: the *curse of dimensionality*.

Blue Noise

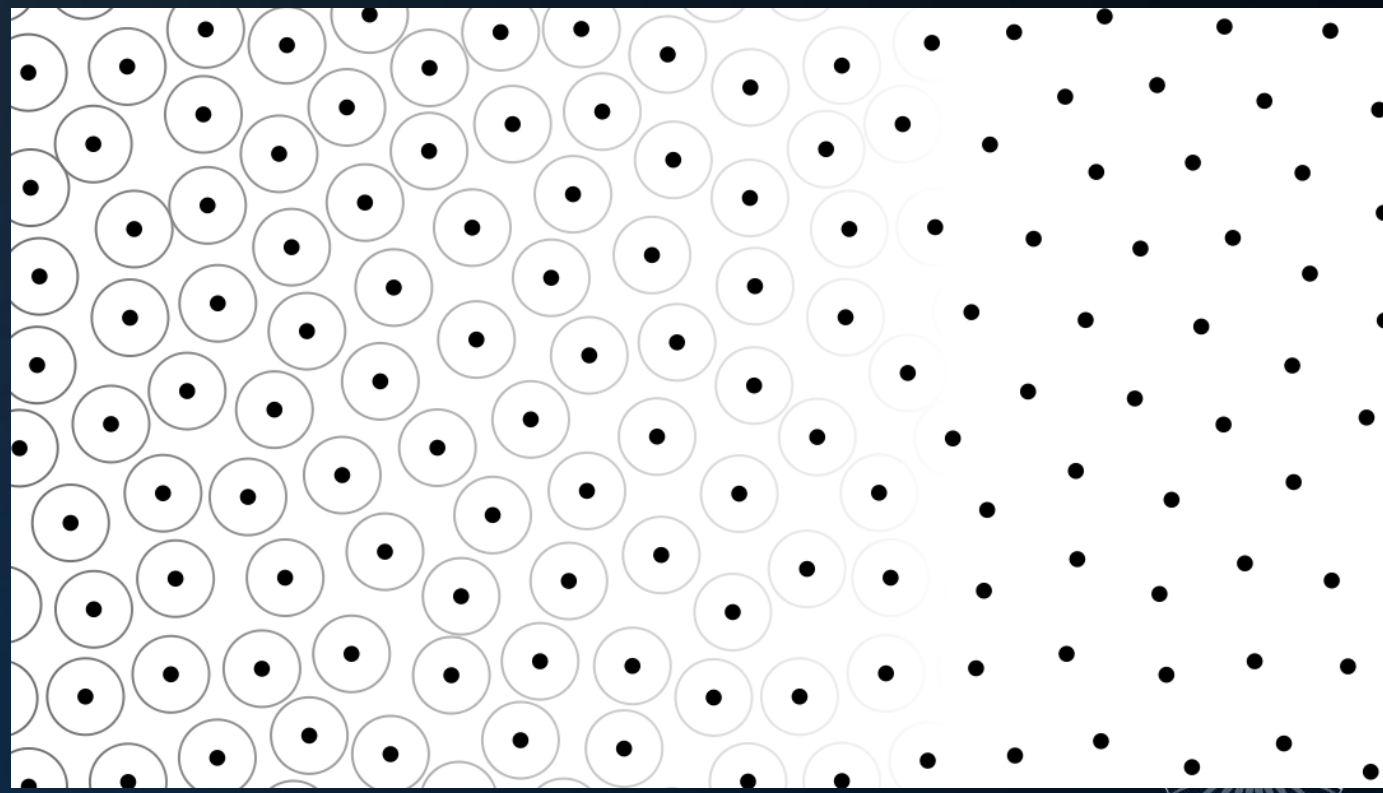
Uniform Random Numbers

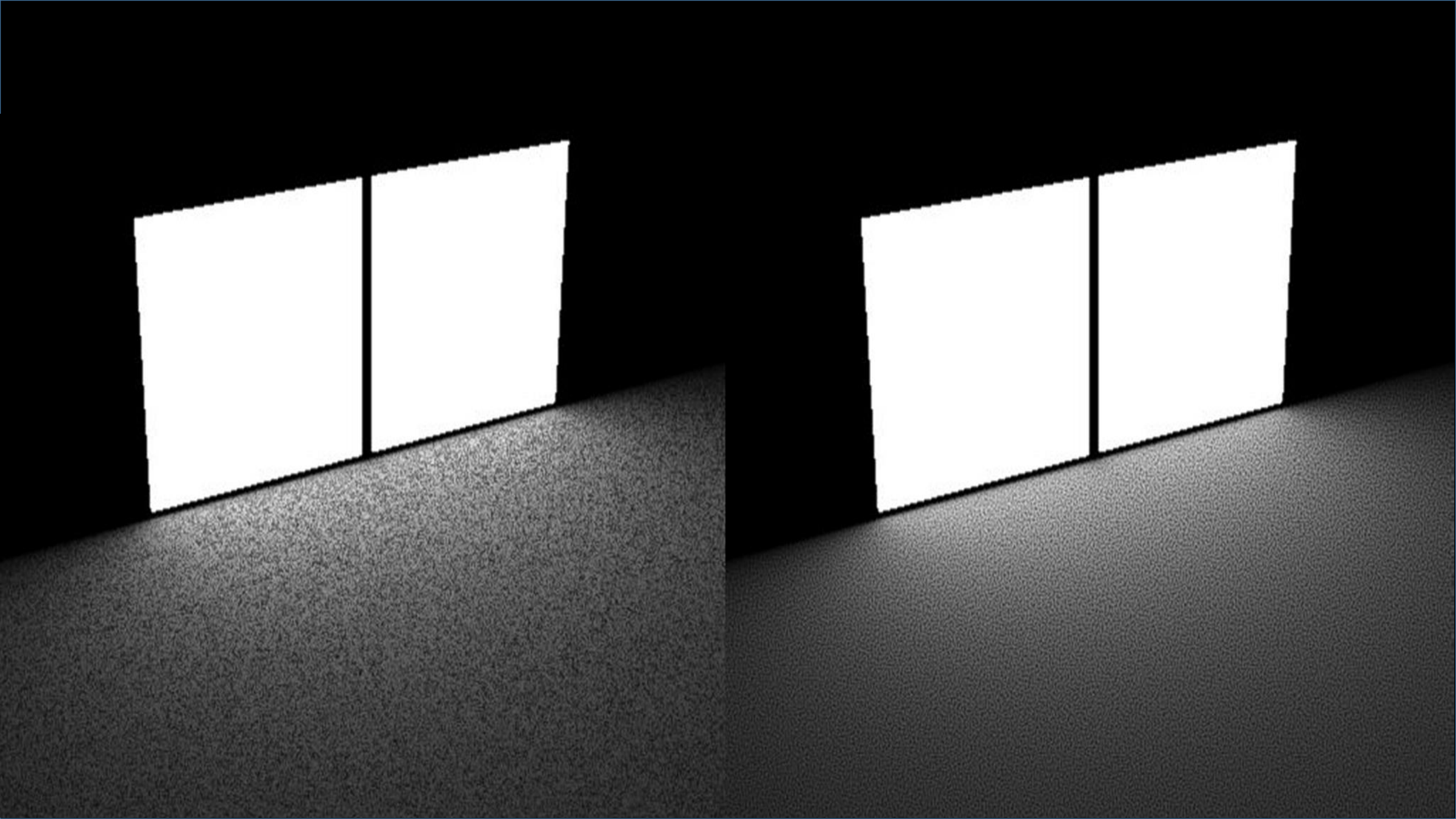
Stratification helps, because it improves the *uniformity* of random numbers.

Other approaches to achieve this:

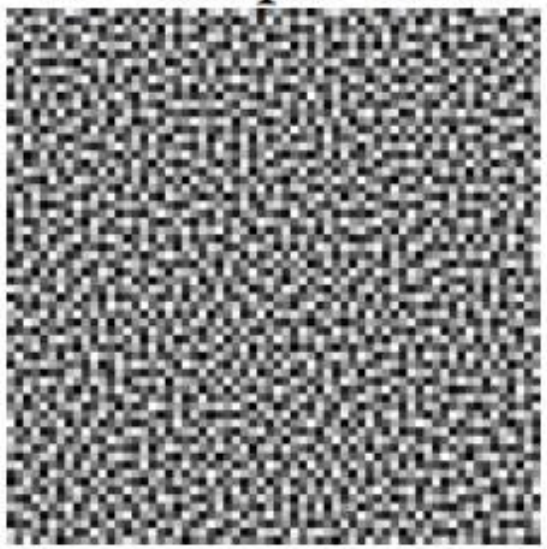
Poisson-disc distributions

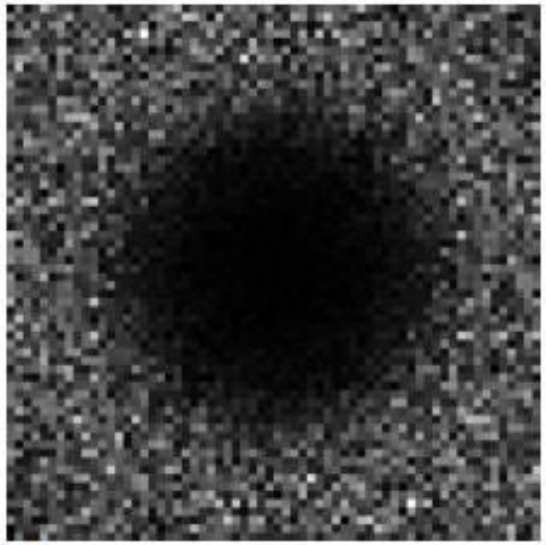
Also known as: *blue noise*.





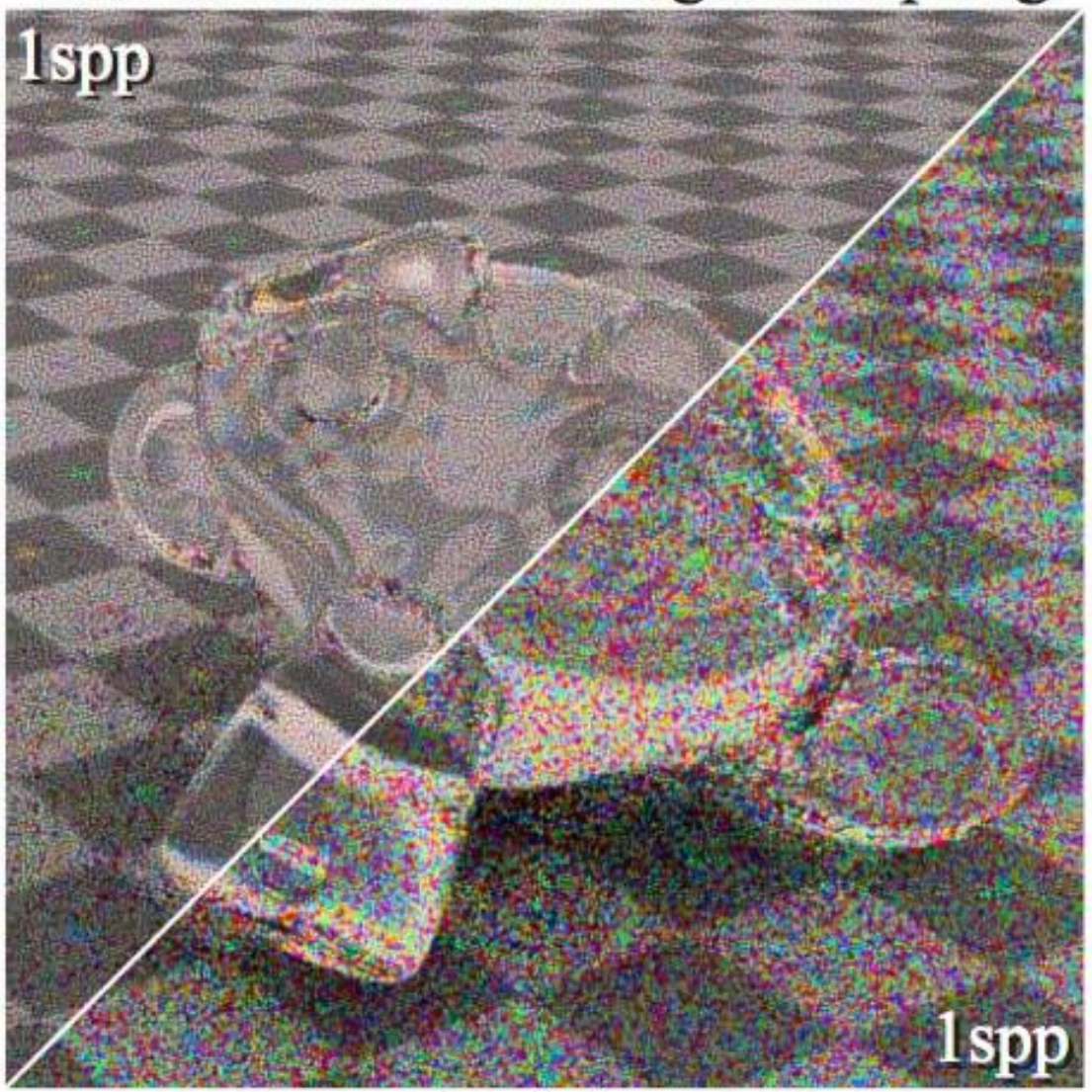
1D-sample mask



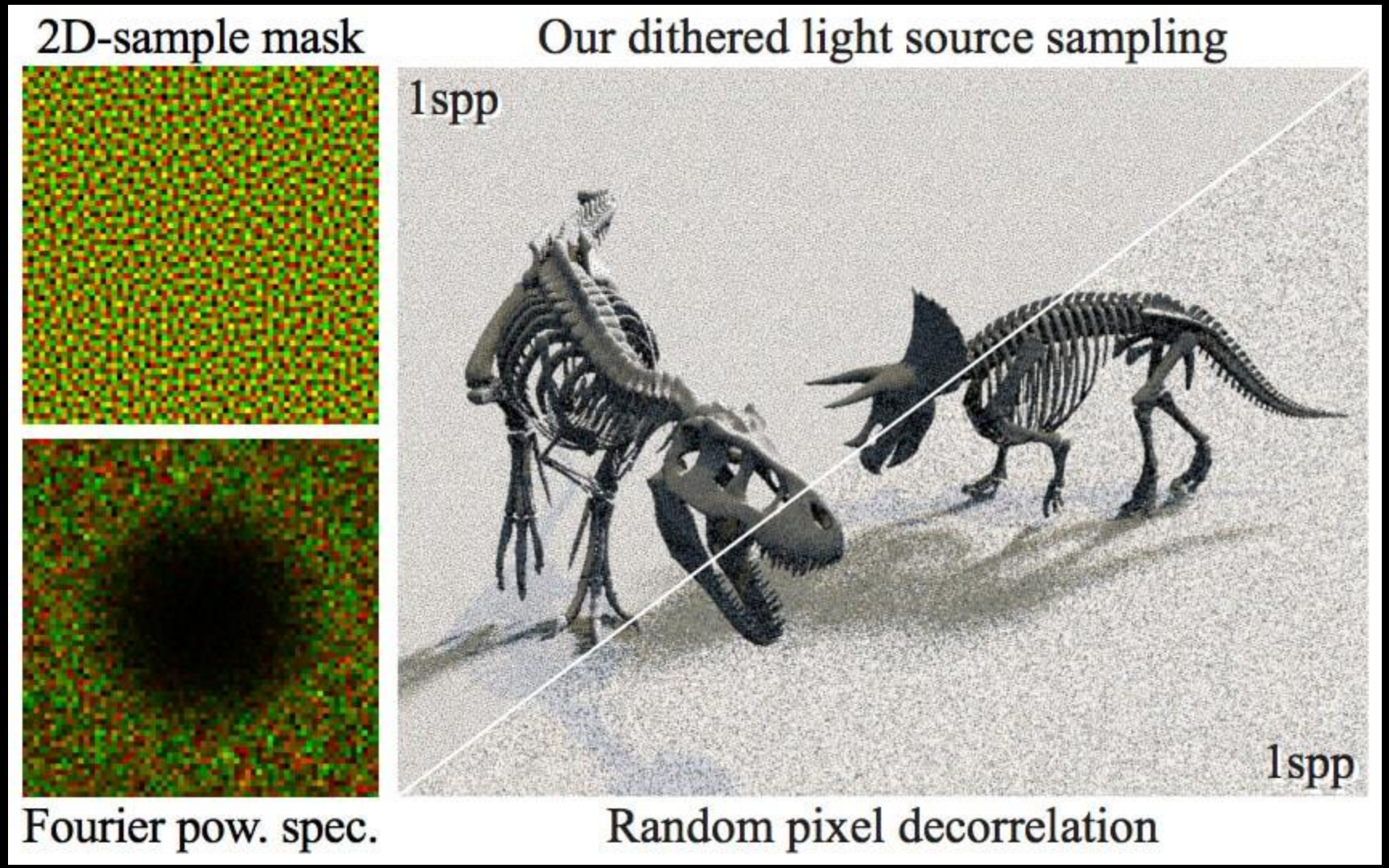


Fourier pow. spec.

