



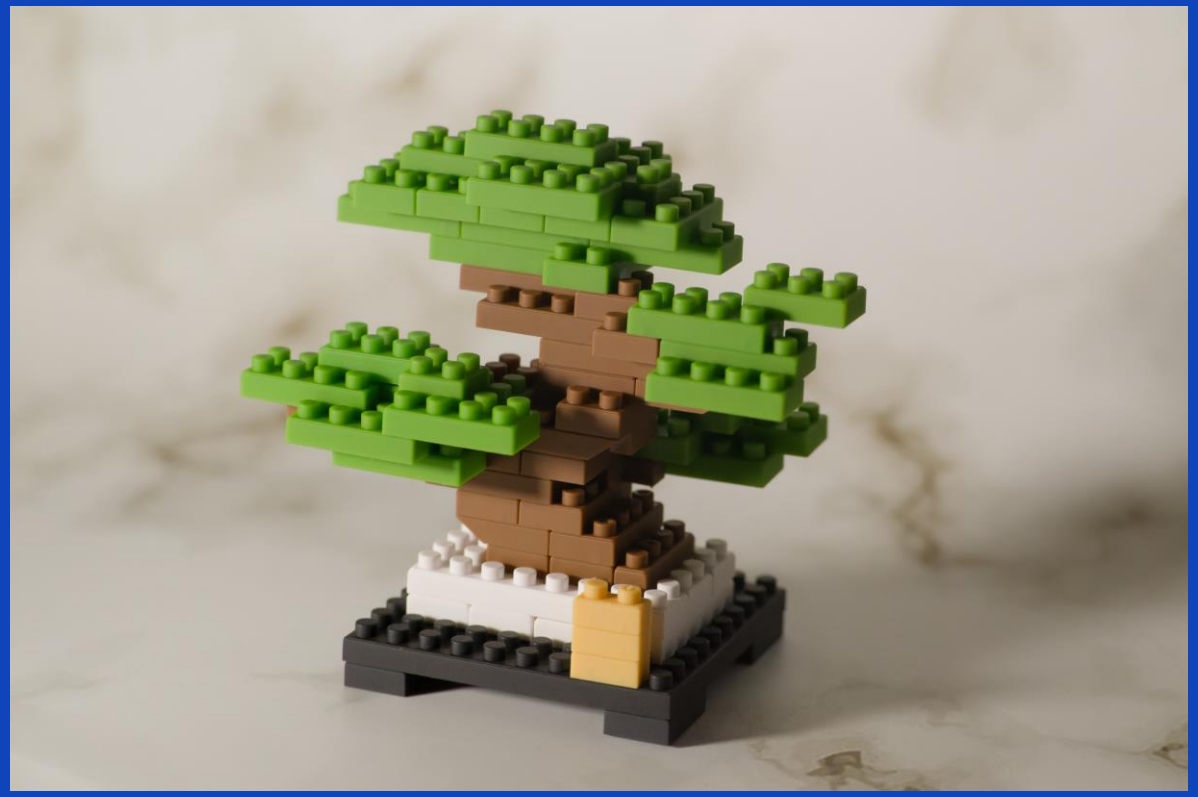
# 手軽に始めるGPUレイトレーシング！
# GPUプログラミングの基本からReSTIRまで

江藤 健汰
吉村 篤

ARR
Advanced Rendering Research Group

# Quiz – Which is the “Ray Tracing”?

A

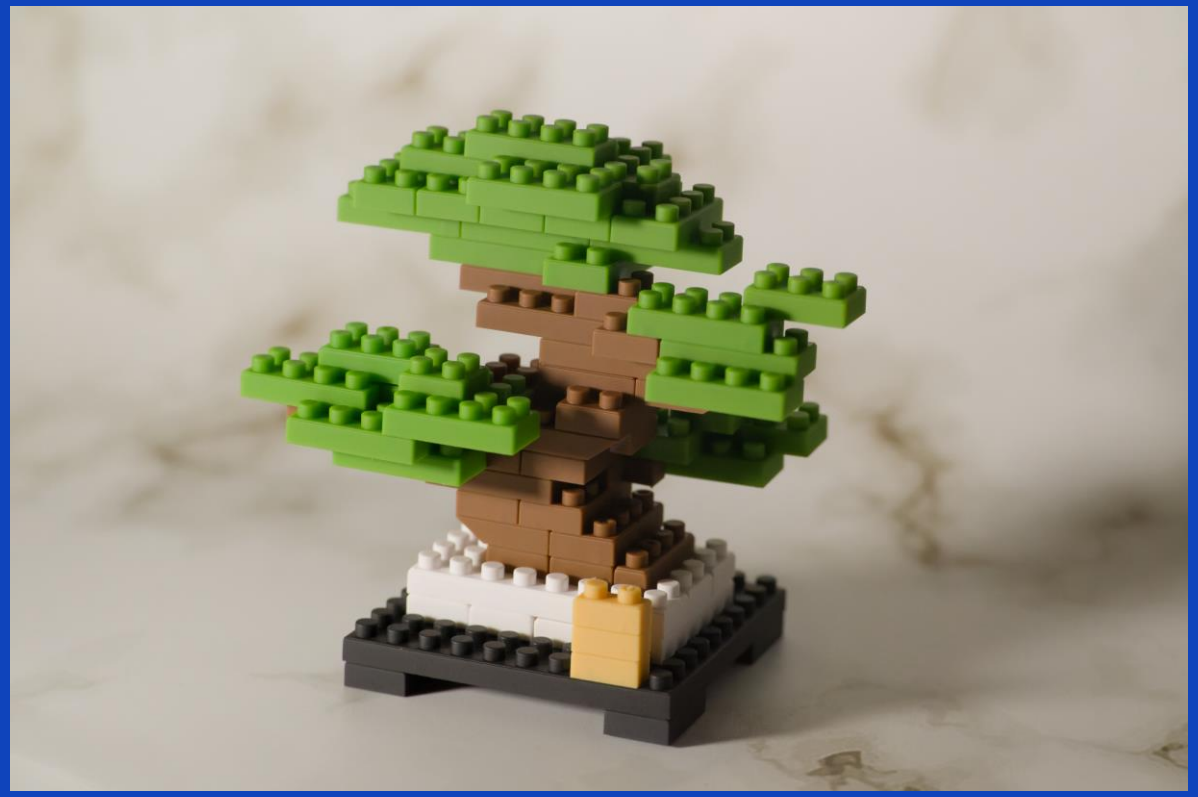


B

# Quiz – Which is the “Ray Tracing”?

**A** **Ray Tracing**

※ Radeon™ ProRender

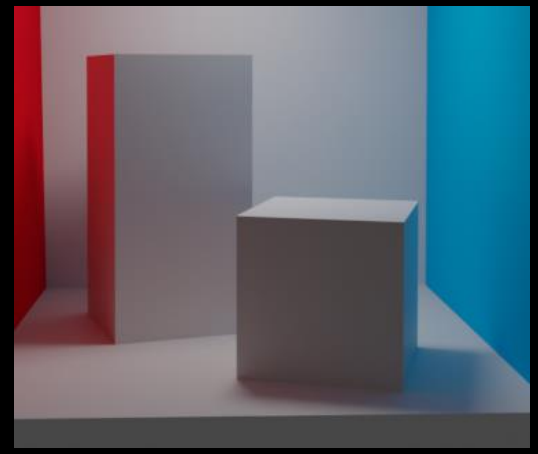


**B** **Photo**

※Nikon Z30 / DX 18-140

# Motivation to Ray Tracing

- Photorealistic Rendering
  - Soft shadow
  - Color Bleeding
  - Reflection / Refraction
  - Depth of Field

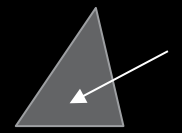


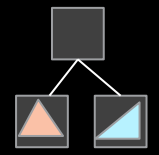
このような光の現象を単純で統一的な枠組みでシミュレート

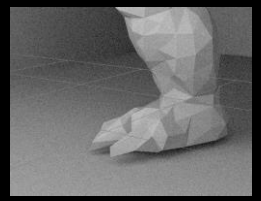

→ ・リアルタイム向けレンダリングエンジンのリファレンスとして
・視覚効果の一部として

# Agenda

- Overview of ray-tracing based rendering
- Ray vs Triangle
- Ambient Occlusion
- Optimization
  - Bounding volume hierarchy
  - HIPRT
- Path Tracing
  - NEE
  - RIS
  - ReSTIR

