Path tracing, part 1 of 2: Introduction

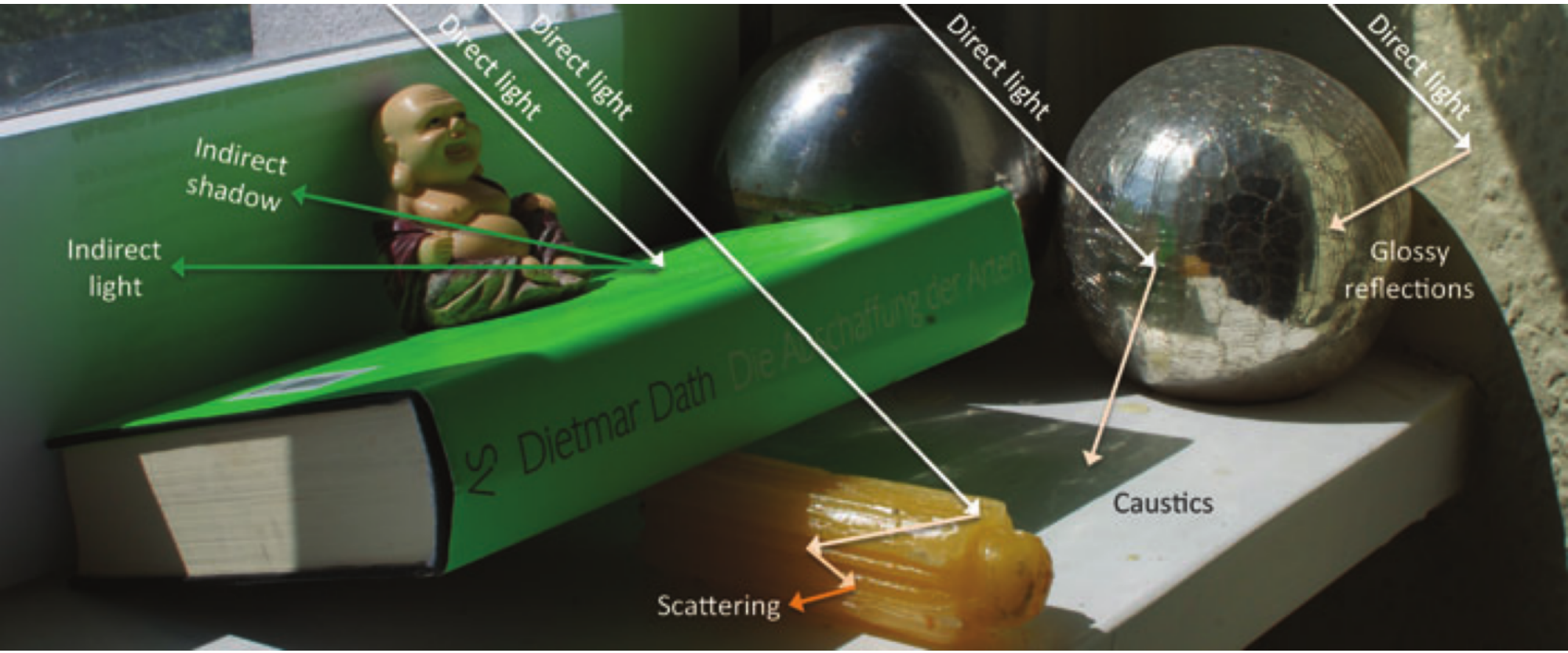
Christoph Peters
Delft University of Technology (TU Delft)
Computer Graphics and Visualization Group

Goal: Full global illumination

Rasterization struggles with many light-transport phenomena.

Path tracing handles them all accurately.

But it takes effort to make it efficient.



doi.org/10.1111/j.1467-8659.2012.02093.x

Light transport

Light transport

Light transport

Light transport

Light transport

Hardware-accelerated ray tracing

NVIDIA introduced GPUs with ray tracing cores in 2018.

Up to 10 billion rays per second on an RTX 2080 Ti.

Hardware-accelerated ray tracing

NVIDIA introduced GPUs with ray tracing cores in 2018.

Up to 10 billion rays per second on an RTX 2080 Ti.

Available in all NVIDIA RTX GPUs, AMD RDNA 2 and newer, Playstation 5, Xbox Series X/S, Intel Arc, Apple M3/A17 Pro.

AMD GPUs (and thus consoles) are slower than others for now.

Hardware-accelerated ray tracing

NVIDIA introduced GPUs with ray tracing cores in 2018.

Up to 10 billion rays per second on an RTX 2080 Ti.

Available in all NVIDIA RTX GPUs, AMD RDNA 2 and newer,
Playstation 5, Xbox Series X/S, Intel Arc, Apple M3/A17 Pro.

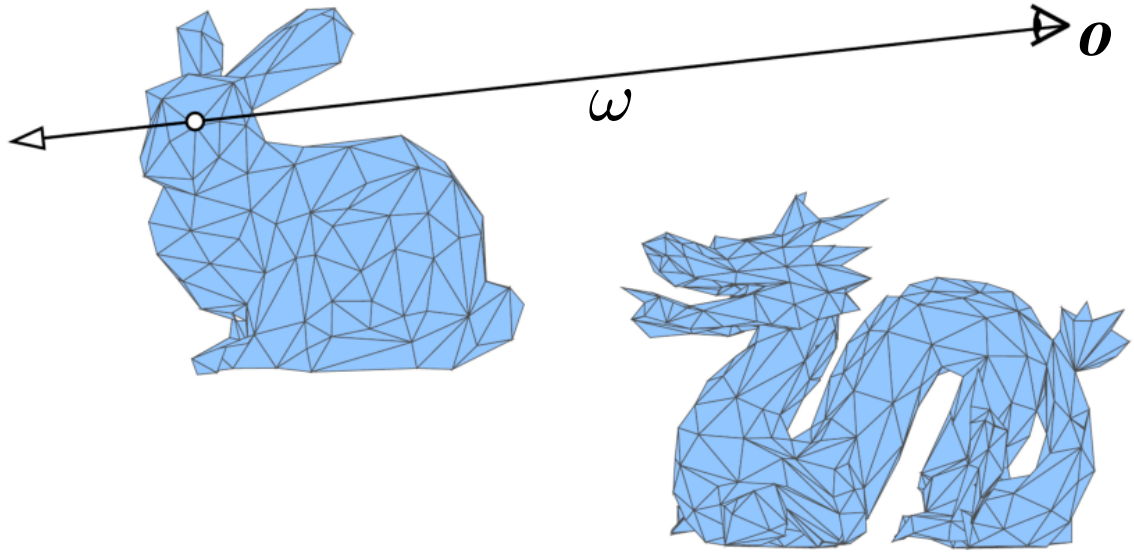
AMD GPUs (and thus consoles) are slower than others for now.

Available through Vulkan, Direct 3D 12, Metal,
NVIDIA OptiX, AMD Radeon Rays, Intel Embree.

Not through OpenGL, Direct3D 11, WebGL.

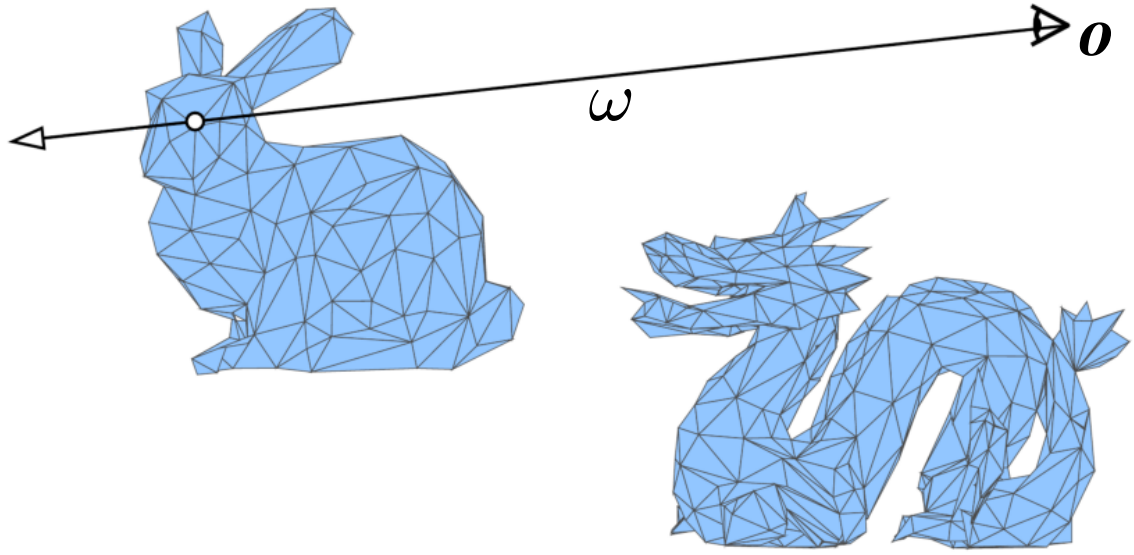
Ray tracing APIs

Goal: For a ray with origin $\boldsymbol{o} \in \mathbb{R}^3$ and direction $\omega \in \mathbb{R}^3$ find the closest scene triangle that it intersects (if any).



Ray tracing APIs

Goal: For a ray with origin $\boldsymbol{o} \in \mathbb{R}^3$ and direction $\omega \in \mathbb{R}^3$ find the closest scene triangle that it intersects (if any).



Use API to build acceleration structure for all scene triangles.

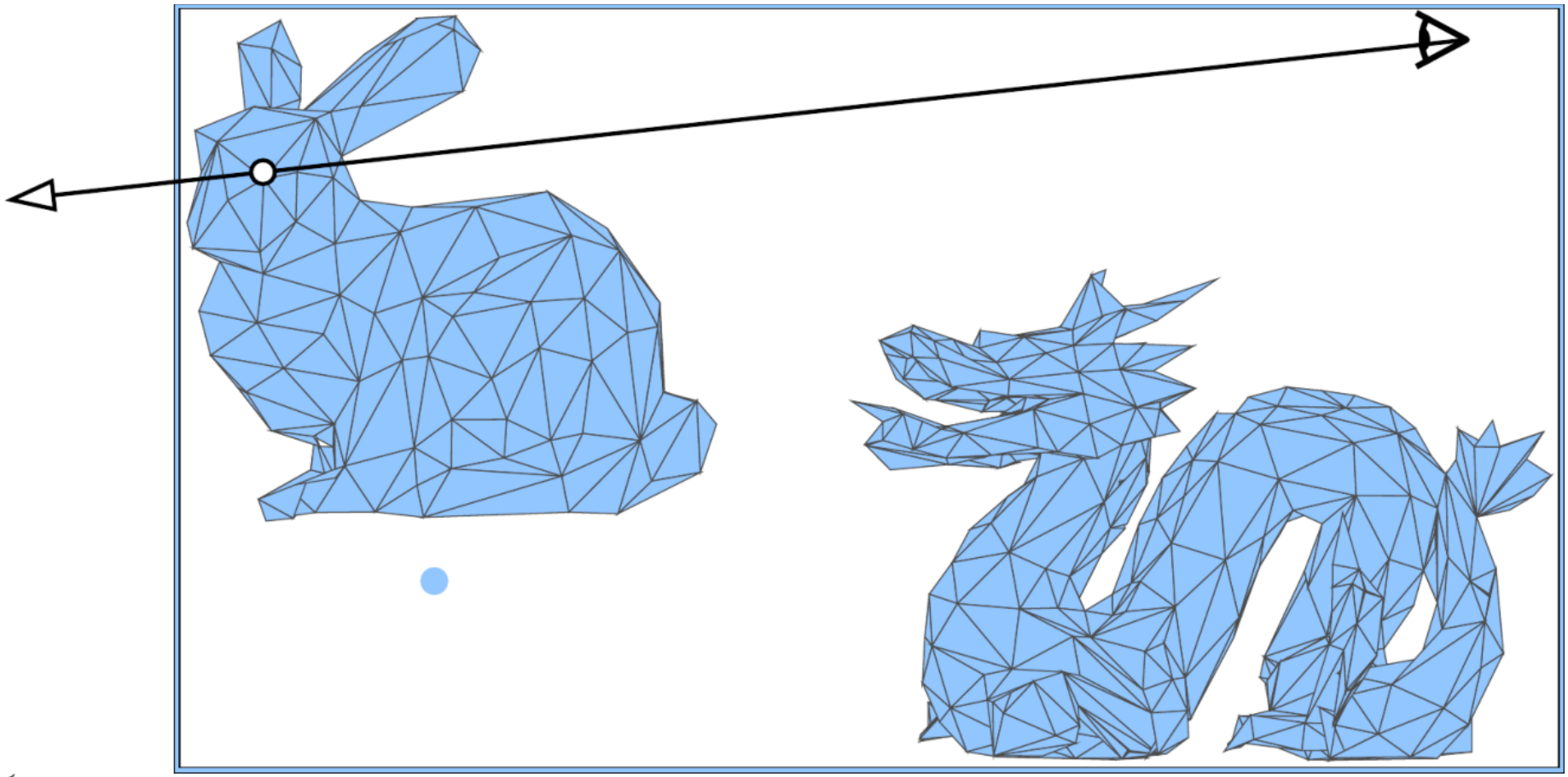
In a shader, pass $\boldsymbol{o}$, ω and acceleration structure to the API.

Then retrieve info about the hit triangle (if any).

Bounding volume hierarchy (BVH)

BVH = tree of bounding boxes with triangles in leafs.