Our dithered wavelength sampling



Random pixel decorrelation



https://www.arnoldrenderer.com/research/dither_abstract.pdf

Stratification

Troubleshooting Path Tracing Experiments

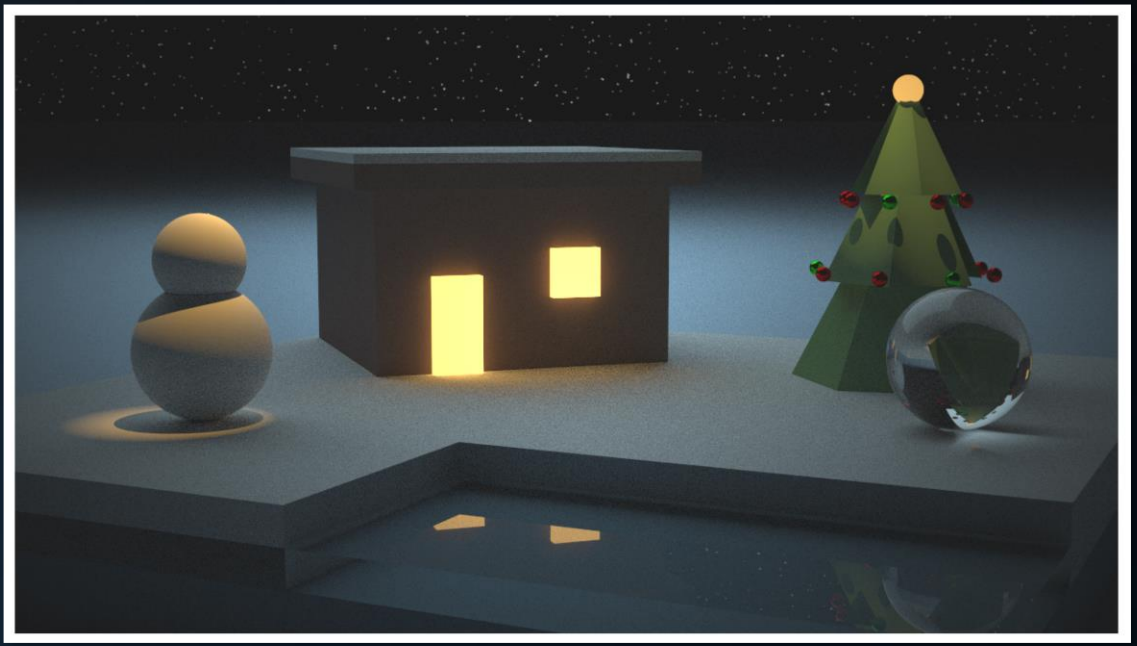
When experimenting with stratification and other variance reduction methods you will frequently produce incorrect images.

Tip:

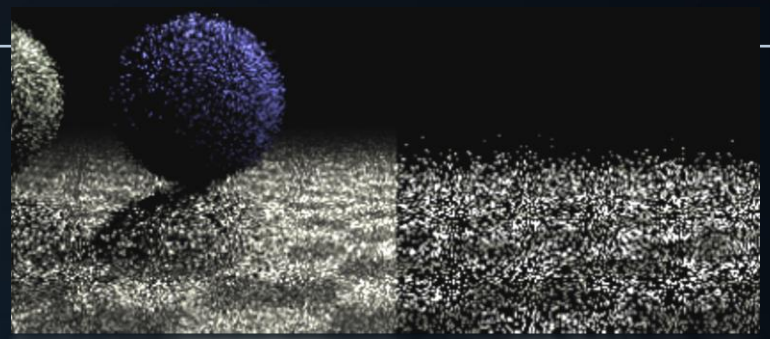
Keep a simple reference path tracer without any tricks.
Compare your output to this reference solution frequently.

Today's Agenda:

- Introduction
- Random Samples
- Next Event Estimation
- Importance Sampling
- Russian Roulette



NEE



Next Event Estimation

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

Also recall that we had two ways to sample direct illumination:

integrating over the hemisphere

$$L_o(p,\omega_o) = \int_\Omega f_r(p,\omega_o,\omega_i)\, L_d(p,\omega_i) \cos\theta_i \, d\omega_i$$

integrating over the lights

$$L(s \leftarrow x) = \sum_{j=1}^{lights} \int_{A_j} f_r(s \leftarrow x \leftarrow x'_j)\, L(x \leftarrow x'_j)\, G(x \leftrightarrow x'_j)\, dA(x')$$

…and the way we sampled it using Monte Carlo:

```
Vector3 L = RandomPointOnLight() - I;
float dist = L.Length();
L /= dist;
float cos_o = Dot( -L, lightNormal );
float cos_i = Dot( L, ray.N );
if (cos_o <= 0 || cos_i <= 0) return BLACK;
// trace shadow ray
Ray r = new Ray(…);
Scene.Intersect( r );
if (r.objIdx != -1) return BLACK;
// V(p,p')=1; calculate transport
Vector3 BRDF = material.diffuse * INVPI;
float solidAngle = …;
return solidAngle * BRDF * lightColor * cos_i;
```

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

Can we apply this to the full rendering equation, instead of just direct illumination?

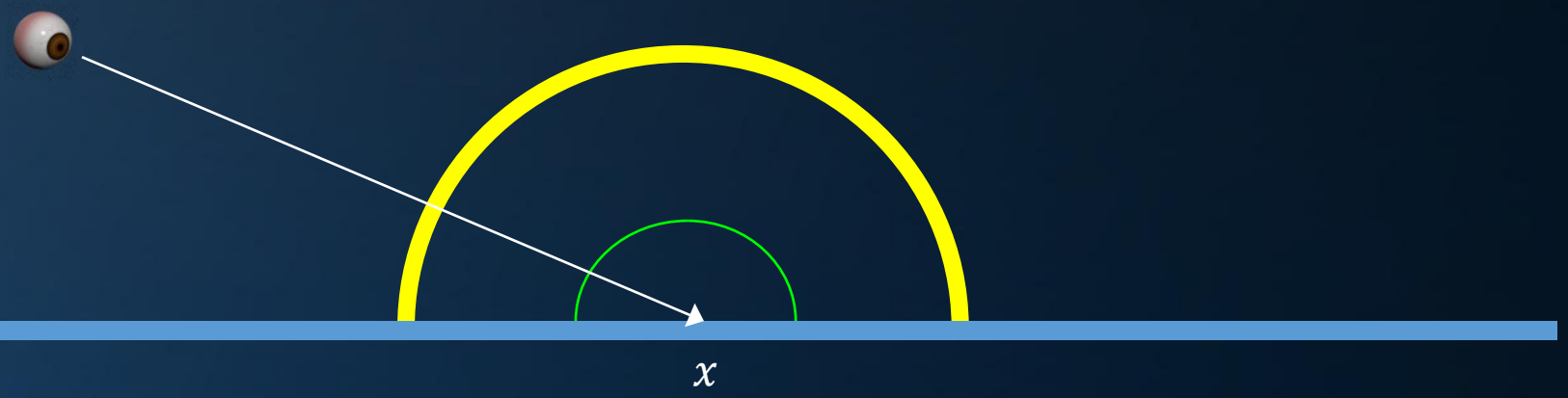
NEE

Incoming direct light

$$= \int_{\Omega} L_d(x, \omega_i) \cos\theta_i \, d\omega_i$$

$$\approx \frac{2\pi}{N} \sum_{i=1}^{N} L_d(p, \omega_i) \cos\theta_i$$

$-\frac{1}{2}\pi$ $+\frac{1}{2}\pi$



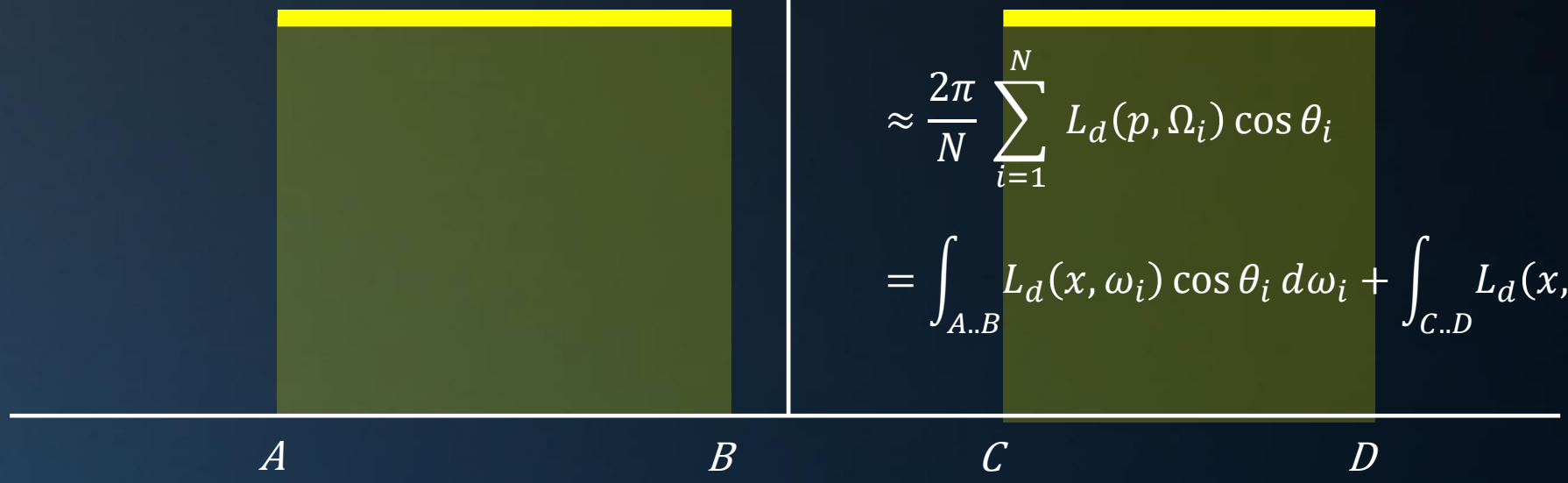
NEE

Incoming direct light

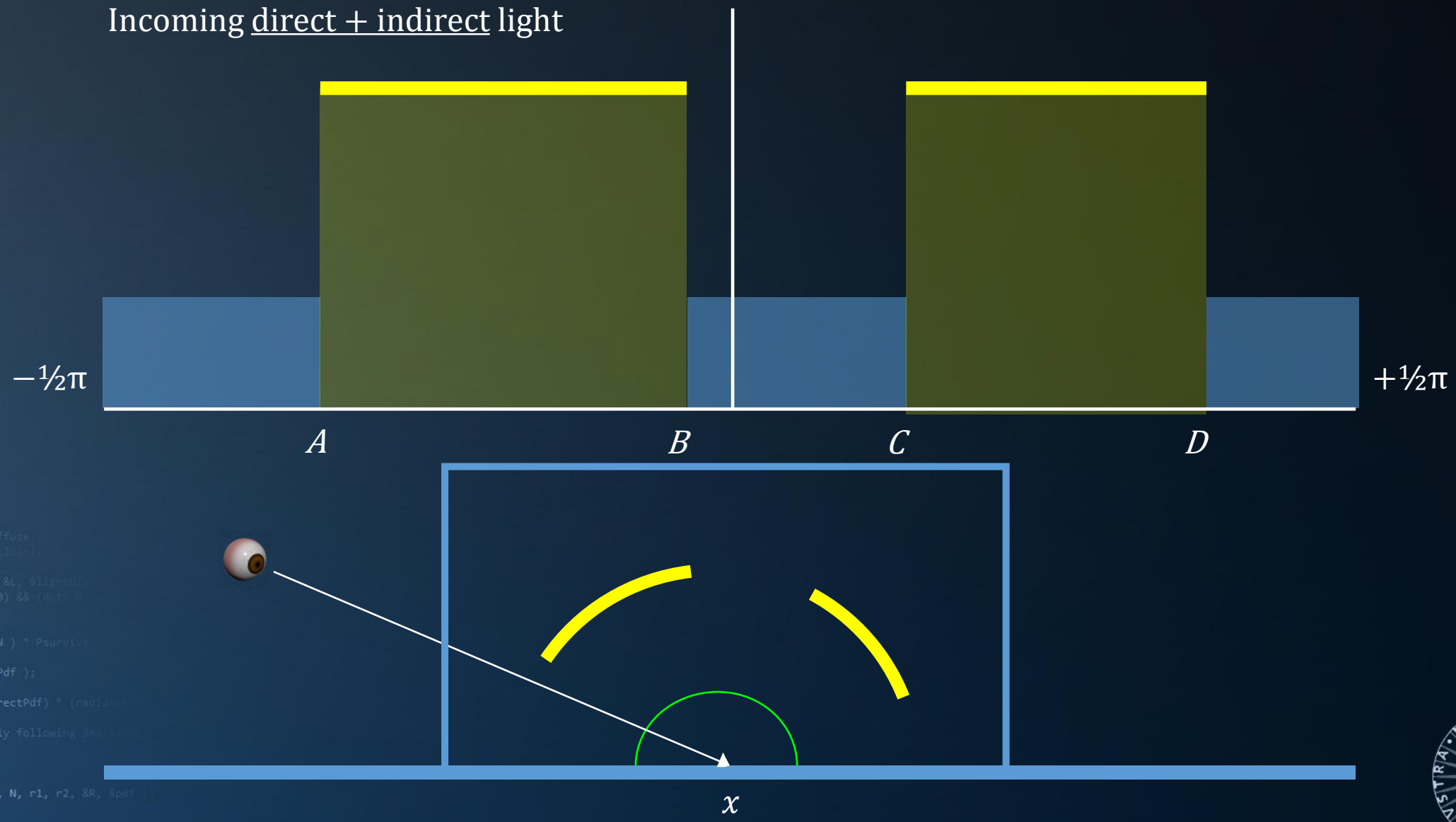
$$= \int_{\Omega} L_d(x, \omega_i) \cos\theta_i \, d\omega_i$$