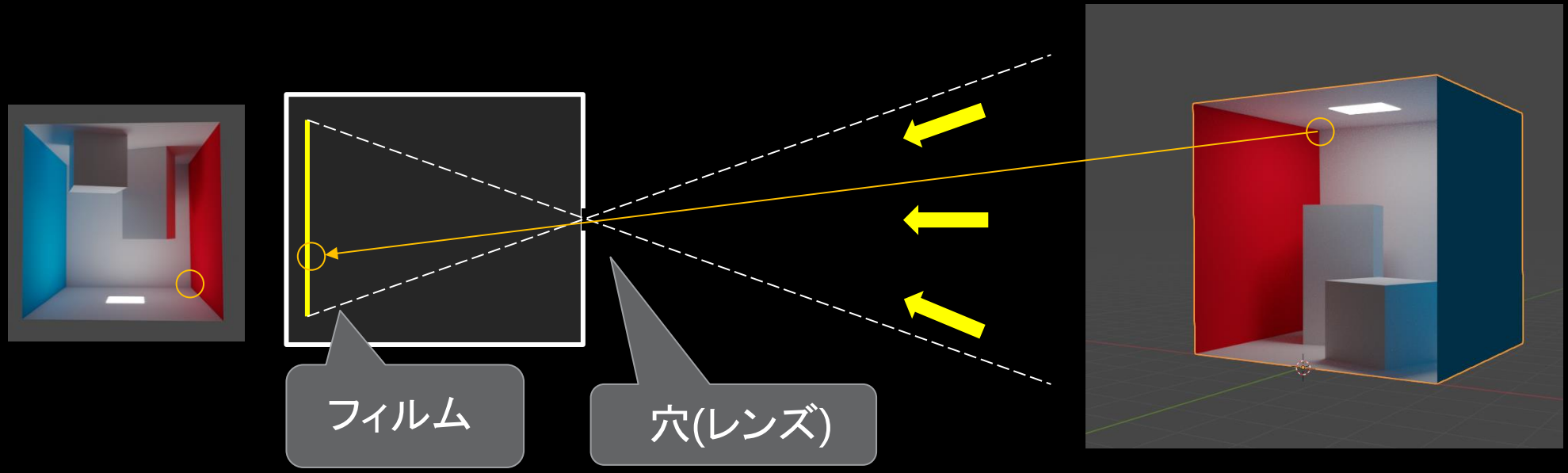






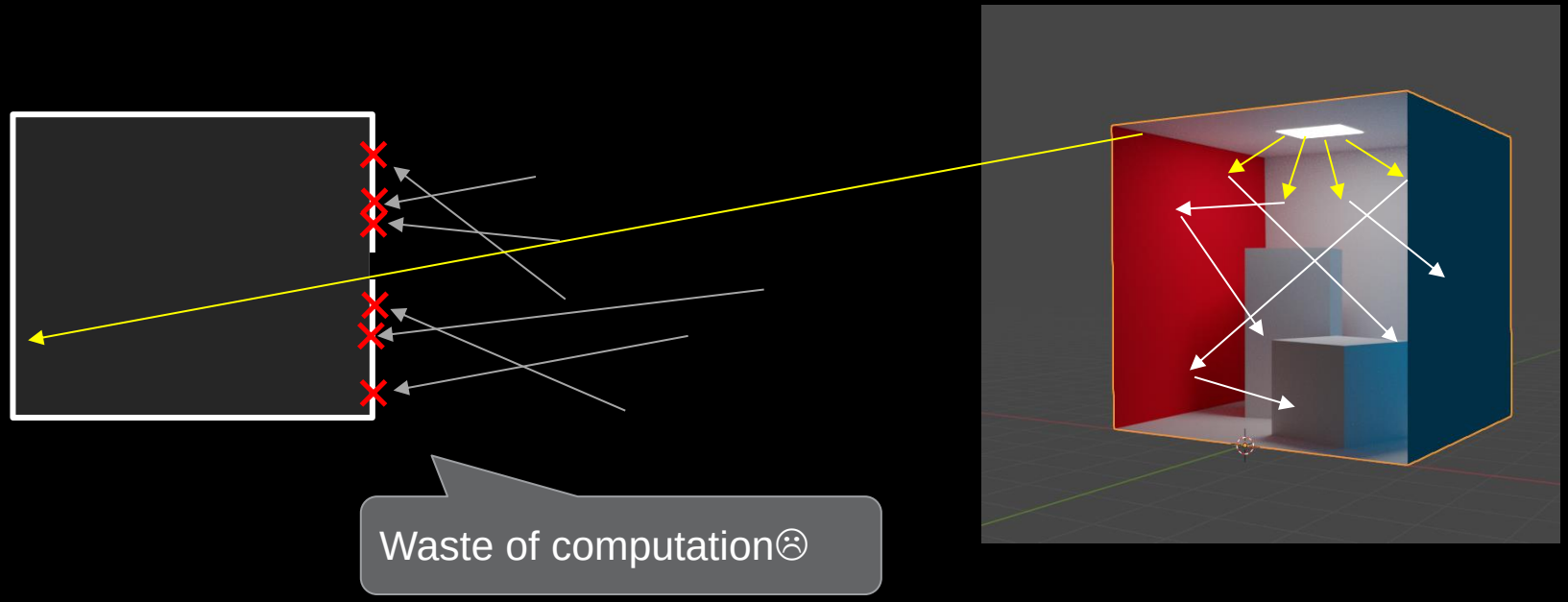
# Raytracing-based rendering algorithm

- A pinhole camera



# Raytracing-based rendering algorithm

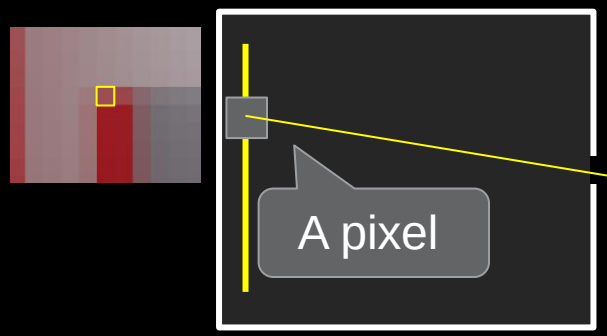
Approach 1 –Trace from light

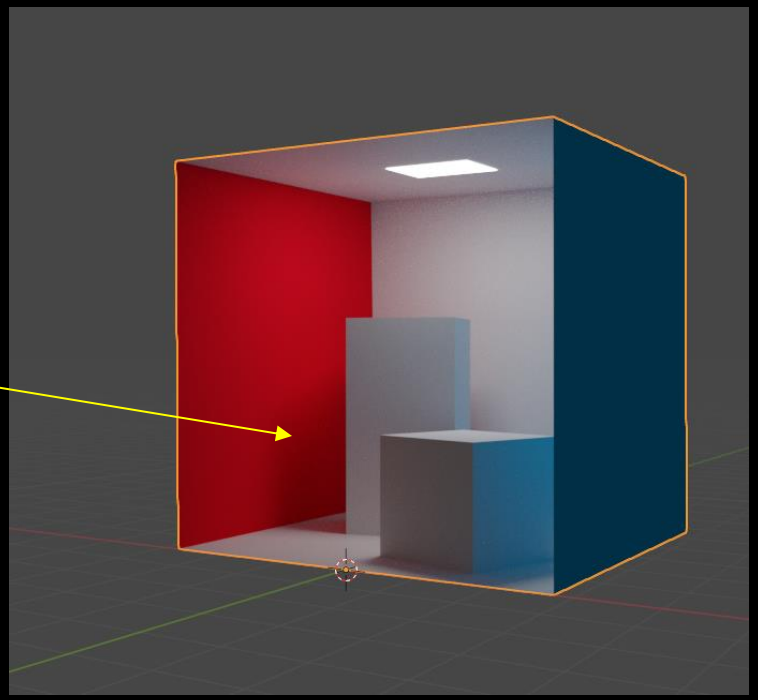


# Raytracing-based rendering algorithm

Approach 2 - Trace from camera

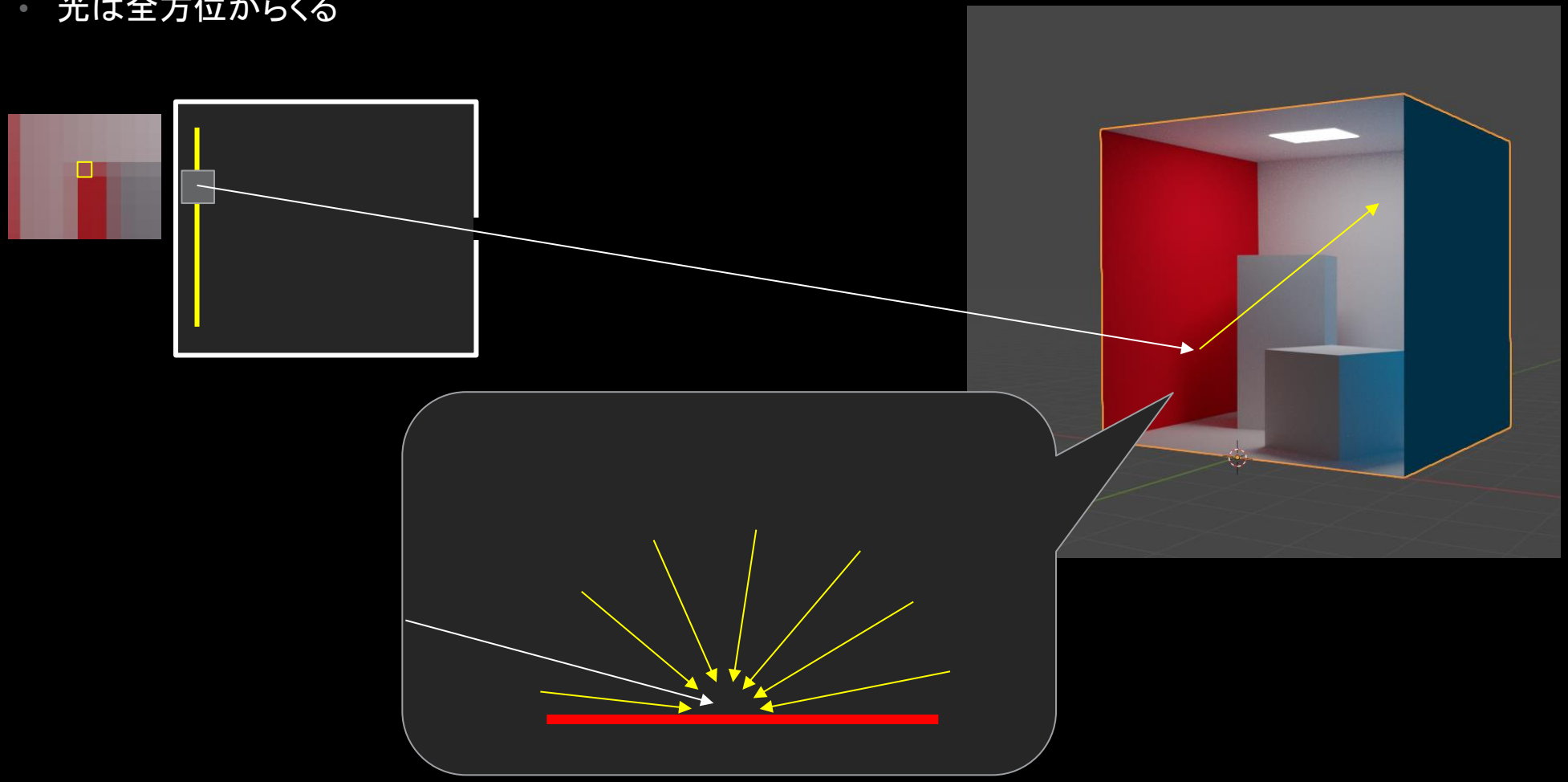
- カメラから見える光のみに計算を集中
- 多くの場合効率的



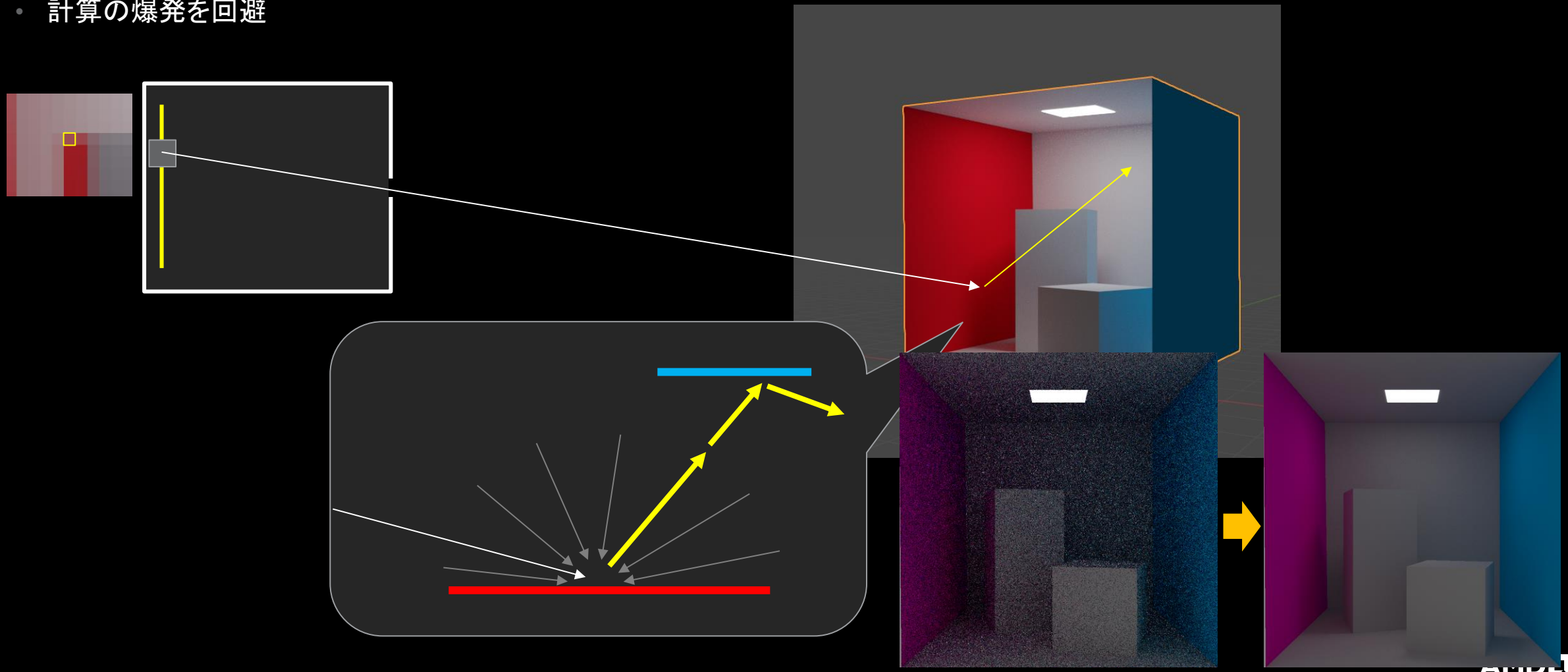
# Raytracing-based rendering algorithm

- ざらざらした表面にヒットした場合
  - 光は全方位からくる



# Raytracing-based rendering algorithm

- 確率的に一つの方向を選ぶ
  - 計算の爆発を回避



# HIP/CUDA

GPU向け C++ APIs / kernel language

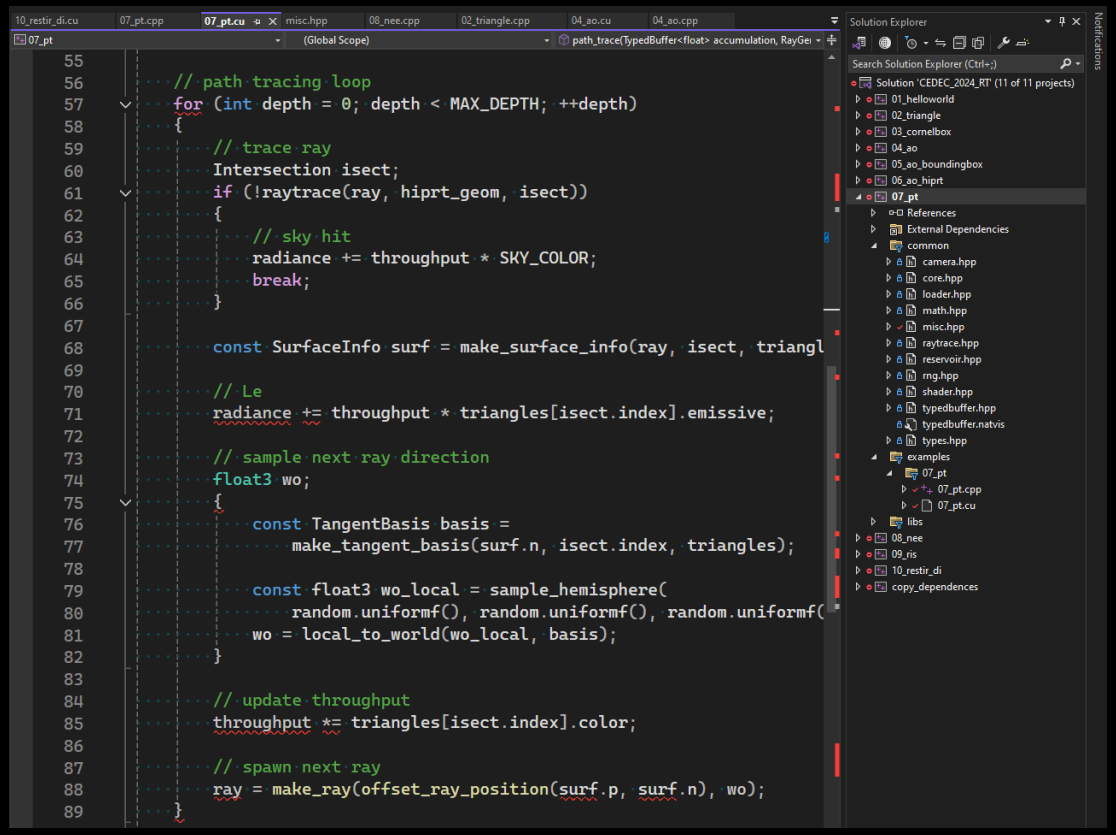
- AMD GPUs, CUDA devicesどちらでも実行可能
  - HIPのCUDA APIラッパー
- 並列化可能 / 計算コストの高い処理に向いている
  - CPUとは桁違いのスレッドを並列で動作
- GPUの並列性能を生かしたインタラクティブなデモ

すべてのサンプルコードは後ほど公開します

1. CEDIL
2. GPUOpen Blog

A HIP kernel example

```
__global__ void twiceKernel(float* xs)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    xs[index] *= 2.0f;
}
```






AMD

# Ray-scene intersection

- レイトレーシングではレイとシーンの高速な交差判定が必要不可欠

| ベンターによるライブラリを利用 | | 独自実装 |
|---|---|---|
| ハードウェアの機能を最大限利用できる | ⟷ | ハードウェアによる高速化に限界がある |
| 高速化などの実装コストがゼロ | | 数値安定性や高速化のための労力を払う必要がある |
| 中身がブラックボックス | | 中身を完全にコントロール可能 |
| 技術的知見の蓄積が難しい | | 技術的知見を蓄積できる |
| アプリケーションに特化した最適化が難しい | | アプリケーションに特化した最適化が可能 |

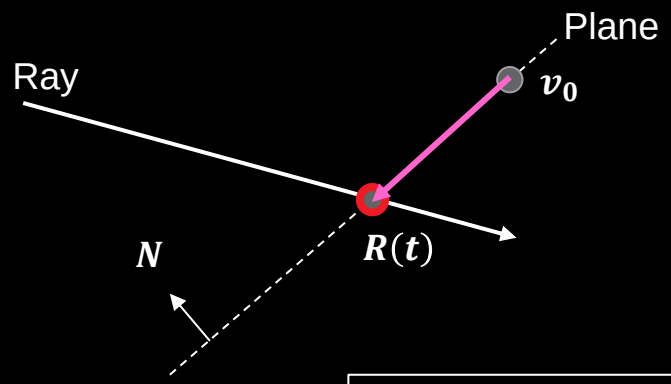
➡ これらのトレードオフの理解のため、最初は独自実装の例を紹介
後ほどライブラリを紹介

# Ray-triangle intersection

- 三角形 - 最もよく使われるジオメトリの形式
- 候補:
  - Tomas Möller and Ben Trumbore, “Fast Minimum Storage Ray-Triangle Intersection” a.k.a Möller-Trumbore
  - Woop et al. “Watertight Ray/Triangle Intersection”
- ここでは理解のしやすさ、単純さを優先した手法を紹介

# Ray-triangle intersection

1. レイと平面の衝突




$$N \cdot (R(t) - v_0) = 0$$

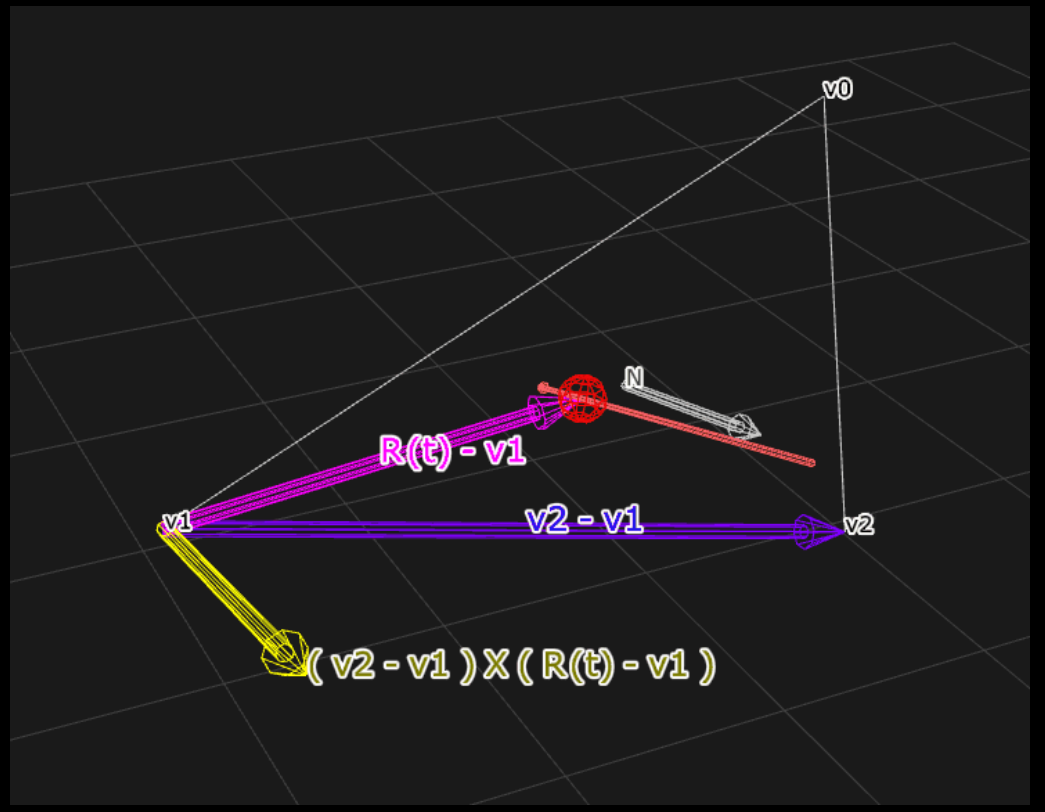
$$t = \frac{N \cdot (v_0 - r_o)}{N \cdot r_d}$$

| | Symbol |
|---|---|
| Ray origin | $r_o$ |
| Ray direction | $r_d$ |
| Ray | $R(t) = r_o + t r_d$ |
| Triangle normal | $N$ |
| Triangle vertices | $v_0, v_1, v_2$ |

2. 衝突点 $R(t)$ が三角形の内側かを判定

$R(t)$ がすべてのエッジの内側にあるか

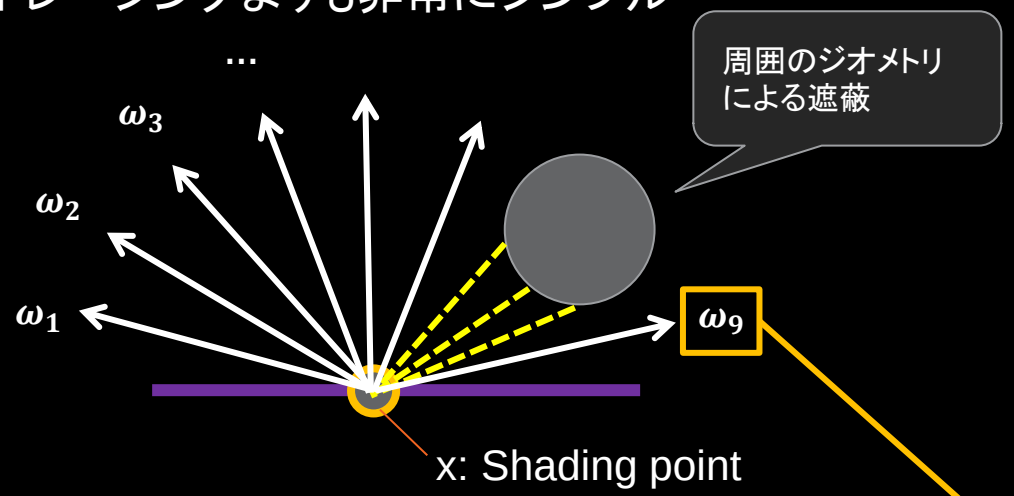
$$\Leftrightarrow \begin{aligned} 0 &\leq N \cdot \{(v_1 - v_0) \times (R(t) - v_0)\} \\ 0 &\leq N \cdot \{(v_2 - v_1) \times (R(t) - v_1)\} \\ 0 &\leq N \cdot \{(v_0 - v_2) \times (R(t) - v_2)\} \end{aligned}$$



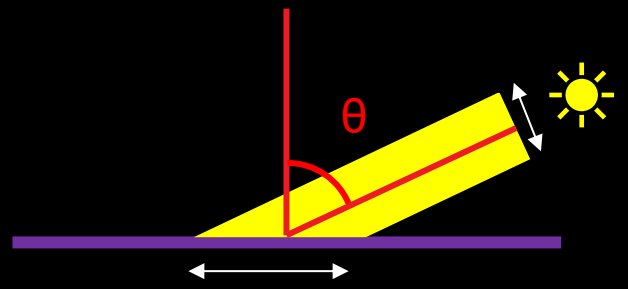
# Ambient Occlusion

# Ambient Occlusion

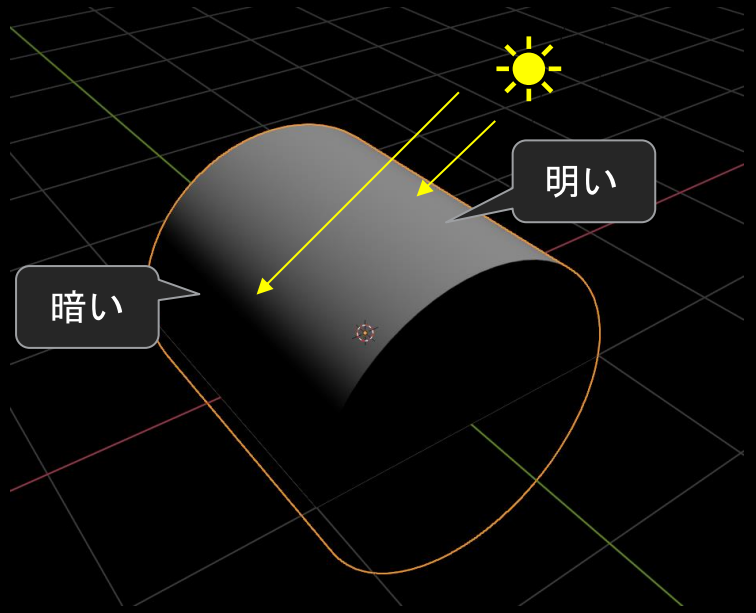
- その点がどの程度遮蔽物に囲まれているか
  - 近似的な陰影づけ
  - ローカルな遮蔽のみを考慮
  - パストレーシングよりも非常にシンプル




ランベルトのコサイン則



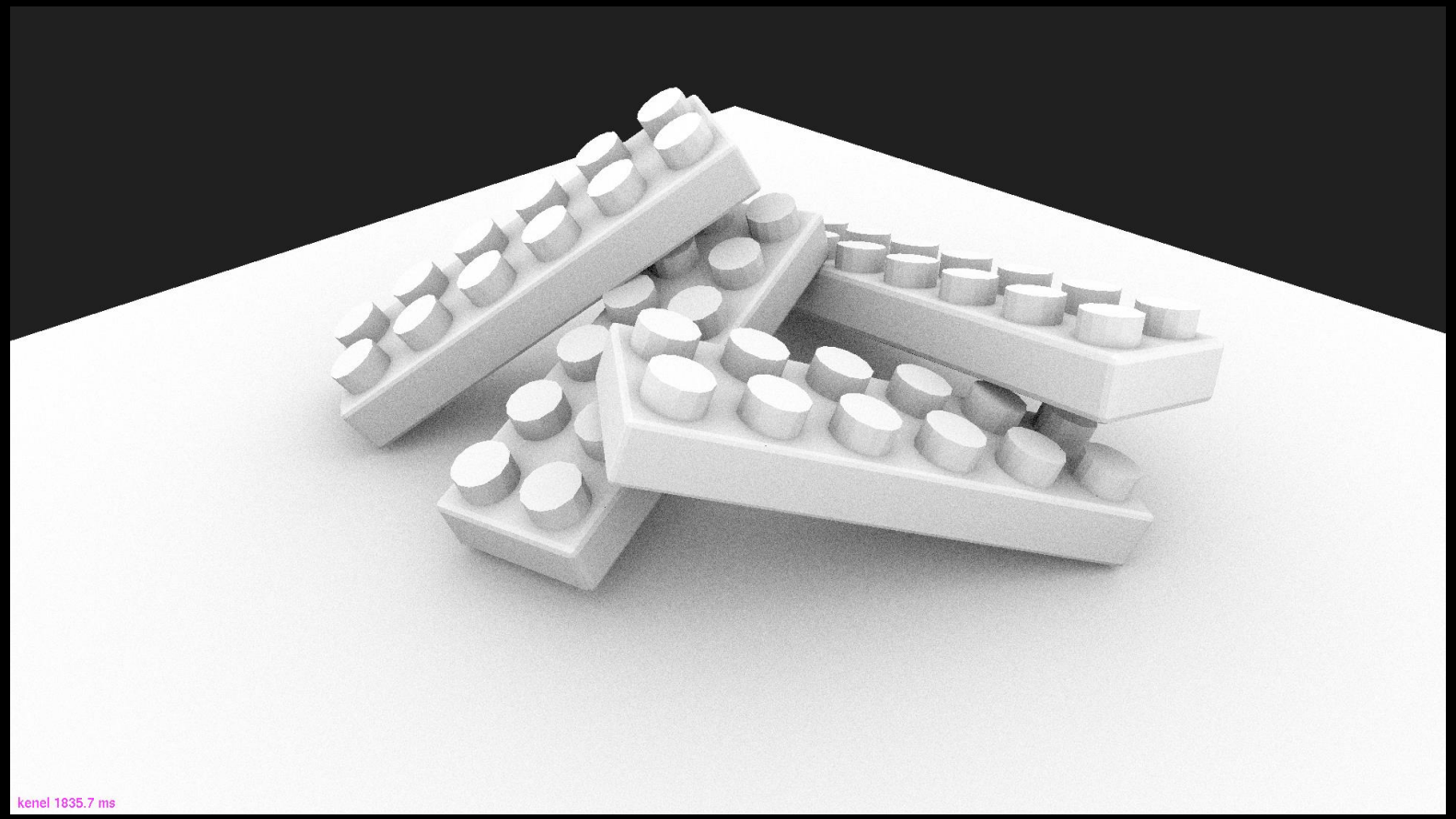

入射角度により表面に届く光の密度が下がる




$\omega$ を cosθ に比例するように選択

# Ambient Occlusion

- 1ピクセルあたり 64 レイ
  - 3600 ポリゴン, 総当たり
  - フルHD で1フレーム 2 秒ほど, Radeon™ RX 7900XTX ☹

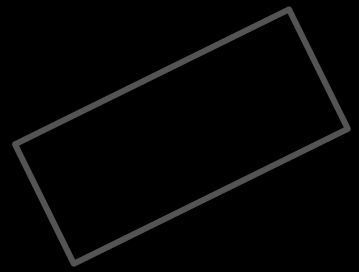


# Bounding Volume

# Bounding Volume

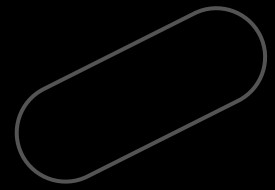
- より単純な図形での早期カリング
  - AABB( Axis-Aligned Bounding Box ) がレイトレーシングではよくつかわれる
    - 単純な交差判定
    - 安価な衝突判定コスト