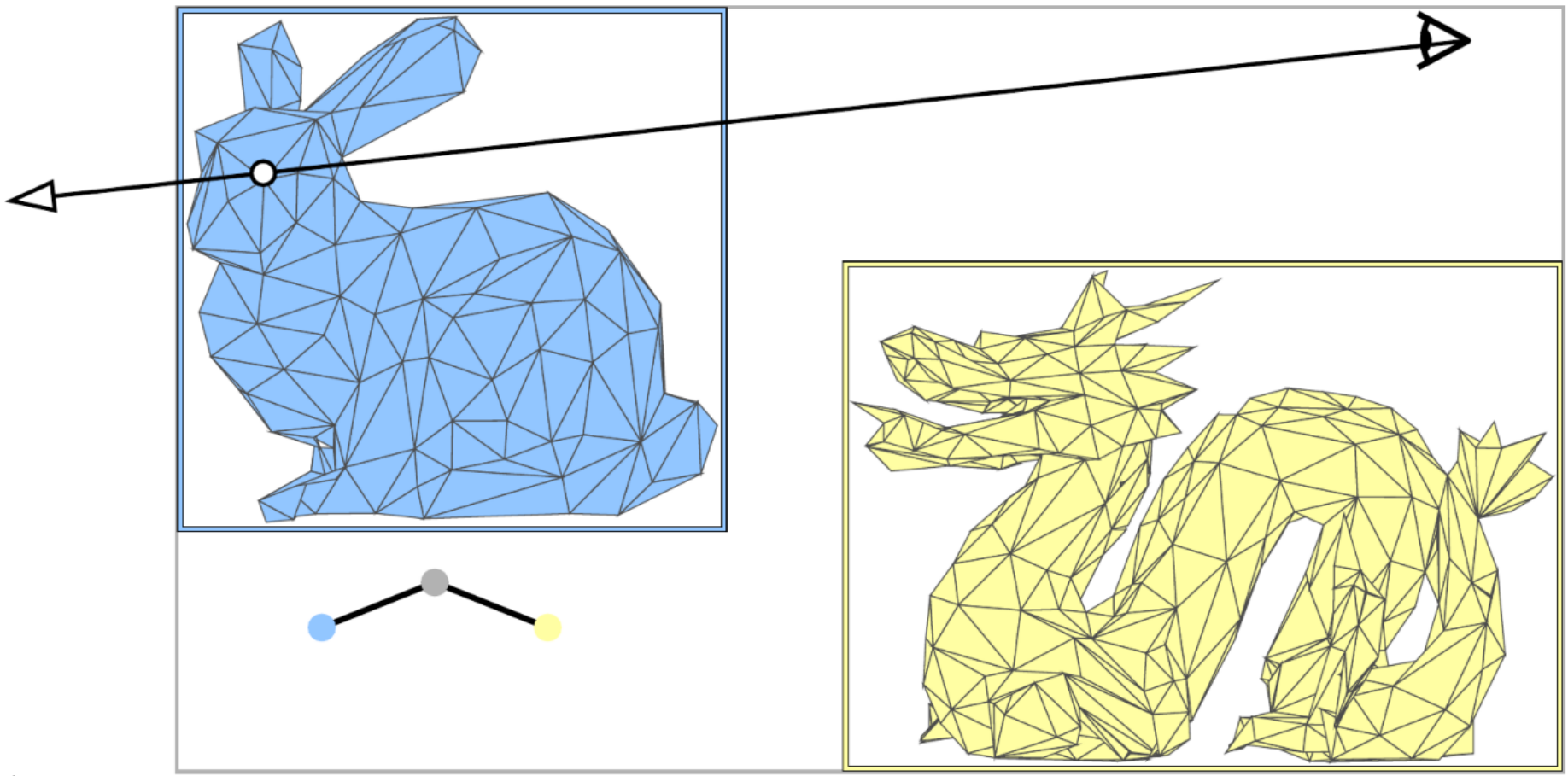
If a ray does not hit a box, ray tracing ignores its subtree.



Bounding volume hierarchy (BVH)

BVH = tree of bounding boxes with triangles in leafs.

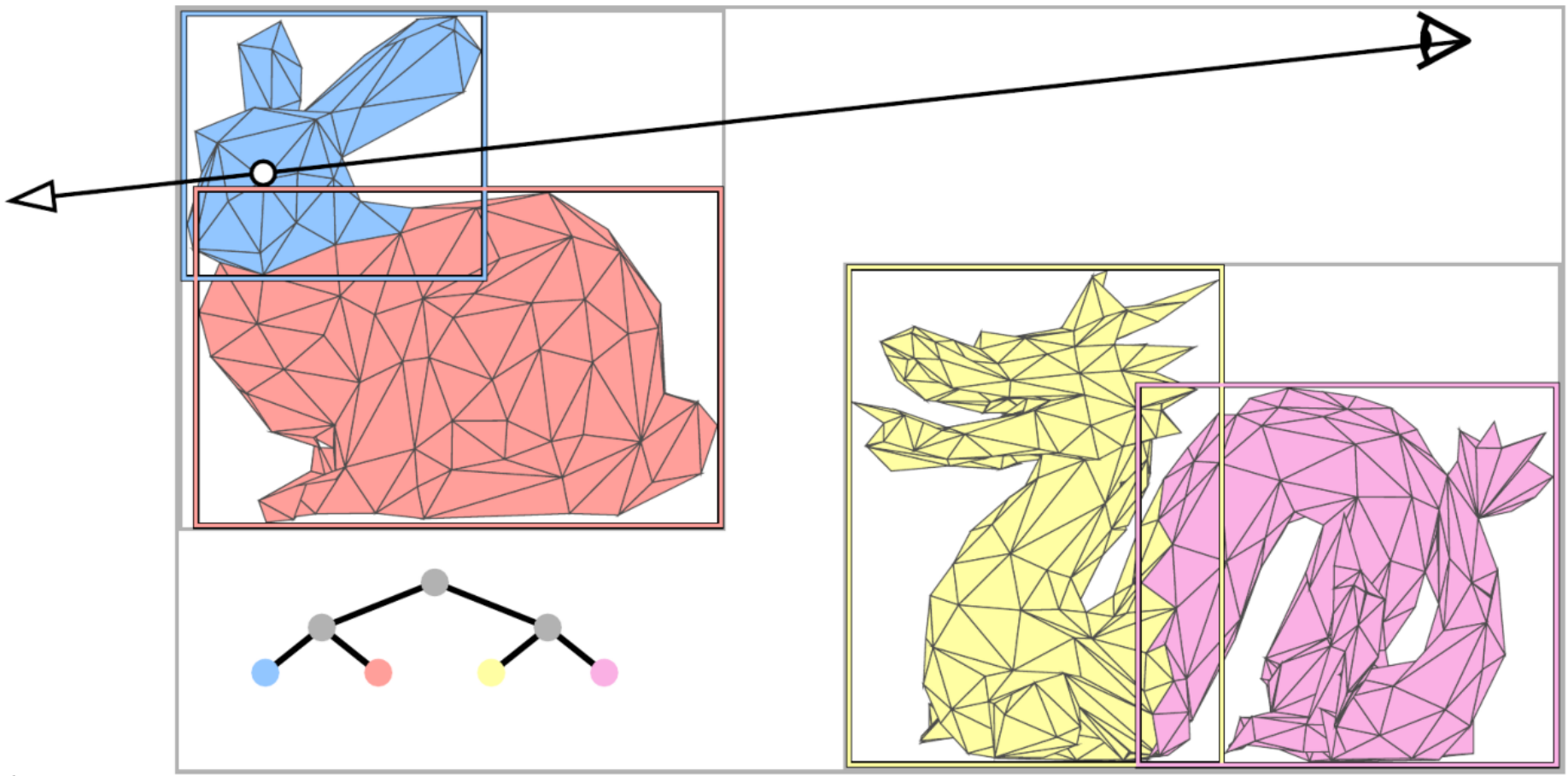
If a ray does not hit a box, ray tracing ignores its subtree.



Bounding volume hierarchy (BVH)

BVH = tree of bounding boxes with triangles in leafs.

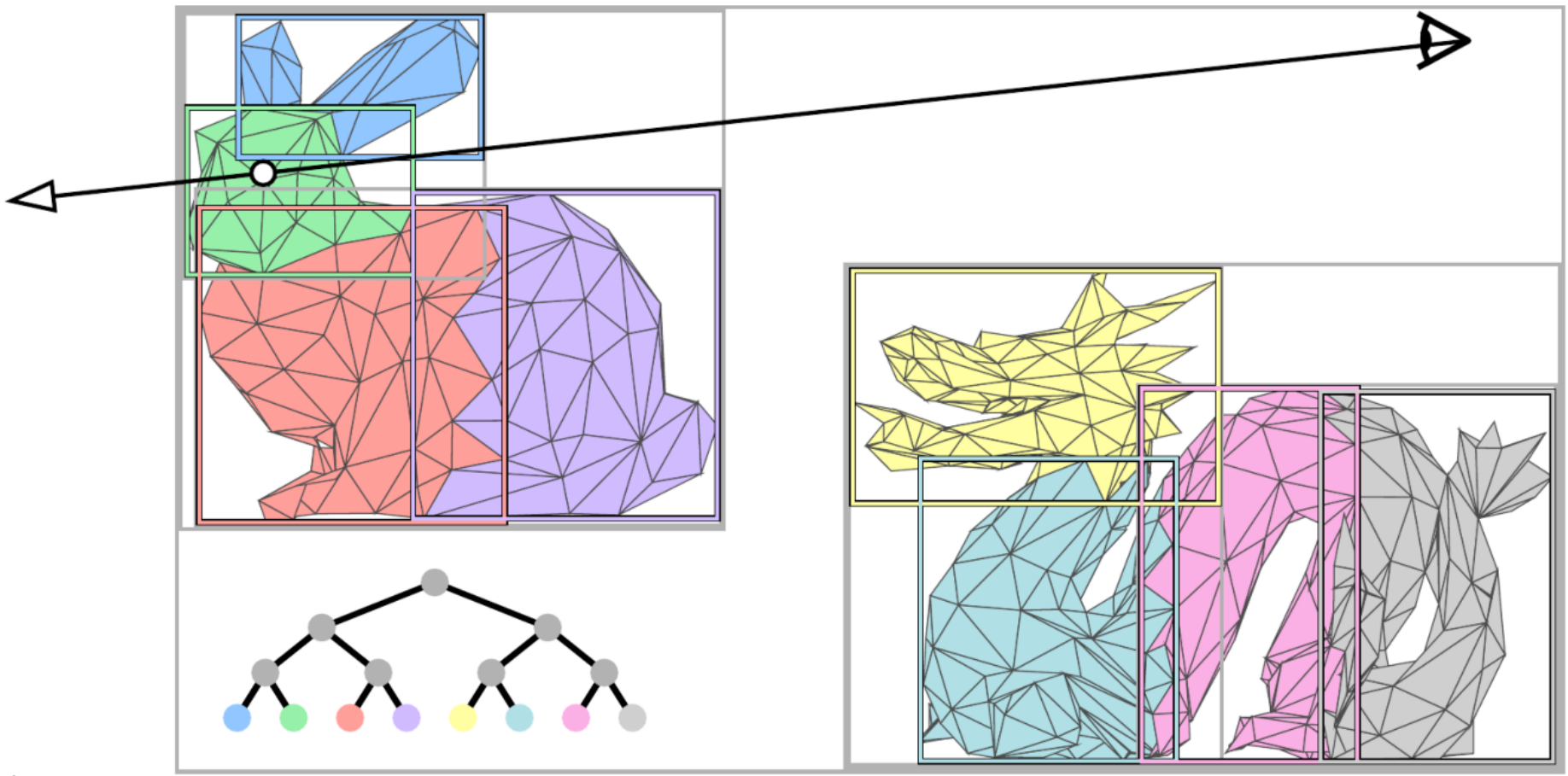
If a ray does not hit a box, ray tracing ignores its subtree.



Bounding volume hierarchy (BVH)

BVH = tree of bounding boxes with triangles in leafs.

If a ray does not hit a box, ray tracing ignores its subtree.



Shading data

For an intersection at $\boldsymbol{x}$, the API provides the triangle index and barycentric coordinates $w_0, w_1, w_2 \in [0, 1]$.

Load the triangle vertex positions $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2$.

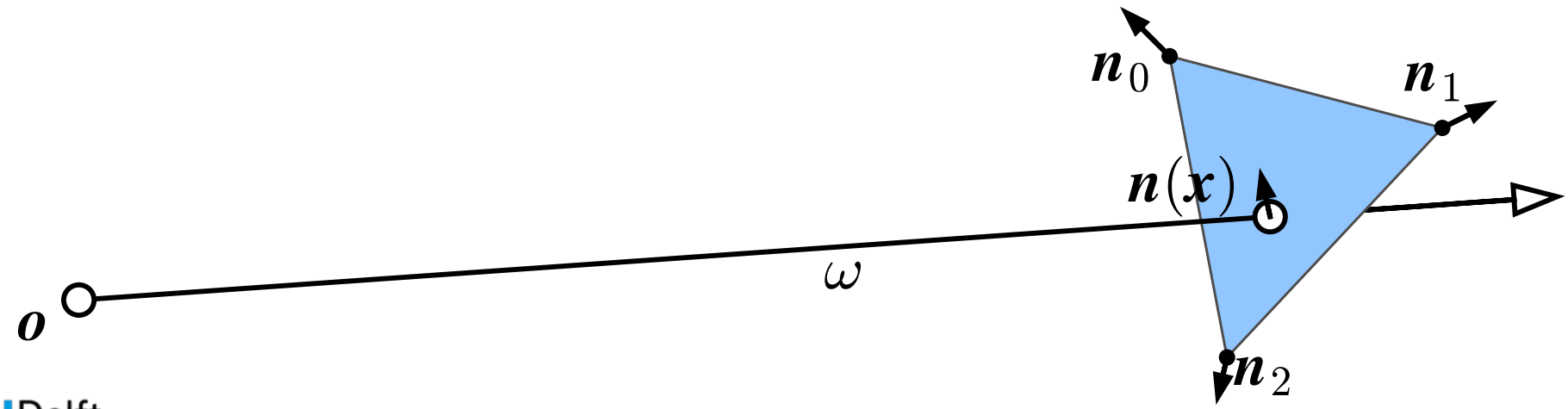
Interpolate: $\boldsymbol{x} = w_0 \boldsymbol{x}_0 + w_1 \boldsymbol{x}_1 + w_2 \boldsymbol{x}_2$

Shading data

For an intersection at $\boldsymbol{x}$, the API provides the triangle index and barycentric coordinates $w_0, w_1, w_2 \in [0, 1]$.