$$\approx \frac{2\pi}{N} \sum_{i=1}^{N} L_d(p, \Omega_i) \cos\theta_i$$

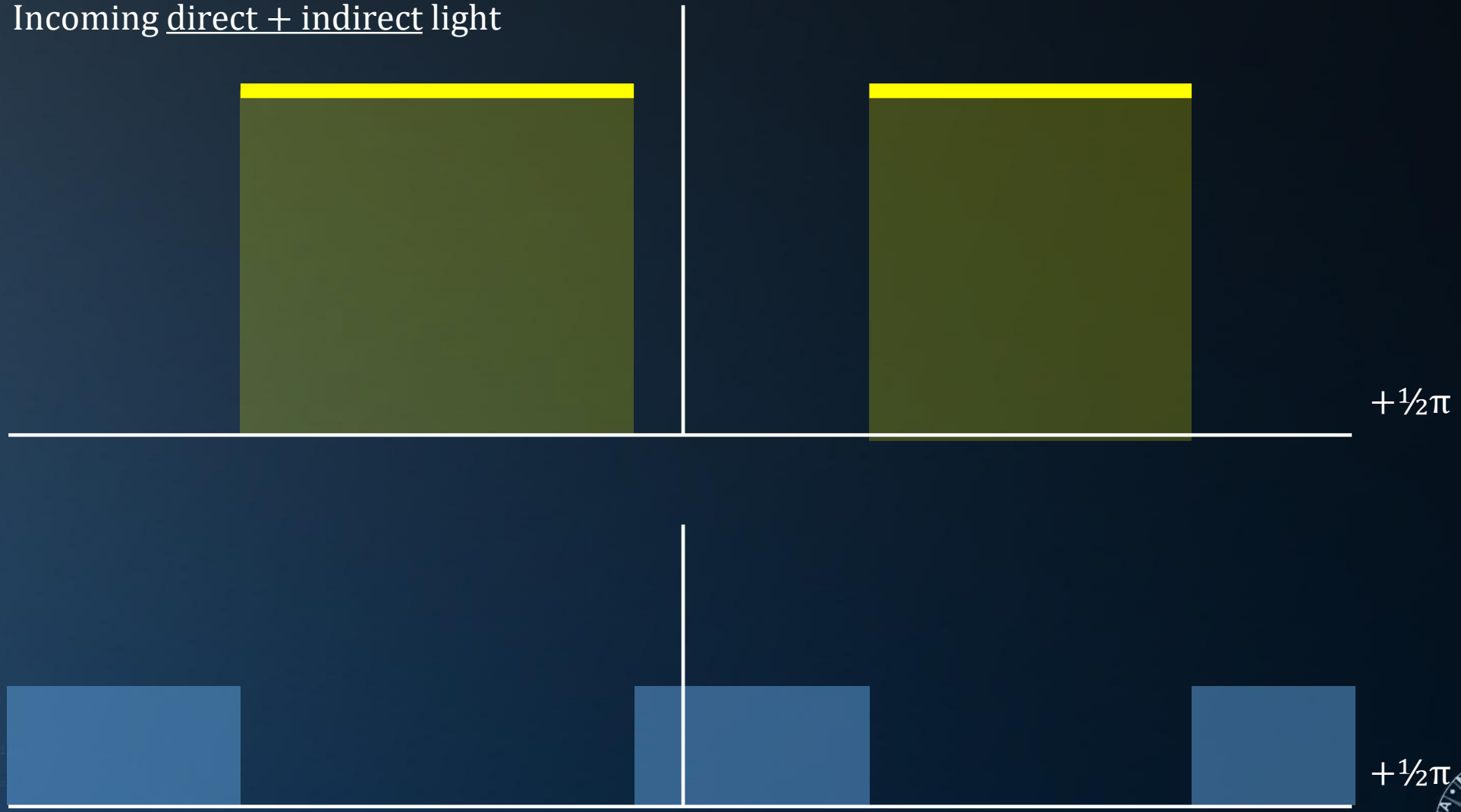
$$= \int_{A..B} L_d(x, \omega_i) \cos\theta_i \, d\omega_i + \int_{C..D} L_d(x, \omega_i) \cos\theta_i \, d\omega_i$$



NEE



NEE

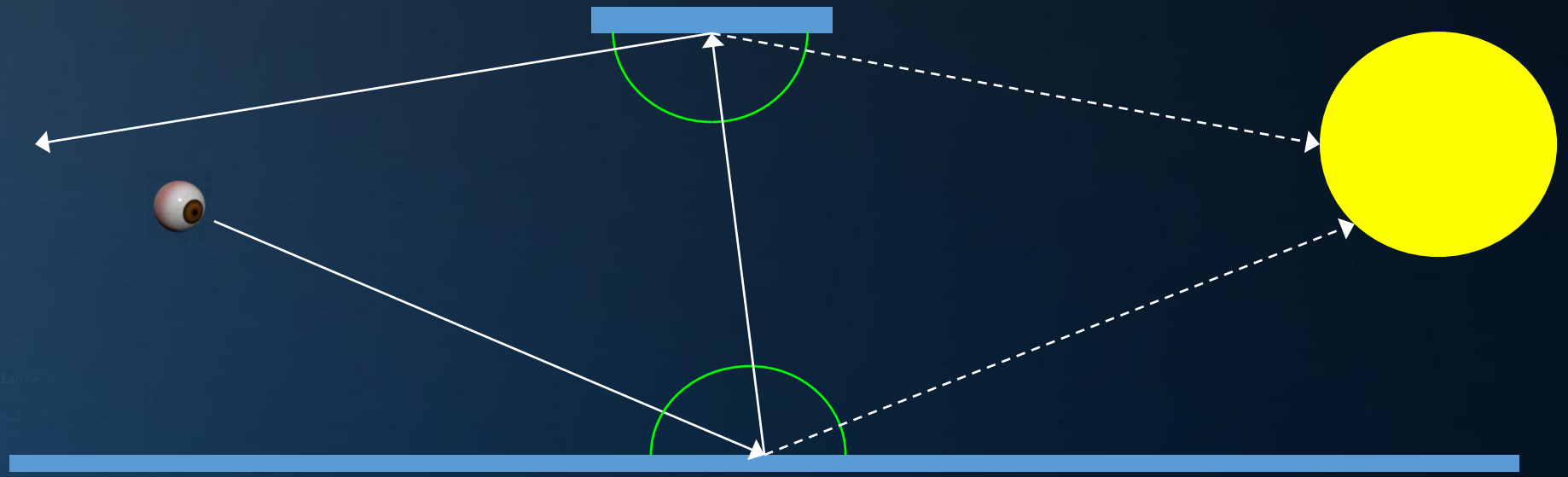


NEE

Next Event Estimation

Observation: light travelling via any vertex on the path consists of indirect light and direct light *for that vertex.*

Next Event Estimation: sampling direct and indirect *separately*.

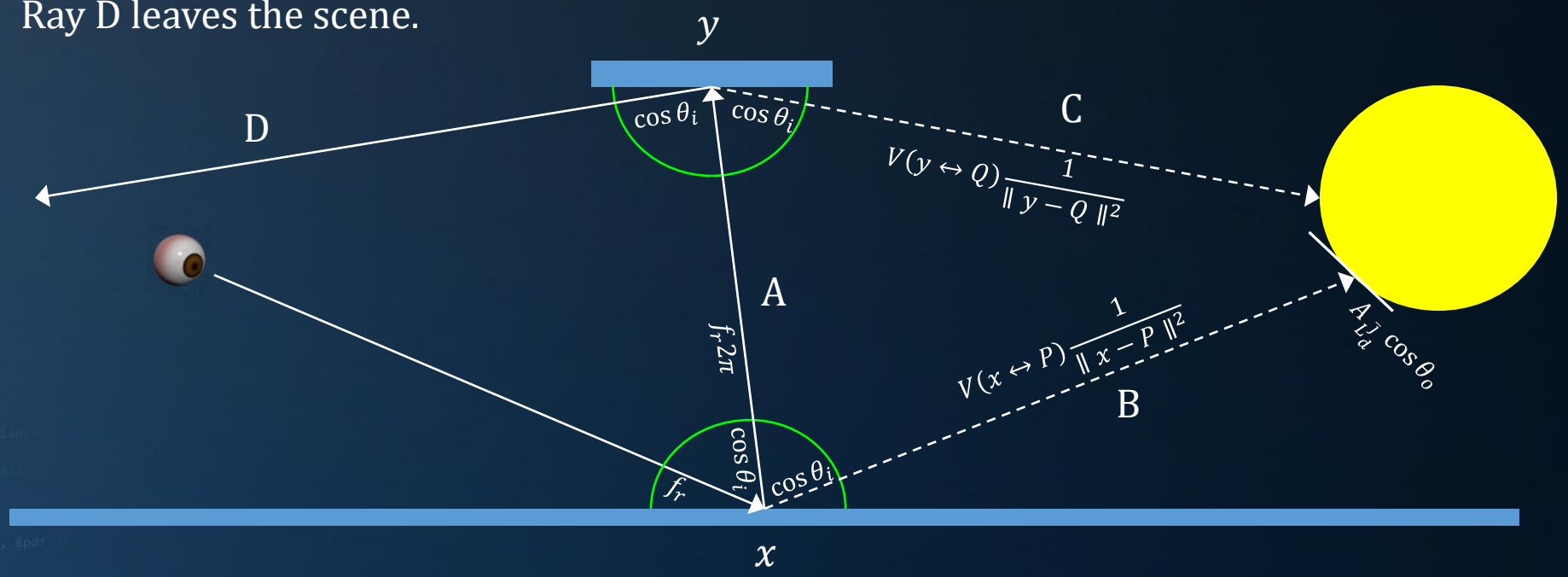


NEE

Next Event Estimation

Per surface interaction, we trace *two* random rays.

- Ray A returns (via point x) the energy reflected by y (estimates indirect light for x).
- Ray B returns the direct illumination on point x (estimates direct light on x).
- Ray C returns the direct illumination on point y, which will reach the sensor via ray A.
- Ray D leaves the scene.

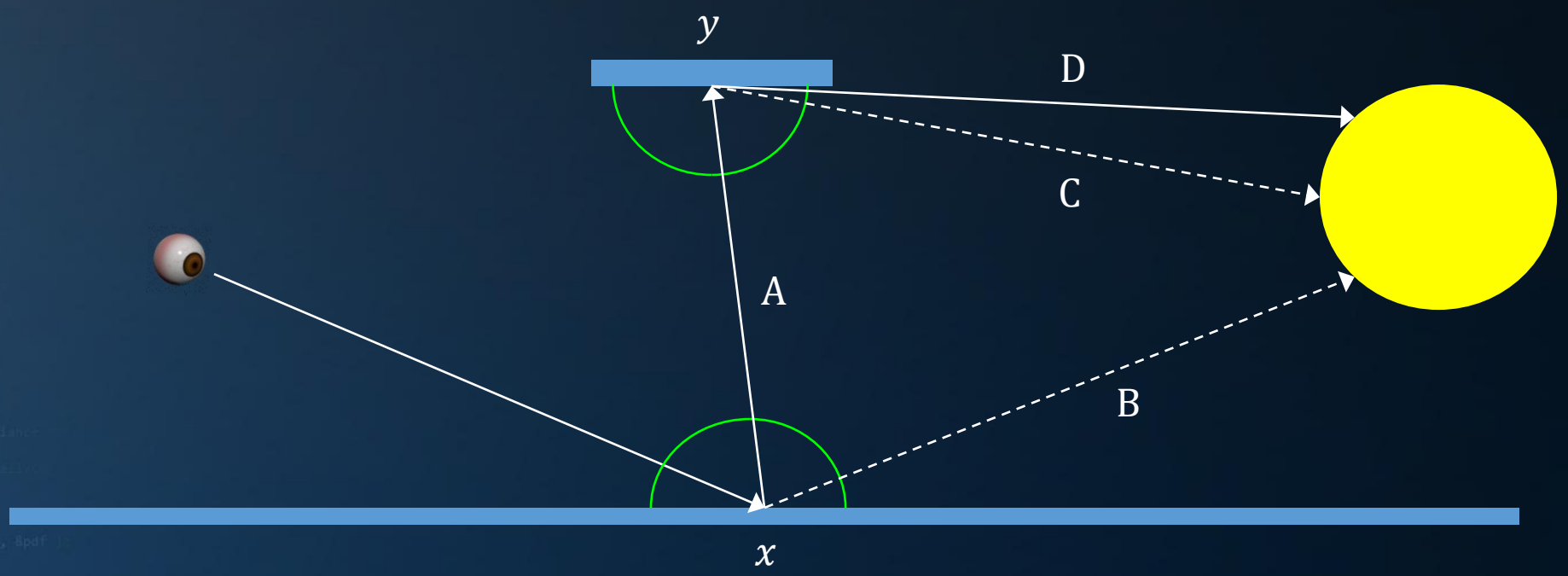


NEE

Next Event Estimation

When a ray for indirect illumination stumbles upon a light, the path is terminated and no energy is transported via ray D:

This way, we prevent accounting for direct illumination on point y twice.



NEE

Next Event Estimation

We thus split the hemisphere into two distinct areas:

1. The area that has the projection of the light source on it;
2. The area that is not covered by this projection.

We can now safely send a ray to each of these areas and sum whatever we find there.

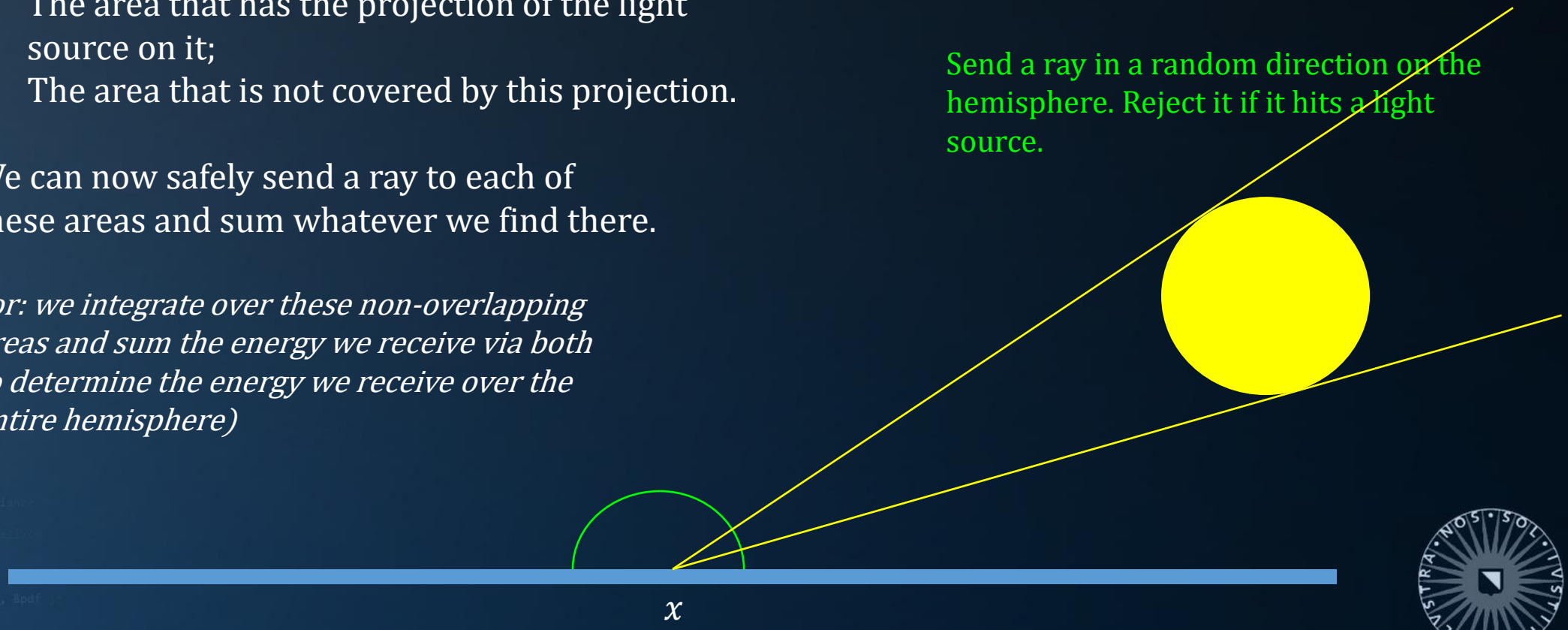
(or: we integrate over these non-overlapping areas and sum the energy we receive via both to determine the energy we receive over the entire hemisphere)

Area 1:

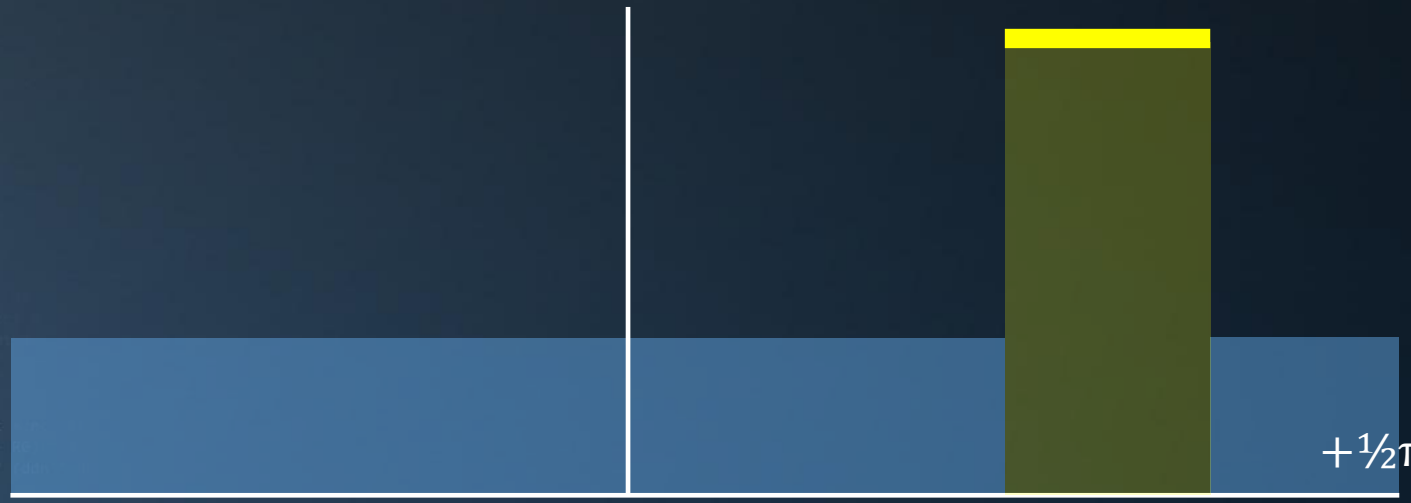
Send a ray directly to a random light source. Reject it (return 0) if it hits anything else than the targeted light.