AABB

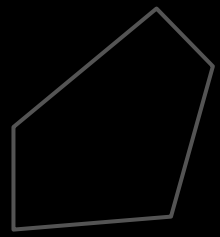


OBB

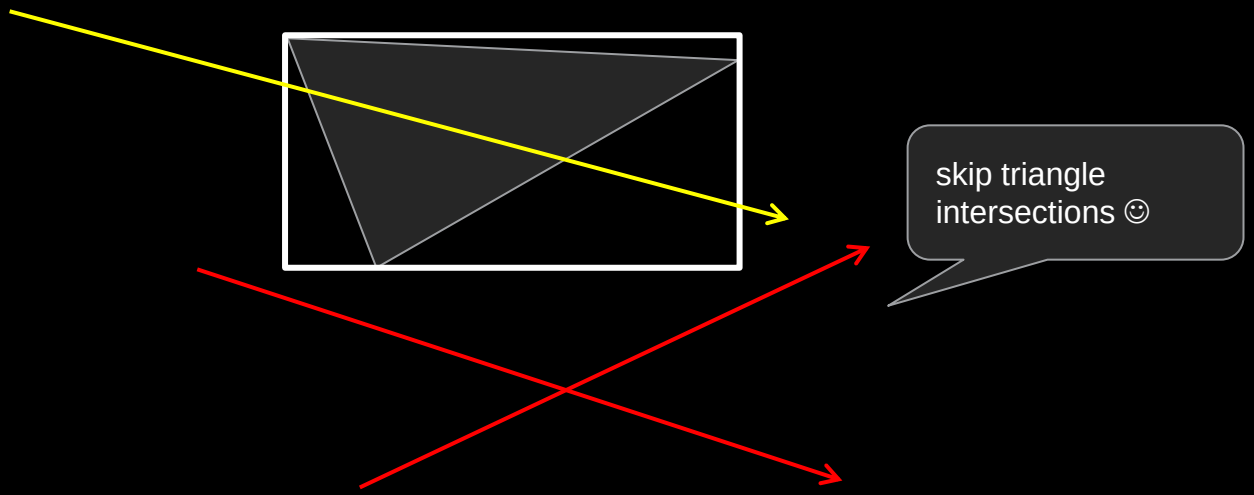
K-DOPs



Capsule

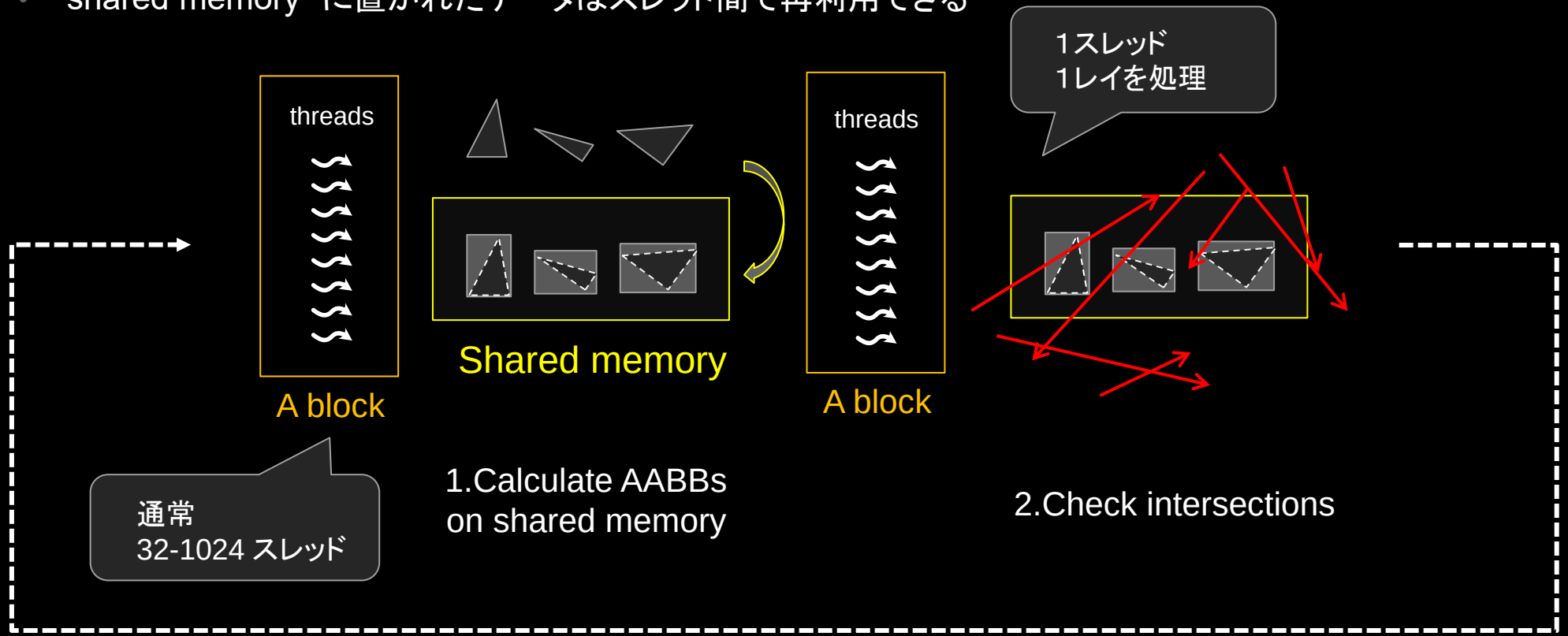


Convex Hull

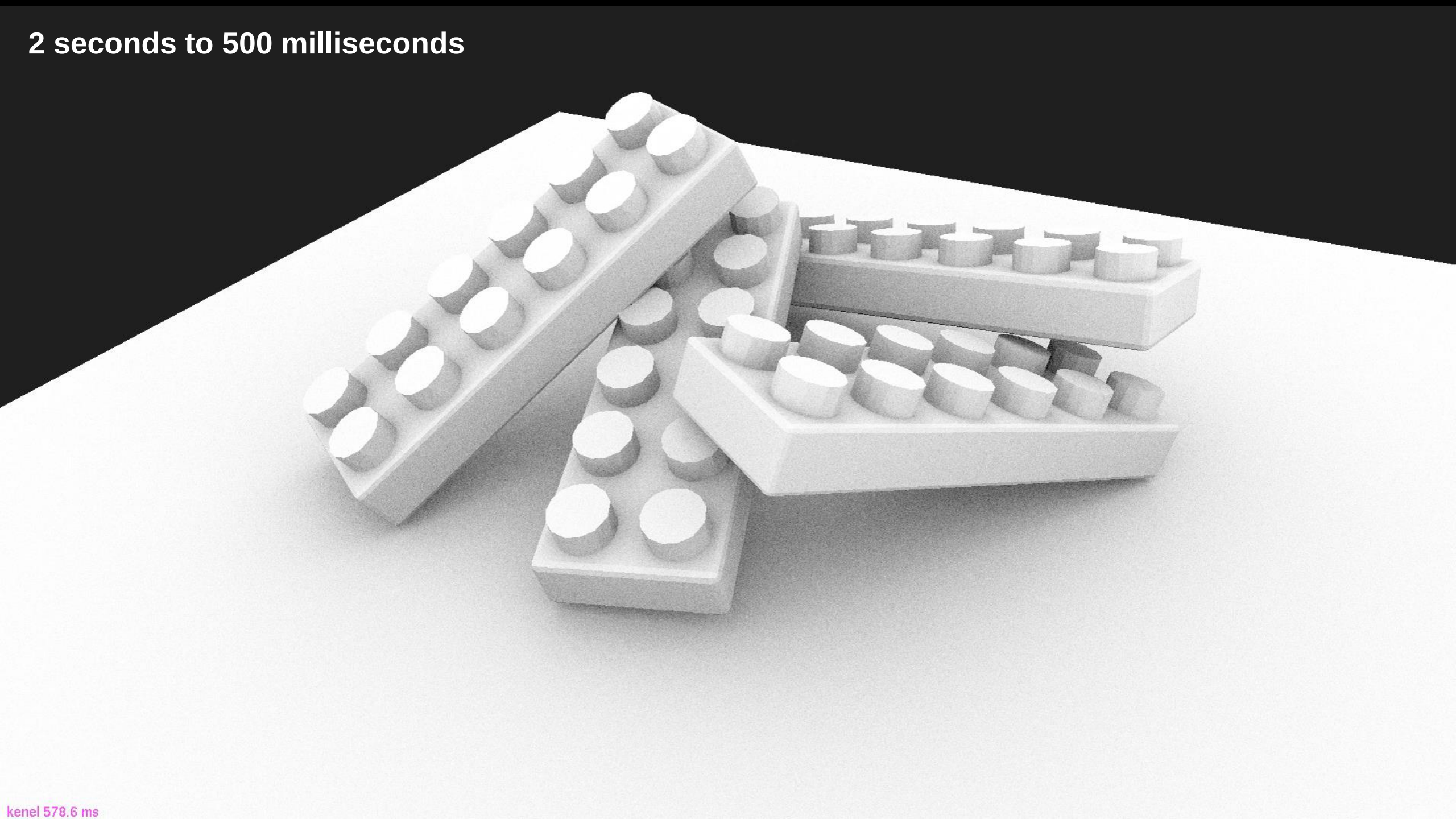


# Bounding Volume

- オンザフライでの AABB の計算（GPU向け）
  - それぞれのスレッドで独立してすべての三角形のAABBを計算するのは無駄が多い
  - 小さいものの高速なメモリ - “shared memory” が HIP/CUDA では利用可能
  - “shared memory” に置かれたデータはスレッド間で再利用できる

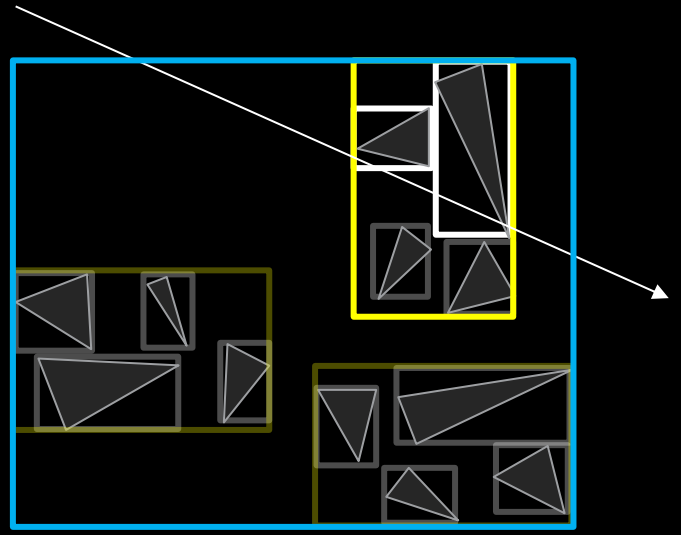


# 2 seconds to 500 milliseconds



kenel 578.6 ms

# 高速化の追求



- 追加のデータ構造による高速化の可能性
- しかしながら、先の実験には数多くの最適化の余地がある
  - BVH - Bounding Volume Hierarches
  - よりAABBを小さくするには？
    - ジオメトリの分布を考慮
    - 近いもの同士を同じAABBに
  - データ構造そのものの構築コストを抑えるには？
    - リアルタイムのアプリケーションでは、ジオメトリにアニメーションが含まれることもある
    - 再構築

➡ 多くの研究と成果

For the enthusiastic listener:

Meister et al., “A Survey on Bounding Volume Hierarchies for Ray Tracing”, EG 2021

Benthin et al., “H-PLOC Hierarchical Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction”, HPG 2024

# ライブラリを使う - HIPRT

# HIPRT

- 汎用レイトレーシングライブラリ
  - クロスプラットフォーム – AMD GPU / CUDA devices
  - AMD GPUでのハードウェアレイトレーシングのサポート
  - **シンプルで使いやすいAPI**
    - SBTの設定などの複雑性を回避
- Shared Memoryをスタックメモリに利用推奨
  - BVH トラバーサル用

```
hiprtTriangleMeshPrimitive mesh = {};
mesh.triangleCount = ...
mesh.vertexCount = ...
mesh.vertexStride = ...
mesh.vertices = ...

hiprtGeometryBuildInput geomInput = {};
geomInput.type = hiprtPrimitiveTypeTriangleMesh;
geomInput.primitive.triangleMesh = mesh;

...

hiprtGeometry geom = 0;
hiprtCreateGeometry(hContext, geomInput, buildOptions, geom);
hiprtBuildGeometry(hContext, hiprtBuildOperationBuild, geomInput,
                   buildOptions, geomTemp.data(), 0 /*stream*/, geom);
```

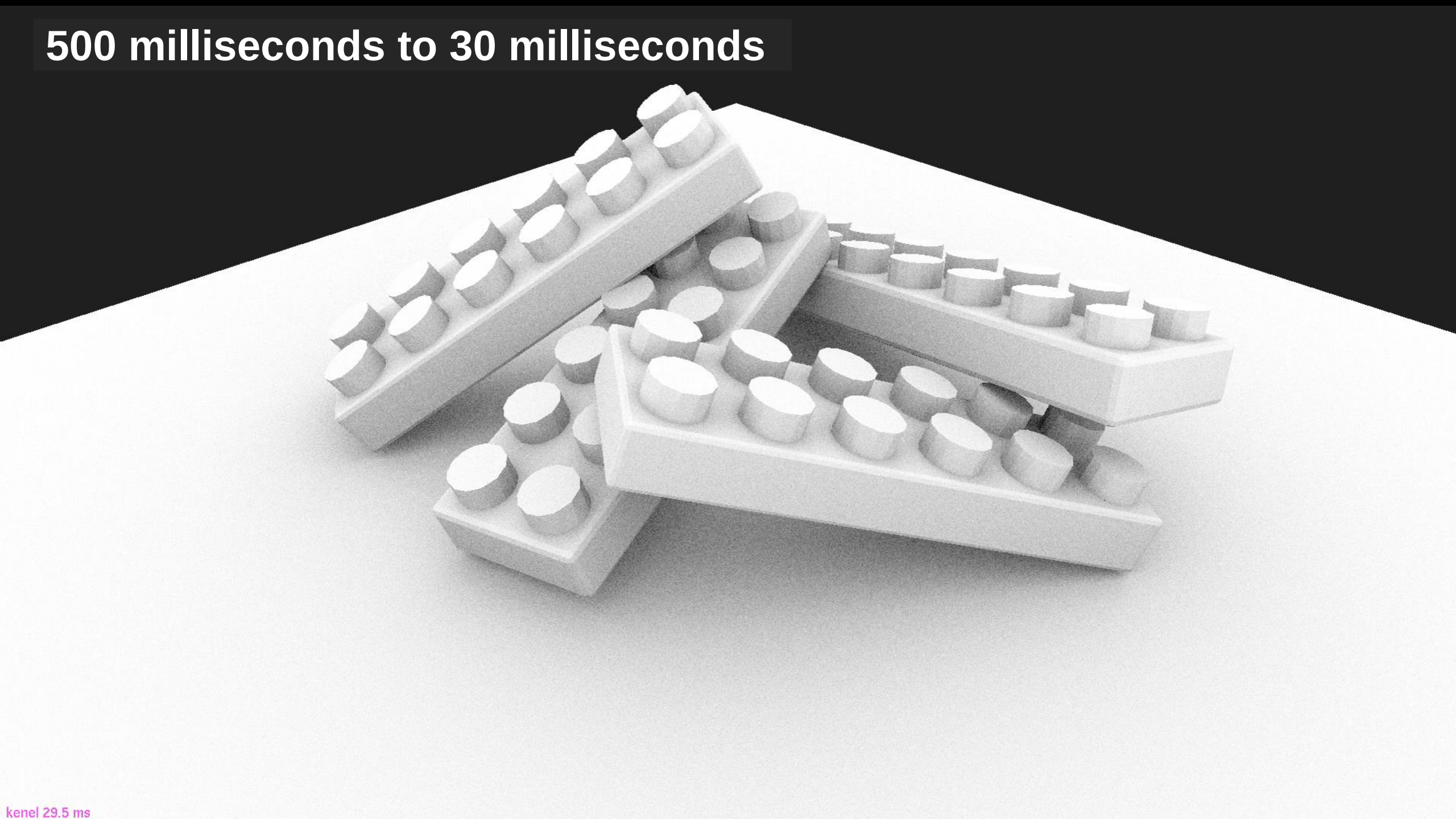
Prepare acceleration structure

```
#define SHARED_STACK_SIZE 32
#define BLOCK_SIZE 256

__device__ bool hiprt_closeset_hit(I
                                   float
{
    __shared__ int s_mem[SHARED_STACK_SIZE * BLOCK_SIZE];
    hiprtGlobalStack stack(
        {}, hiprtSharedStackBuffer{SHARED_STACK_SIZE, (void*)s_mem});

    hiprtRay ray;
    ray.origin = …;
    ray.direction = …;
    hiprtGeomTraversalClosestCustomStack<hiprtGlobalStack> tr(geom, ray,
                                                              stack);

    hiprtHit hit = tr.getNextHit();

    if (hit.hasHit() == false) { return false; }
    intersection->t = hit.t;
    intersection->index = hit.primID;
    return true;
}
```