Load the triangle vertex normals $\boldsymbol{n}_0, \boldsymbol{n}_1, \boldsymbol{n}_2$.

Interpolate: $\boldsymbol{n}(\boldsymbol{x}) = \dfrac{w_0 \boldsymbol{n}_0 + w_1 \boldsymbol{n}_1 + w_2 \boldsymbol{n}_2}{\| w_0 \boldsymbol{n}_0 + w_1 \boldsymbol{n}_1 + w_2 \boldsymbol{n}_2 \|}$



Shading data

For an intersection at $\boldsymbol{x}$, the API provides the triangle index and barycentric coordinates $w_0, w_1, w_2 \in [0, 1]$.

Load the triangle vertex texture coordinates $\boldsymbol{t}_0, \boldsymbol{t}_1, \boldsymbol{t}_2$.

Interpolate: $\boldsymbol{t}(\boldsymbol{x}) = w_0\,\boldsymbol{t}_0 + w_1\,\boldsymbol{t}_1 + w_2\,\boldsymbol{t}_2$

Shading data

For an intersection at $\boldsymbol{x}$, the API provides the triangle index and barycentric coordinates $w_0, w_1, w_2 \in [0, 1]$.

Load the triangle vertex texture coordinates $\boldsymbol{t}_0, \boldsymbol{t}_1, \boldsymbol{t}_2$.

Interpolate: $\boldsymbol{t}(\boldsymbol{x}) = w_0\,\boldsymbol{t}_0 + w_1\,\boldsymbol{t}_1 + w_2\,\boldsymbol{t}_2$

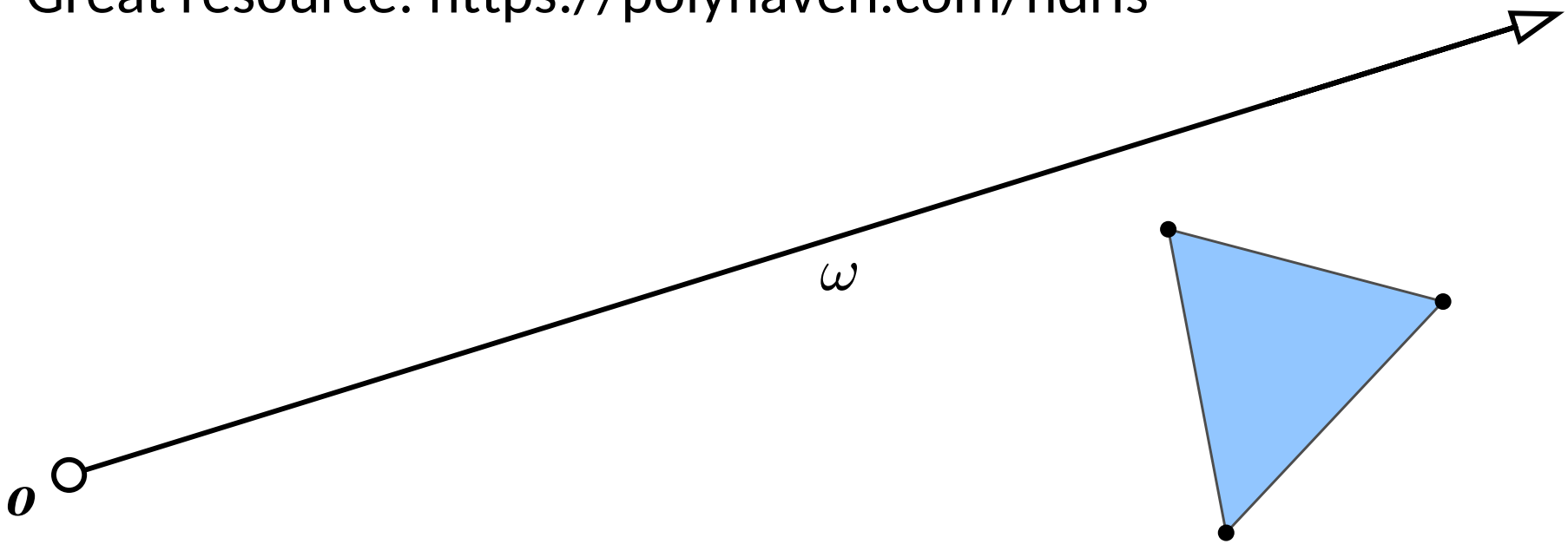
Sample textures at $\boldsymbol{t}(\boldsymbol{x})$, prepare BRDF parameters and determine the emitted radiance $L_e(\boldsymbol{x}, -\omega)$.

Shading data

If the ray misses all scene triangles,
we only compute an emitted radiance $L_e(-\omega)$.

E.g. a constant sky color,
or we sample a background texture (a.k.a. light probe/HDRI).

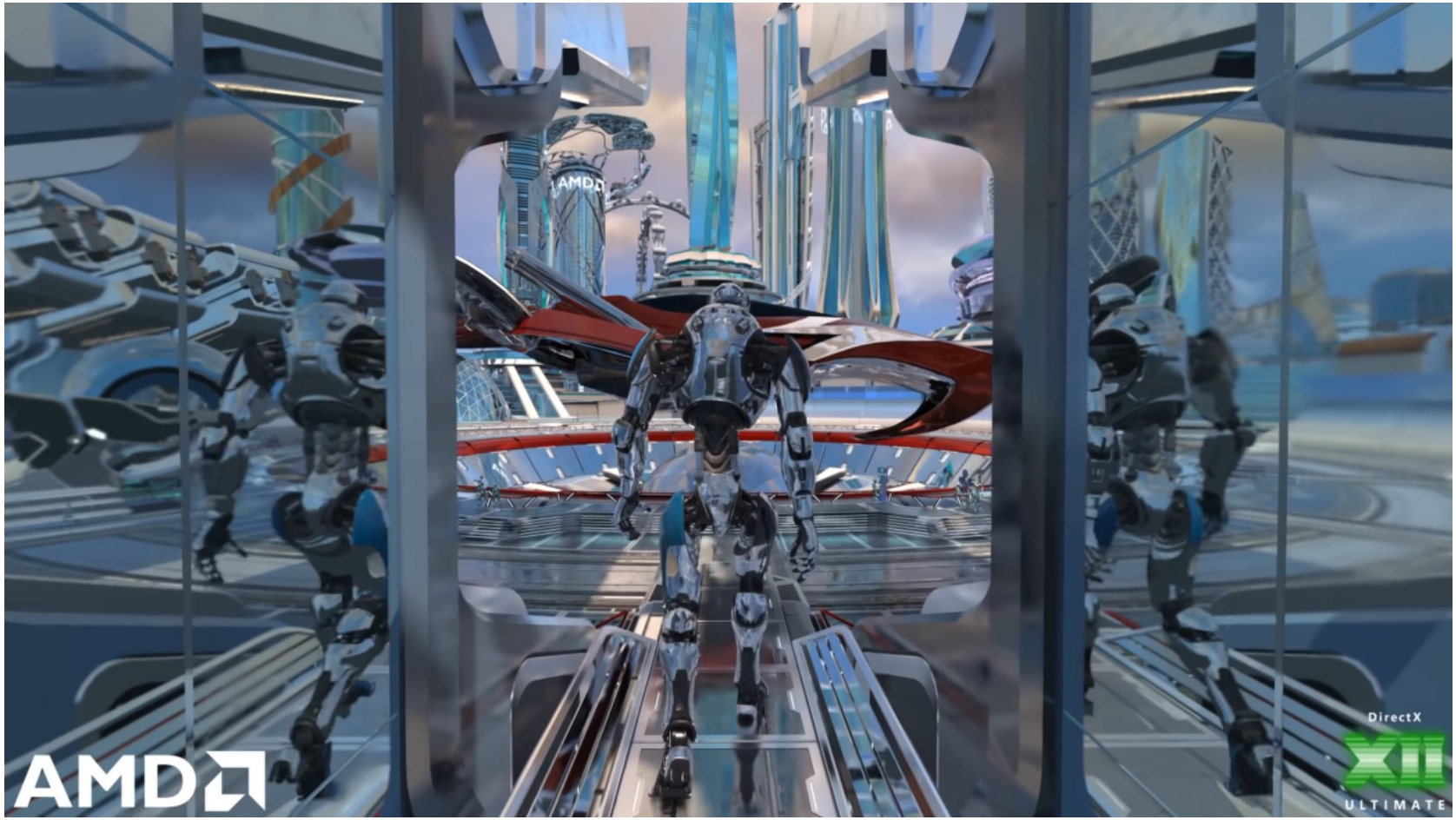
Great resource: https://polyhaven.com/hdris



Ray tracing effects

Gamedevs tend to embrace ray tracing for familiar effects:

Glossy reflections, soft shadows, ambient occlusion.



community.amd.com/t5/gaming/powering-next-generation-gaming-visuals-with-amd-rdna-2-and/ba-p/414239

Ray tracing effects

Gamedevs tend to embrace ray tracing for familiar effects:

Glossy reflections, soft shadows, ambient occlusion.

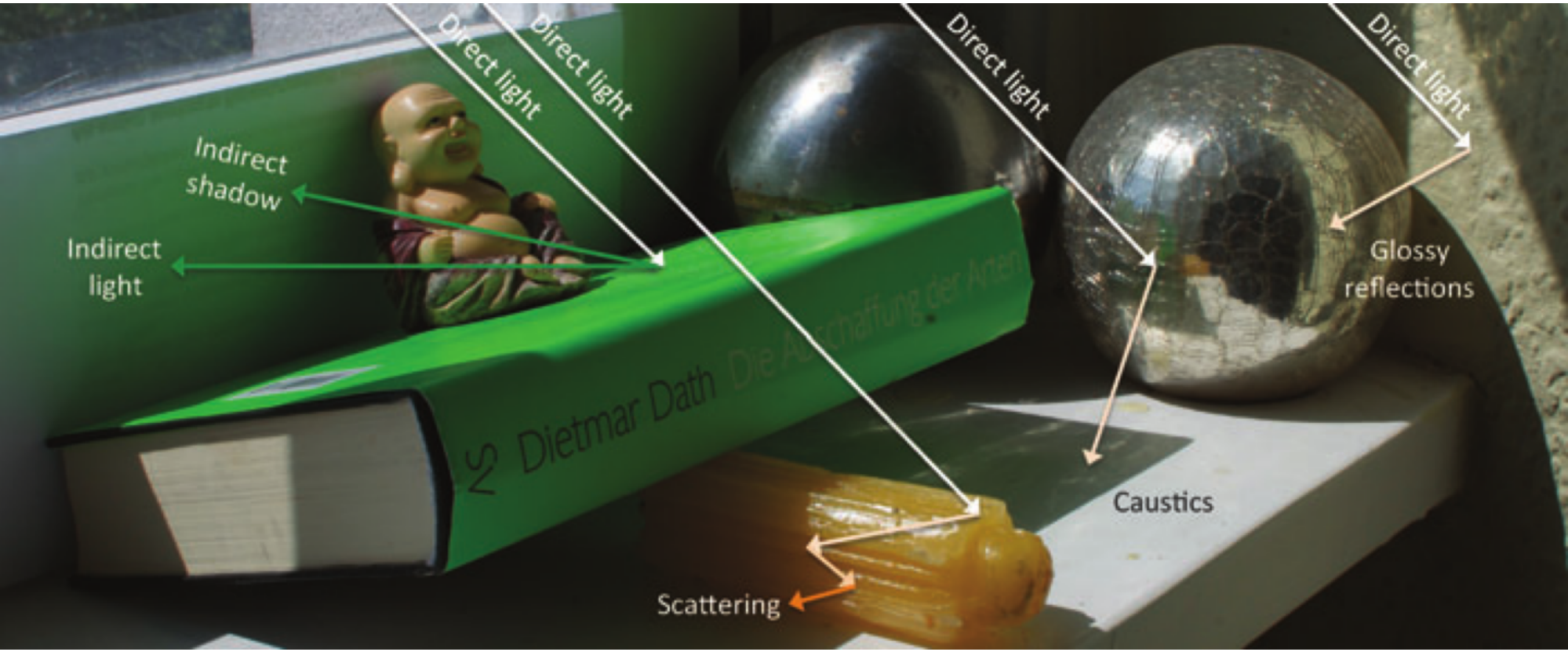
community.amd.com/t5/gaming/powering-next-generation-gaming-visuals-with-amd-rdna-2-and/ba-p/414239

Ray tracing effects

Gamedevs tend to embrace ray tracing for familiar effects:

Glossy reflections, soft shadows, ambient occlusion.

But we aim for full global illumination with path tracing.




TUDelft

The rendering equation

The rendering equation

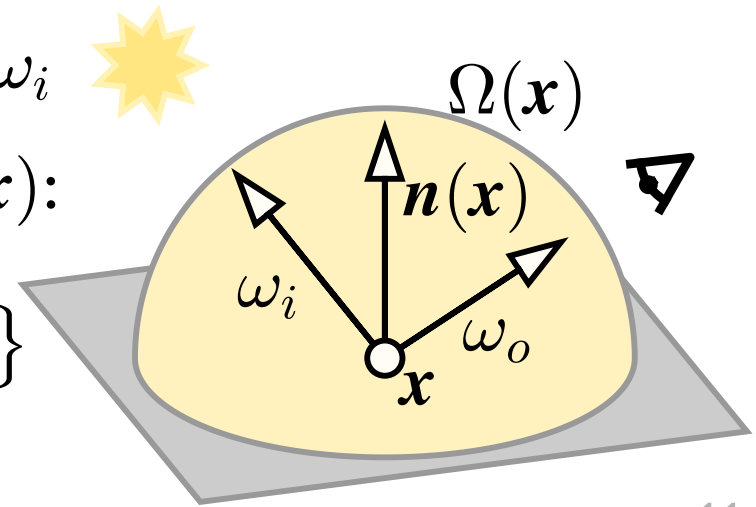
$$L_o(\boldsymbol{x}, \omega_o) = L_e(\boldsymbol{x}, \omega_o) + \int_{\Omega(\boldsymbol{x})} L(\boldsymbol{x}, \omega_i)\, \rho(\boldsymbol{x}, \omega_i, \omega_o)\, \boldsymbol{n}(\boldsymbol{x}) \cdot \omega_i \, \mathrm{d}\omega_i$$

Defines outgoing radiance $L_o(\boldsymbol{x}, \omega_o)$, i.e. color of surface point $\boldsymbol{x} \in \mathbb{R}^3$ seen from direction $\omega_o \in \Omega(\boldsymbol{x})$.