Area 2:

Send a ray in a random direction on the hemisphere. Reject it if it hits a light source.



NEE

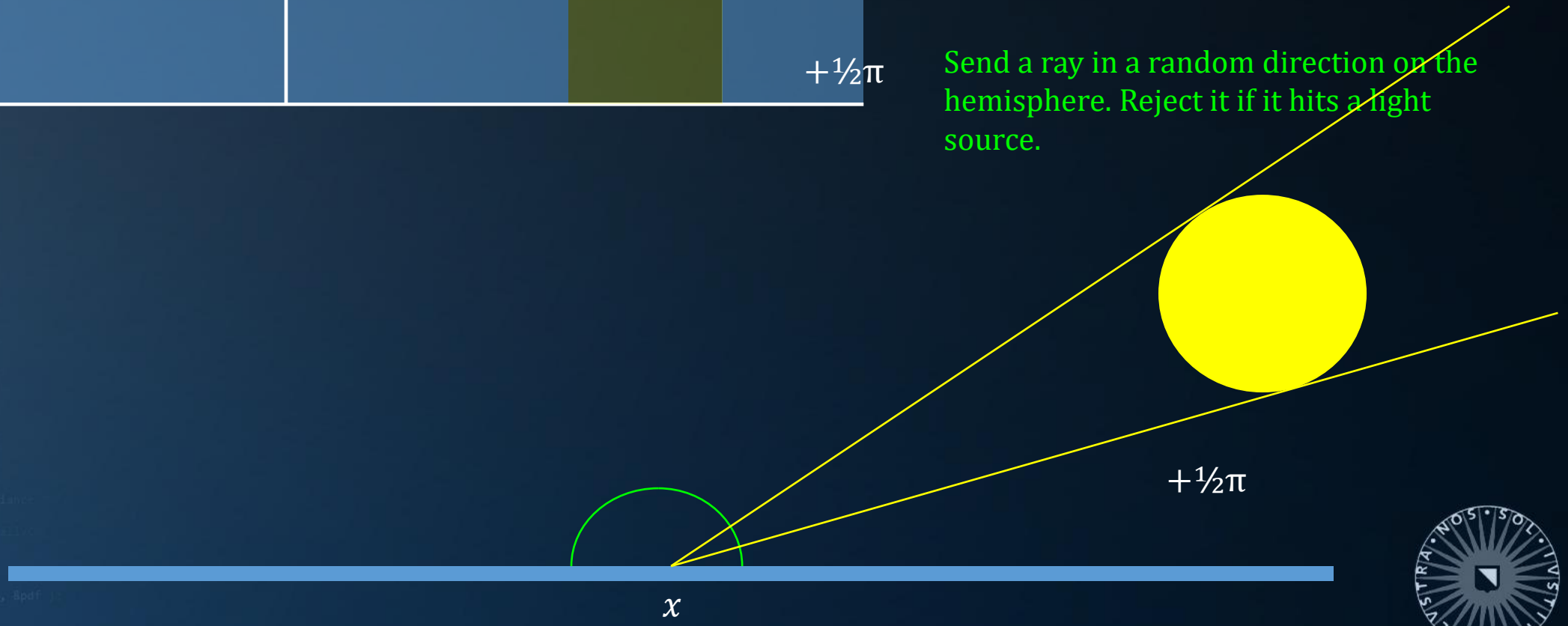


Area 1:

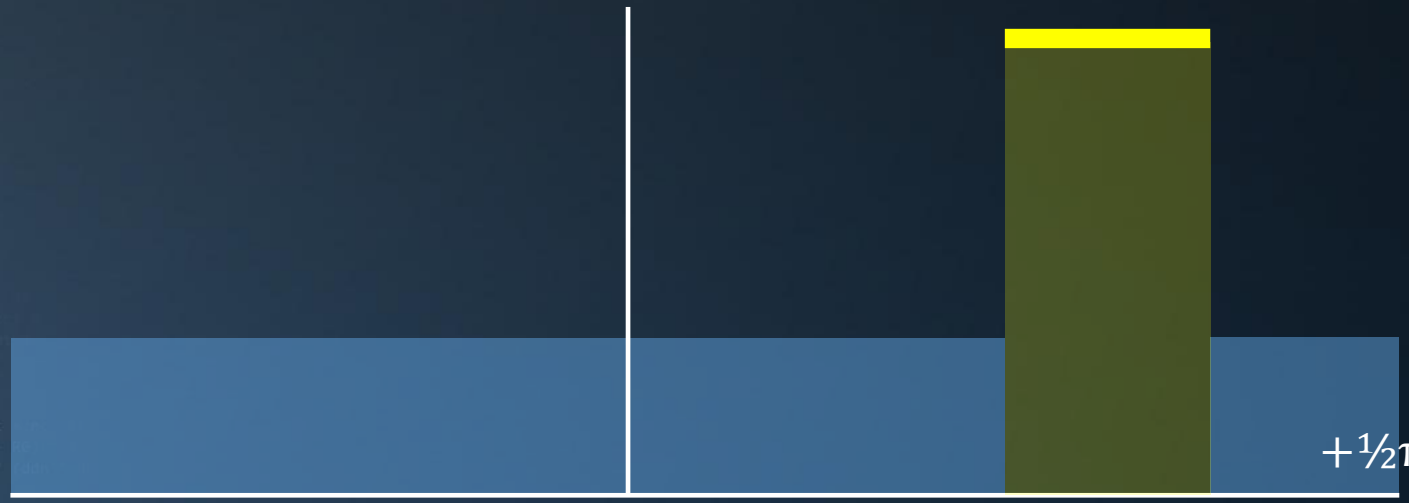
Send a ray directly to a random light source. Reject it if it hits anything else than the targeted light.

Area 2:

Send a ray in a random direction on the hemisphere. Reject it if it hits a light source.



NEE

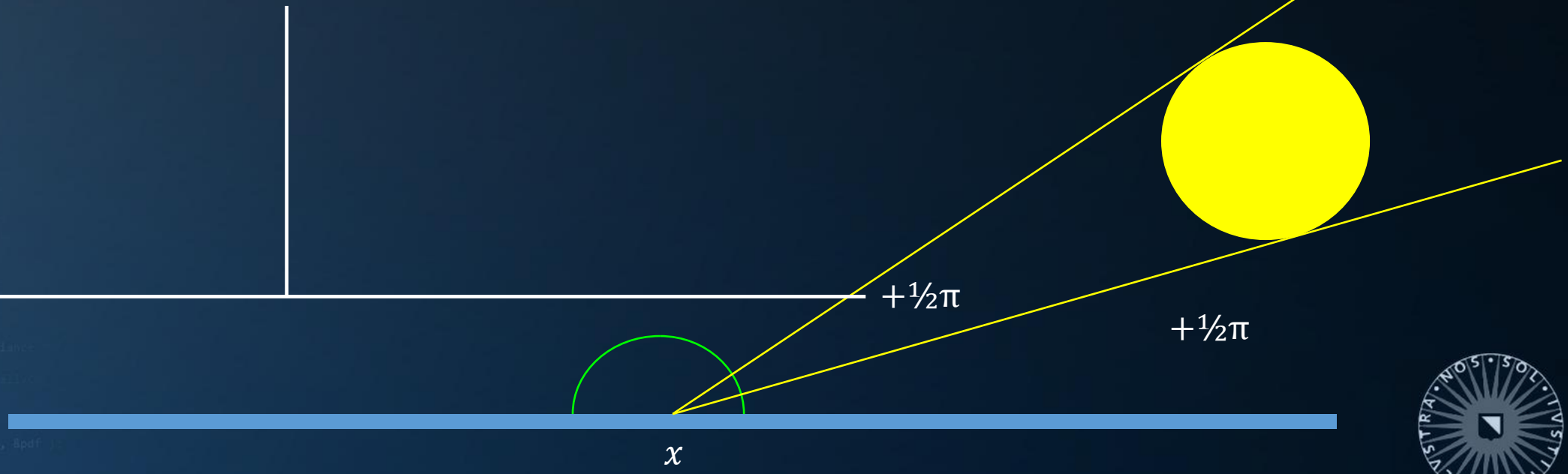


Area 1:

Send a ray directly to a random light source. Reject it if it hits anything else than the targeted light.

Area 2:

Send a ray in a random direction on the hemisphere. Reject it if it hits a light source.



NEE

Next Event Estimation

```
Color Sample( Ray ray )
{
   // trace ray
   I, N, material = FindNearest( ray );
   BRDF = material.albedo / PI;
   // terminate if ray left the scene
   if (ray.NOHIT) return BLACK;
   // terminate if we hit a light source
   if (material.isLight) return LIGHT_COLOR;
   // continue random walk
   R = DiffuseReflection( N );
   Ray r( I, R );
   Ei = Sample( r ) * (N·R);
   return PI * 2.0f * BRDF * Ei;
}
```

NEE

Next Event Estimation

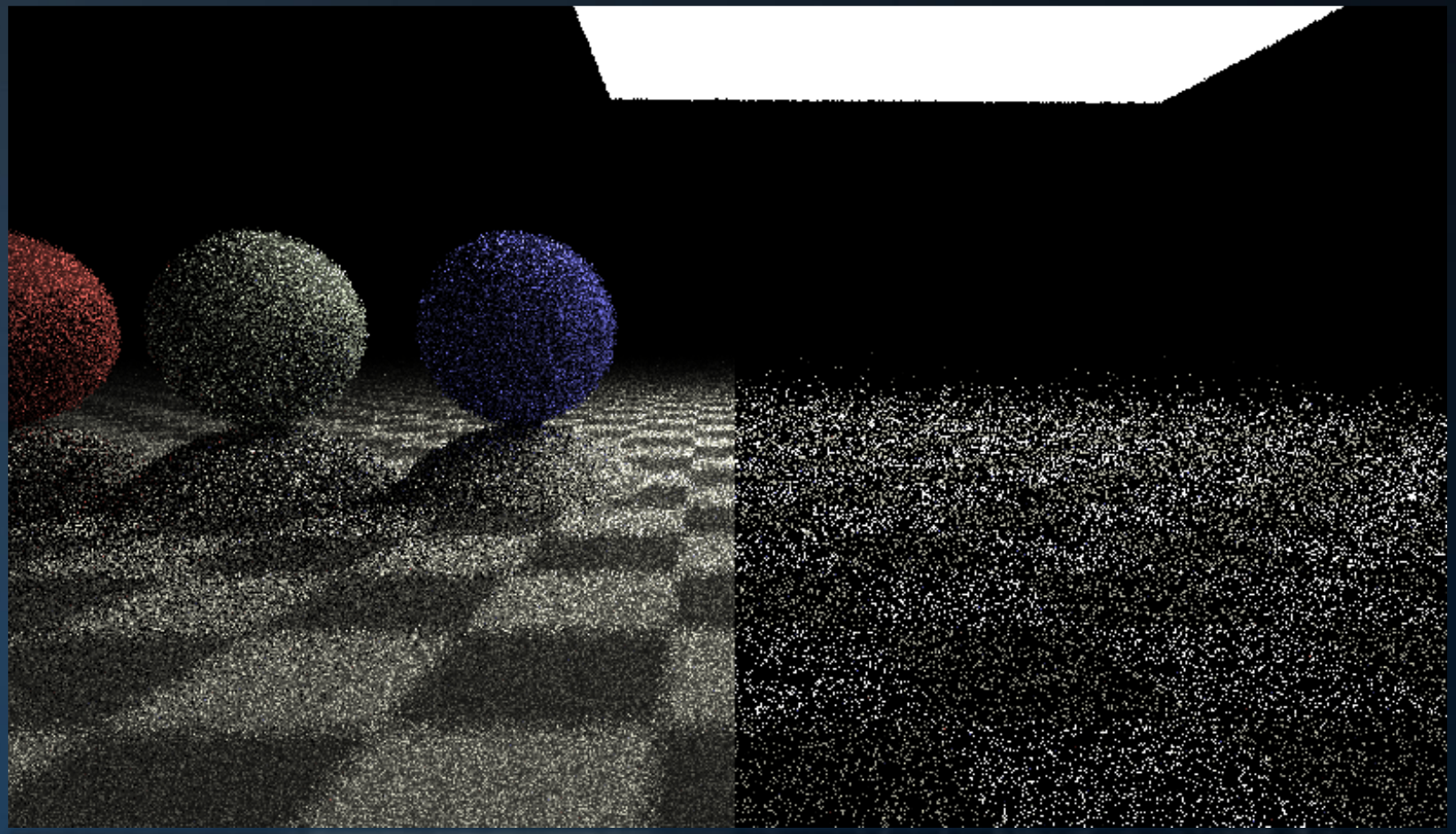
```
Color Sample( Ray ray )
{
   // trace ray
   I, N, material = FindNearest( ray );
   BRDF = material.albedo / PI;
   // terminate if ray left the scene
   if (ray.NOHIT) return BLACK;
   // terminate if we hit a light source
   if (material.isLight) return LIGHT_COLOR;
   // sample a random light source
   L, Nl, dist, A = RandomPointOnLight();
   Ray lr( I, L, dist );
   if (N·L > 0 && Nl·-L > 0) if (!Occluded( lr ))
   {
      solidAngle = ((Nl·-L) * A) / dist²;
      Ld = lightColor * solidAngle * BRDF *
           N·L * lightCount;
   }
   // continue random walk
   R = DiffuseReflection( N );
   Ray r( I, R );
   Ei = Sample( r ) * (N·R);
   return PI * 2.0f * BRDF * Ei + Ld;
}
```