Use shared memory as stack memory

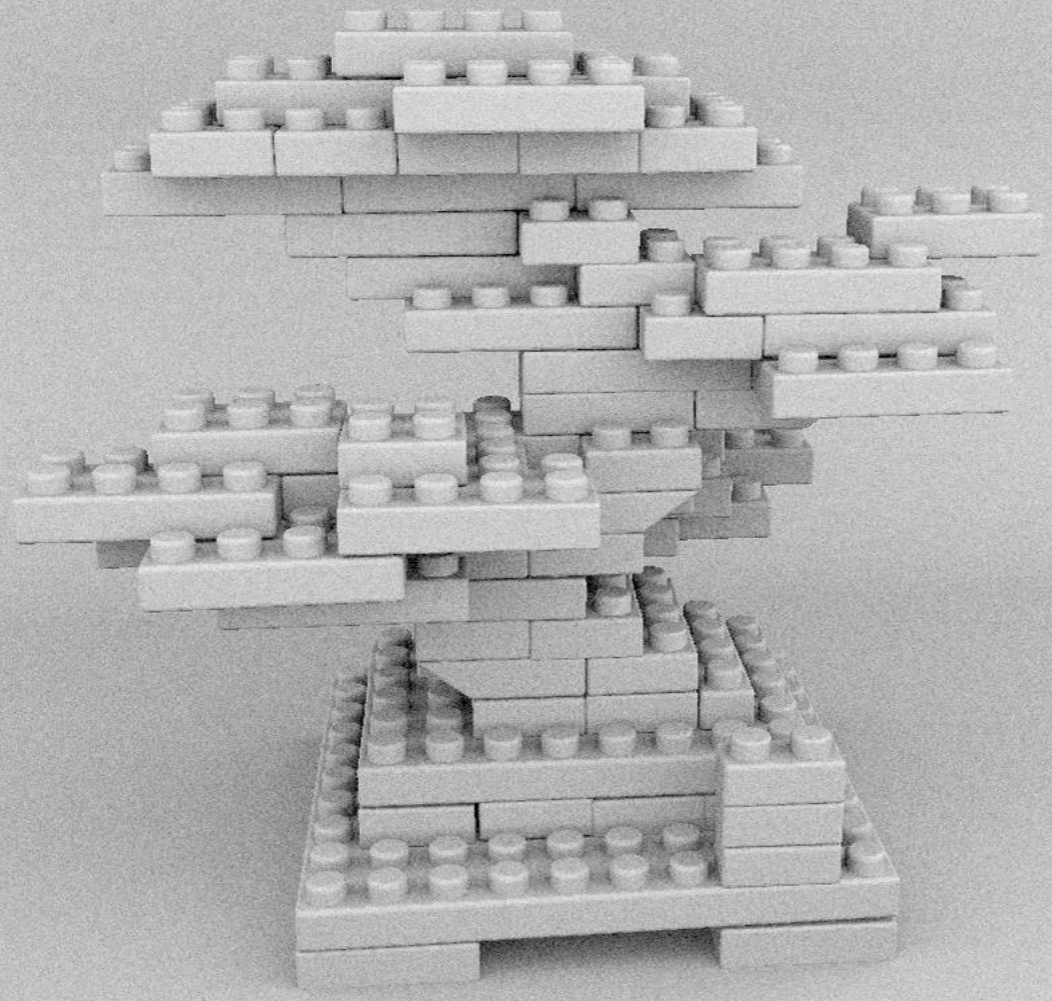
Ray casting

# 500 milliseconds to 30 milliseconds



kenel 29.5 ms

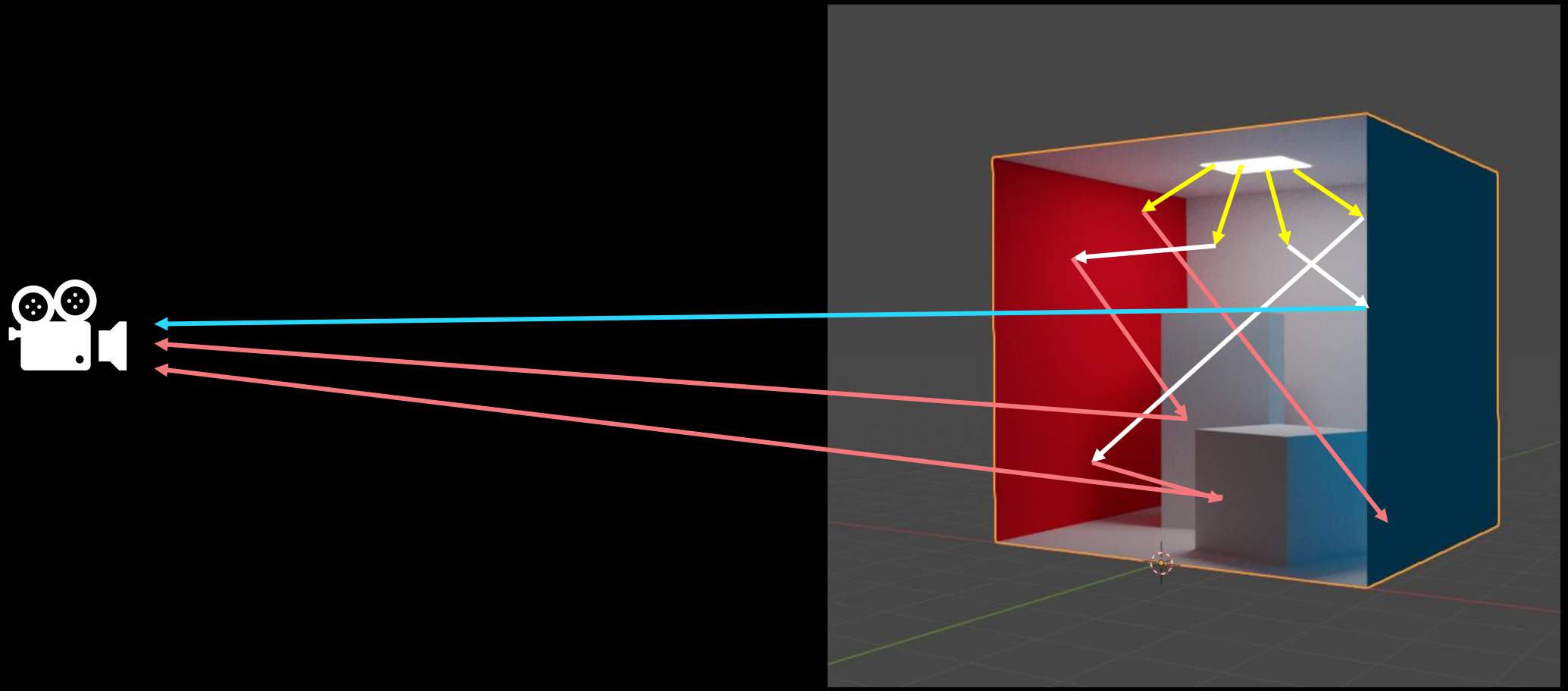
# 850k triangles, 55 – 60 milliseconds



kenel 49.8 ms

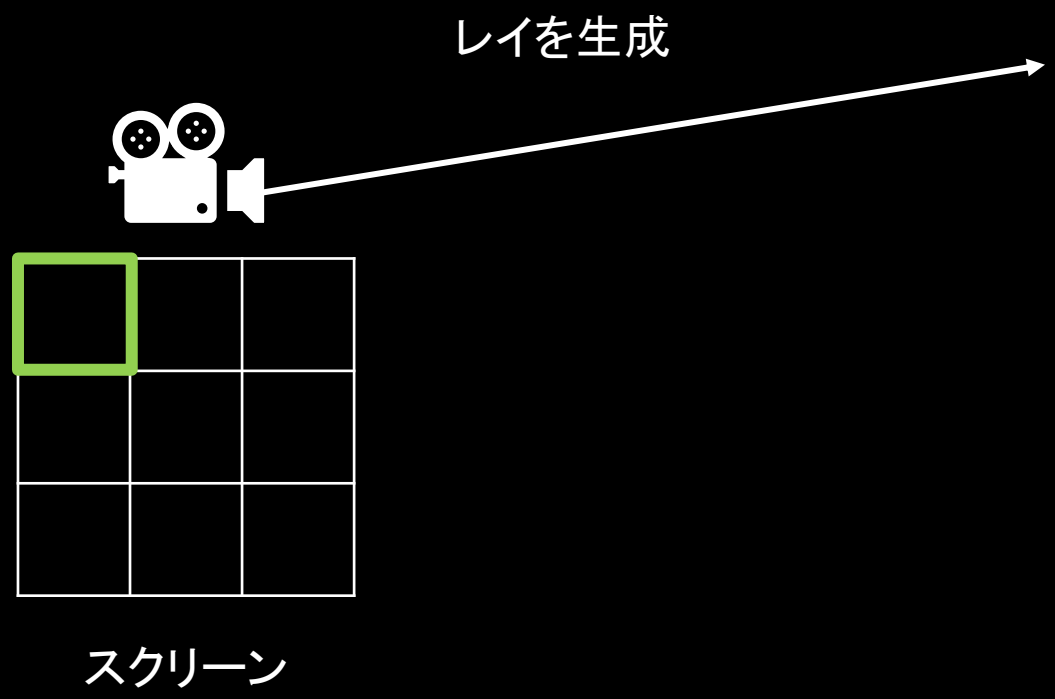
# パストレーシング

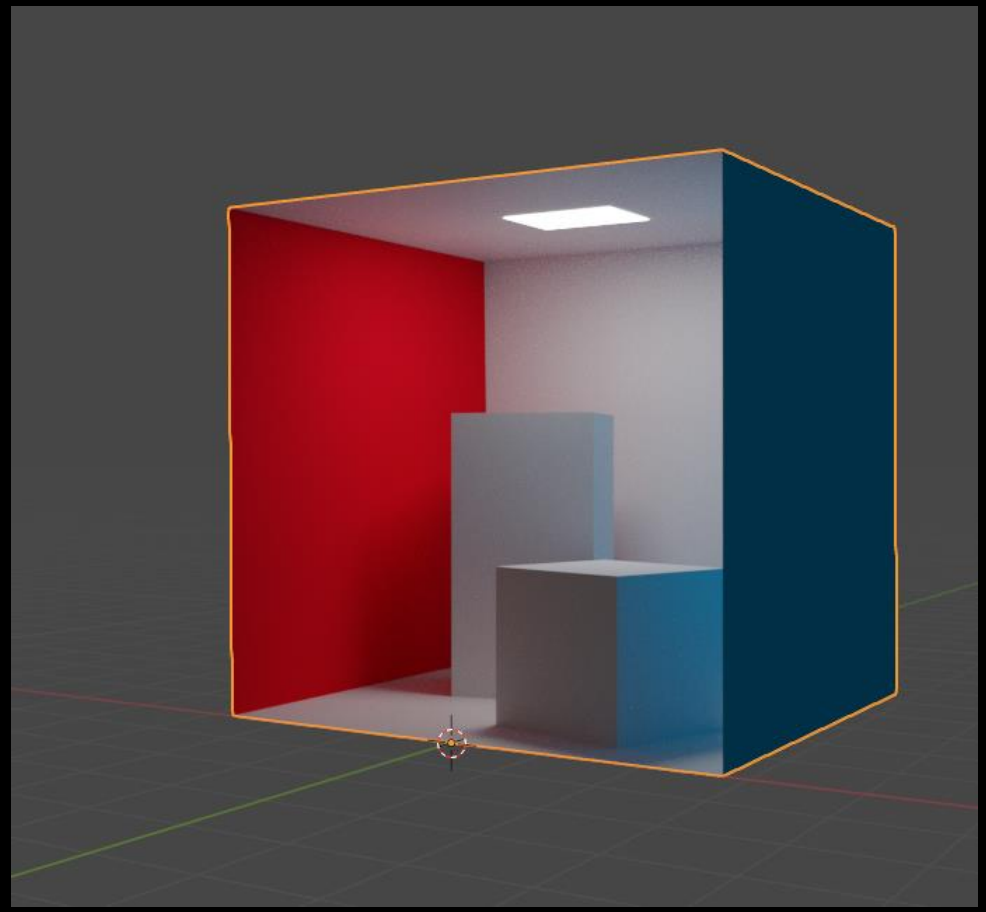
# 光の伝達



# パストレーシング

throughput = (1.0,1.0,1.0)

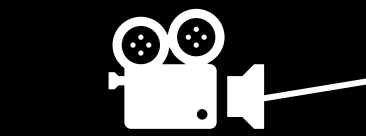


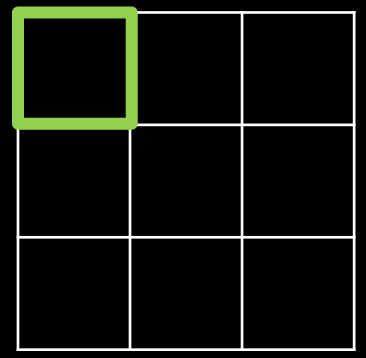



# パストレーシング

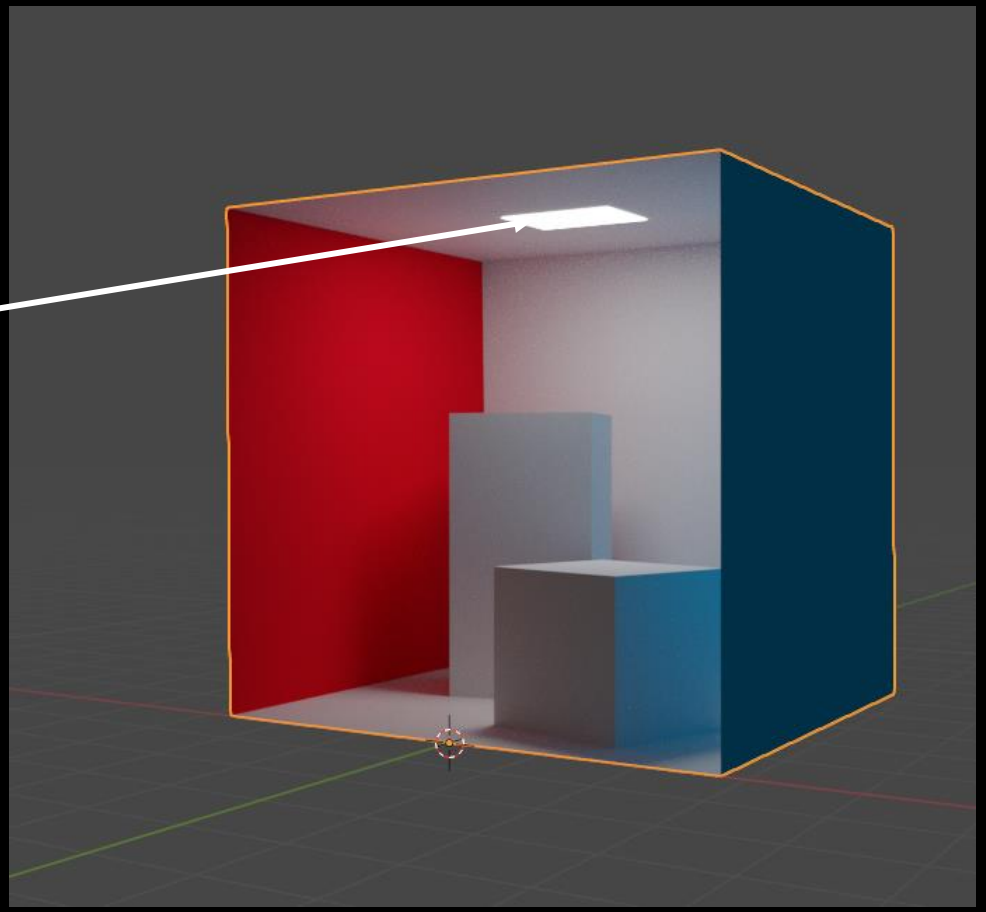
throughput = (1.0,1.0,1.0)

レイトレーシング



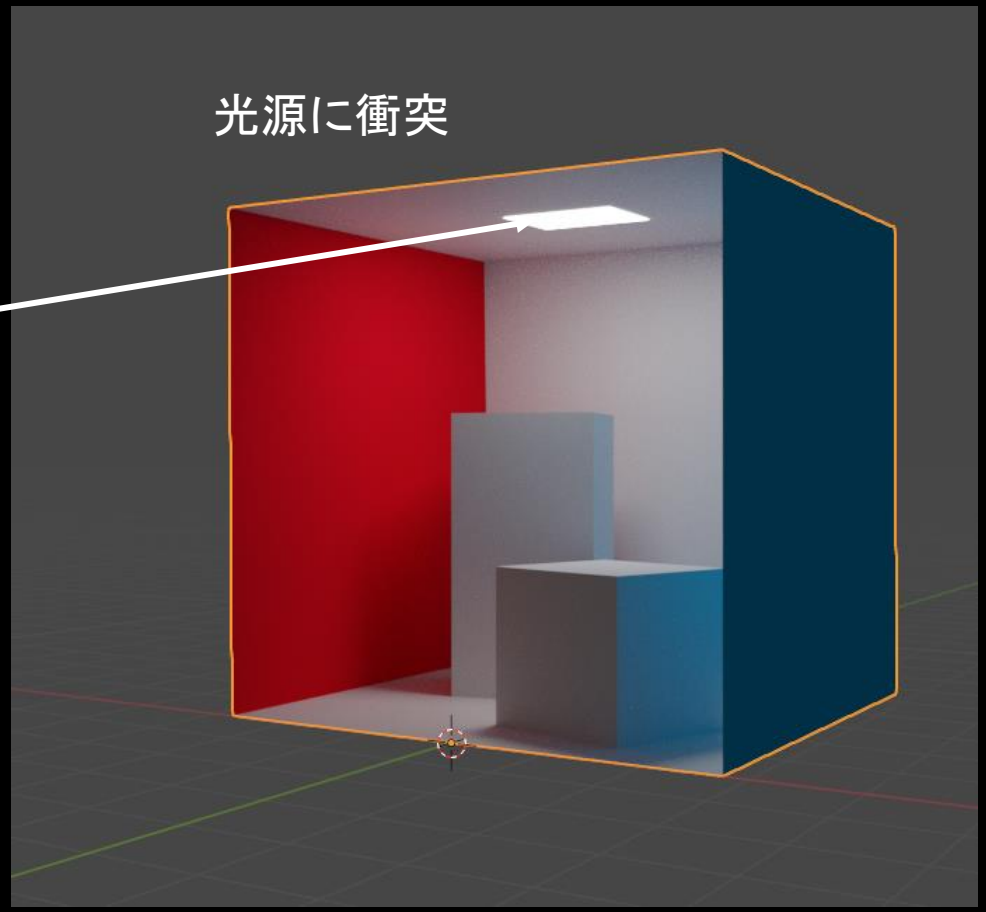


スクリーン

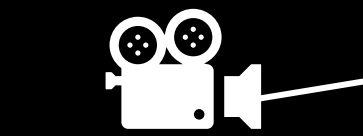


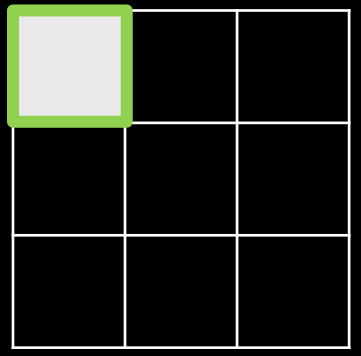
# パストレーシング

throughput = (1.0,1.0,1.0)



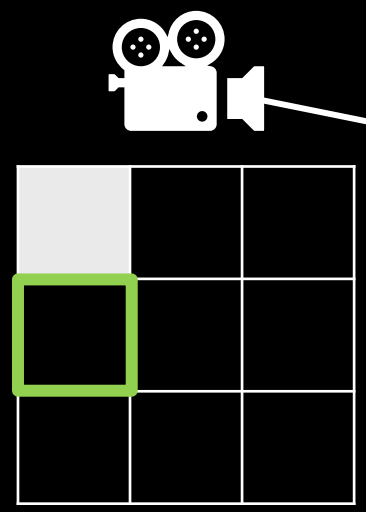

throughput * 光源の明るさ



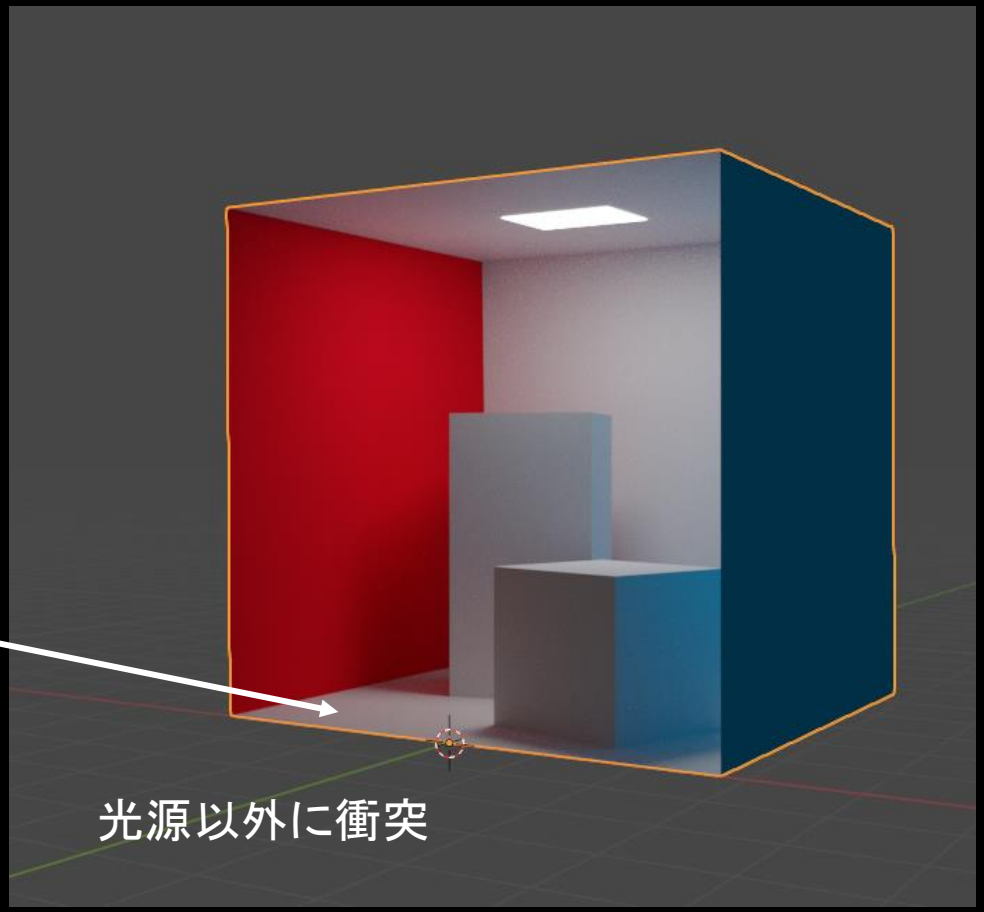


スクリーン

# パストレーシング



スクリーン

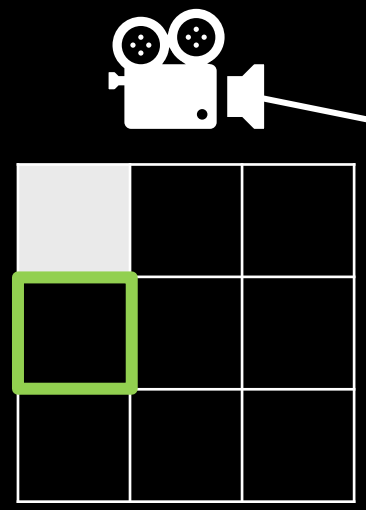





AMD

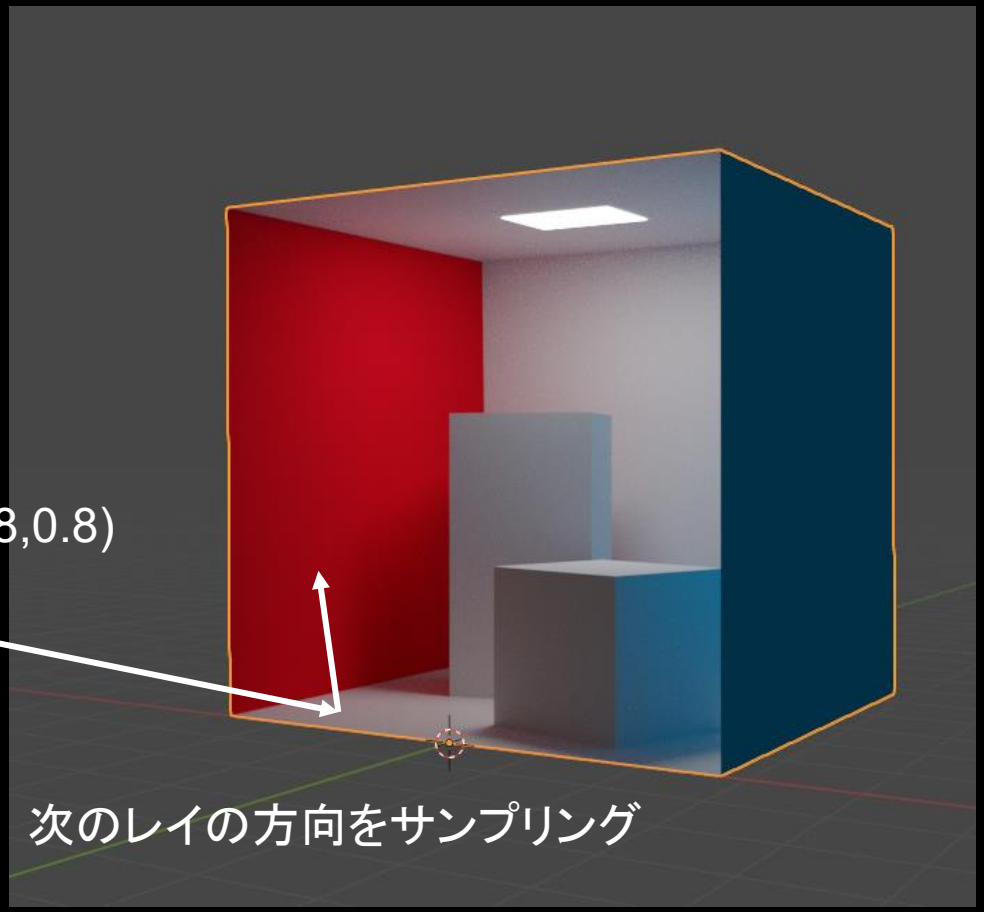
# パストレーシング

throughput = (0.8,0.8,0.8)