Emitted radiance $L_e(\boldsymbol{x}, \omega_o)$ is non-zero on lights.

Integral over incoming light directions ω_i in hemisphere $\Omega(\boldsymbol{x})$ around normal $\boldsymbol{n}(\boldsymbol{x})$:

$$\Omega(\boldsymbol{x}) := \left\{ \omega \in \mathbb{R}^3 \mid \|\omega\| = 1, \boldsymbol{n}(\boldsymbol{x}) \cdot \omega \geq 0 \right\}$$

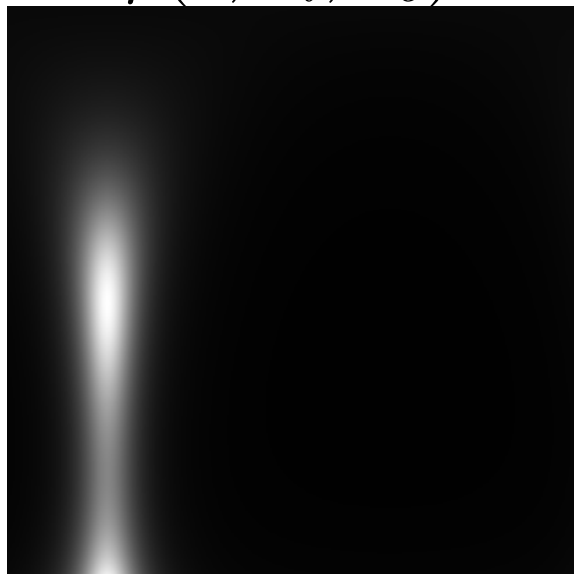


The rendering equation: Integrand

$$L_o(\boldsymbol{x}, \omega_o) = L_e(\boldsymbol{x}, \omega_o) + \int_{\Omega(\boldsymbol{x})} L(\boldsymbol{x}, \omega_i)\, \rho(\boldsymbol{x}, \omega_i, \omega_o)\, \boldsymbol{n}(\boldsymbol{x}) \cdot \omega_i \, \mathrm{d}\omega_i$$

Incoming radiance
$L(\boldsymbol{x}, \omega_i)$

BRDF
$\rho(\boldsymbol{x}, \omega_i, \omega_o)$



Geometry term
$\boldsymbol{n}(\boldsymbol{x}) \cdot \omega_i$

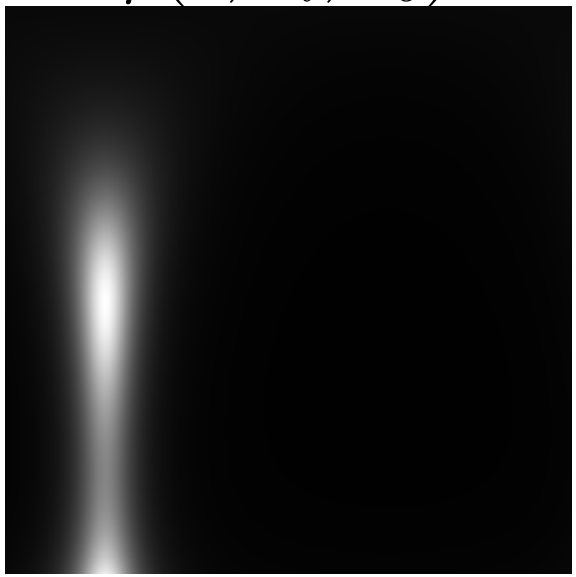
The rendering equation: Challenges

$$L_o(\boldsymbol{x}, \omega_o) = L_e(\boldsymbol{x}, \omega_o) + \int_{\Omega(\boldsymbol{x})} L(\boldsymbol{x}, \omega_i)\, \rho(\boldsymbol{x}, \omega_i, \omega_o)\, \boldsymbol{n}(\boldsymbol{x}) \cdot \omega_i \, \mathrm{d}\omega_i$$

How do we compute incoming radiance?

Incoming radiance
$L(\boldsymbol{x}, \omega_i)$

BRDF
$\rho(\boldsymbol{x}, \omega_i, \omega_o)$



Geometry term
$\boldsymbol{n}(\boldsymbol{x}) \cdot \omega_i$

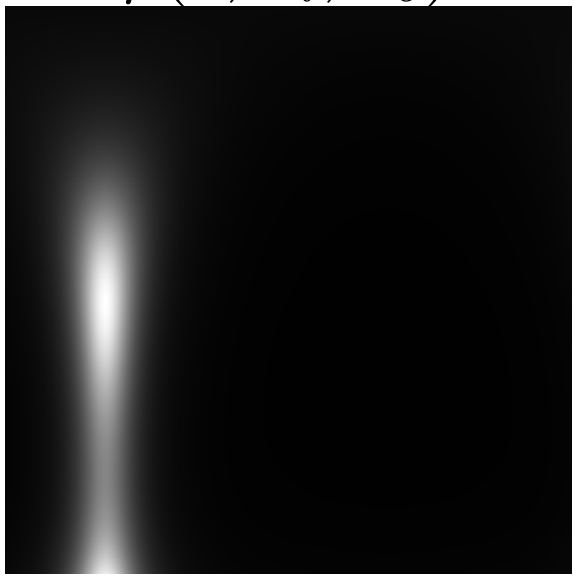
The rendering equation: Challenges

$$L_o(\boldsymbol{x}, \omega_o) = L_e(\boldsymbol{x}, \omega_o) + \int_{\Omega(\boldsymbol{x})} L(\boldsymbol{x}, \omega_i)\, \rho(\boldsymbol{x}, \omega_i, \omega_o)\, \boldsymbol{n}(\boldsymbol{x}) \cdot \omega_i \, \mathrm{d}\omega_i$$

How do we integrate over the hemisphere $\Omega(\boldsymbol{x})$?

Incoming radiance
$L(\boldsymbol{x}, \omega_i)$

BRDF
$\rho(\boldsymbol{x}, \omega_i, \omega_o)$



Geometry term
$\boldsymbol{n}(\boldsymbol{x}) \cdot \omega_i$

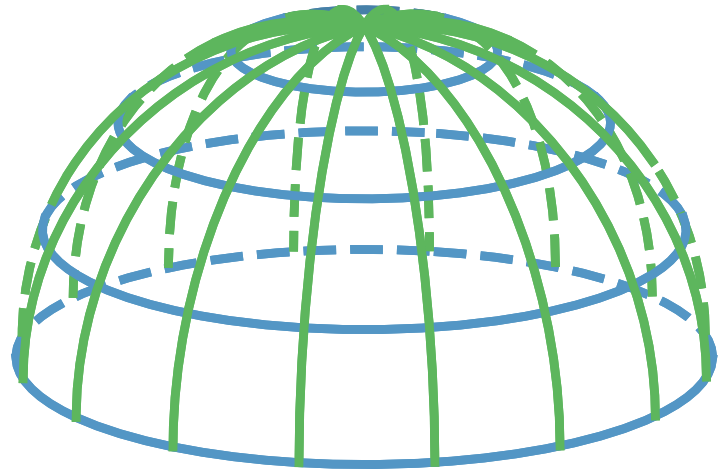
Spherical coordinates

Inclination $\theta \in [0, \pi]$, azimuth $\varphi \in [-\pi, \pi)$.

$$\omega_x(\theta, \varphi) := \cos(\varphi)\sin(\theta)$$

$$\omega_y(\theta, \varphi) := \sin(\varphi)\sin(\theta)$$

$$\omega_z(\theta, \varphi) := \cos(\theta)$$



Parametrization of the unit sphere:

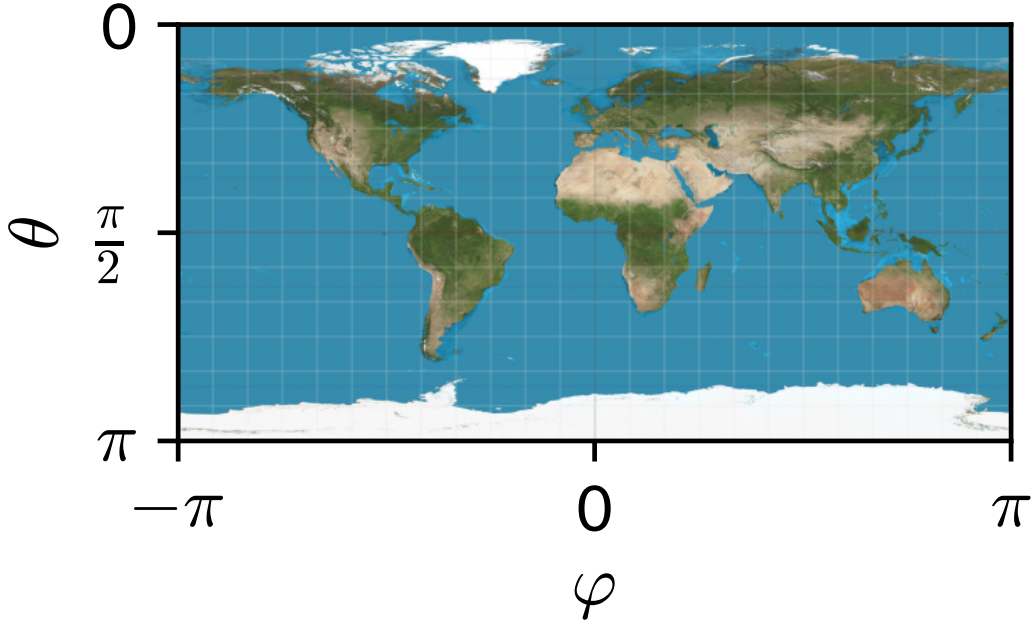
$$\omega_x^2 + \omega_y^2 + \omega_z^2 = (\cos^2\varphi + \sin^2\varphi)\sin^2\theta + \cos^2\theta = 1$$

For a fixed θ, we get a circle with constant z-coordinate.

$\theta \in [0, \frac{\pi}{2}]$ gives hemisphere Ω with $\omega_z(\theta, \varphi) \geq 0$.

Spherical integrals

Spherical coordinates are not area preserving.



To get correct integrals, we need the correction factor $\sin(\theta)$:

$$\int_{\Omega} f(\omega)\,\mathrm{d}\omega = \int_{-\pi}^{\pi}\int_{0}^{\frac{\pi}{2}} f(\omega(\theta,\varphi))\sin(\theta)\,\mathrm{d}\theta\,\mathrm{d}\varphi$$

Hemispherical integral $=$ two nested 1D integrals.

Interlude: Greek alphabet

Confused by those odd characters? Let me help.

Name	Lowercase	Uppercase
alpha	α	A
theta	θ	Θ
pi	π	Π
rho	ρ	P
phi	φ	Φ
omega	ω	Ω

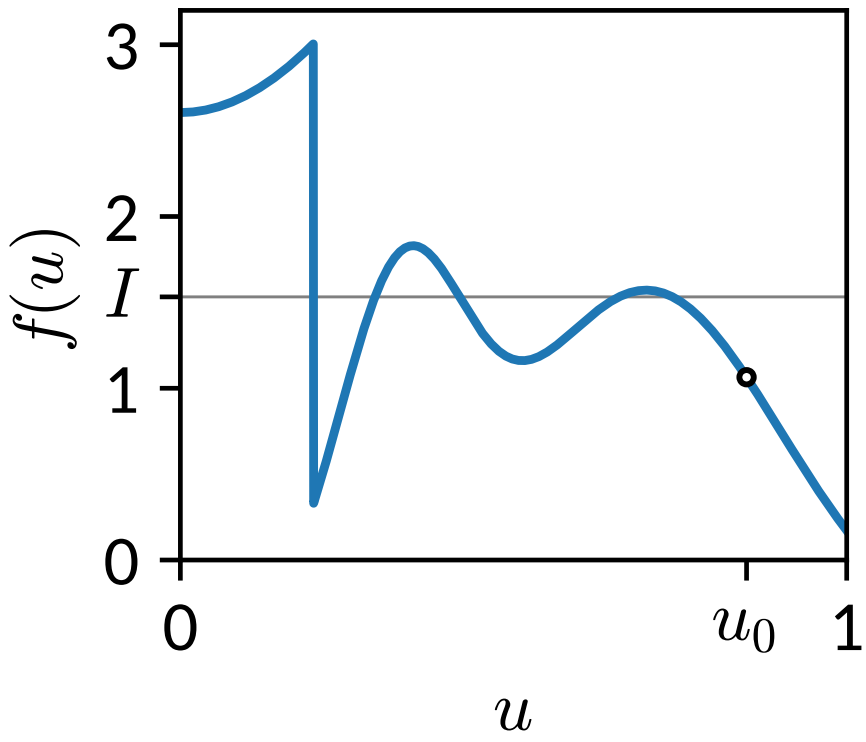
en.wikipedia.org/wiki/Greek_alphabet

Monte Carlo integration

1D Monte Carlo integration

Let u_0 be a random sample distributed uniformly in $[0, 1)$.