NEE

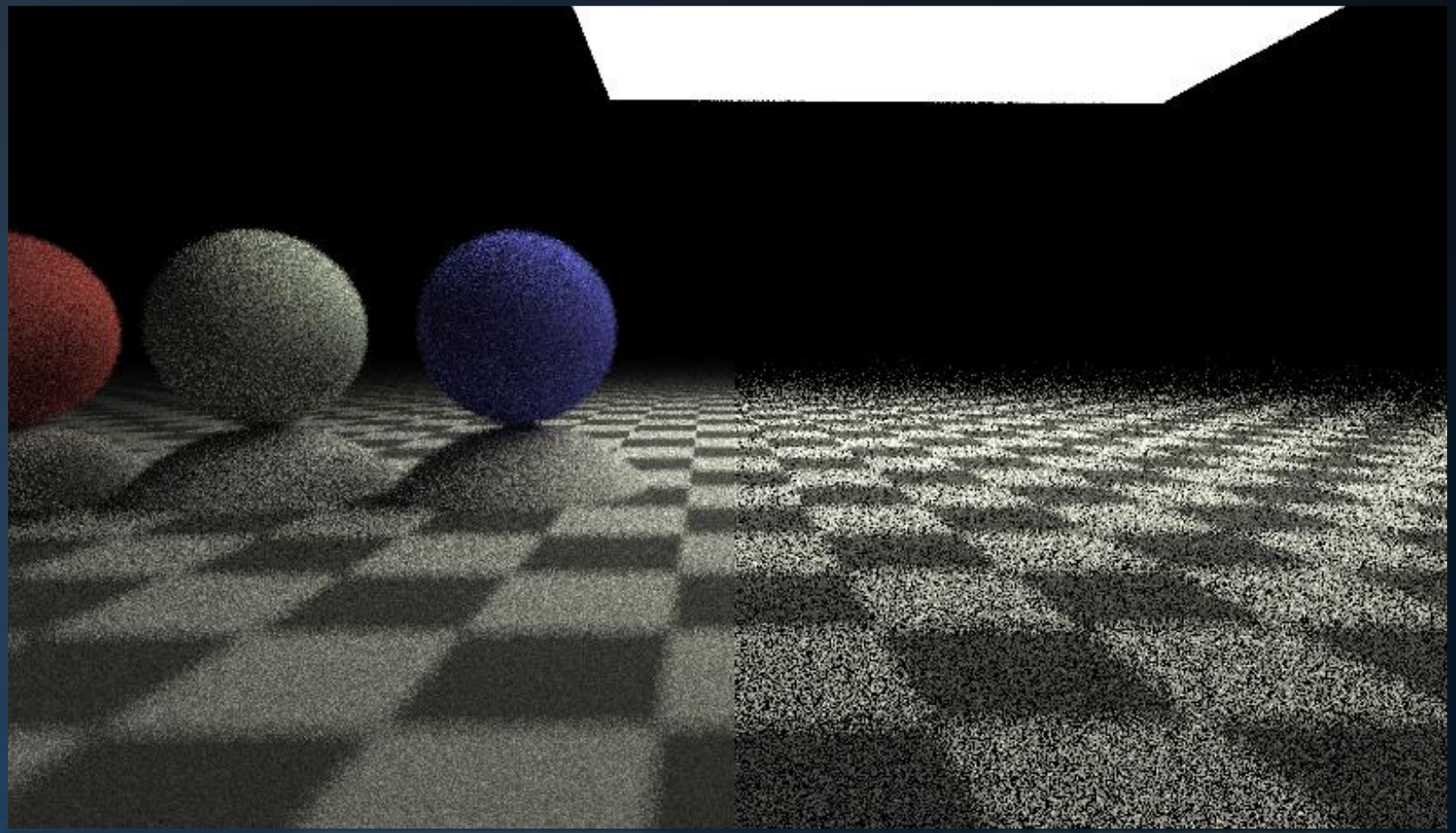
Next Event Estimation

```
Color Sample( Ray ray )
{
   // trace ray
   I, N, material = FindNearest( ray );
   BRDF = material.albedo / PI;
   // terminate if ray left the scene
   if (ray.NOHIT) return BLACK;
   // terminate if we hit a light source
   if (material.isLight) return BLACK;
   // sample a random light source
   L, Nl, dist, A = RandomPointOnLight();
   Ray lr( I, L, dist );
   if (N·L > 0 && Nl·-L > 0) if (!Occluded( lr ))
   {
      solidAngle = ((Nl·-L) * A) / dist²;
      Ld = lightColor * solidAngle * BRDF *
           N·L * lightCount;
   }
   // continue random walk
   R = DiffuseReflection( N );
   Ray r( I, R );
   Ei = Sample( r ) * (N·R);
   return PI * 2.0f * BRDF * Ei + Ld;
}
```

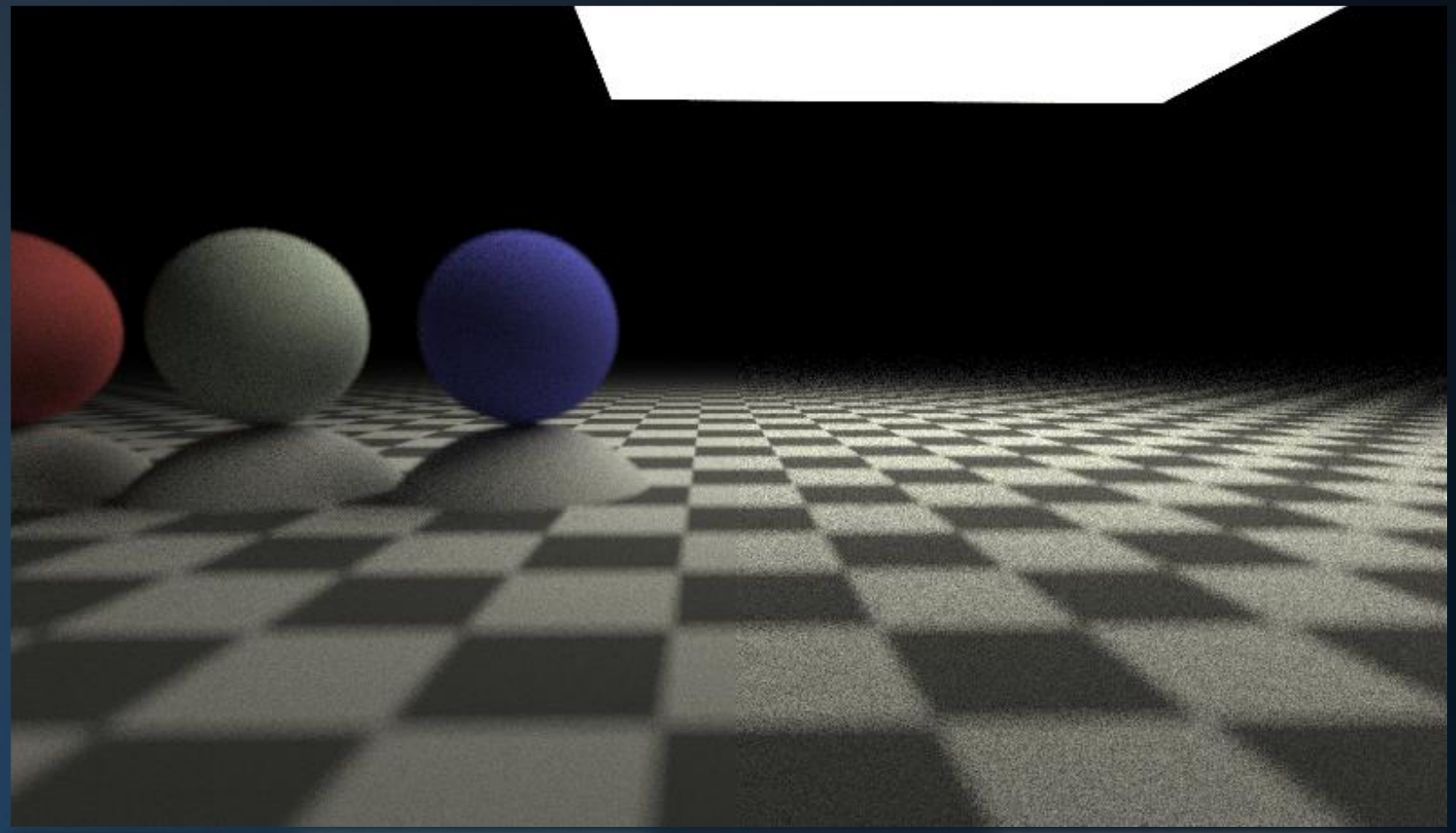
NEE



NEE



NEE



NEE

Next Event Estimation

Some vertices require special attention:

- If the first vertex after the camera is emissive, its energy can't be reflected to the camera.
- For specular surfaces, the BRDF to a light is always 0.

Since a light ray doesn't make sense for specular vertices, we will include emission from a vertex directly following a specular vertex.

The same goes for the first vertex after the camera: if this is emissive, we will also include this.

This means we need to keep track of the type of the previous vertex during the random walk.

NEE

```
Color Sample( Ray ray, bool lastSpecular )
{
   // trace ray
   I, N, material = Trace( ray );
   BRDF = material.albedo / PI;
   // terminate if ray left the scene
   if (ray.NOHIT) return BLACK;
   // terminate if we hit a light source
   if (material.isLight)
      if (lastSpecular) return material.emissive;
                   else return BLACK;
   // sample a random light source
   L, Nl, dist, A = RandomPointOnLight();
   Ray lr( I, L, dist );
   if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
   {
      solidAngle = ((Nl·-L) * A) / dist²;
      Ld = lightColor * solidAngle * BRDF * N·L;
   }
   // continue random walk
   R = DiffuseReflection( N );
   Ray r( I, R );
   Ei = Sample( r, false ) * (N·R);
   return PI * 2.0f * BRDF * Ei + Ld;
}
```

Further Reading

Part 1:

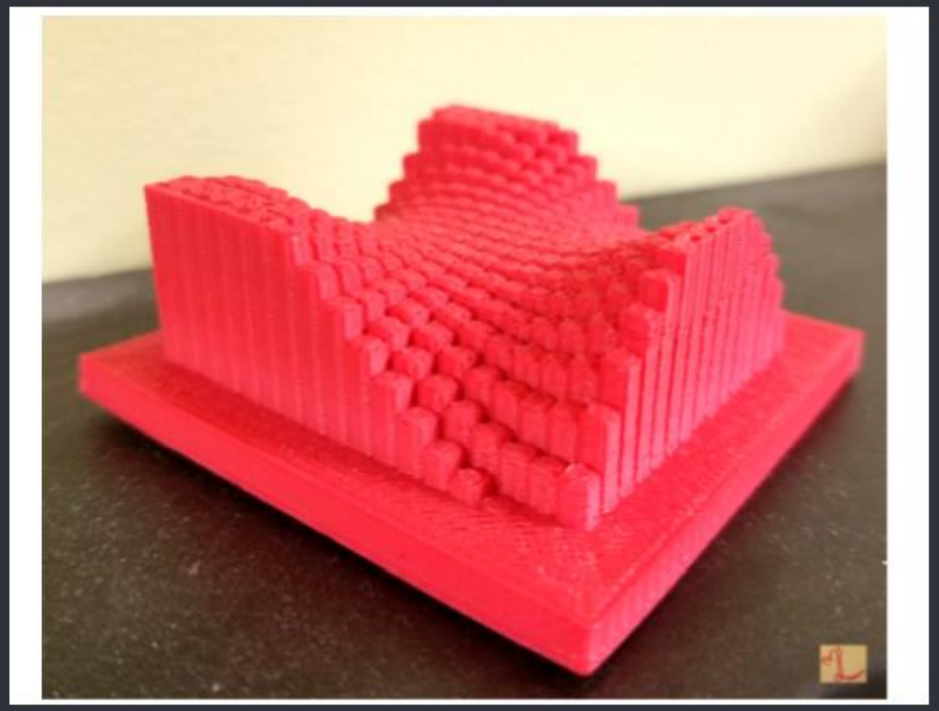
https://jacco.ompf2.com/2019/12/11/probability-theory-for-physically-based-rendering

Part 2:

https://jacco.ompf2.com/2019/12/13/probability-theory-for-physically-based-rendering-part-2/

Jacco's Blog Links, Code & Articles

HOME PROJECTS PUBLICATIONS PERSONAL



Posted in Article, Tutorial

18 Comments

Probability Theory for Physically Based Rendering

by jbikker on December 11, 2019

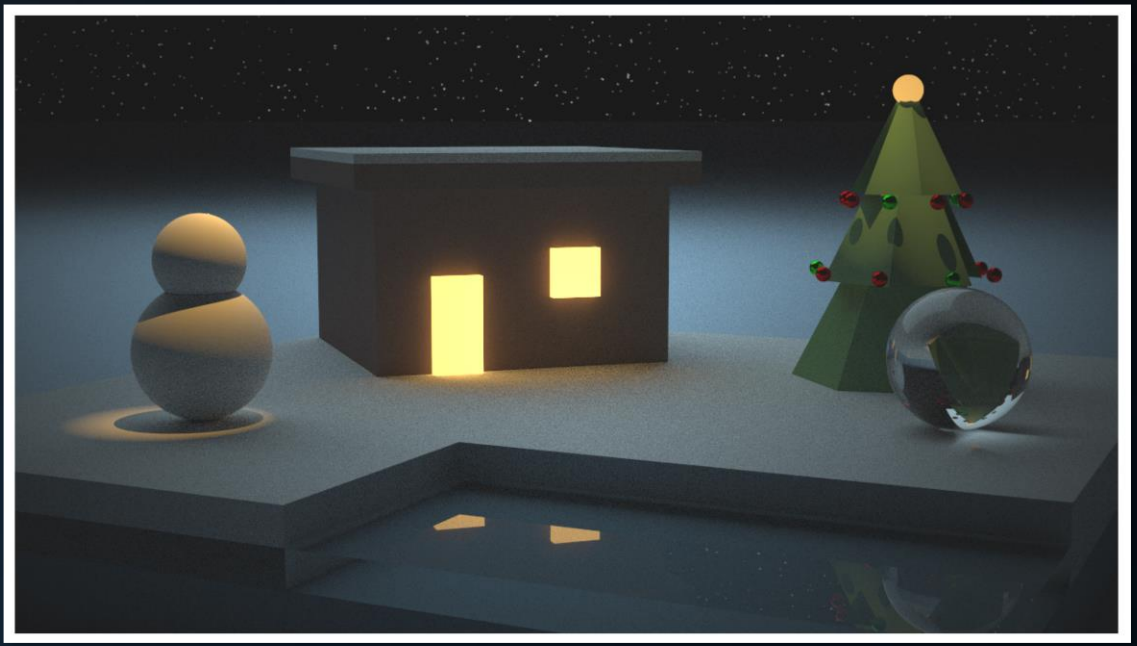
Introduction

Rendering frequently involves the evaluation of multidimensional definite integrals: e.g., the visibility of an area light, radiance arriving over the area of a pixel, radiance arriving over a period of time, and the irradiance arriving over the hemisphere of a surface point. Evaluation of these integrals is typically done using Monte-Carlo integration, where the integral is replaced by the expected value of a stochastic

Today's Agenda:

- Introduction
- Random Samples
- Next Event Estimation

- Importance Sampling
- Russian Roulette



Importance Sampling

Importance Sampling for Monte Carlo

Monte Carlo integration:

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

Example 1: rolling two dice D_1 and D_2, the outcome is $6D_1 + D_2$.
What is the expected value of this experiment?

*(Answer: average die value is 3.5, so the answer is 3.5 * 6 + 3.5 = 24.5)*

Using Monte Carlo:

$$V = \frac{1}{N}\sum_{i=1}^{N} f(D_1) + g(D_2) \quad where: \quad D_1, D_2 \in \{1,2,3,4,5,6\}, \quad f(x) = 6x, g(x) = x$$

Importance Sampling

Importance Sampling for Monte Carlo

Changing the experiment slightly: each sample is one roll of one die.