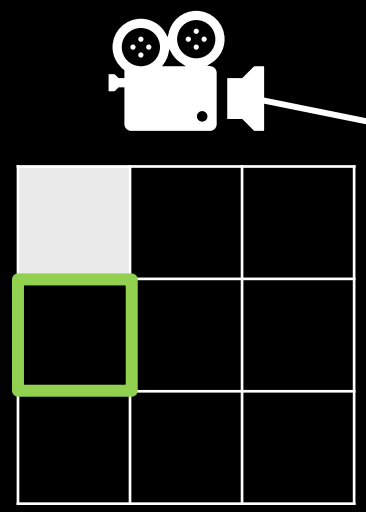
スクリーン

反射率: (0.8,0.8,0.8)

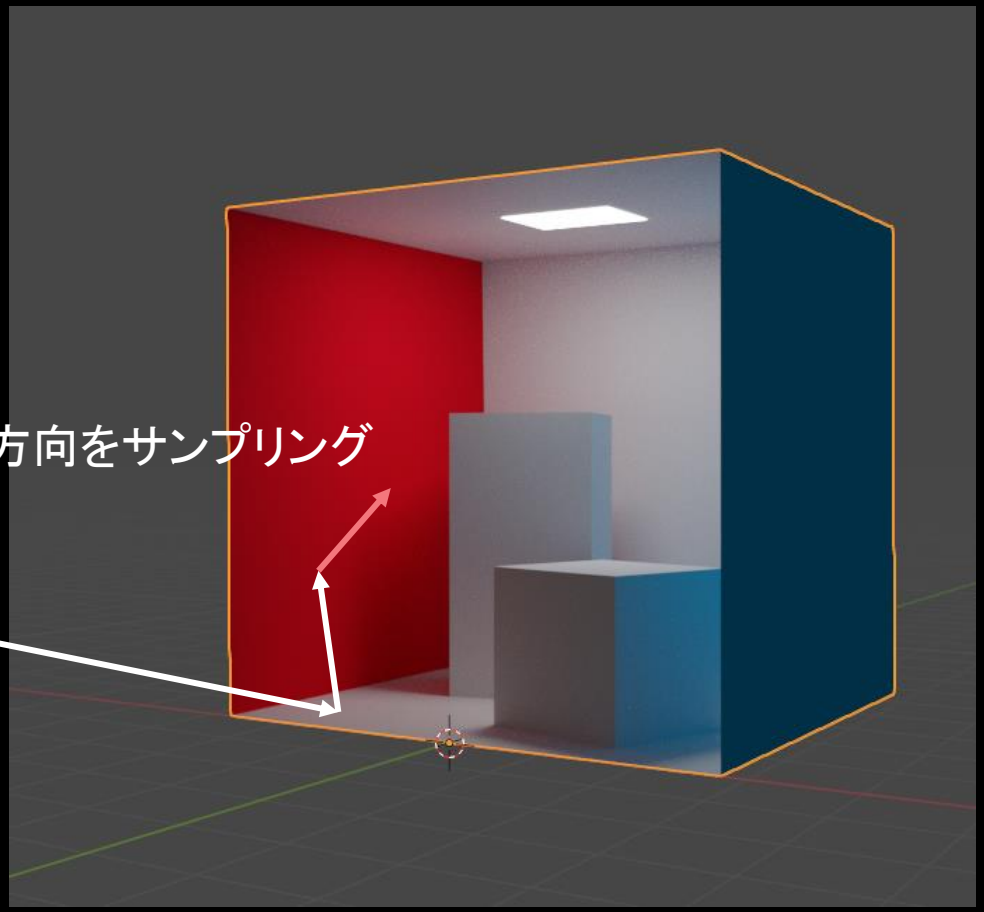


# パストレーシング

throughput = (0.8,0.8,0.8) * (1.0, 0.2, 0.2)

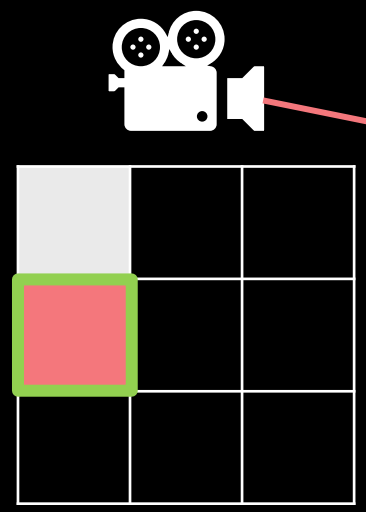


スクリーン



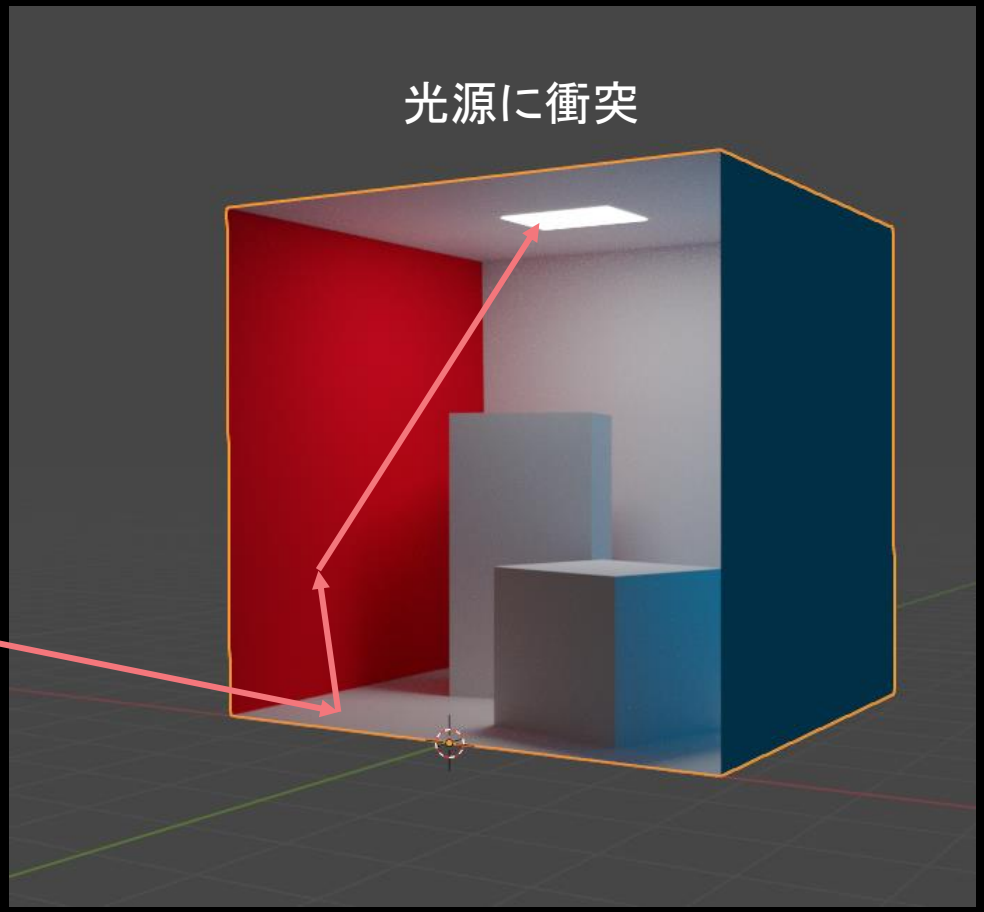
# パストレーシング

throughput = (0.8,0.8,0.8) * (1.0, 0.2, 0.2)



スクリーン

throughput * 光源の明るさ



# 実装例

- 各ピクセルで右のようなfor文を実行

```
// 初期値をセット
Ray ray = make_ray(ro, rd);
float3 radiance = {0.0f, 0.0f, 0.0f};
float3 throughput = {1.0f, 1.0f, 1.0f};

// パストレーシングのループ
for (int depth = 0; depth < options.max_depth; ++depth) {
    // レイトレース
    Intersection isect;
    if (!raytrace(ray, hiprt_geom, isect)) {
        // 空に当たった場合
        radiance += throughput * options.sky_color;
        break;
    }

    const Triangle& hit_triangle = triangles[isect.index];
    if (has_emission(hit_triangle)) {
        // 光源に当たった場合
        radiance += throughput * triangles[isect.index].emissive;
        break;
    }

    // 交差点における法線などを計算
    const SurfaceInfo surf = make_surface_info(ray, isect, triangles);

    // 次のレイの方向を生成
    float3 wo; {
        // …
    }

    // スループットを更新
    throughput *= hit_triangle.color;

    // 次のレイを生成
    ray = make_ray(offset_ray_position(surf.p, surf.n), wo);
}
```

# 実装例

- 各ピクセルで右のようなfor文を実行

```
 // 初期値をセット
Ray ray = make_ray(ro, rd);
float3 radiance = {0.0f, 0.0f, 0.0f};
float3 throughput = {1.0f, 1.0f, 1.0f};

// パストレーシングのループ
for (int depth = 0; depth < options.max_depth; ++depth) {
    // レイトレース
    Intersection isect;
    if (!raytrace(ray, hiprt_geom, isect)) {
        // 空に当たった場合
        radiance += throughput * options.sky_color;
        break;
    }

    const Triangle& hit_triangle = triangles[isect.index];
    if (has_emission(hit_triangle)) {
        // 光源に当たった場合
        radiance += throughput * triangles[isect.index].emissive;
        break;
    }

    // 交差点における法線などを計算
    const SurfaceInfo surf = make_surface_info(ray, isect, triangles);

    // 次のレイの方向を生成
    float3 wo; {
        // …
    }

    // スループットを更新
    throughput *= hit_triangle.color;

    // 次のレイを生成
    ray = make_ray(offset_ray_position(surf.p, surf.n), wo);
}
```

# 実装例

- 各ピクセルで右のようなfor文を実行

```
// 初期値をセット
Ray ray = make_ray(ro, rd);
float3 radiance = {0.0f, 0.0f, 0.0f};
float3 throughput = {1.0f, 1.0f, 1.0f};

// パストレーシングのループ
for (int depth = 0; depth < options.max_depth; ++depth) {
    // レイトレース
    Intersection isect;
    if (!raytrace(ray, hiprt_geom, isect)) {
        // 空に当たった場合
        radiance += throughput * options.sky_color;
        break;
    }

    const Triangle& hit_triangle = triangles[isect.index];
    if (has_emission(hit_triangle)) {
        // 光源に当たった場合
        radiance += throughput * triangles[isect.index].emissive;
        break;
    }

    // 交差点における法線などを計算
    const SurfaceInfo surf = make_surface_info(ray, isect, triangles);

    // 次のレイの方向を生成
    float3 wo; {
        // …
    }

    // スループットを更新
    throughput *= hit_triangle.color;

    // 次のレイを生成
    ray = make_ray(offset_ray_position(surf.p, surf.n), wo);
}
```

# 実装例

- 各ピクセルで右のようなfor文を実行

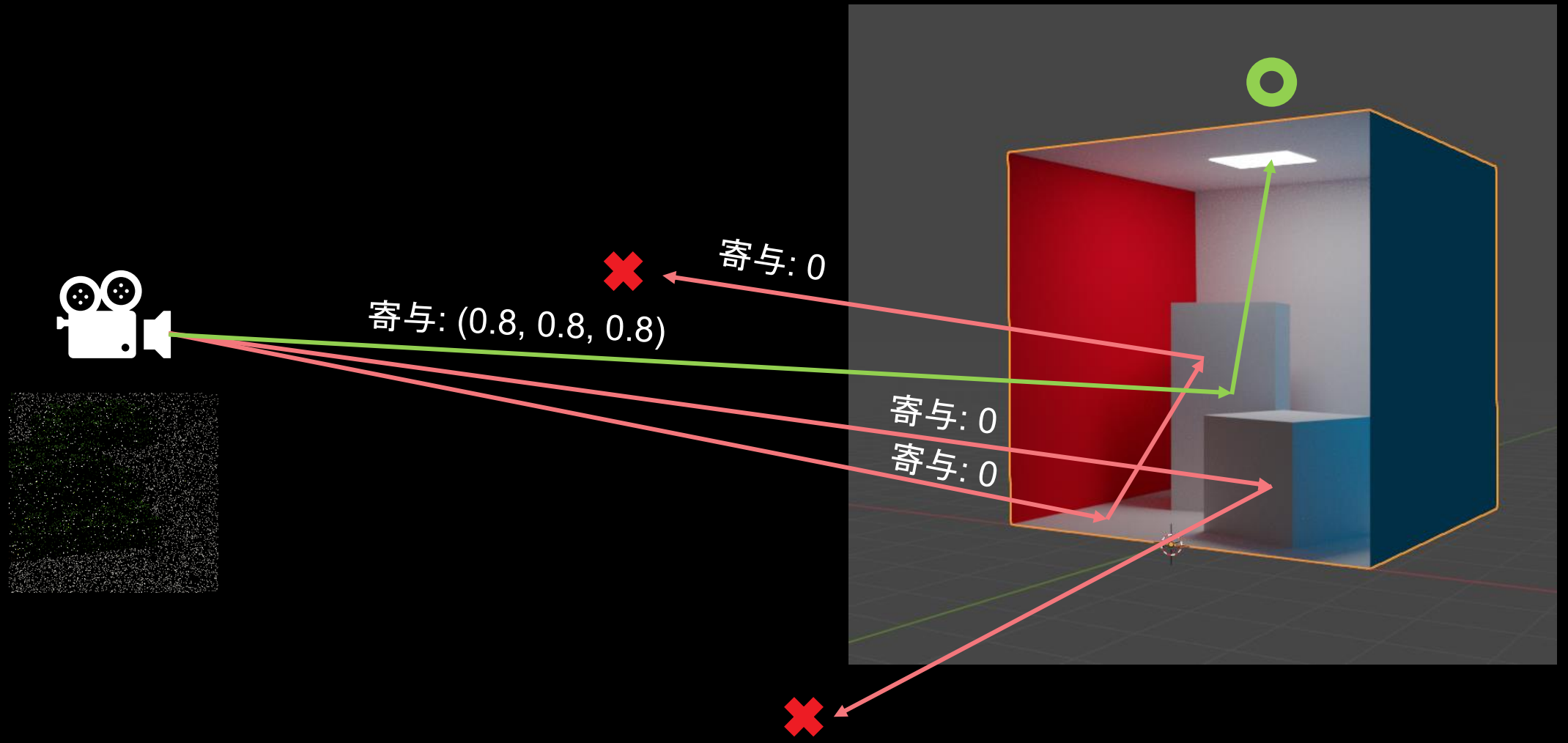
```
// 初期値をセット
Ray ray = make_ray(ro, rd);
float3 radiance = {0.0f, 0.0f, 0.0f};
float3 throughput = {1.0f, 1.0f, 1.0f};

// パストレーシングのループ
for (int depth = 0; depth < options.max_depth; ++depth) {
    // レイトレース
    Intersection isect;
    if (!raytrace(ray, hiprt_geom, isect)) {
        // 空に当たった場合
        radiance += throughput * options.sky_color;
        break;
    }

    const Triangle& hit_triangle = triangles[isect.index];
    if (has_emission(hit_triangle)) {
        // 光源に当たった場合
        radiance += throughput * triangles[isect.index].emissive;
        break;
    }

    // 交差点における法線などを計算
    const SurfaceInfo surf = make_surface_info(ray, isect, triangles);

    // 次のレイの方向を生成
    float3 wo; {
        // …
    }

    // スループットを更新
    throughput *= hit_triangle.color;

    // 次のレイを生成
    ray = make_ray(offset_ray_position(surf.p, surf.n), wo);
}
```