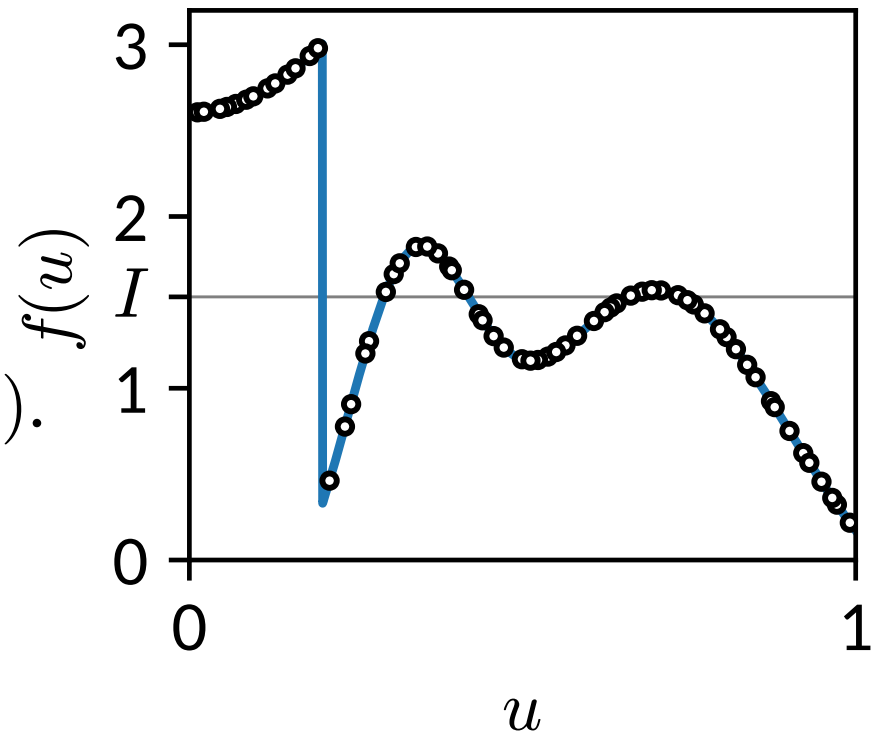
$$I := \int_0^1 f(u)\,\mathrm{d}u \approx f(u_0)$$



1D Monte Carlo integration

Let u_0 be a random sample distributed uniformly in $[0, 1)$.

$$I := \int_0^1 f(u)\,\mathrm{d}u \approx f(u_0)$$



More generally: $u_0, \ldots, u_{N-1}$

independent and uniform in $[0, 1)$.

$$\int_0^1 f(u)\,\mathrm{d}u \approx \tfrac{1}{N} \sum_{j=0}^{N-1} f(u_j)$$

Law of large numbers:

$$I = \lim_{N\to\infty} \tfrac{1}{N} \sum_{j=0}^{N-1} f(u_j)$$ with 100% probability. Unbiased!

kD Monte Carlo integration

To integrate the kD-function $f\colon [0,1]^k \to \mathbb{R}$,

sample $\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}$ uniformly in $[0,1)^k$.

$$\int_{[0,1]^k} f(\boldsymbol{u})\,\mathrm{d}\boldsymbol{u} \approx \frac{1}{N}\sum_{j=0}^{N-1} f(\boldsymbol{u}_j)$$

kD Monte Carlo integration

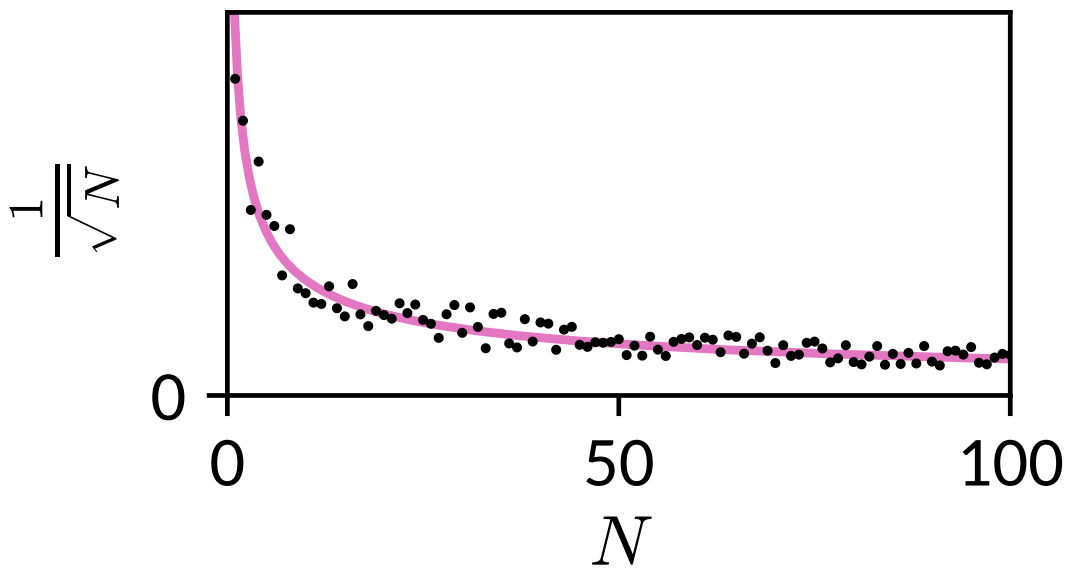
To integrate the kD-function $f\colon [0,1]^k \to \mathbb{R}$,

sample $\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}$ uniformly in $[0,1)^k$.

$$\int_{[0,1]^k} f(\boldsymbol{u})\,\mathrm{d}\boldsymbol{u} \approx \frac{1}{N}\sum_{j=0}^{N-1} f(\boldsymbol{u}_j)$$

Central limit theorem: The error is proportional to $\frac{1}{\sqrt{N}}$.

Four times more samples, half the error.



kD Monte Carlo integration

To integrate the kD-function $f\colon [0,1]^k \to \mathbb{R}$,

sample $\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}$ uniformly in $[0,1)^k$.

$$\int_{[0,1]^k} f(\boldsymbol{u})\,\mathrm{d}\boldsymbol{u} \approx \frac{1}{N}\sum_{j=0}^{N-1} f(\boldsymbol{u}_j)$$

Central limit theorem: The error is proportional to $\frac{1}{\sqrt{N}}$.

Four times more samples, half the error.

Sampling a kD grid gives an error proportional to $N^{-\frac{1}{k}}$.

Much worse for $k \geq 3$.

In path tracing, we essentially care about $k = \infty$.

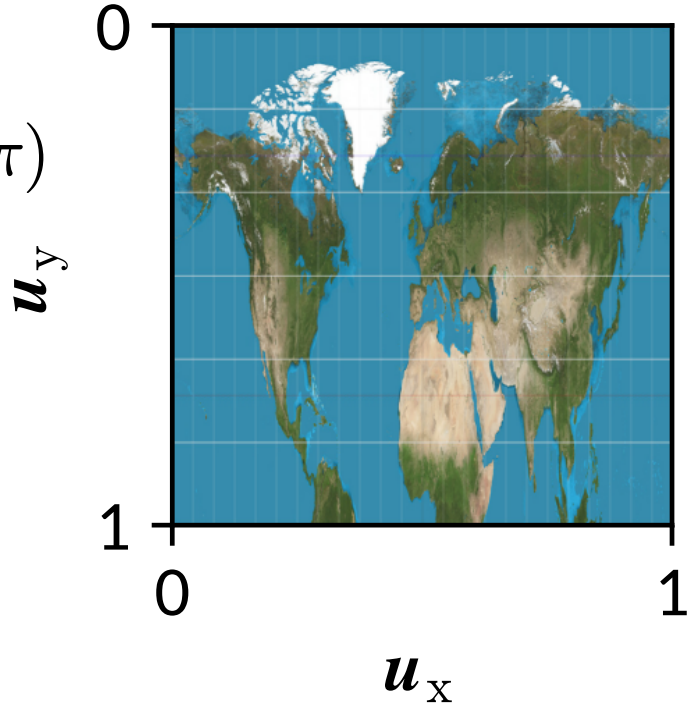
Hemispherical Monte Carlo integration

Sample $\boldsymbol{u}$ uniformly in $[0,1)^2$.

$\theta = \frac{\pi}{2}\,\boldsymbol{u}_{\mathrm{y}} \in [0, \frac{\pi}{2}), \quad \varphi = 2\pi\,\boldsymbol{u}_{\mathrm{x}} - \pi \in [-\pi, \pi)$

Parametrization of Ω:

$\omega(\boldsymbol{u}) := \omega(\theta, \varphi) = \omega(\frac{\pi}{2}\boldsymbol{u}_{\mathrm{y}}, 2\pi\,\boldsymbol{u}_{\mathrm{x}} - \pi)$



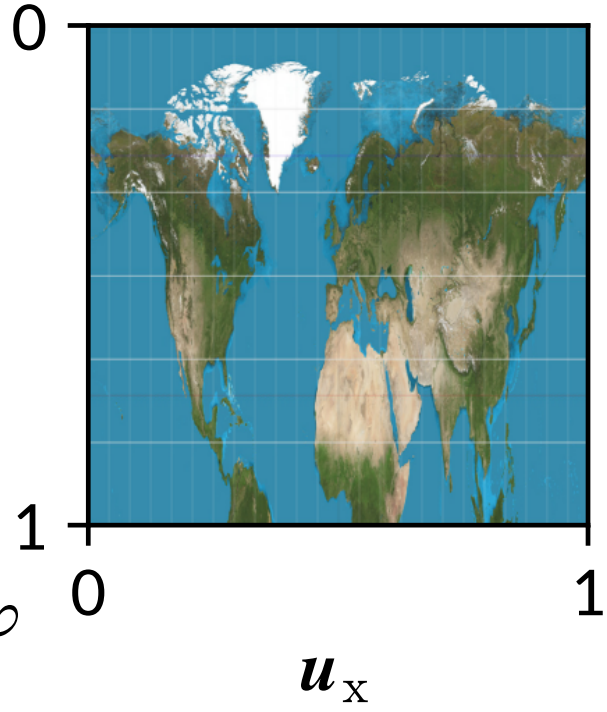
Hemispherical Monte Carlo integration

Sample $\boldsymbol{u}$ uniformly in $[0,1)^2$.

$\theta = \frac{\pi}{2}\,\boldsymbol{u}_{\mathrm{y}} \in [0, \frac{\pi}{2}), \quad \varphi = 2\pi\,\boldsymbol{u}_{\mathrm{x}} - \pi \in [-\pi, \pi)$

Parametrization of Ω:

$\omega(\boldsymbol{u}) := \omega(\theta, \varphi) = \omega(\frac{\pi}{2}\boldsymbol{u}_{\mathrm{y}}, 2\pi\,\boldsymbol{u}_{\mathrm{x}} - \pi)$



$$\int_{\Omega} f(\omega)\,\mathrm{d}\omega = \int_{-\pi}^{\pi}\int_{0}^{\frac{\pi}{2}} f(\omega(\theta,\varphi))\sin(\theta)\,\mathrm{d}\theta\,\mathrm{d}\varphi$$

$$= 2\pi\frac{\pi}{2}\int_{[0,1]^2} f(\omega(\boldsymbol{u}'))\sin(\tfrac{\pi}{2}\boldsymbol{u}'_{\mathrm{y}})\,\mathrm{d}\boldsymbol{u}'$$

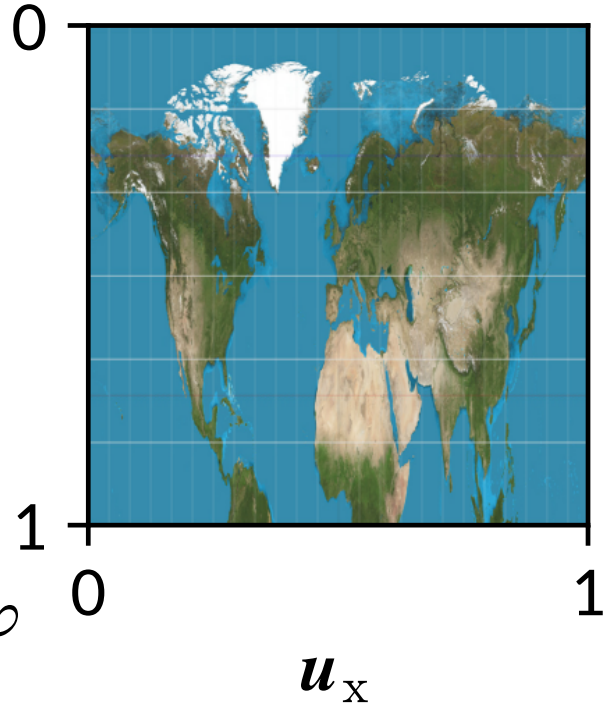
Hemispherical Monte Carlo integration

Sample $\boldsymbol{u}$ uniformly in $[0,1)^2$.

$\theta = \frac{\pi}{2}\,\boldsymbol{u}_{\mathrm{y}} \in [0, \frac{\pi}{2}), \quad \varphi = 2\pi\,\boldsymbol{u}_{\mathrm{x}} - \pi \in [-\pi, \pi)$

Parametrization of Ω:

$\omega(\boldsymbol{u}) := \omega(\theta, \varphi) = \omega(\frac{\pi}{2}\boldsymbol{u}_{\mathrm{y}}, 2\pi\,\boldsymbol{u}_{\mathrm{x}} - \pi)$



$$\int_{\Omega} f(\omega)\,\mathrm{d}\omega = \int_{-\pi}^{\pi}\int_{0}^{\frac{\pi}{2}} f(\omega(\theta,\varphi))\sin(\theta)\,\mathrm{d}\theta\,\mathrm{d}\varphi$$

$$= \pi^2 \int_{[0,1]^2} f(\omega(\boldsymbol{u}'))\sin(\tfrac{\pi}{2}\boldsymbol{u}'_{\mathrm{y}})\,\mathrm{d}\boldsymbol{u}'$$

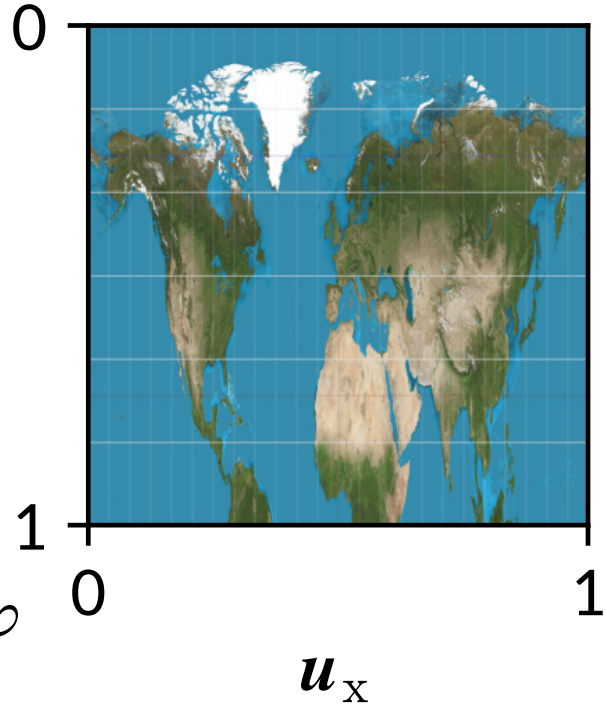
Hemispherical Monte Carlo integration

Sample $\boldsymbol{u}$ uniformly in $[0,1)^2$.