Using Monte Carlo:

$$V = \frac{1}{N}\sum_{i=1}^{N}\frac{f(T,D)}{0.5} \quad where: D \in \{1,2,3,4,5,6\}, T \in \{0,1\}, f(t,d) = (5t+1)\,d$$

0.5: Probability of using die T.

```
for( int i = 0; i < 1000; i++ )
{
   int D1 = IRand( 6 ) + 1;
   int D2 = IRand( 6 ) + 1;
   float f = (float)(6 * D1 + D2);
   total += f;
   rolls++;
}
```

```
for( int i = 0; i < 2000; i++ )
{
   int D = IRand( 6 ) + 1;
   int T = IRand( 2 );
   float f = (float)((5 * T + 1) * D) / 0.5f;
   total += f;
   rolls++;
}
```

Importance Sampling

Importance Sampling for Monte Carlo

What happens when we don't pick each die with the same probability?

```
float D1_prob = 0.8f;

for( int i = 0; i < 1000; i++ )
{
   int D = IRand( 6 ) + 1;
   float r = Rand(); // uniform 0..1
   int T = (r < D1_prob) ? 0 : 1;
   float p = (T == 0) ? D1_prob : (1 – D1_prob);
   float f = (float)((5 * T + 1) * D) / p;
   total += f;
   rolls++;
}
```

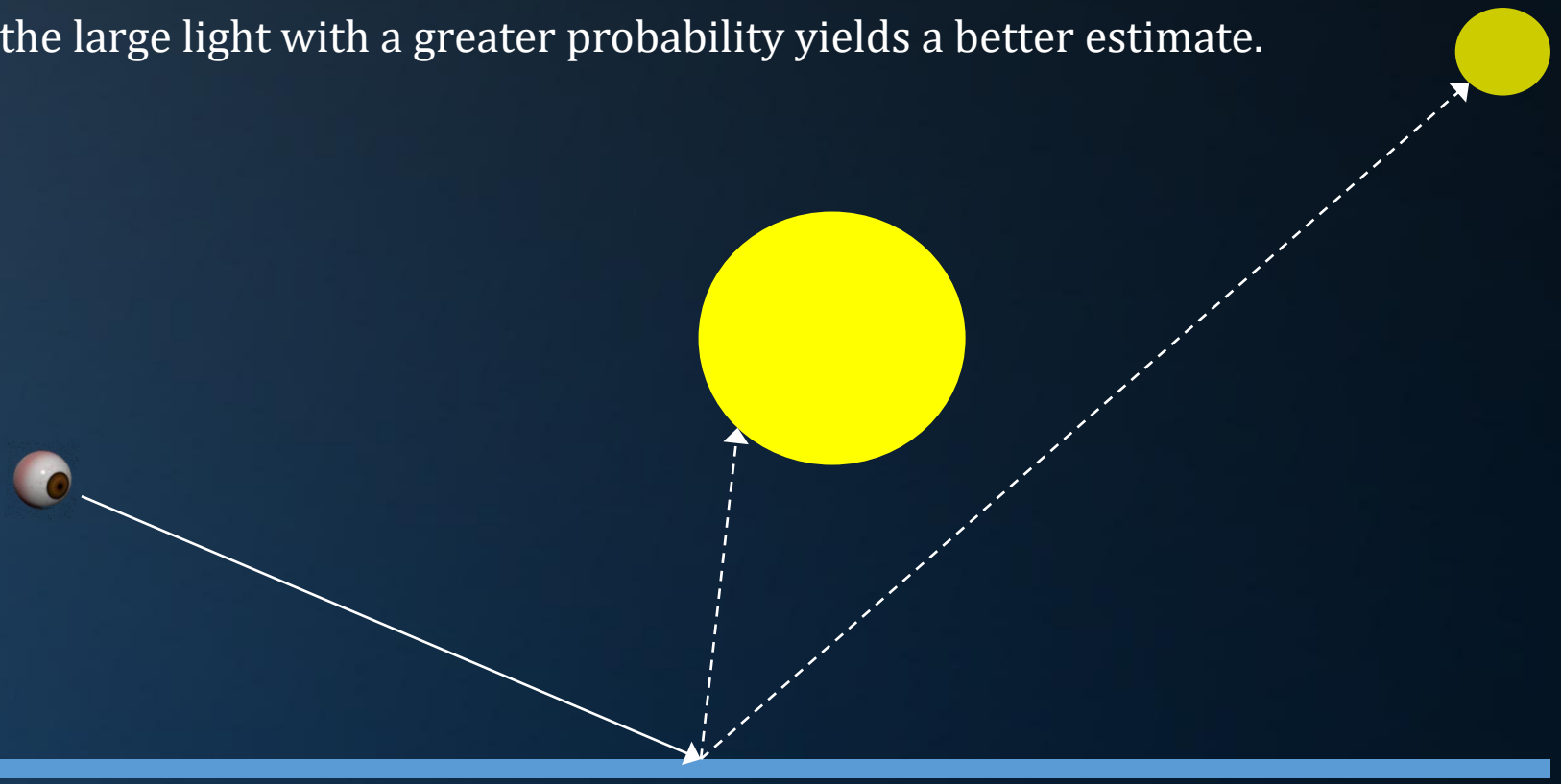
- we get the correct answer;
- we get lower variance.

Importance Sampling

Importance Sampling for Monte Carlo

Example 2: sampling two area lights.

Sampling the large light with a greater probability yields a better estimate.

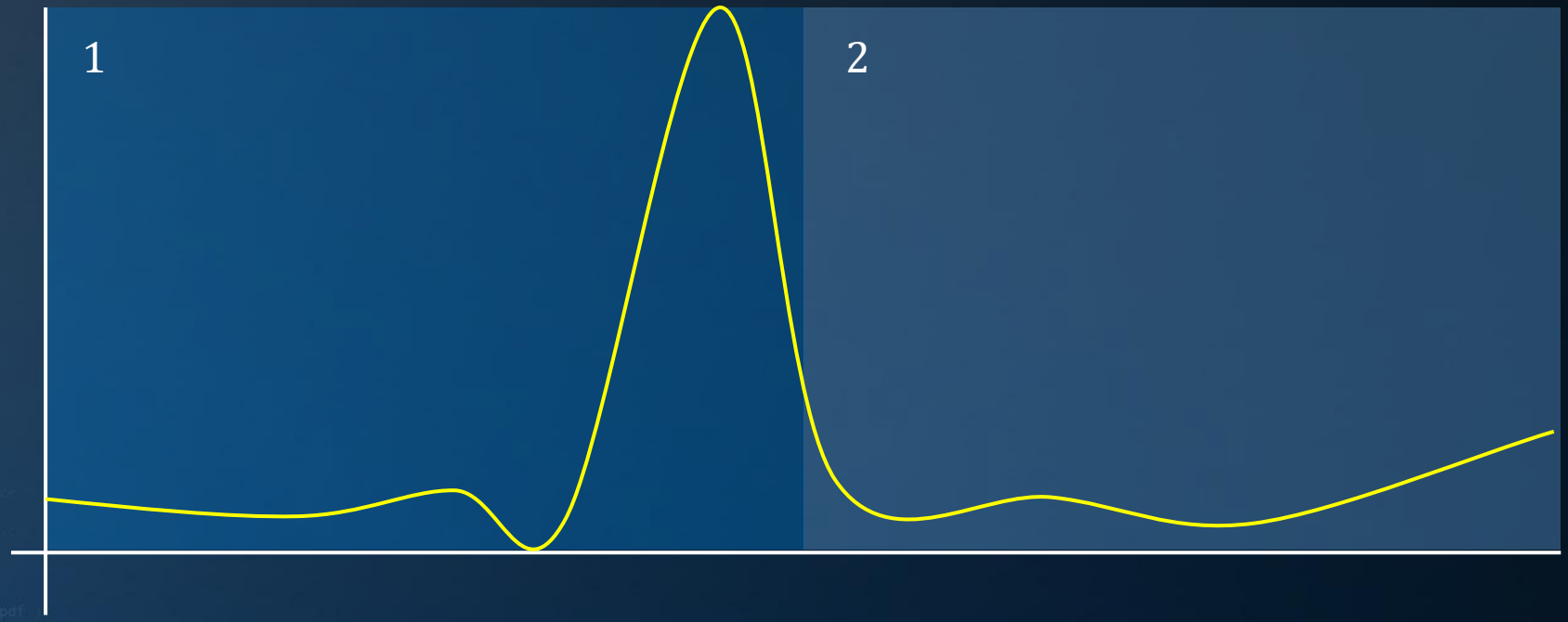


Importance Sampling

Importance Sampling for Monte Carlo

Example 3: sampling an integral.

Considering the previous experiments, which stratum should be sample more often?

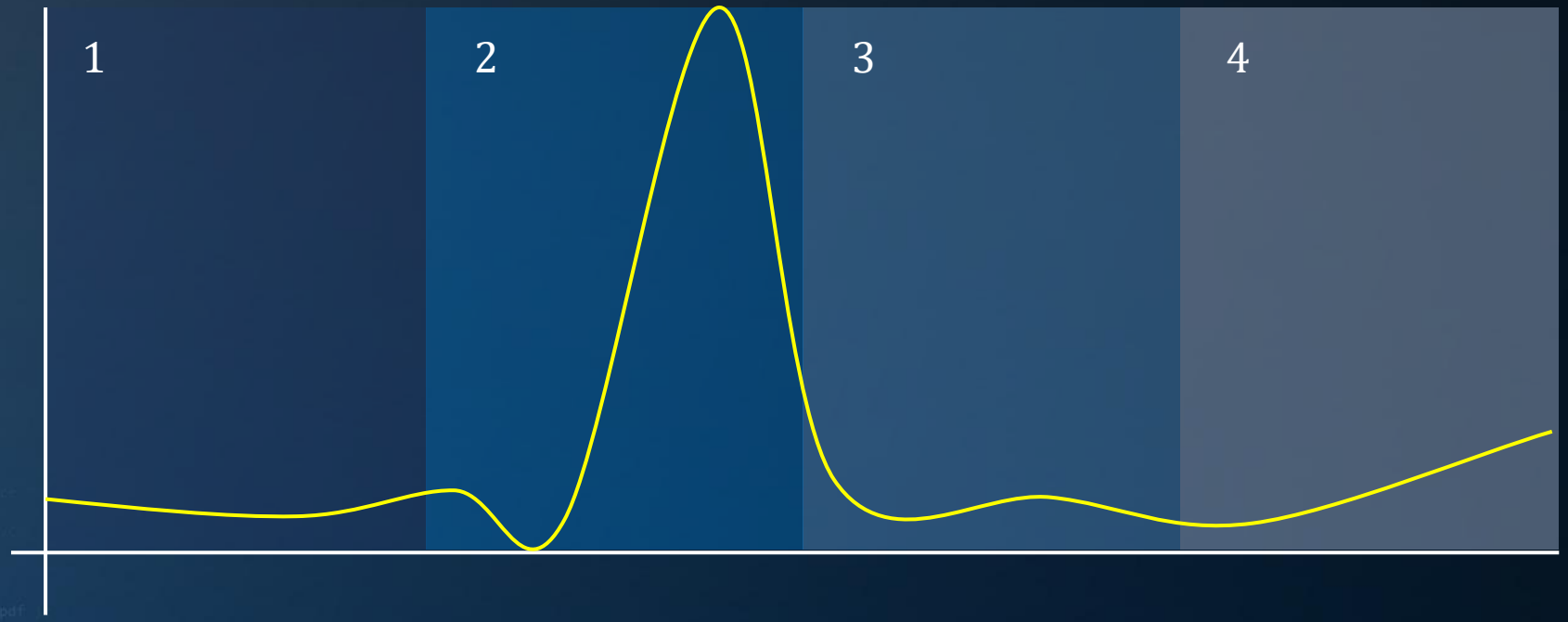


Importance Sampling

Importance Sampling for Monte Carlo

Example 3: sampling an integral.

Considering the previous experiments, which stratum should be sample more often?

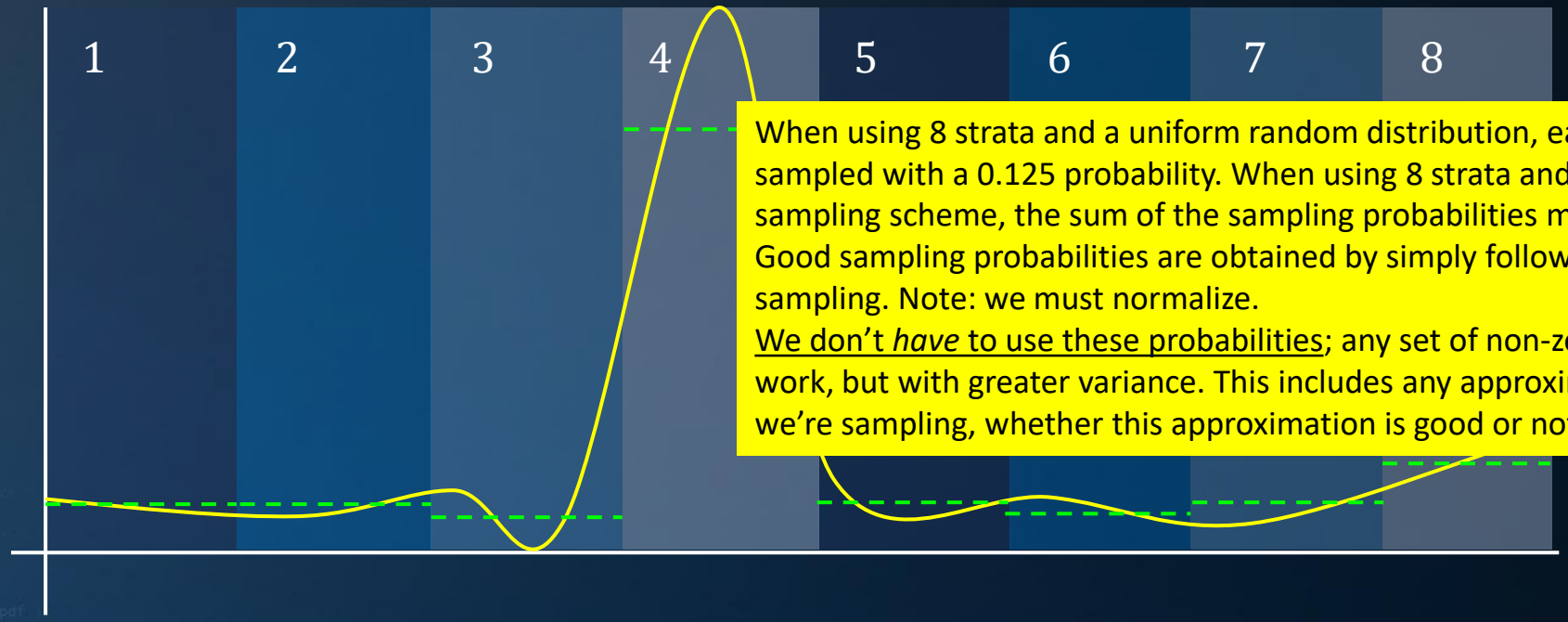


Importance Sampling

Importance Sampling for Monte Carlo

Example 3: sampling an integral.

Considering the previous experiments, which stratum should be sample more often?