# Result

# Result (1サンプル)

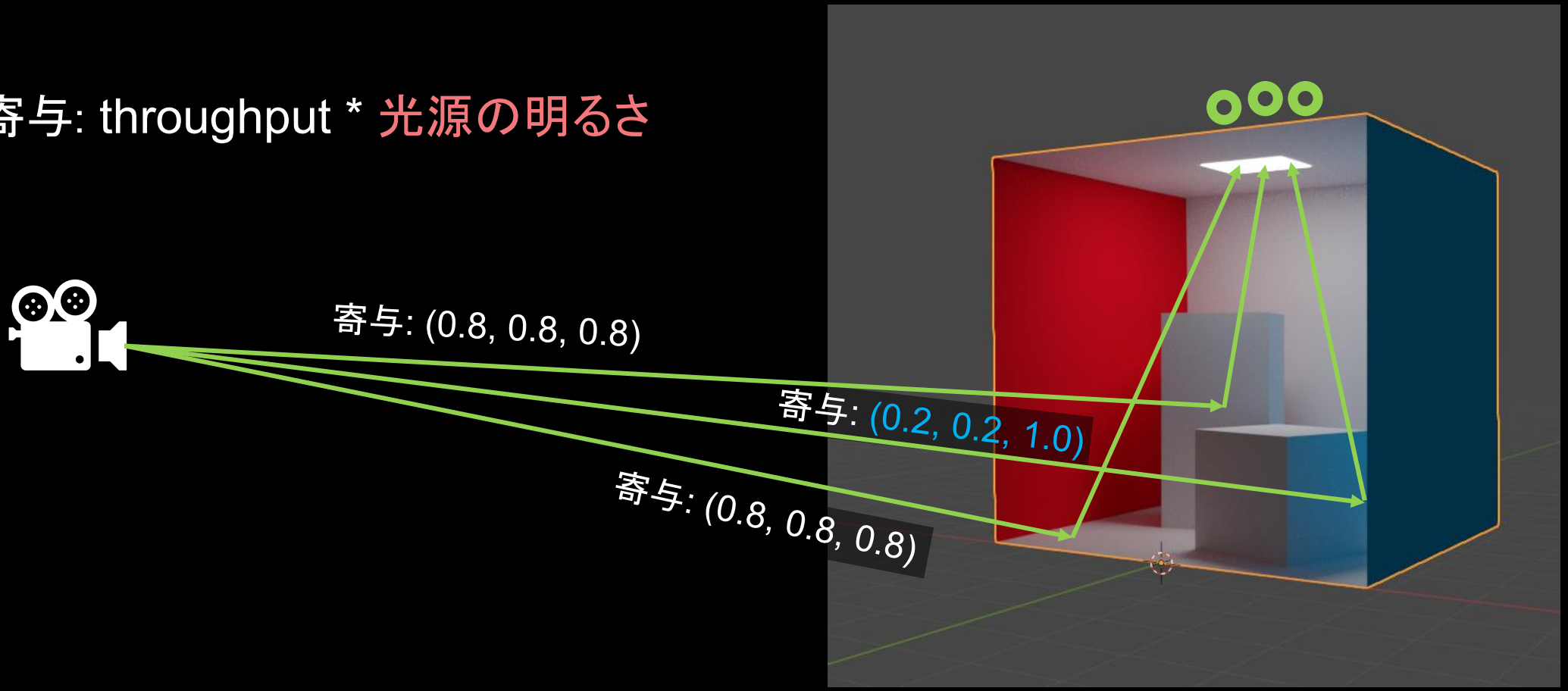
kernel 4.5 ms

# なぜノイズが生まれる?



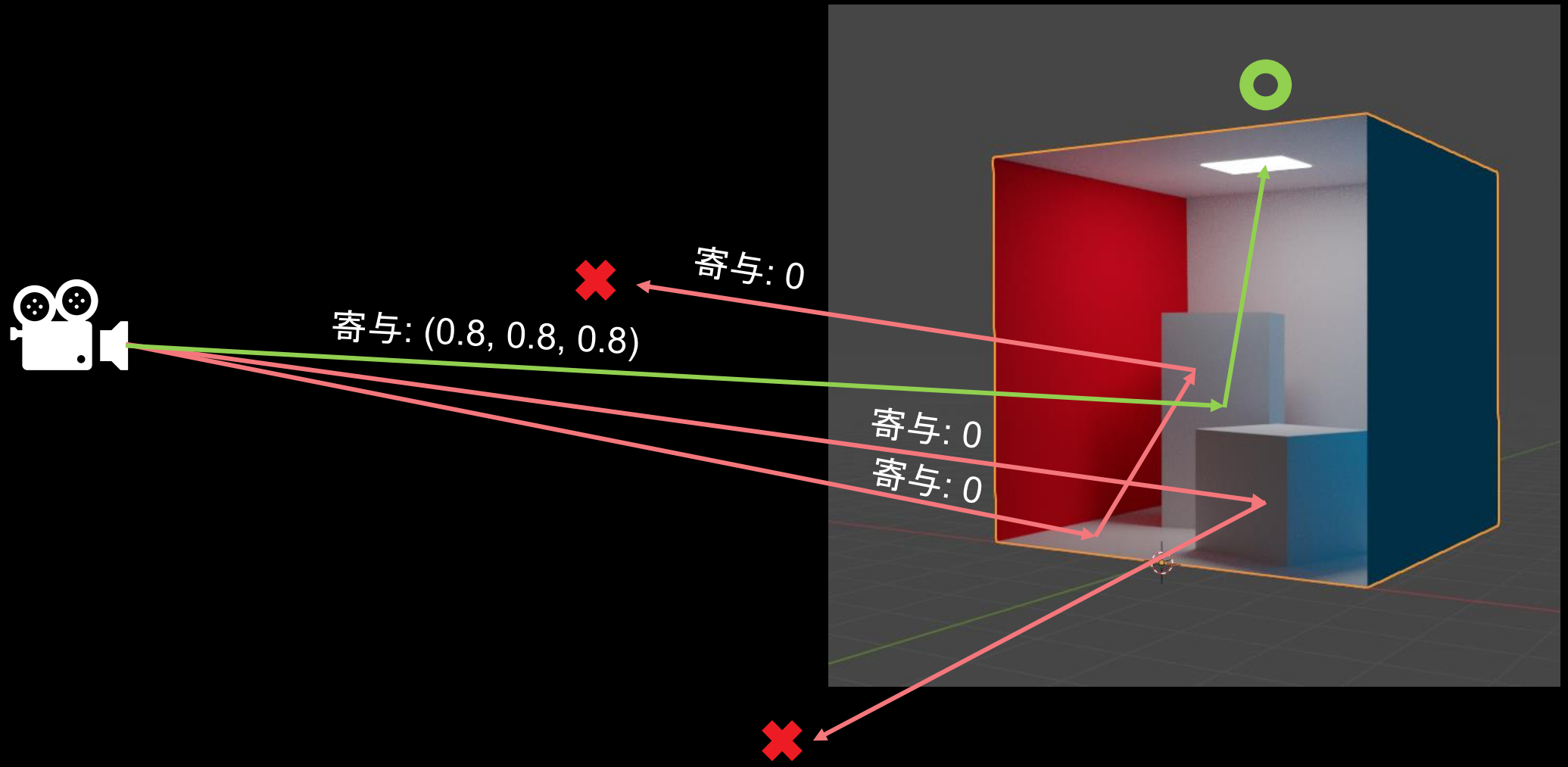
# 重点的サンプリング

経路の寄与: throughput * 光源の明るさ

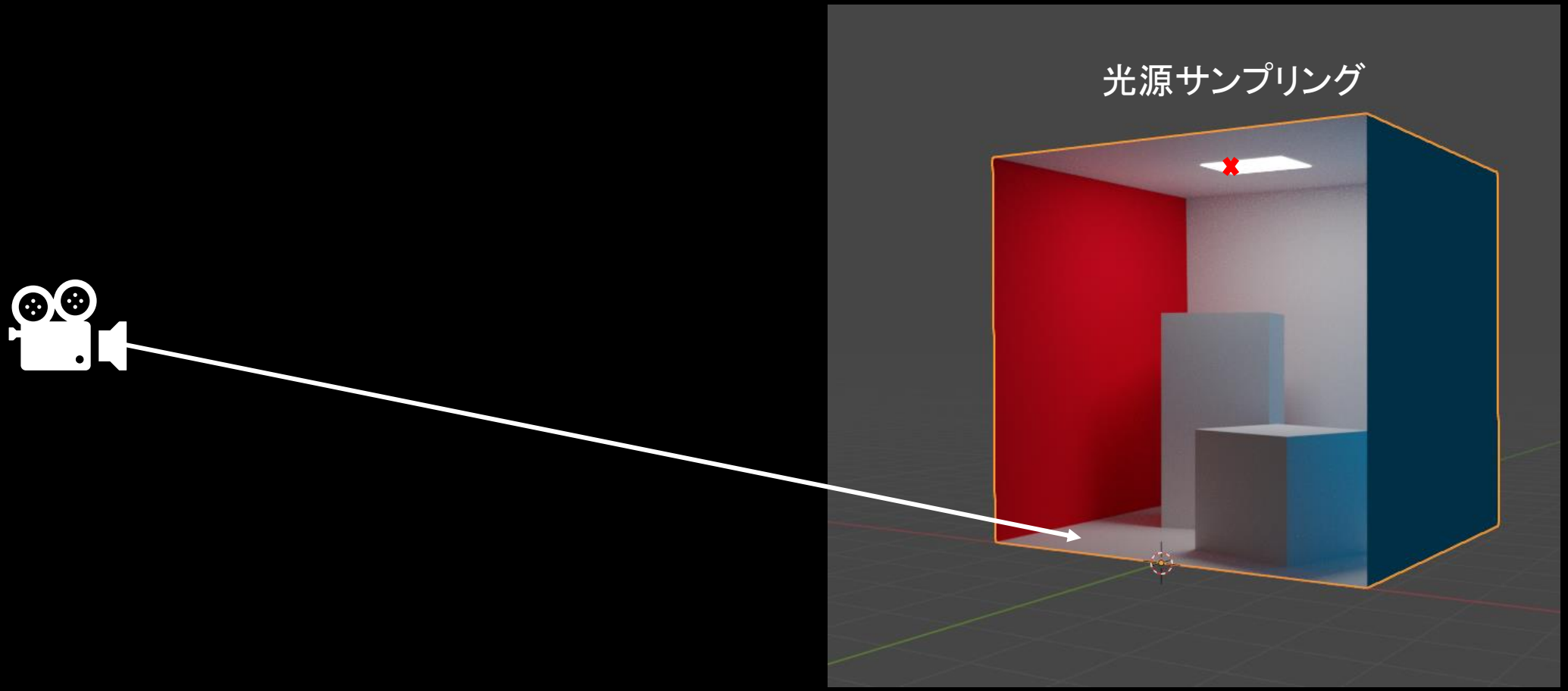


# Next Event Estimation (NEE)

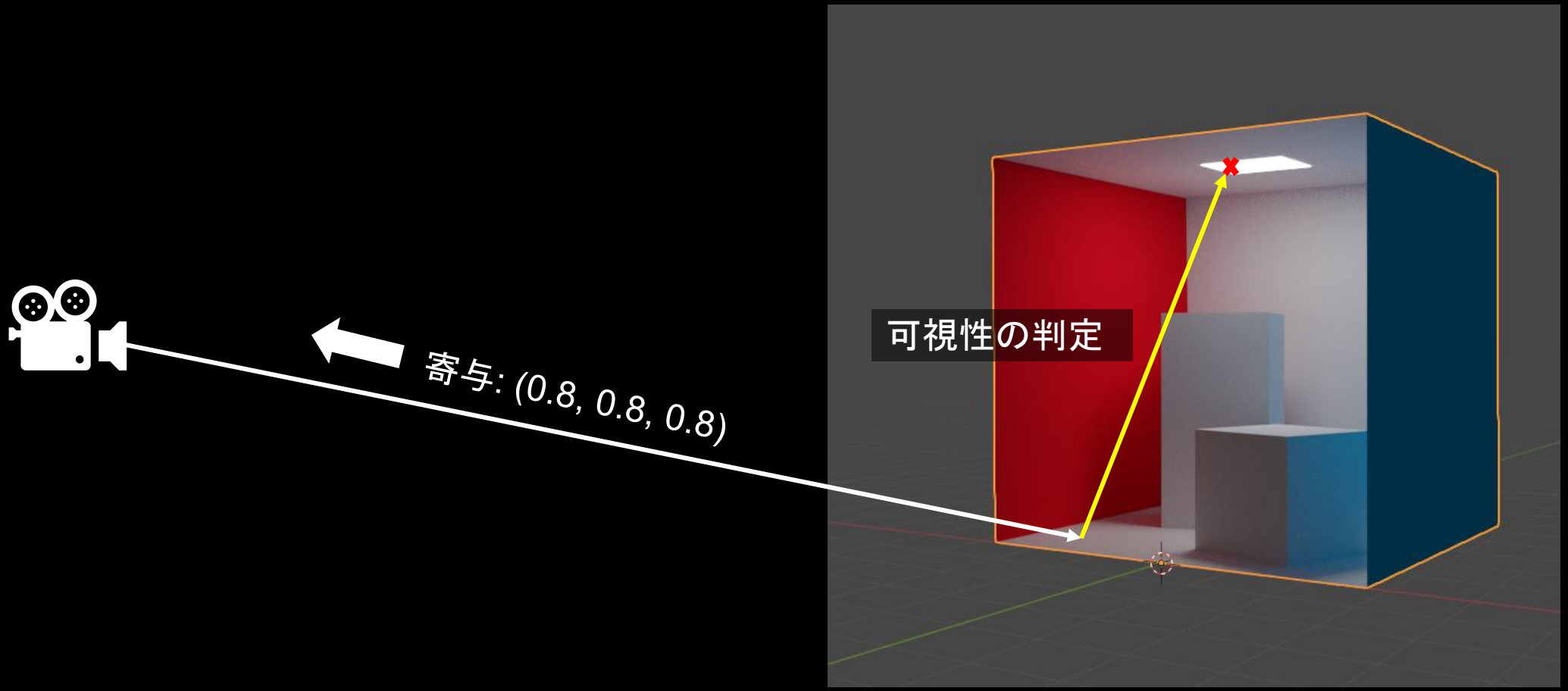
# パストレーシングの問題点



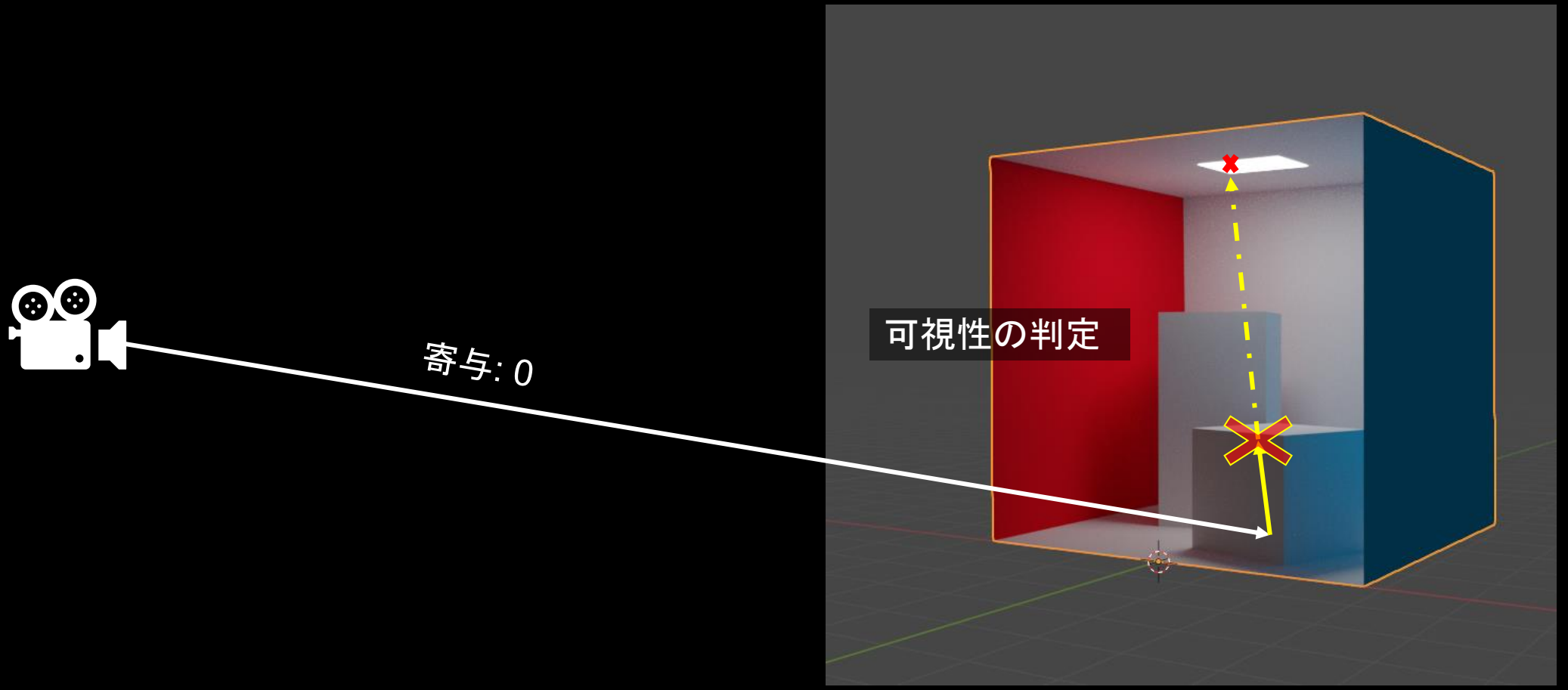
# Next event estimation



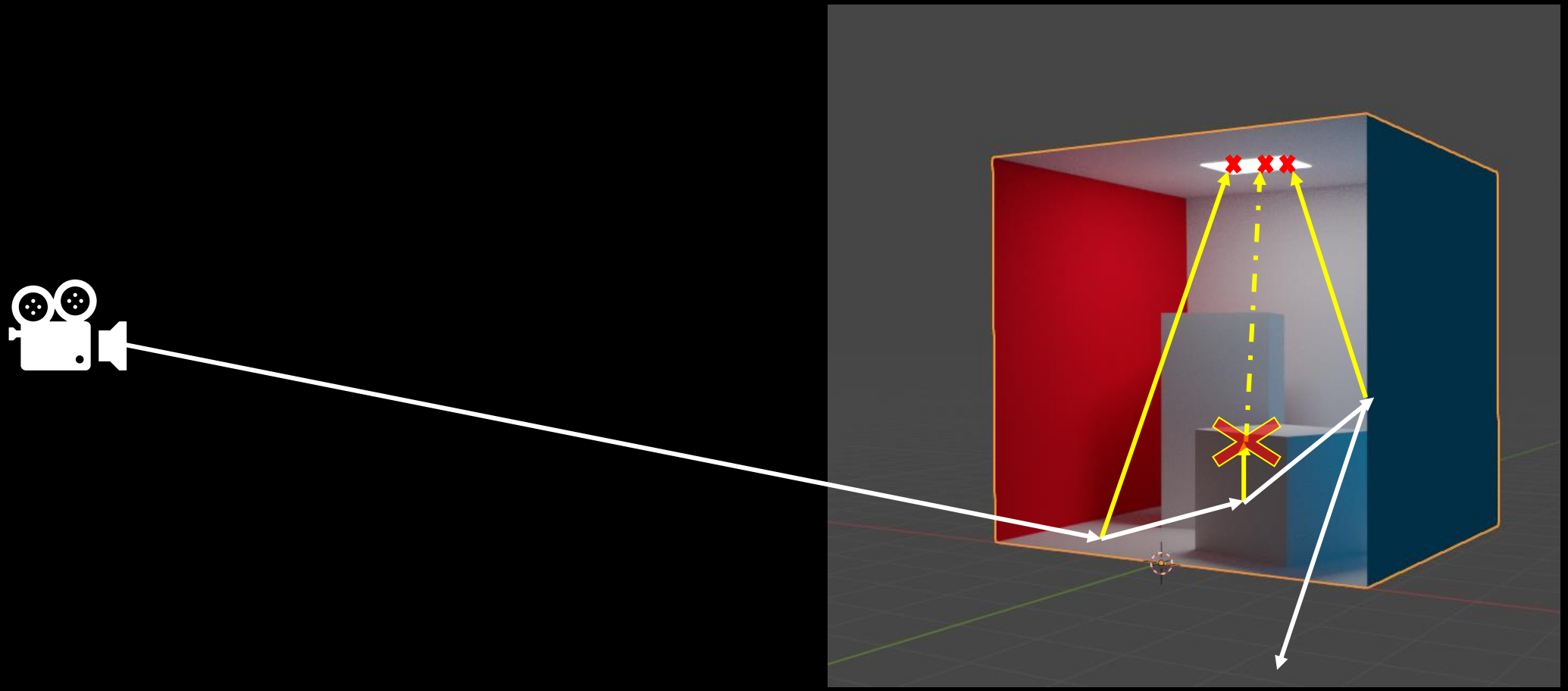
# Next event estimation



# Next event estimation



# Next event estimation



# 実装例

```
// 光源上の点をサンプリング
const LightSample light_sample =
    sample_light(triangles, lights, random.uniformf(),
                 random.uniformf(), random.uniformf());

// 寄与の計算
{
    const Triangle& light_triangle = triangles[light_sample.index];

    // 光源の可視性のチェック
    const float V =
        check_visibility(surf.p, surf.n, light_sample.p, hiprt_geom);

    // 重要な項
    const float3 brdf = 1.0f / PI * hit_triangle.color;
    const float G =
        geometry_term(surf.p, surf.n, light_sample.p, light_sample.n);
    const float light_pdf =
        1.0f / lights.size() * 1.0f / area_of(light_triangle);

    // 寄与を加算
    radiance +=
        throughput * brdf * G * V * light_triangle.emissive / light_pdf;
}
```

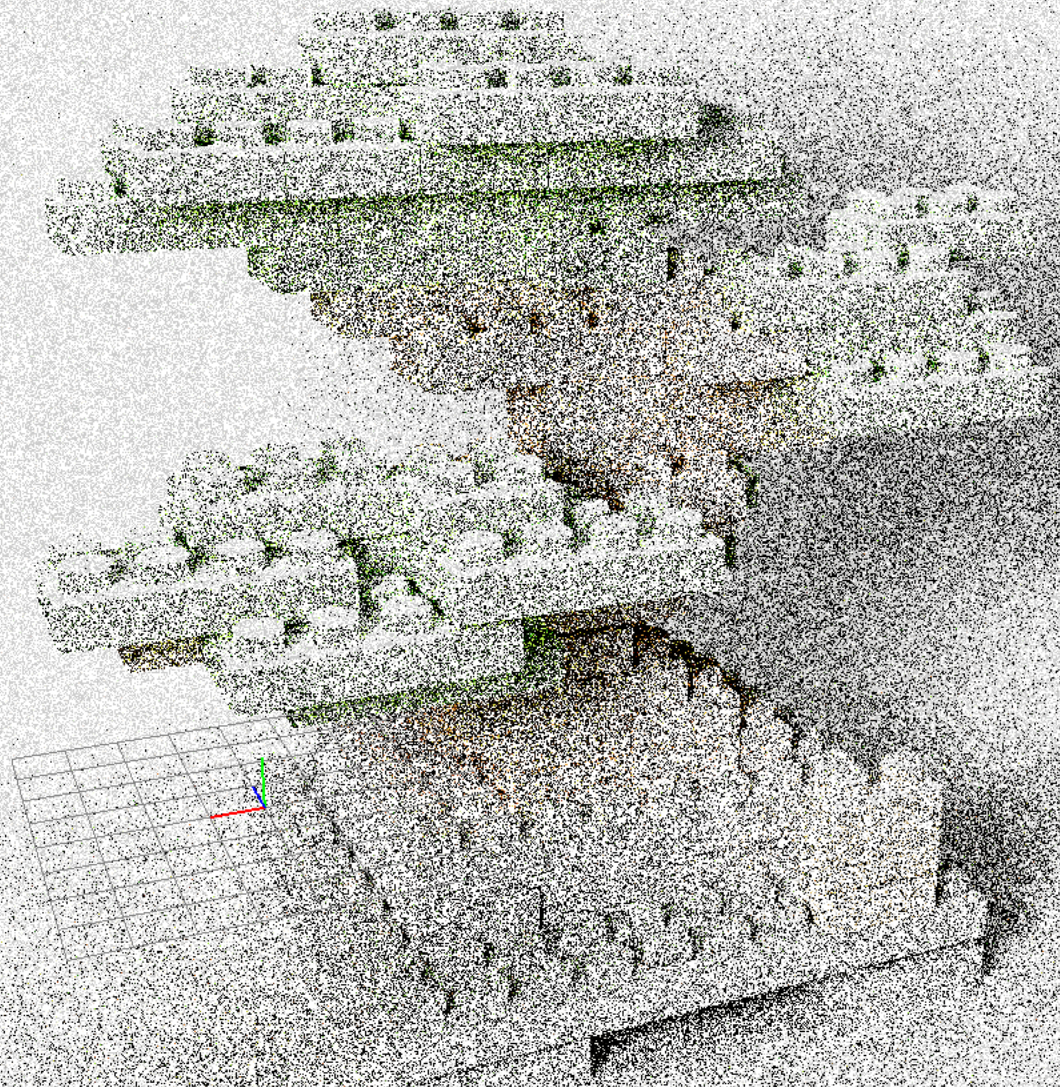
# 実装例

```
// 光源上の点をサンプリング
const LightSample light_sample =
    sample_light(triangles, lights, random.uniformf(),
                 random.uniformf(), random.uniformf());

// 寄与の計算
{
    const Triangle& light_triangle = triangles[light_sample.index];

    // 光源の可視性のチェック
    const float V =
        check_visibility(surf.p, surf.n, light_sample.p, hiprt_geom);

    // 重要な項
    const float3 brdf = 1.0f / PI * hit_triangle.color;
    const float G =
        geometry_term(surf.p, surf.n, light_sample.p, light_sample.n);
    const float light_pdf =
        1.0f / lights.size() * 1.0f / area_of(light_triangle);

    // 寄与を加算
    radiance +=
        throughput * brdf * G * V * light_triangle.emissive / light_pdf;
}
```