$\theta = \frac{\pi}{2}\,\boldsymbol{u}_{\mathrm{y}} \in [0, \frac{\pi}{2}), \quad \varphi = 2\pi\,\boldsymbol{u}_{\mathrm{x}} - \pi \in [-\pi, \pi)$

Parametrization of Ω:

$\omega(\boldsymbol{u}) := \omega(\theta, \varphi) = \omega(\frac{\pi}{2}\boldsymbol{u}_{\mathrm{y}}, 2\pi\,\boldsymbol{u}_{\mathrm{x}} - \pi)$



$$\int_{\Omega} f(\omega)\,\mathrm{d}\omega = \int_{-\pi}^{\pi}\int_{0}^{\frac{\pi}{2}} f(\omega(\theta,\varphi))\sin(\theta)\,\mathrm{d}\theta\,\mathrm{d}\varphi$$

$$= \pi^2 \int_{[0,1]^2} f(\omega(\boldsymbol{u}'))\sin(\tfrac{\pi}{2}\boldsymbol{u}'_{\mathrm{y}})\,\mathrm{d}\boldsymbol{u}'$$

$$\approx \pi^2 f(\omega(\boldsymbol{u}))\sin(\tfrac{\pi}{2}\boldsymbol{u}_{\mathrm{y}})$$

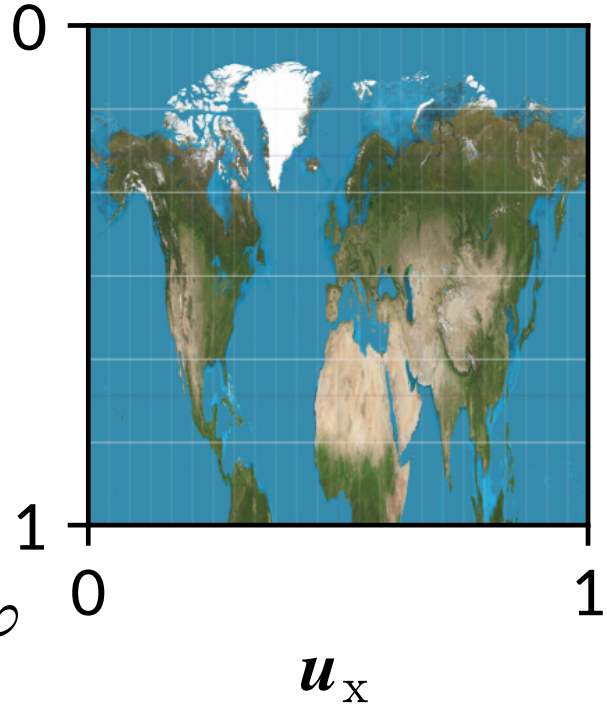
Hemispherical Monte Carlo integration

Sample $\boldsymbol{u}$ uniformly in $[0,1)^2$.

$\theta = \frac{\pi}{2}\,\boldsymbol{u}_{\mathrm{y}} \in [0, \frac{\pi}{2}), \quad \varphi = 2\pi\,\boldsymbol{u}_{\mathrm{x}} - \pi \in [-\pi, \pi)$

Parametrization of Ω:

$\omega(\boldsymbol{u}) := \omega(\theta, \varphi) = \omega(\frac{\pi}{2}\boldsymbol{u}_{\mathrm{y}}, 2\pi\,\boldsymbol{u}_{\mathrm{x}} - \pi)$



$$\int_{\Omega} f(\omega)\,\mathrm{d}\omega = \int_{-\pi}^{\pi}\int_{0}^{\frac{\pi}{2}} f(\omega(\theta,\varphi))\sin(\theta)\,\mathrm{d}\theta\,\mathrm{d}\varphi$$

$$= \pi^2 \int_{[0,1]^2} f(\omega(\boldsymbol{u}'))\sin(\tfrac{\pi}{2}\boldsymbol{u}'_{\mathrm{y}})\,\mathrm{d}\boldsymbol{u}'$$

$$\approx \pi^2 f(\omega(\boldsymbol{u}))\sin(\tfrac{\pi}{2}\boldsymbol{u}_{\mathrm{y}})$$

For hemisphere $\Omega(\boldsymbol{x})$, rotate coordinates to turn it into Ω.

Pseudorandom number generator

GLSL implementation of PCG 2D.

Returns $\boldsymbol{u}$ distributed uniformly in $[0, 1)^2$.

The seed can be e.g. pixel_index + 2^{16}frame_index.

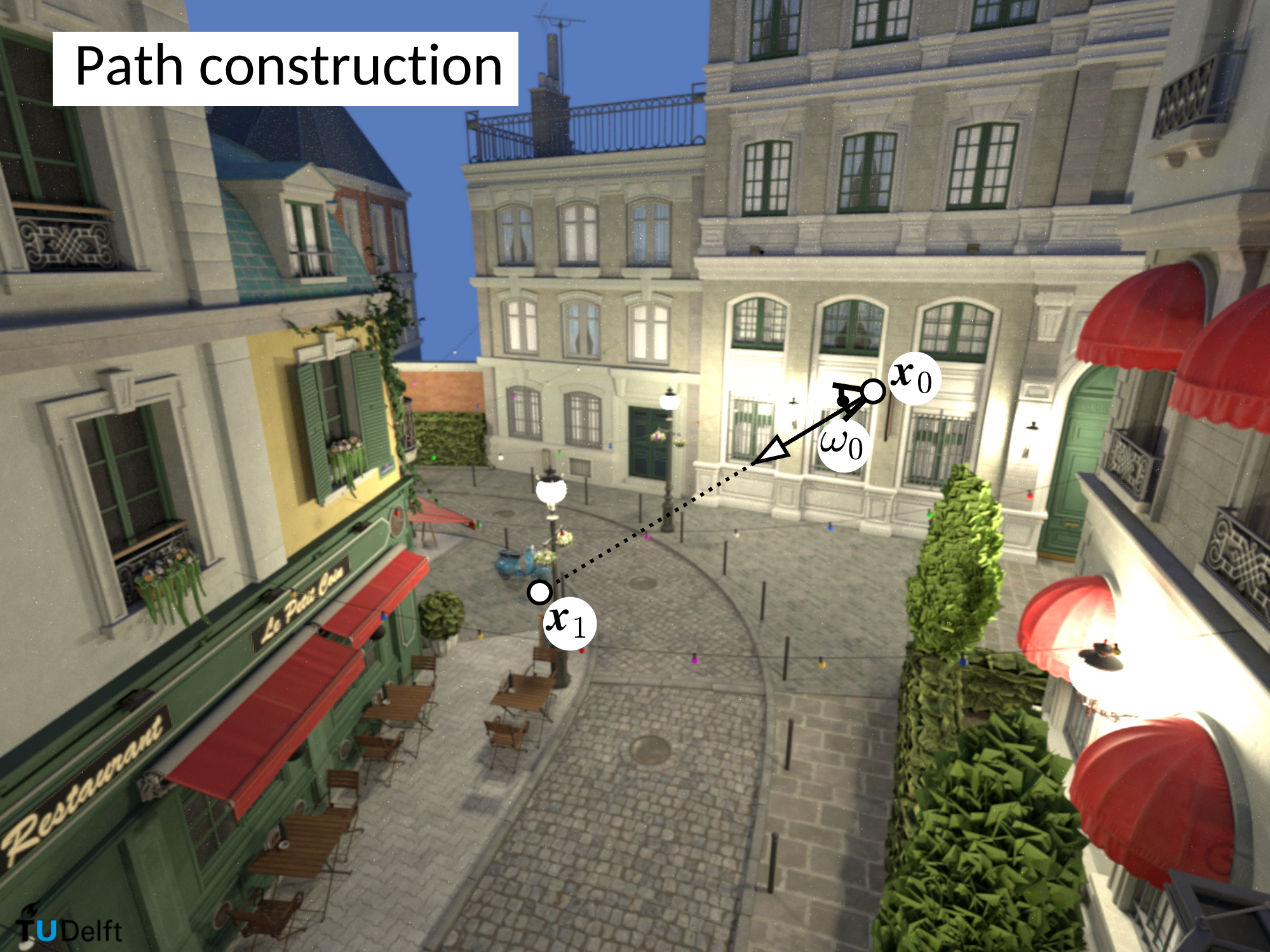
The seed gets updated and can be reused.

```
vec2 get_random_numbers(inout uvec2 seed) {
    seed = 1664525u * seed + 1013904223u;
    seed.x += 1664525u * seed.y;
    seed.y += 1664525u * seed.x;
    seed ^= (seed >> 16u);
    seed.x += 1664525u * seed.y;
    seed.y += 1664525u * seed.x;
    seed ^= (seed >> 16u);
    return vec2(seed) * pow(0.5, 32.0);
}
```