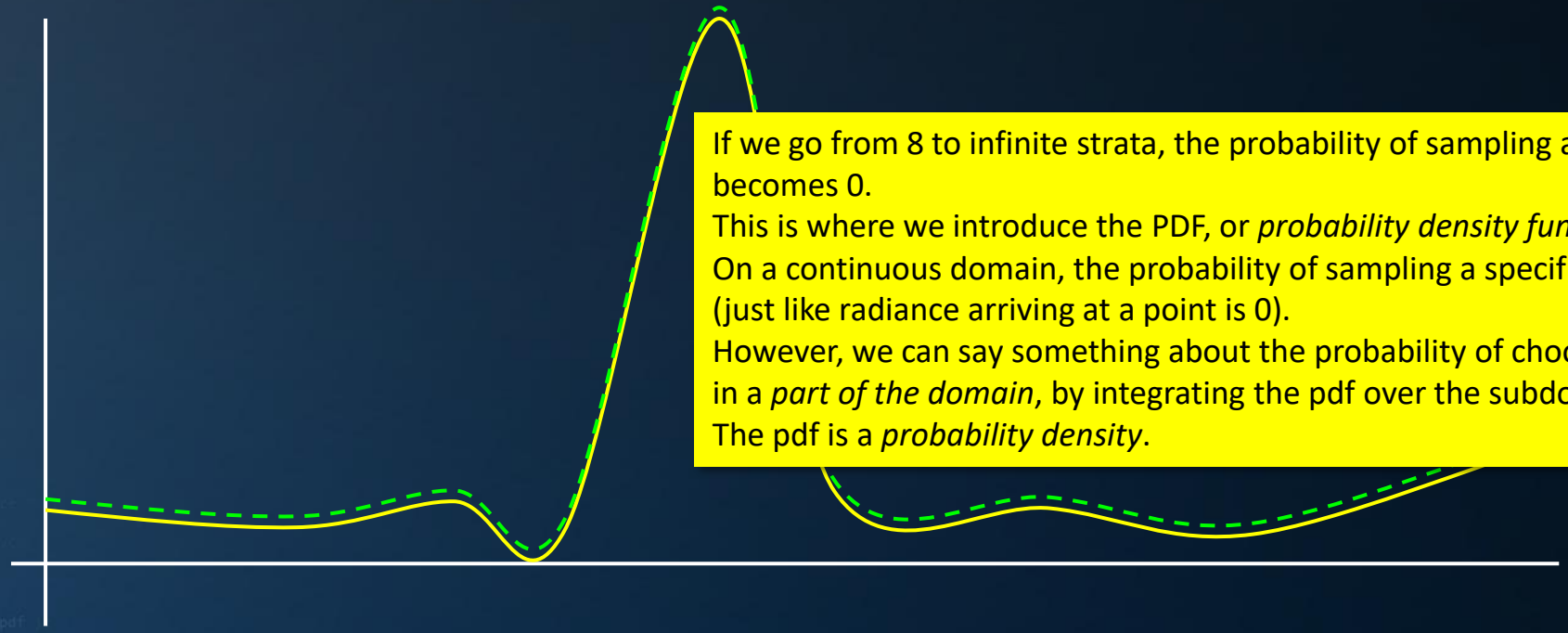
When using 8 strata and a uniform random distribution, each stratum will be sampled with a 0.125 probability. When using 8 strata and a non-uniform sampling scheme, the sum of the sampling probabilities must be 1.
Good sampling probabilities are obtained by simply following the function we're sampling. Note: we must normalize.
We don't *have* to use these probabilities; any set of non-zero probabilities will work, but with greater variance. This includes any approximation of the function we're sampling, whether this approximation is good or not.

Importance Sampling

Importance Sampling for Monte Carlo

Example 3: sampling an integral.

Considering the previous experiments, which stratum should be sample more often?

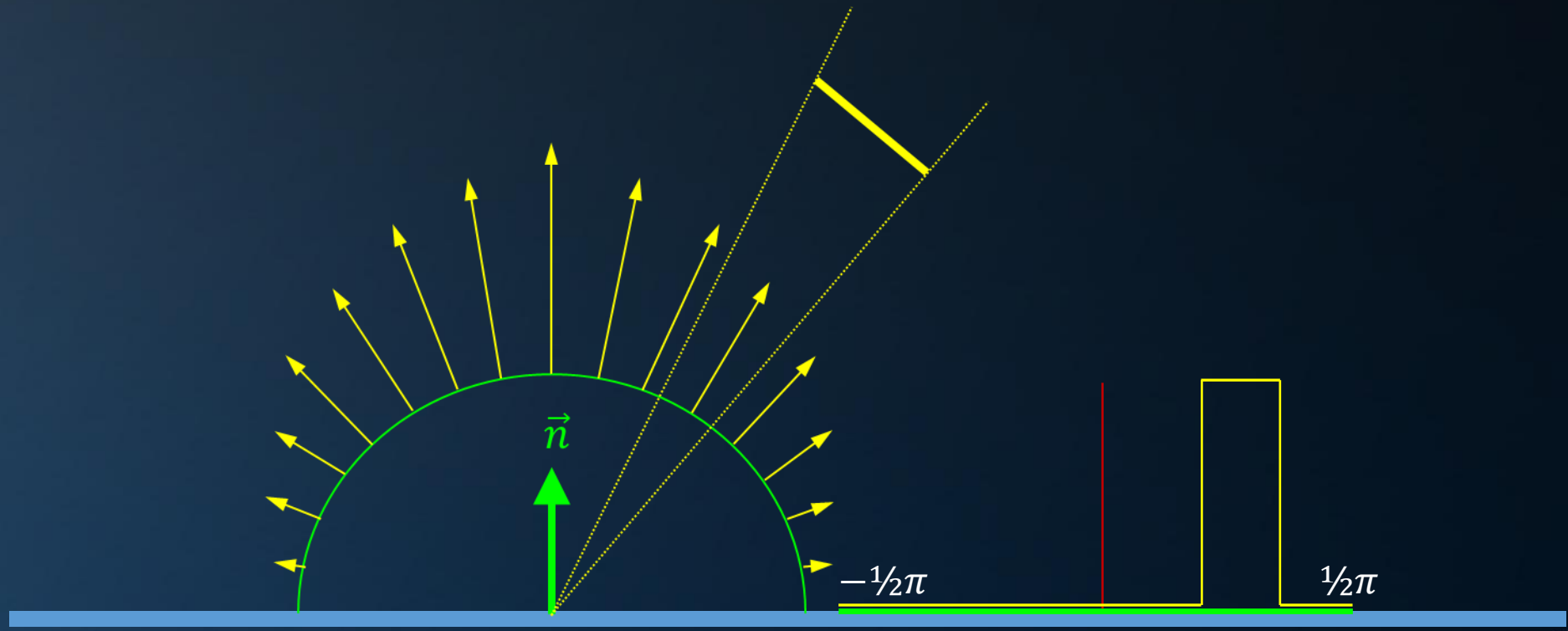


If we go from 8 to infinite strata, the probability of sampling a stratum becomes 0.
This is where we introduce the PDF, or *probability density function*.
On a continuous domain, the probability of sampling a specific X is 0 (just like radiance arriving at a point is 0).
However, we can say something about the probability of choosing X in a *part of the domain*, by integrating the pdf over the subdomain.
The pdf is a *probability density*.

Importance Sampling

Importance Sampling for Monte Carlo

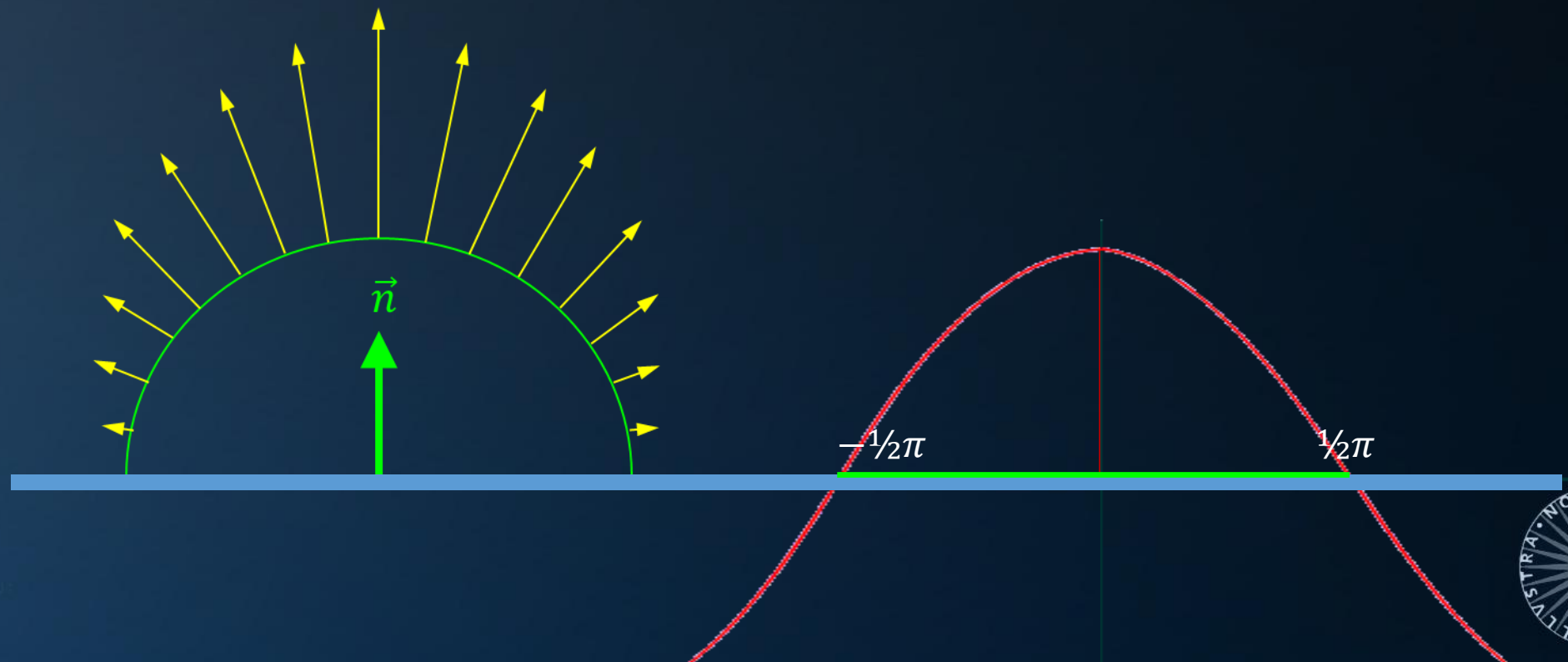
Example 4: sampling the hemisphere.



Importance Sampling

Importance Sampling for Monte Carlo

Example 4: sampling the hemisphere.



Importance Sampling

Importance Sampling for Monte Carlo

Monte Carlo without importance sampling:

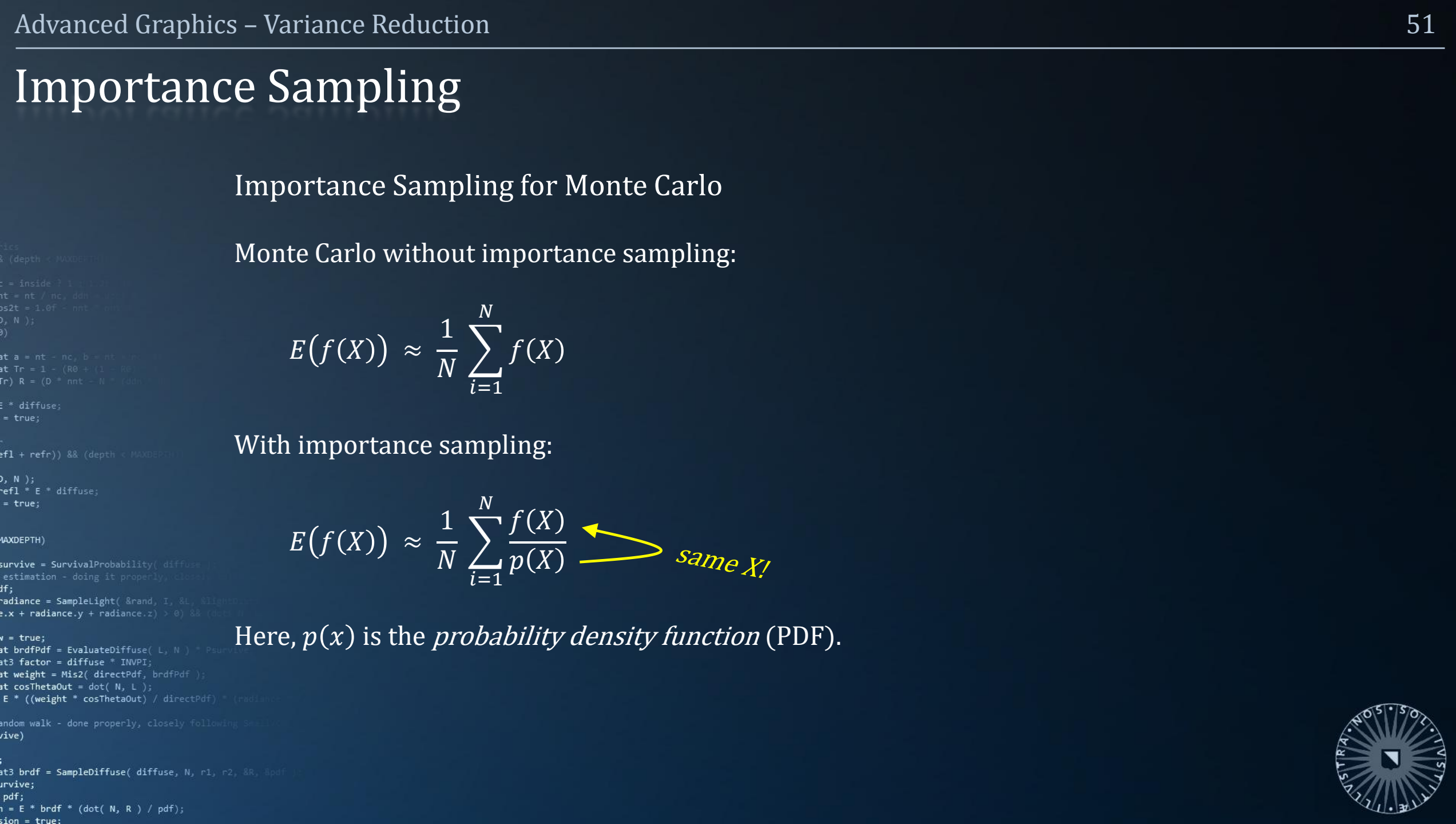
$$E\big(f(X)\big) \approx \frac{1}{N}\sum_{i=1}^{N} f(X)$$

With importance sampling:

$$E\big(f(X)\big) \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(X)}{p(X)}$$

same X!

Here, $p(x)$ is the *probability density function* (PDF).



Importance Sampling

Probability Density Function

Properties of a valid PDF $p(x)$:

1. $p(x) > 0$ for all $x \in D$ where $f(x) \neq 0$
2. $\int_D p(x)d\mu(x) = 1$

Note: $p(x)$ is a density, not a probability; it can (and will) exceed 1 for some x.

Applied to direct light sampling:

$p(x) = C$ for the part of the hemisphere covered by the light source
➔ C = 1 / solid angle to ensure $p(x)$ integrates to 1

➔ Since samples are divided by $p(x)$, we multiply by 1/(1/solid angle)):

$$L_o(p,\omega_i) \approx lights * \frac{1}{N}\sum_{i=1}^{N} f_r(p,\omega_o,P)\, L_d^J(p,P)\, V(p \leftrightarrow P) \frac{A_{L_d^J} \cos\theta_i \cos\theta_o}{\| p - P \|^2}$$

Importance Sampling

Probability Density Function

Applied to hemisphere sampling:

Light arriving over the hemisphere is cosine weighted.
➔ Without further knowledge of the environment, the ideal PDF is the cosine function.

$$PDF\colon\ p(\theta) = \cos\theta$$

Question: how do we normalize this?

$$\int_{\Omega} cos\theta d\theta = \pi \quad \Rightarrow \int_{\Omega} \frac{cos\theta}{\pi} d\theta = 1$$

Question: how do we choose random directions using this PDF?