# 実装例

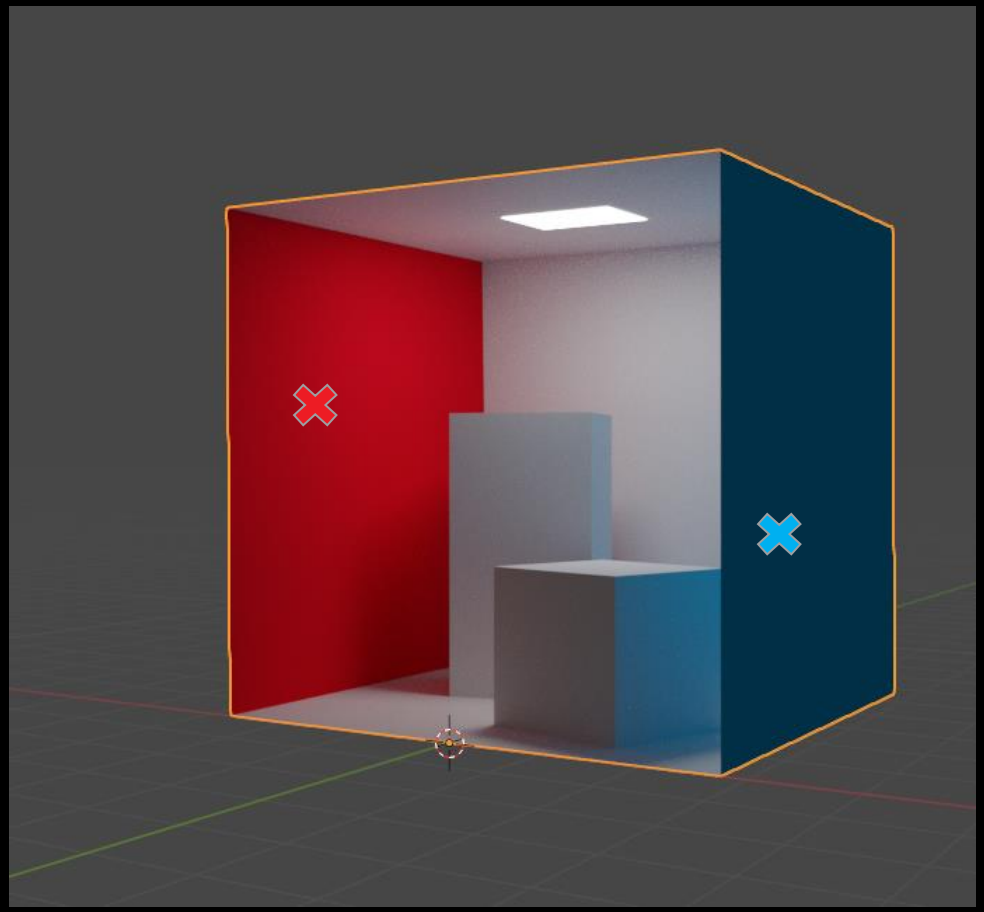
```
// 光源上の点をサンプリング
const LightSample light_sample =
    sample_light(triangles, lights, random.uniformf(),
                 random.uniformf(), random.uniformf());

// 寄与の計算
{
    const Triangle& light_triangle = triangles[light_sample.index];

    // 光源の可視性のチェック
    const float V =
        check_visibility(surf.p, surf.n, light_sample.p, hiprt_geom);

    // 重要な項
    const float3 brdf = 1.0f / PI * hit_triangle.color;
    const float G =
        geometry_term(surf.p, surf.n, light_sample.p, light_sample.n);
    const float light_pdf =
        1.0f / lights.size() * 1.0f / area_of(light_triangle);

    // 寄与を加算
    radiance +=
        throughput * brdf * G * V * light_triangle.emissive / light_pdf;
}
```

# BRDF, G, light_pdfを省くと…？

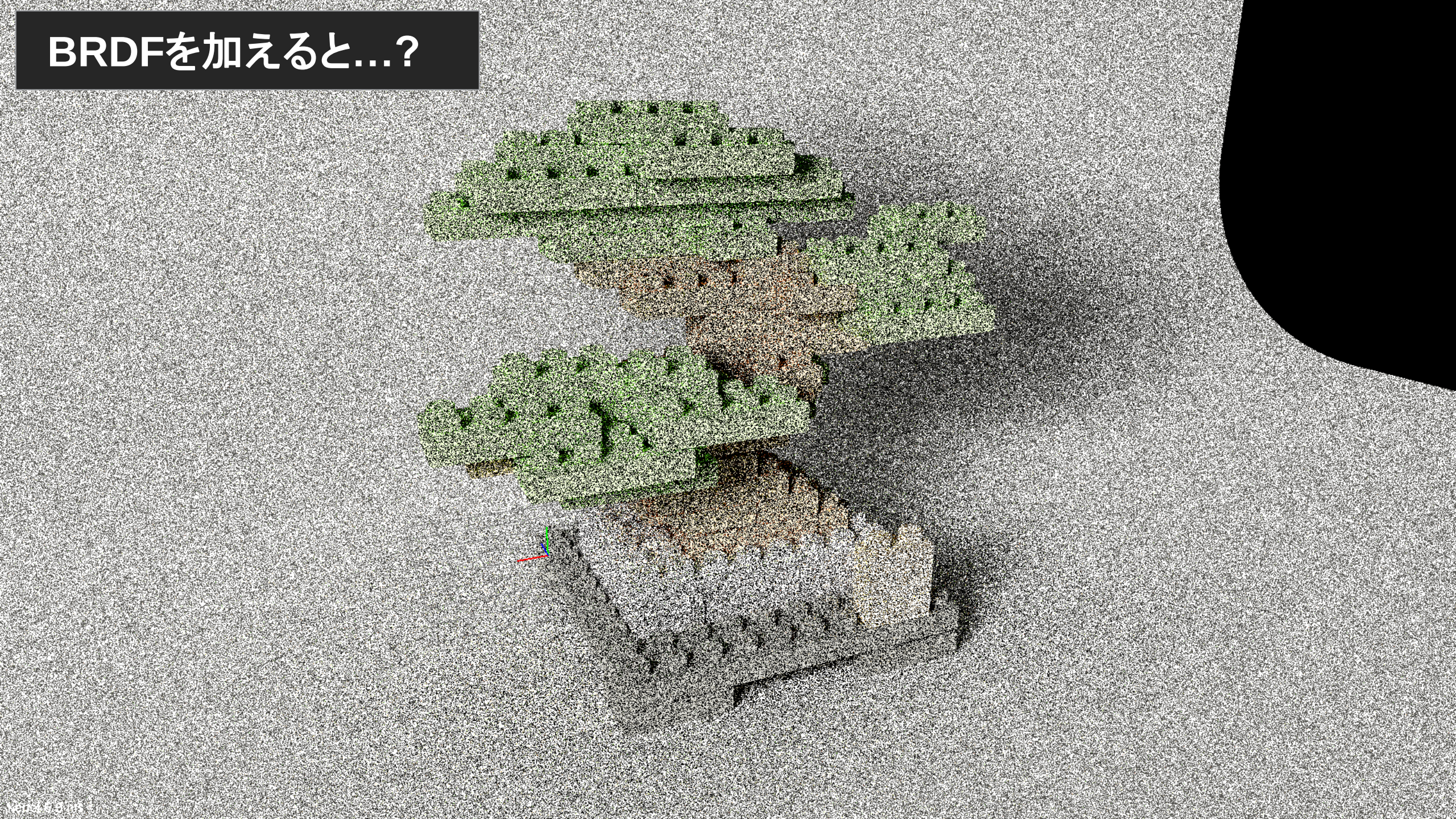


# BRDF

$$Lambert\ BRDF = \frac{1}{\pi} \times \text{反射率}$$

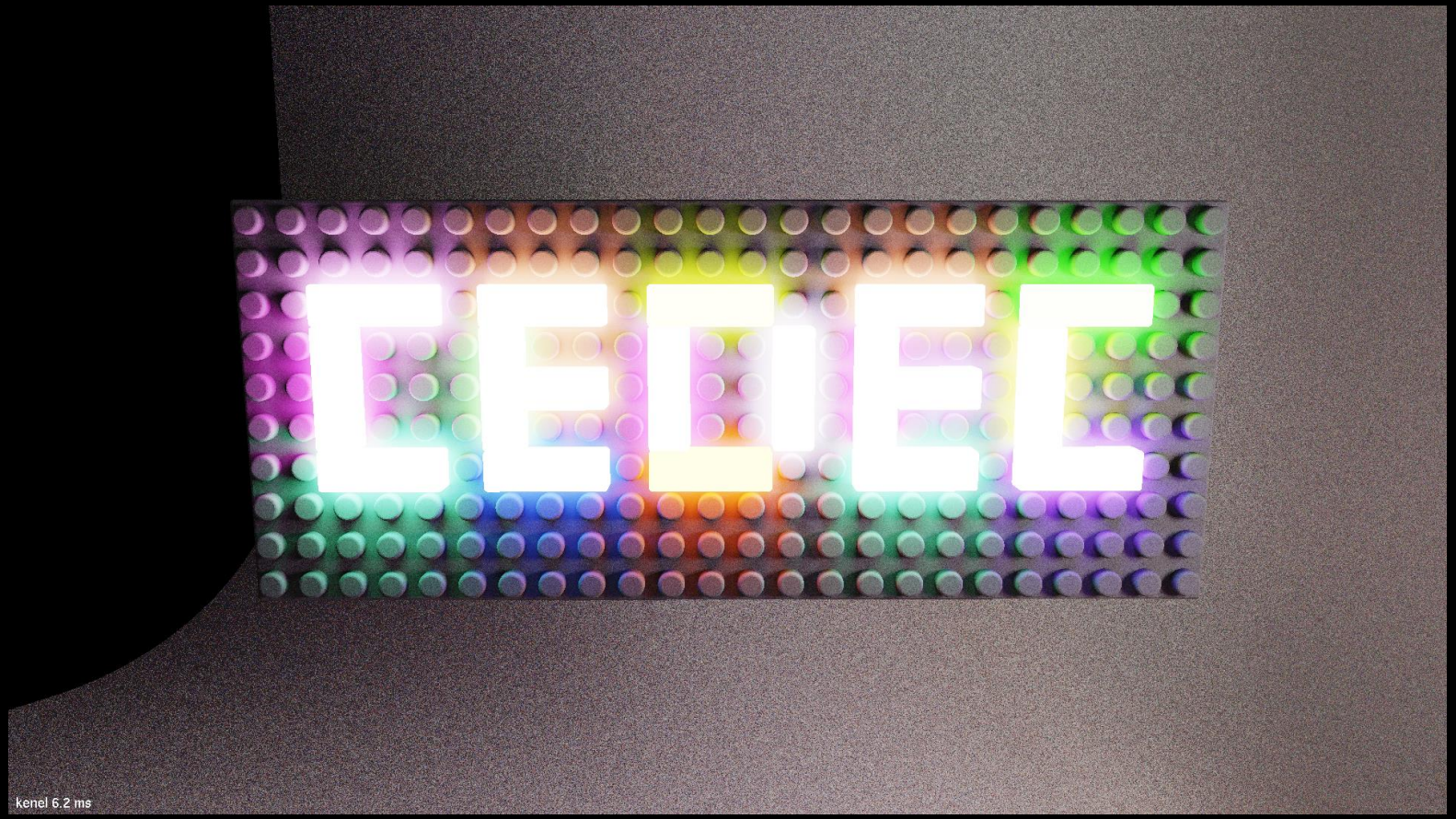


# BRDFを加えると…？

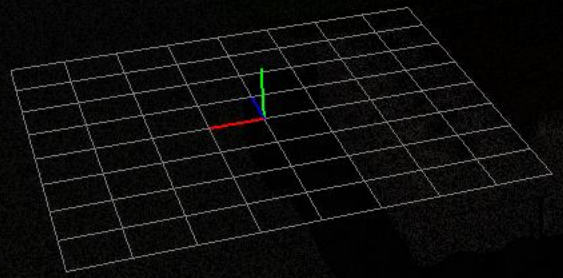


# G (幾何項)

- 幾何的な関係の調整項
  - 近いところの光源ほど寄与が大きい
  - 正面を向いている光源ほど寄与が大きい



# 幾何項も加えると…？



kenel 7.0 ms

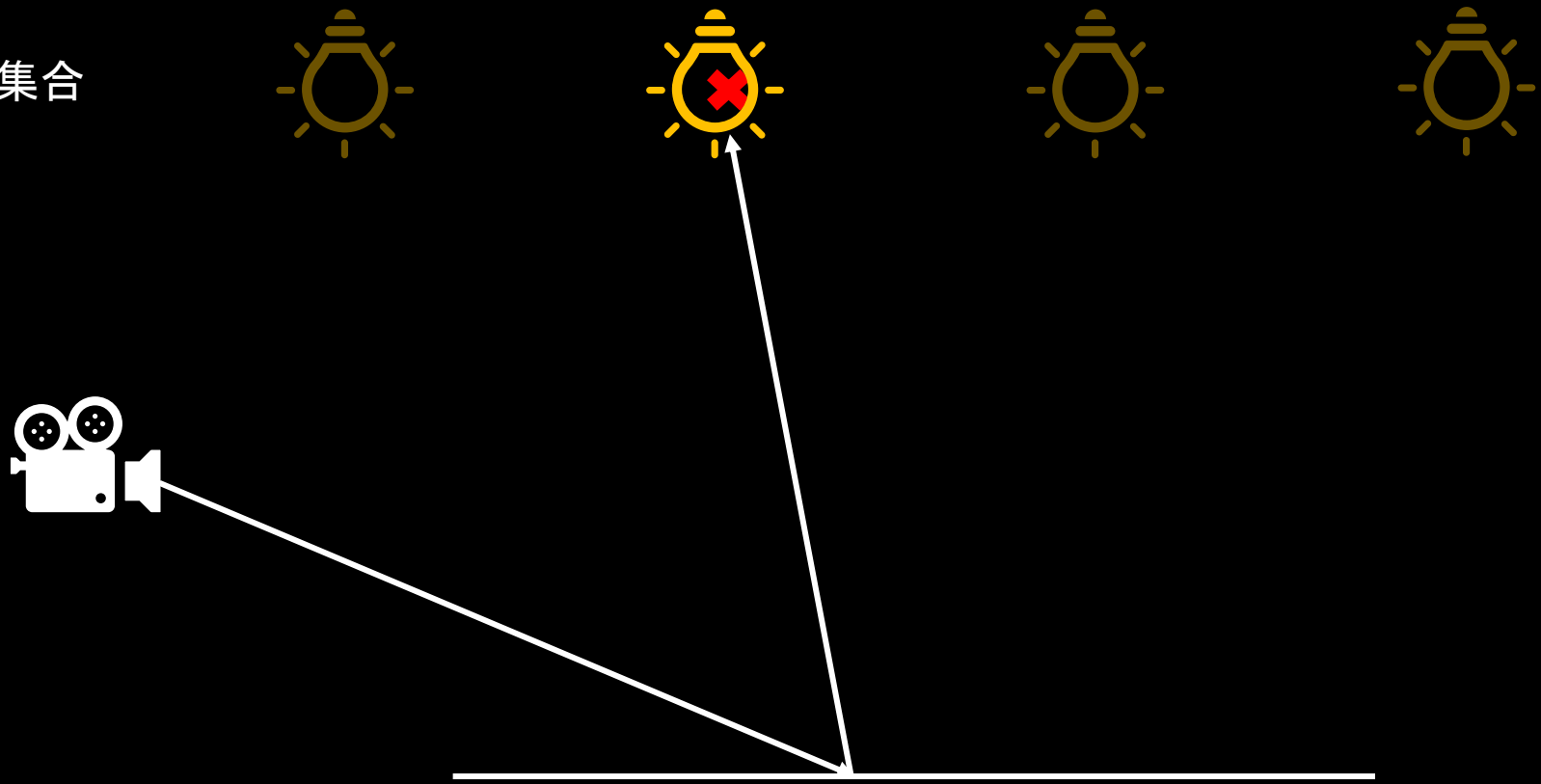

明るさ50倍
kernel 7.0 ms

# Light pdf

$Light\ pdf$ =赤点をサンプリングする確率

寄与 $\propto \frac{\text{光源の明るさ}}{Light\ pdf}$

光源の集合

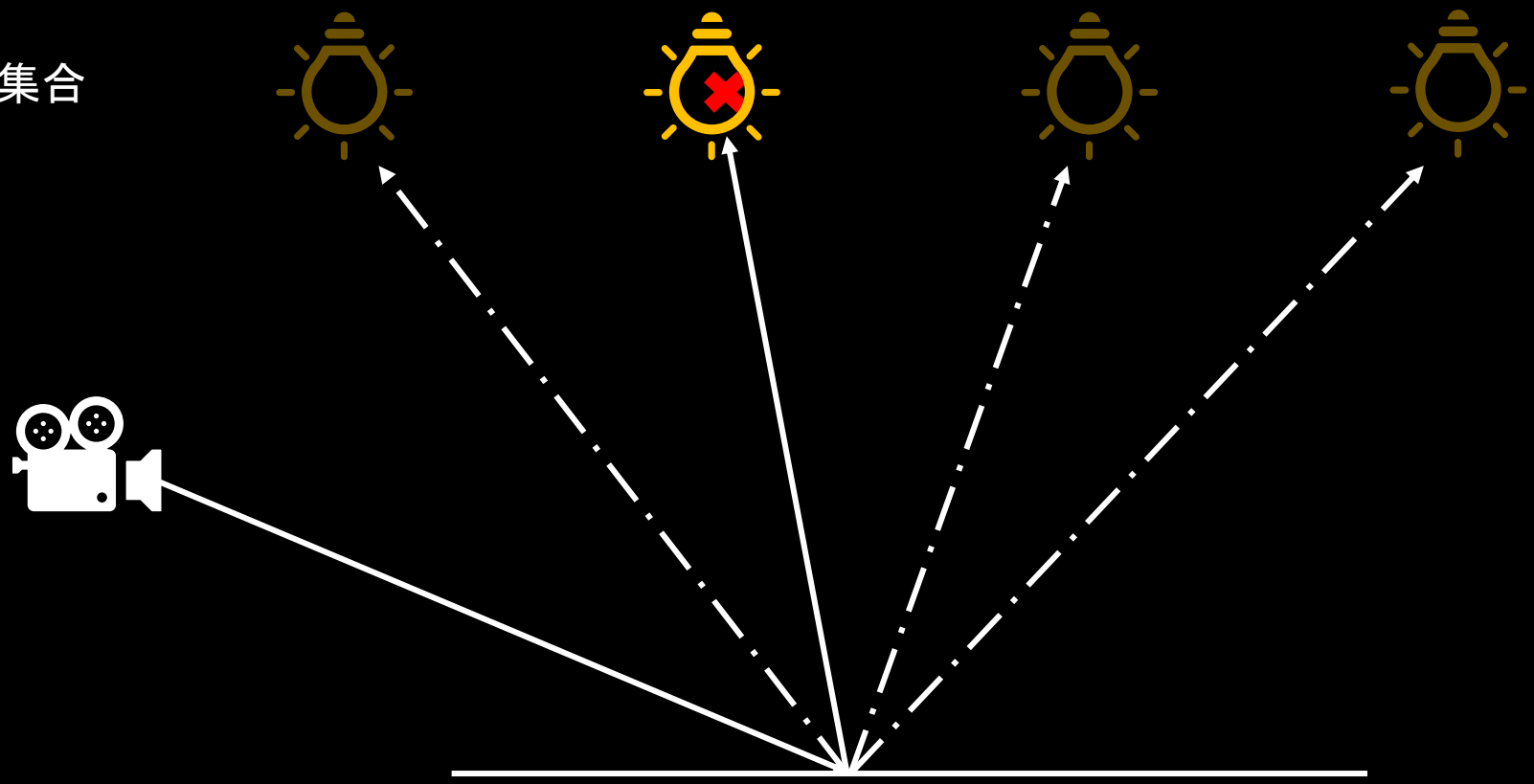


# Light pdf

$Light\ pdf$ =赤点をサンプリングする確率

寄与 $\propto \frac{\text{光源の明るさ}}{Light\ pdf}$

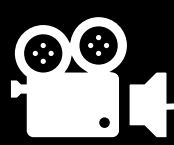
光源の集合



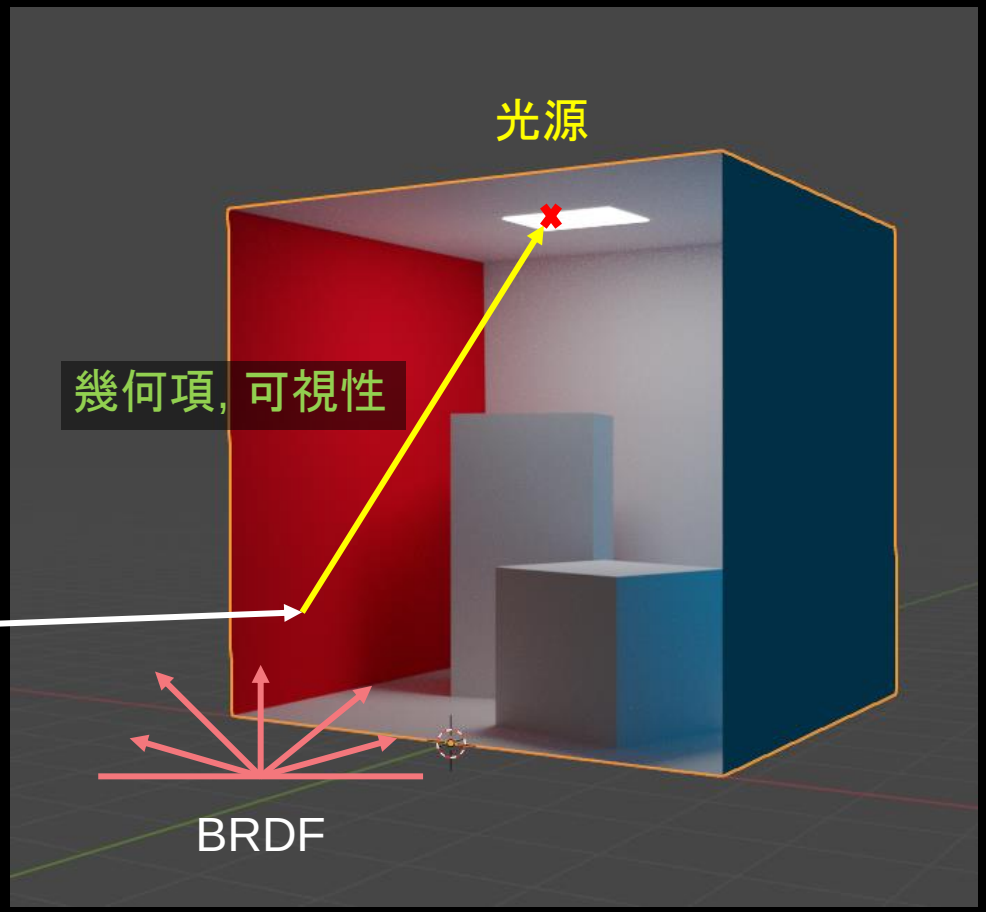
# Light pdfも加えると…？

# 経路の寄与

$$\text{寄与} = \frac{BRDF \times \text{幾何項} \times \text{可視性} \times \text{光源の明るさ}}{pdf}$$