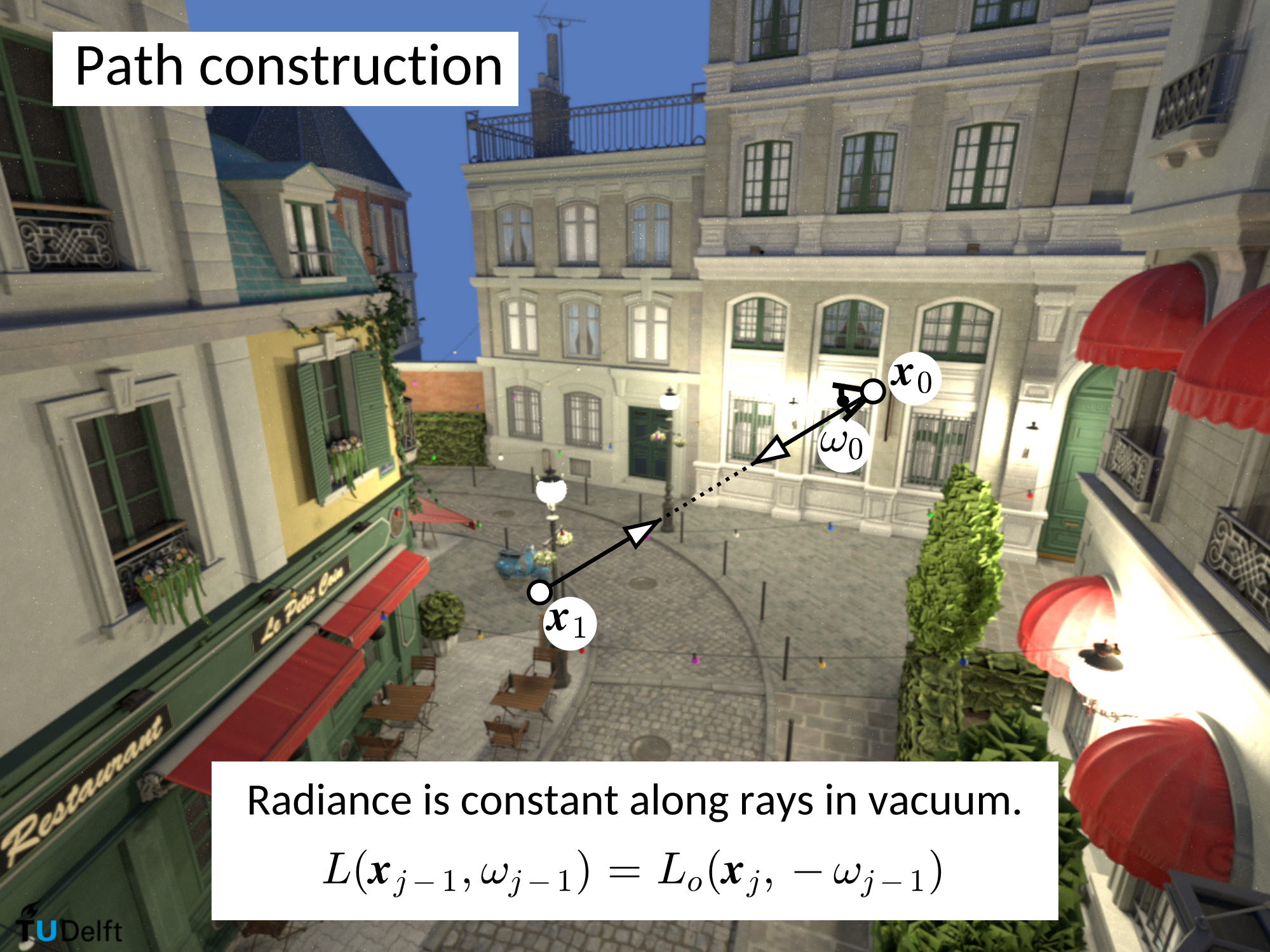
Path tracing

Path construction

Path construction



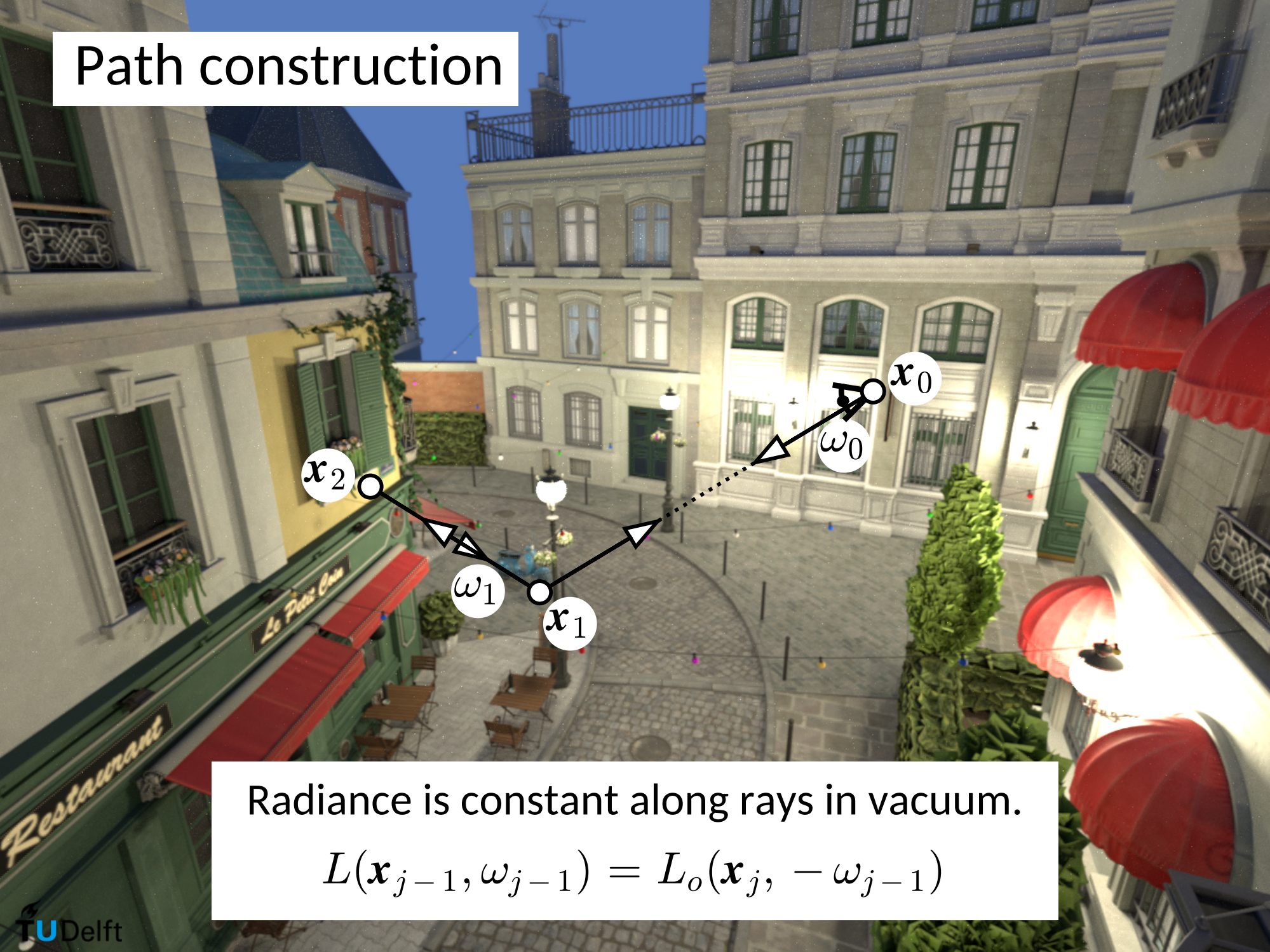
Path construction



Radiance is constant along rays in vacuum.

$$L(\boldsymbol{x}_{j-1}, \omega_{j-1}) = L_o(\boldsymbol{x}_j, -\omega_{j-1})$$

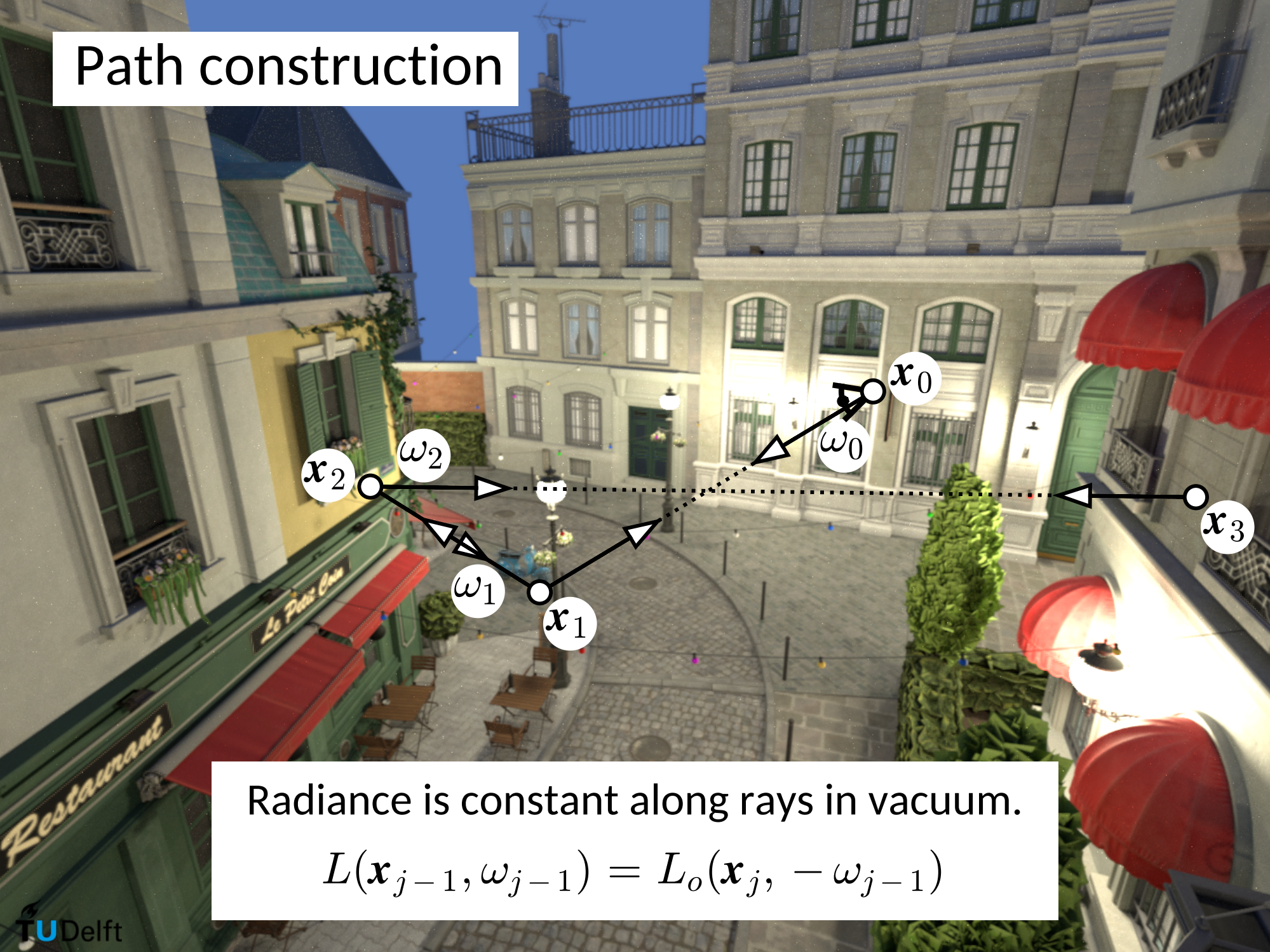
Path construction



Radiance is constant along rays in vacuum.

$$L(\boldsymbol{x}_{j-1}, \omega_{j-1}) = L_o(\boldsymbol{x}_j, -\omega_{j-1})$$

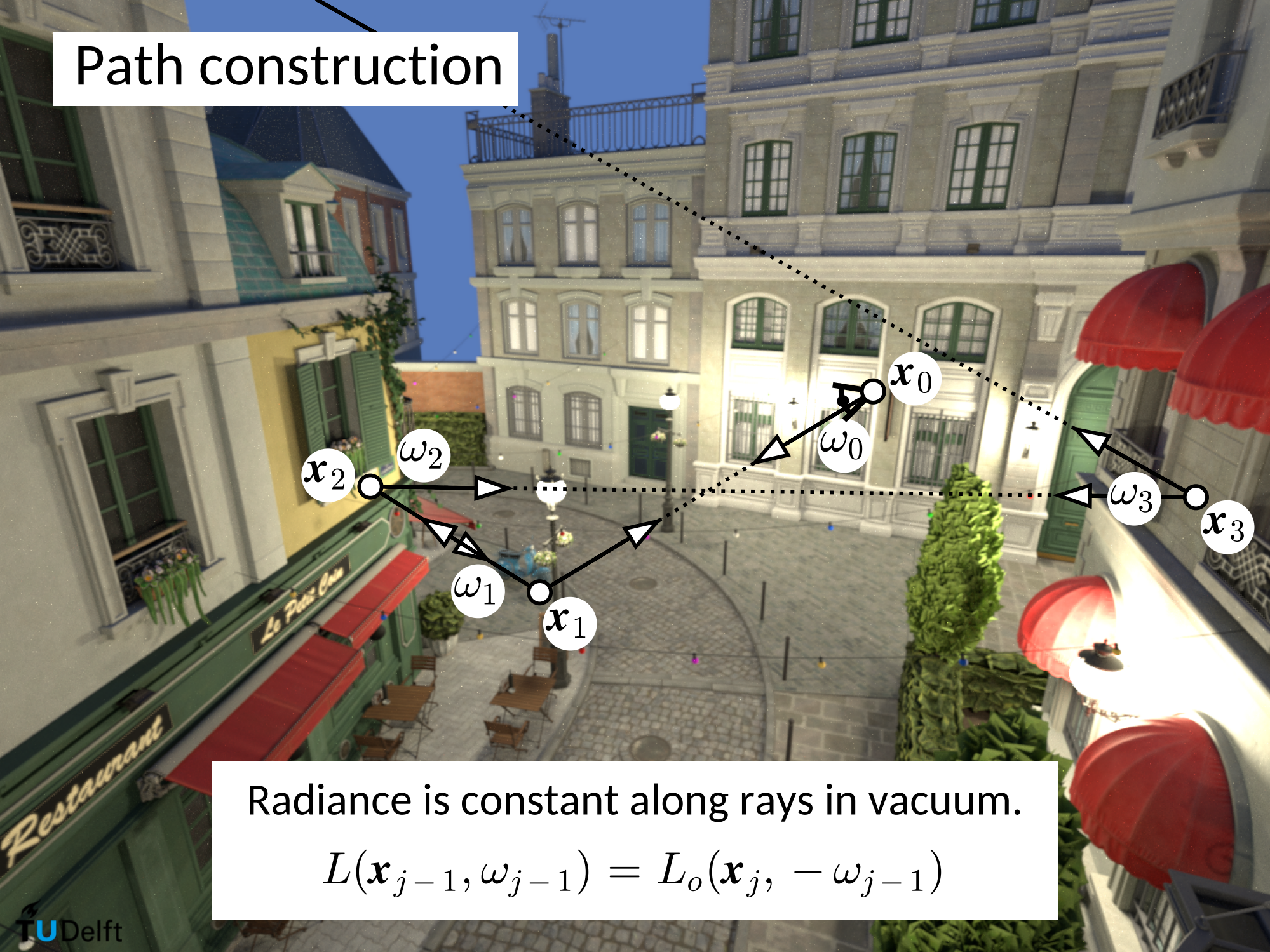
Path construction



Radiance is constant along rays in vacuum.

$$L(\boldsymbol{x}_{j-1}, \omega_{j-1}) = L_o(\boldsymbol{x}_j, -\omega_{j-1})$$

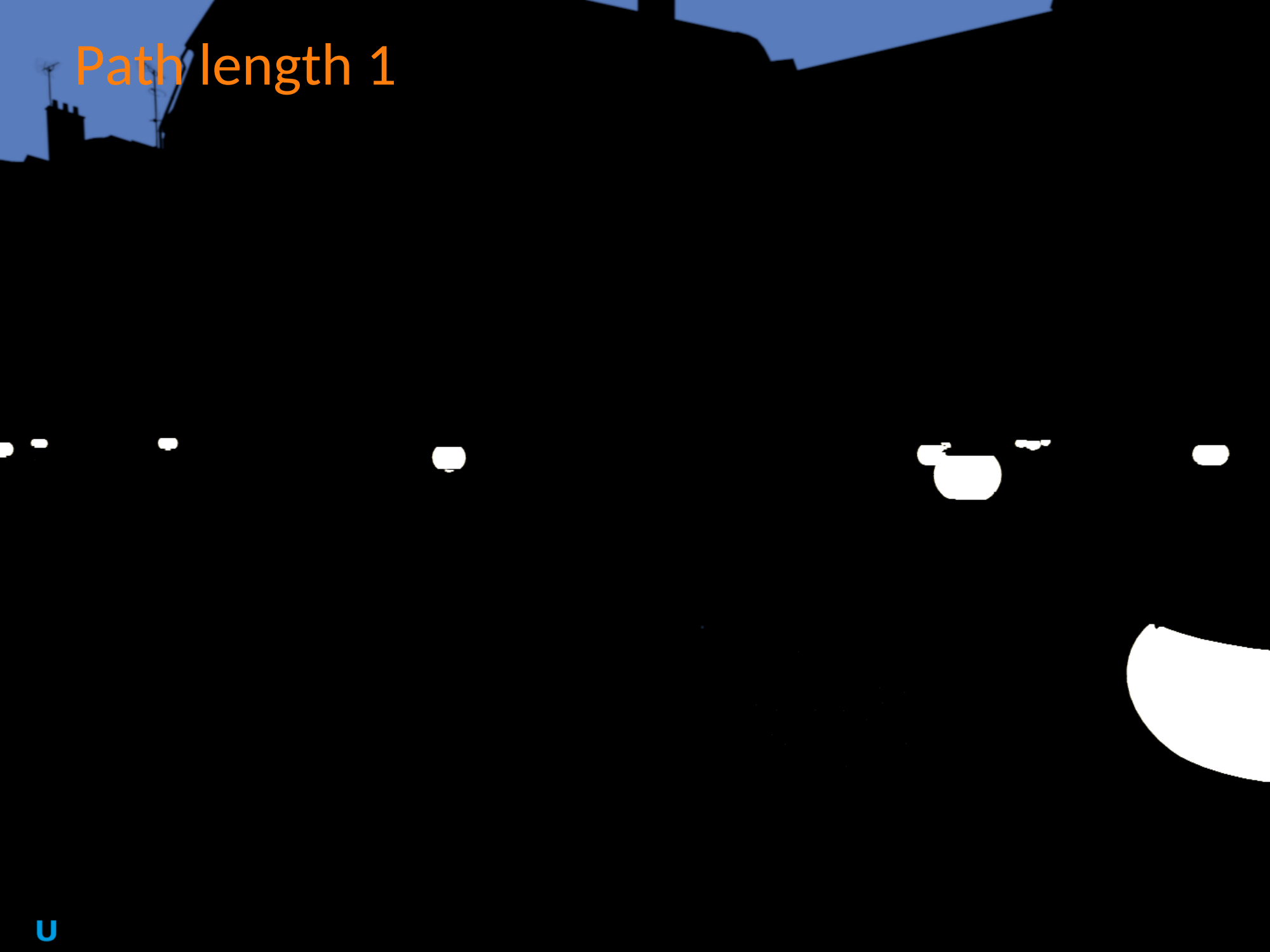
Path construction



Radiance is constant along rays in vacuum.