Importance Sampling

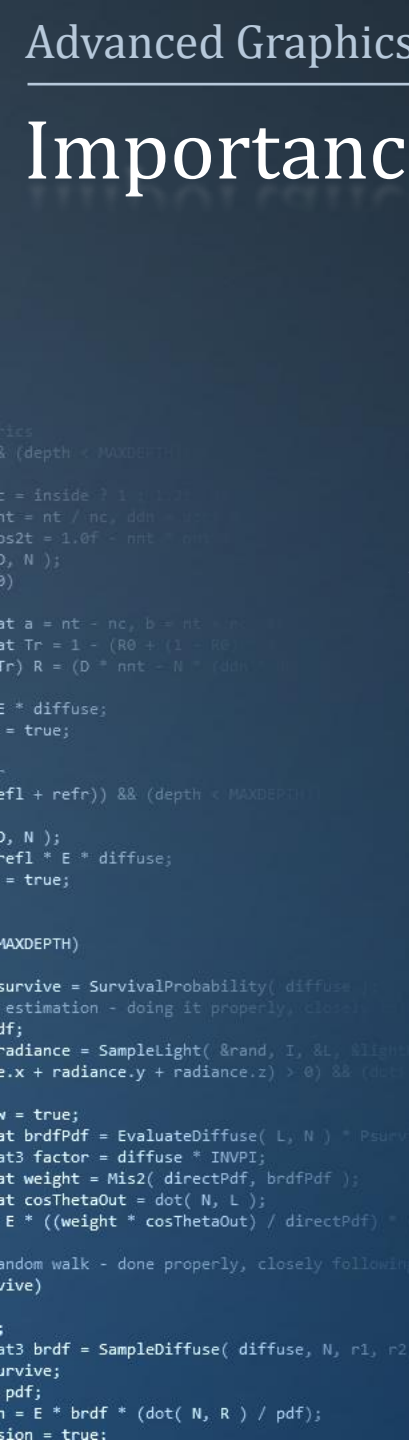
Cosine-weighted Random Direction

Without deriving this in detail:

A cosine-weighted random distribution is obtained by generating points on the unit disc, and projecting the disc on the unit hemisphere. In code:

```
float3 CosineWeightedDiffuseReflection()
{
    float r0 = Rand(), r1 = Rand();
    float r = sqrt( r0 );
    float theta = 2 * PI * r1;
    float x = r * cosf( theta );
    float y = r * sinf( theta );
    return float3( x, y, sqrt( 1 - r0 ) );
}
```

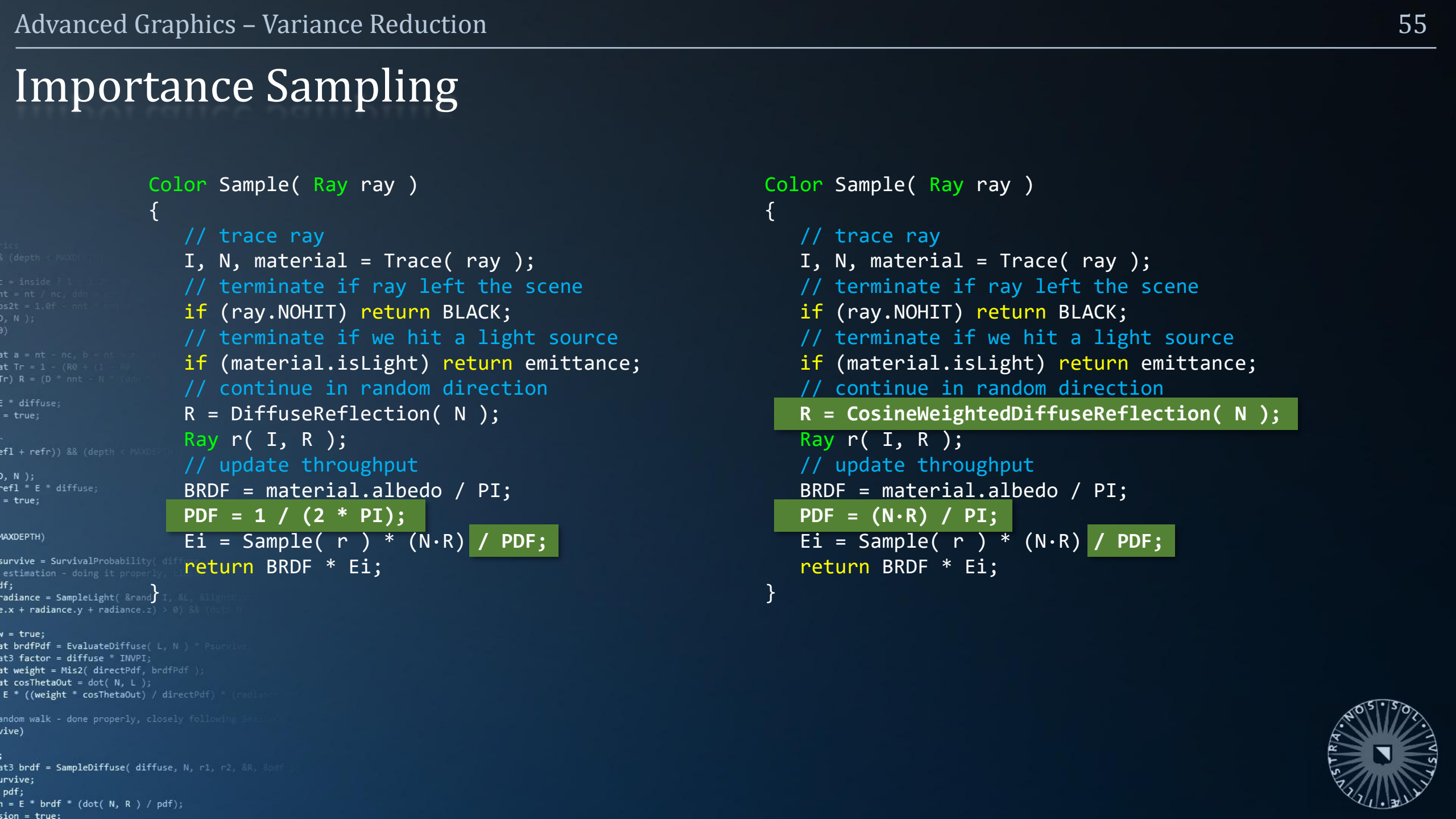
Note: you still have to transform this to tangent space.



Importance Sampling

```
Color Sample( Ray ray )
{
   // trace ray
   I, N, material = Trace( ray );
   // terminate if ray left the scene
   if (ray.NOHIT) return BLACK;
   // terminate if we hit a light source
   if (material.isLight) return emittance;
   // continue in random direction
   R = DiffuseReflection( N );
   Ray r( I, R );
   // update throughput
   BRDF = material.albedo / PI;
   PDF = 1 / (2 * PI);
   Ei = Sample( r ) * (N·R) / PDF;
   return BRDF * Ei;
}
```

```
Color Sample( Ray ray )
{
   // trace ray
   I, N, material = Trace( ray );
   // terminate if ray left the scene
   if (ray.NOHIT) return BLACK;
   // terminate if we hit a light source
   if (material.isLight) return emittance;
   // continue in random direction
   R = CosineWeightedDiffuseReflection( N );
   Ray r( I, R );
   // update throughput
   BRDF = material.albedo / PI;
   PDF = (N·R) / PI;
   Ei = Sample( r ) * (N·R) / PDF;
   return BRDF * Ei;
}
```



Today’s Agenda:

- Introduction
- Random Samples
- Next Event Estimation
- Importance Sampling
- Russian Roulette

RR

```
Color Sample( Ray ray )
{
   // trace ray
   I, N, material = Trace( ray );
   BRDF = material.albedo / PI;
   // terminate if ray left the scene
   if (ray.NOHIT) return BLACK;
   // terminate if we hit a light source
   if (material.isLight) return BLACK;
   // sample a random light source
   L, Nl, dist, A = RandomPointOnLight();
   Ray lr( I, L, dist );
   if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
   {
      solidAngle = ((Nl·-L) * A) / dist²;
      Ld = lightColor * solidAngle * BRDF * N·L;
   }
   // continue random walk
   R = DiffuseReflection( N );
   Ray r( I, R );
   Ei = Sample( r ) * (N·R);
   return PI * 2.0f * BRDF * Ei + Ld;
}
```

```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1) // todo: add 'lastSpecular'
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

RR

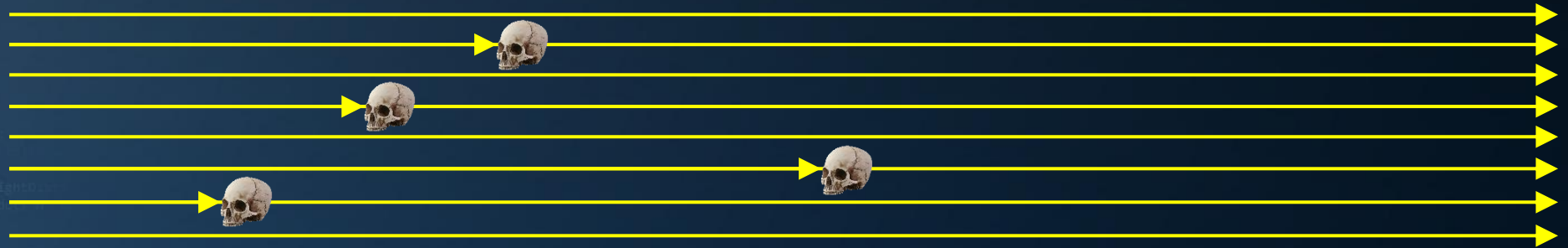
Russian Roulette

Core idea:

The longer a path becomes, the less energy it transports.

Killing half of 16 rays is easy; what do we do with a single path?

➔ Kill it with a probability of 50%.



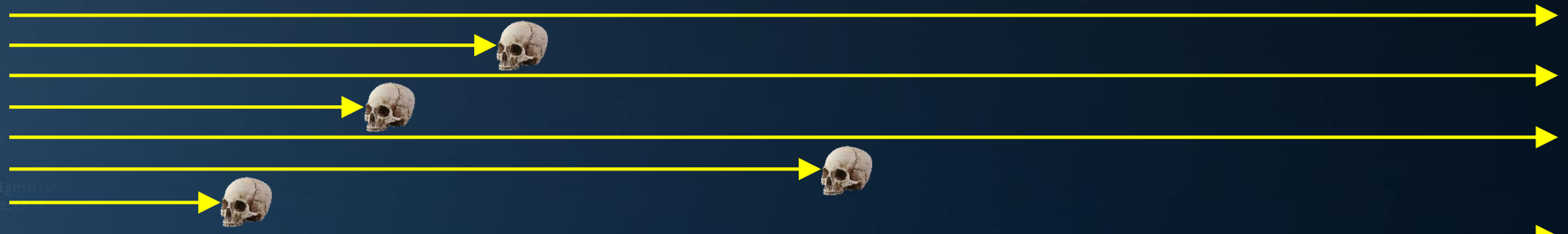
8 rays, returning 16 Watts of radiance each, 128 Watts in total.
= 4 rays, returning 32 Watts of radiance each, 128 Watts in total.

RR

Russian Roulette

Russian roulette is applied to the random walk.

Most basic implementation: just before you start calculating the next random direction, you decide if the path lives or dies.



8 rays, returning 16 Watts of radiance each, 128 Watts in total.
= 4 rays, returning 32 Watts of radiance each, 128 Watts in total.

RR

Better Russian Roulette

The termination probability of 50% is arbitrary.
Any probability is statistically correct.

However: for 50% survival rate, survivors scale up by 2 $\left(= \frac{1}{50\%}\right)$.

➔ In general, for a survival probability ρ, survivors scale up by $\frac{1}{\rho}$.

We can choose the survival probability *per path*. It is typically linked to albedo: the color of the last vertex. A good survival probability is:

$$\rho_{survive} = clamp\left(\frac{red + green + blue}{3}, 0.1, 0.9\right)$$

Note that $\rho_{survive} > 0$ to prevent bias. Also note that $\rho = 1$ is never a good idea.

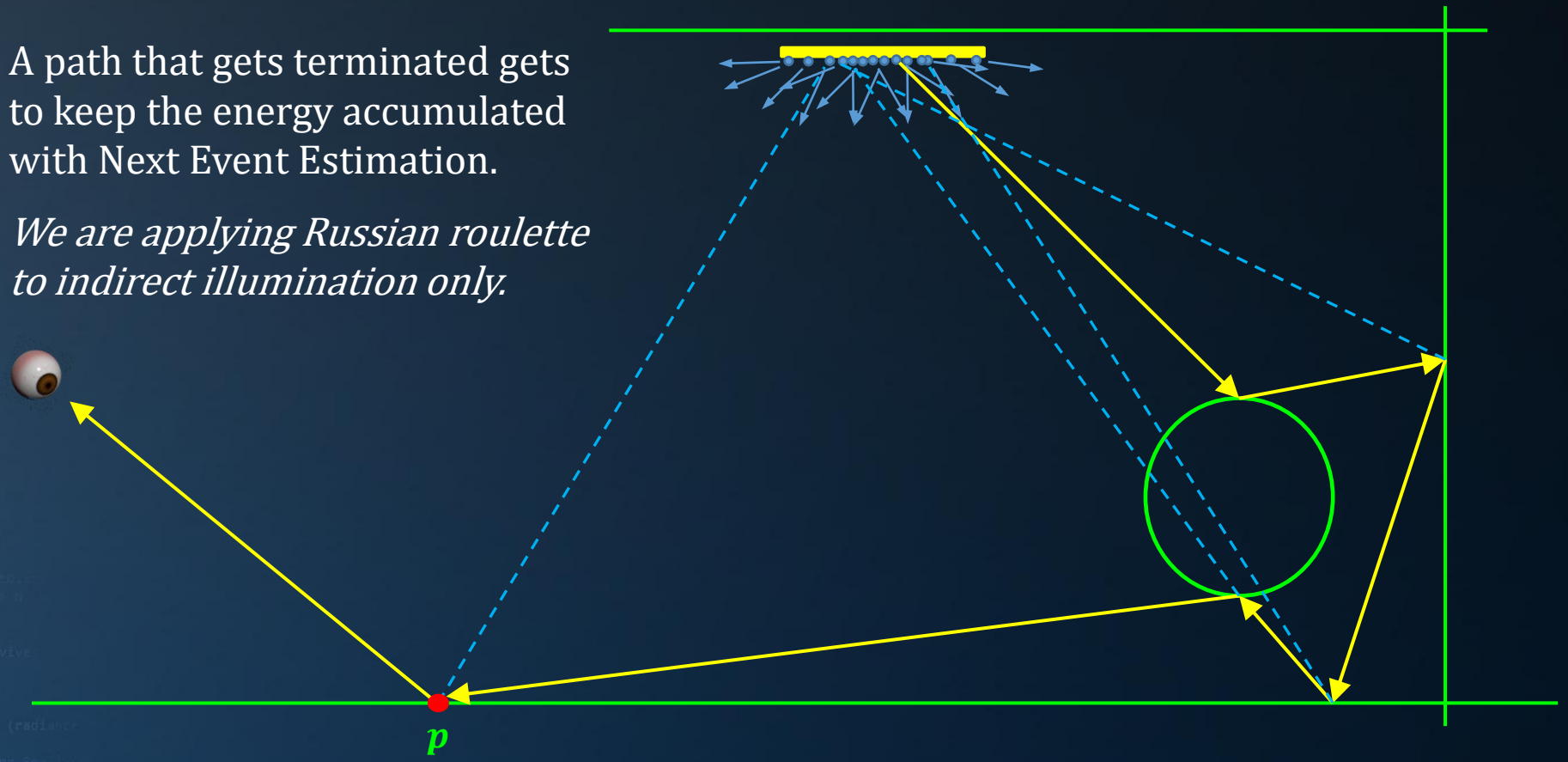
Better:
$\rho_{survive} = clamp(\max(red, green, blue), 0,1)$

RR

RR and Next Event Estimation

A path that gets terminated gets to keep the energy accumulated with Next Event Estimation.

We are applying Russian roulette to indirect illumination only.



RR

```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // Russian Roulette
      p = SurvivalProb( material.albedo );
      if (p < Rand()) break; else /* whew still alive */ T *= 1/p;
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Today’s Agenda:

- Introduction
- Random Samples
- Next Event Estimation
- Importance Sampling
- Russian Roulette

INFOMAGR – Advanced Graphics

Jacco Bikker - November 2021 - February 2022

END of “Variance Reduction (2)”

next lecture: “Various”