$$L(\boldsymbol{x}_{j-1}, \omega_{j-1}) = L_o(\boldsymbol{x}_j, -\omega_{j-1})$$

Path length 1




Path length 2
TU Delft


Path length 3
TU Delft


Path length 4
TU Delft


Path length 5
TU Delft

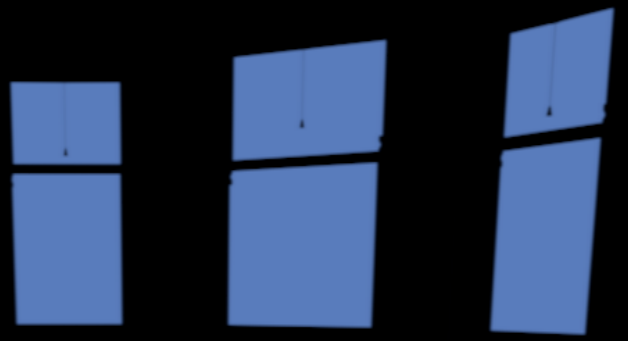

Path length 6
TU Delft


Path length 7
TU Delft

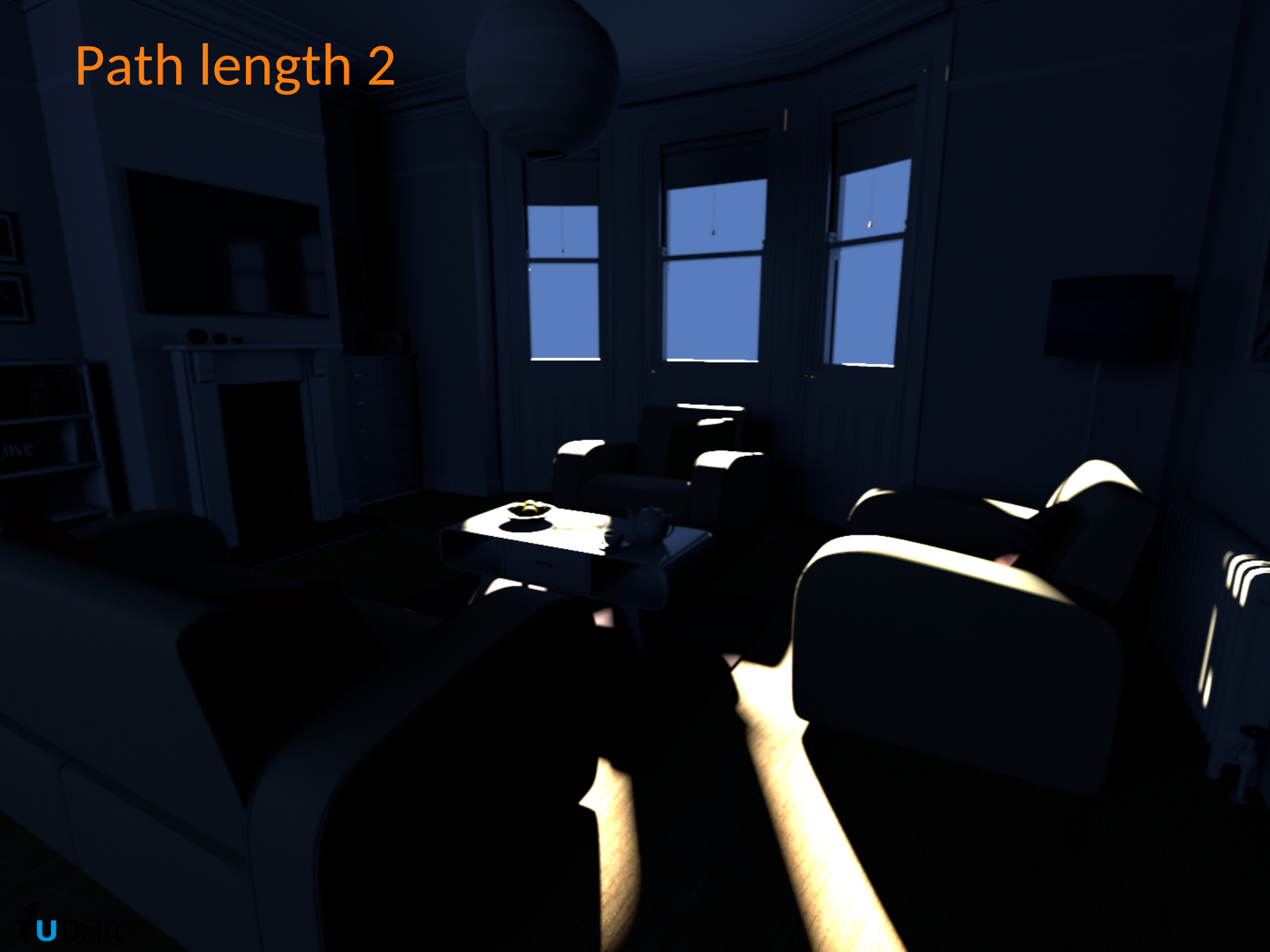

Path length 8
TU Delft

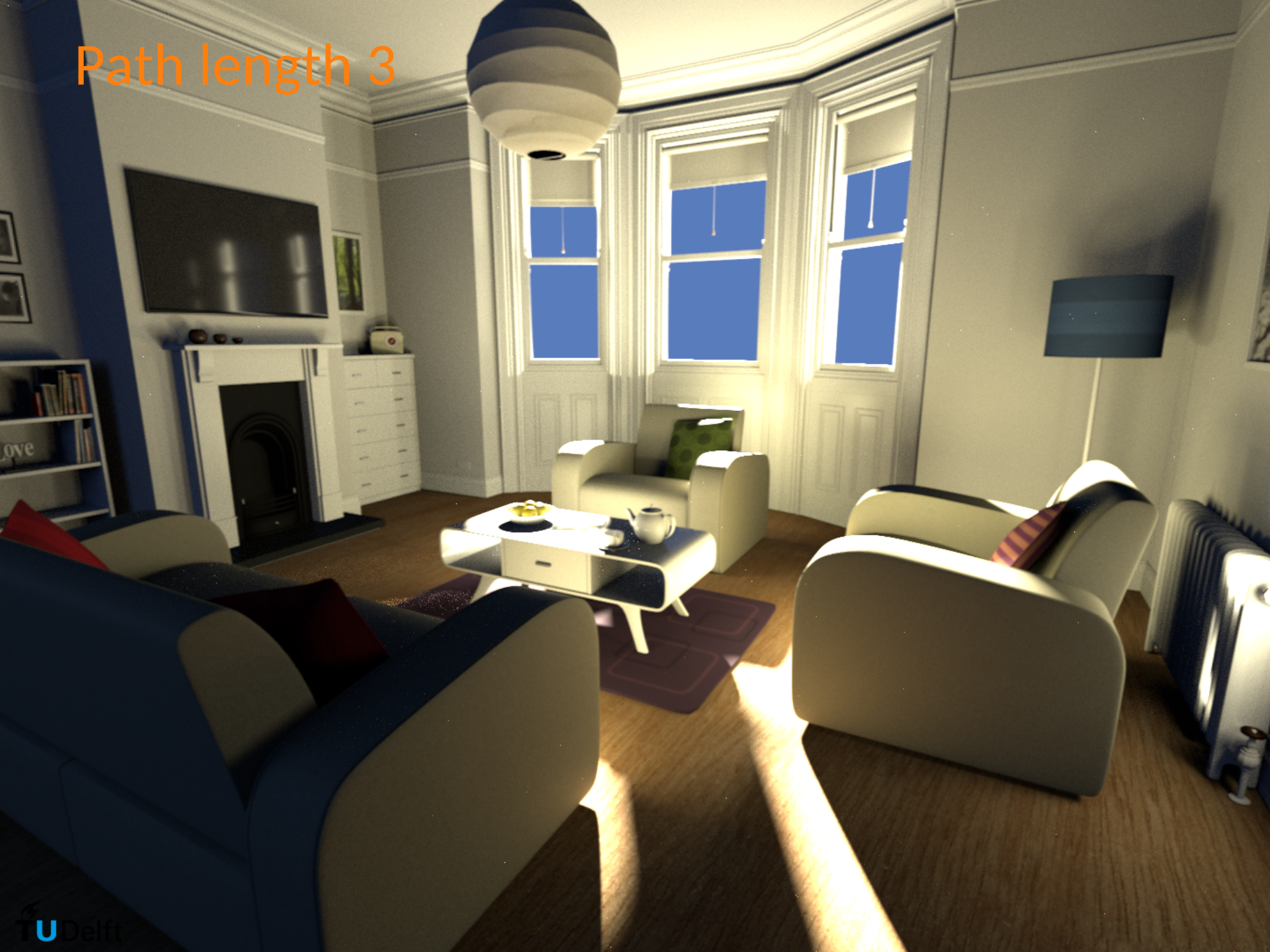

Path length 9
TU Delft

Path length 1



Path length 2




Path length 3
TU Delft


Path length 4
TUDelft


Path length 5
TUDelft


Path length 6
TUDelft


Path length 7
TUDelft


Path length 8
TUDelft


Path length 9
TUDelft

path_trace$(\boldsymbol{x}_{k-1}, \omega_{k-1}, k-1)$

Input: Ray $\boldsymbol{x}_{k-1}, \omega_{k-1}$ and current path length $k-1$.

Output: Estimate of incoming radiance $L(\boldsymbol{x}_{k-1}, \omega_{k-1})$.

path_trace$(\boldsymbol{x}_{k-1}, \omega_{k-1}, k-1)$

Input: Ray $\boldsymbol{x}_{k-1}, \omega_{k-1}$ and current path length $k-1$.

Output: Estimate of incoming radiance $L(\boldsymbol{x}_{k-1}, \omega_{k-1})$.

Trace ray $\boldsymbol{x}_{k-1}, \omega_{k-1}$.

If it missed all scene triangles: Return background emission.

For hit point $\boldsymbol{x}_k$: Get shading data and $L_{e,k} := L_e(\boldsymbol{x}_k, -\omega_{k-1})$.

If $k-1 =$ max. path length: Return $L_{e,k}$.

path_trace$(\boldsymbol{x}_{k-1}, \omega_{k-1}, k-1)$

Input: Ray $\boldsymbol{x}_{k-1}, \omega_{k-1}$ and current path length $k-1$.

Output: Estimate of incoming radiance $L(\boldsymbol{x}_{k-1}, \omega_{k-1})$.

Trace ray $\boldsymbol{x}_{k-1}, \omega_{k-1}$.

If it missed all scene triangles: Return background emission.

For hit point $\boldsymbol{x}_k$: Get shading data and $L_{e,k} := L_e(\boldsymbol{x}_k, -\omega_{k-1})$.

If $k-1 =$ max. path length: Return $L_{e,k}$.

$\omega_k := Q\omega(\boldsymbol{u}) \in \Omega(\boldsymbol{x}_k)$ ($\boldsymbol{u}$ uniform in $[0,1)^2$, $Q \in \mathbb{R}^{3\times 3}$ rotation).

Recurse: $L_k :=$ path_trace$(\boldsymbol{x}_k, \omega_k, k)$

Return $L_{e,k} + \pi^2 L_k \, \rho(\boldsymbol{x}_k, \omega_k, -\omega_{k-1}) \, \boldsymbol{n}(\boldsymbol{x}_k) \cdot \omega_k \sin(\frac{\pi}{2}\boldsymbol{u}_y)$.

Path tracing

Compute camera ray $\boldsymbol{x}_0, \omega_0$.

$$L := \frac{1}{N} \sum_{j=0}^{N-1} \text{path\_trace}(\boldsymbol{x}_0, \omega_0, 0)$$

L is the pixel color.

Path tracing

Compute camera ray $\boldsymbol{x}_0, \omega_0$.

$$L := \frac{1}{N} \sum_{j=0}^{N-1} \text{path\_trace}(\boldsymbol{x}_0, \omega_0, 0)$$

L is the pixel color.

Useful to show intermediate results while summing over j.

That is called progressive rendering.

Start over when the scene or the camera changes.

Path tracing

Compute camera ray $\boldsymbol{x}_0, \omega_0$.

$$L := \frac{1}{N} \sum_{j=0}^{N-1} \text{path\_trace}(\boldsymbol{x}_0, \omega_0, 0)$$

L is the pixel color.

Useful to show intermediate results while summing over j.

That is called progressive rendering.

Start over when the scene or the camera changes.

Now rendering is solved.

Path tracing

Compute camera ray $\boldsymbol{x}_0, \omega_0$.

$$L := \frac{1}{N} \sum_{j=0}^{N-1} \text{path\_trace}(\boldsymbol{x}_0, \omega_0, 0)$$

L is the pixel color.

Useful to show intermediate results while summing over j.

That is called progressive rendering.

Start over when the scene or the camera changes.

Now rendering is solved.

Or is it?


TUDelft


TU Delft