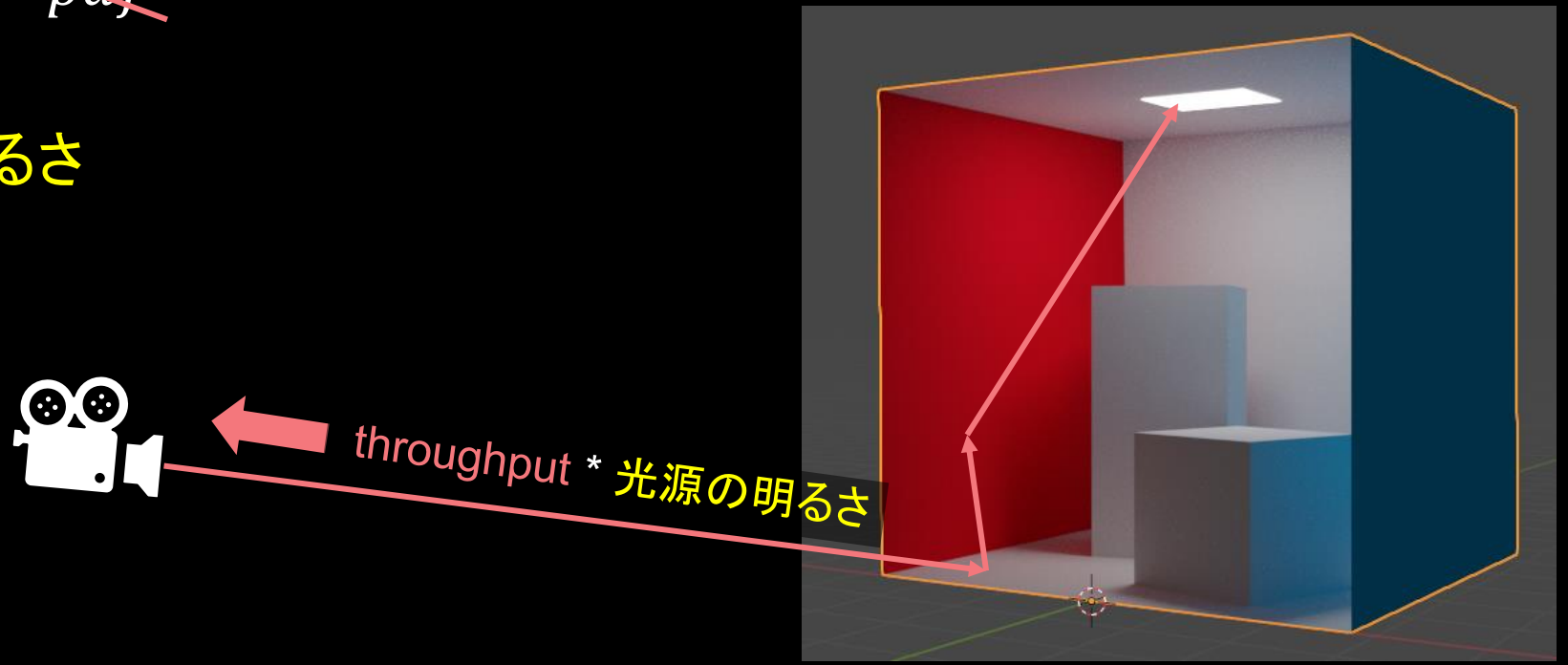
寄与



# パストレーシングにおける経路の寄与

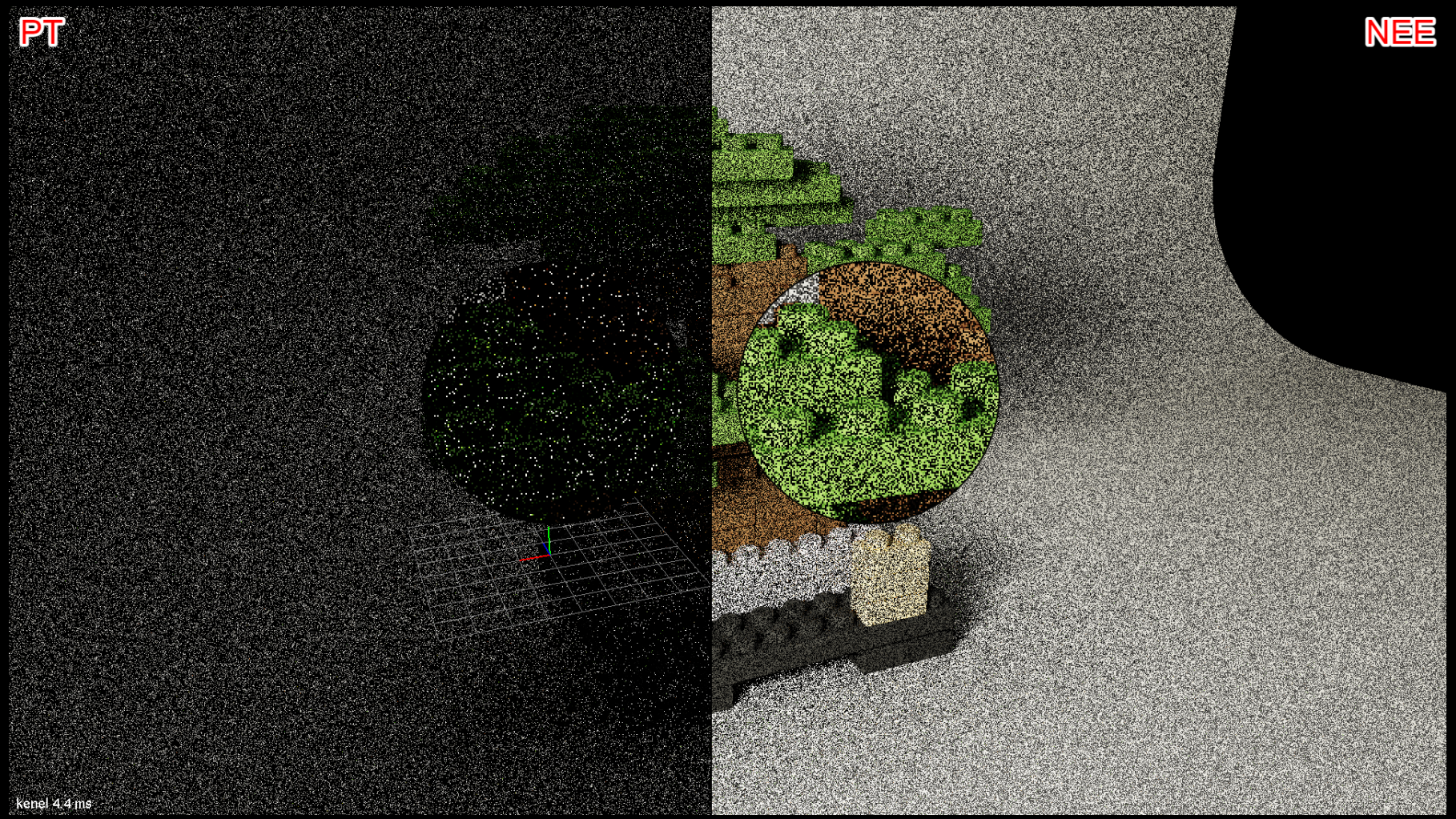
$$\text{寄与} = \frac{BRDF \times \cancel{\text{幾何項}} \times \text{可視性}(=1) \times \text{光源の明るさ}}{\cancel{pdf}}$$

$$= \text{反射率} \times \text{光源の明るさ}$$



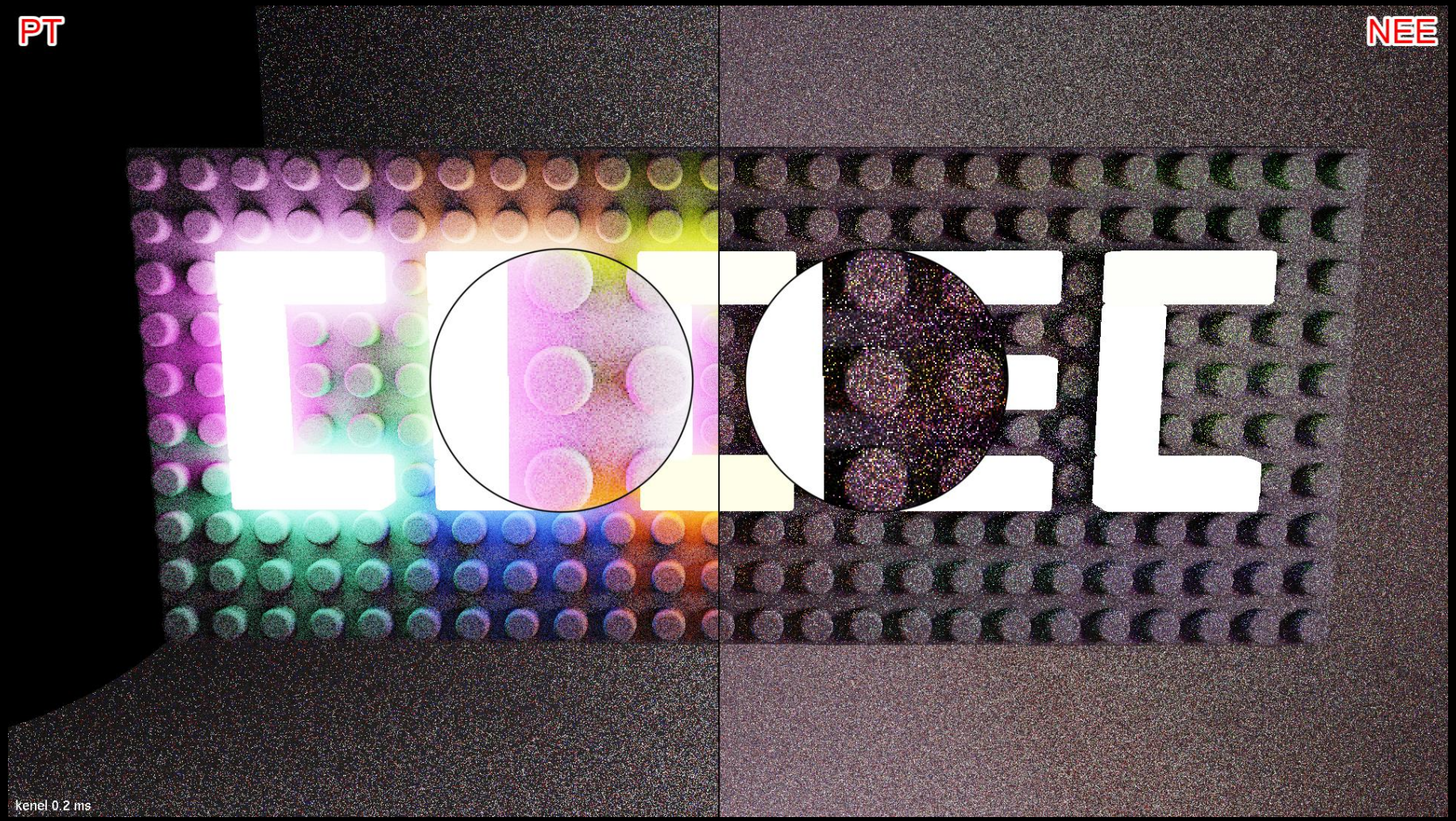
# Result (1サンプル)

- NEEによってノイズがかなり削減された



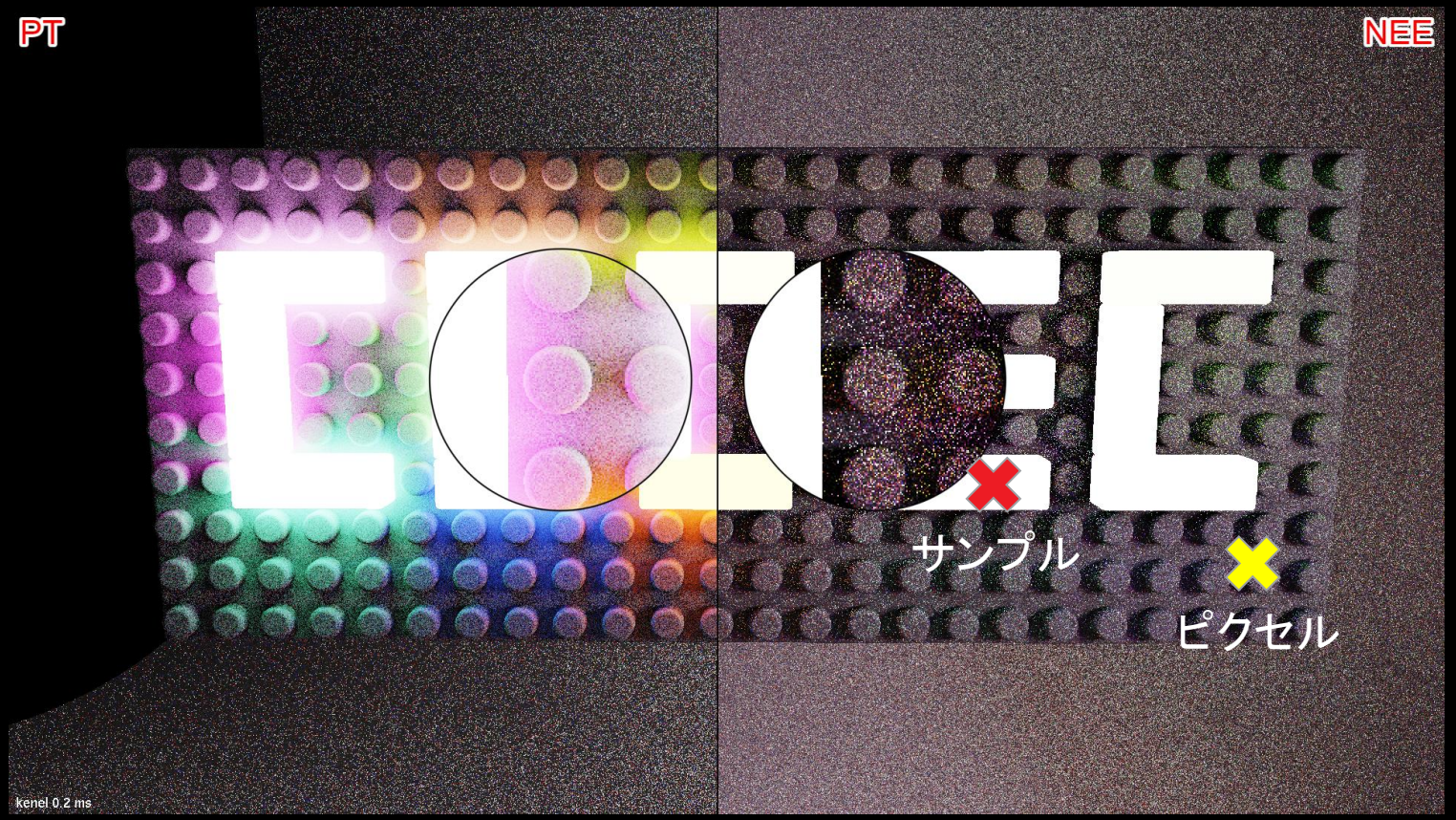
# NEEの問題点 (128サンプル)

- NEEの方がノイズが多いケース



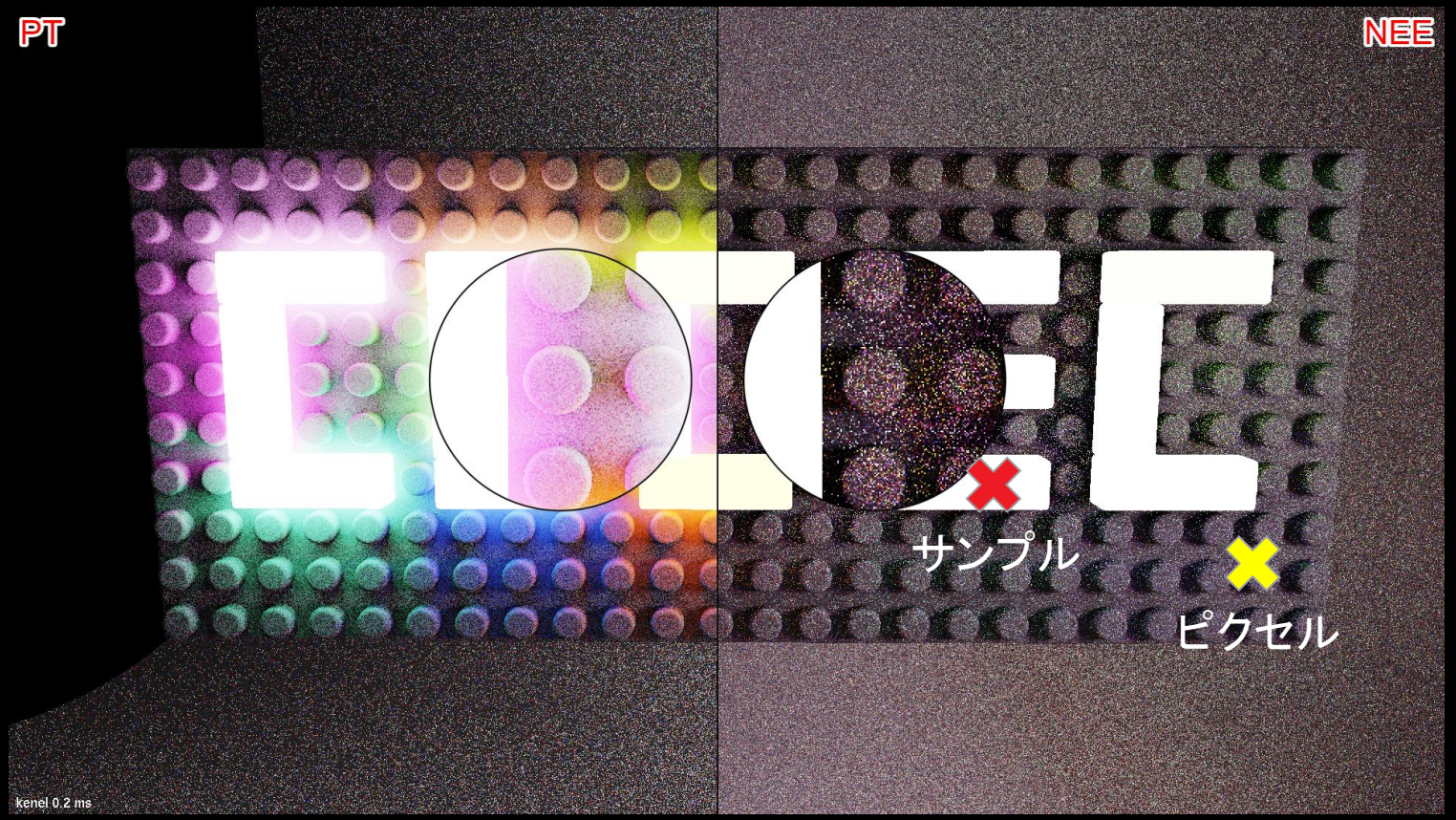
# NEEの問題点

- NEEでは遠い光源も等確率でサンプリングされてしまう



# NEEの問題点

- 寄与 $\propto BRDF \times$ 幾何項$(0.01) \times$ 可視性 $\times$ 光源の明るさ

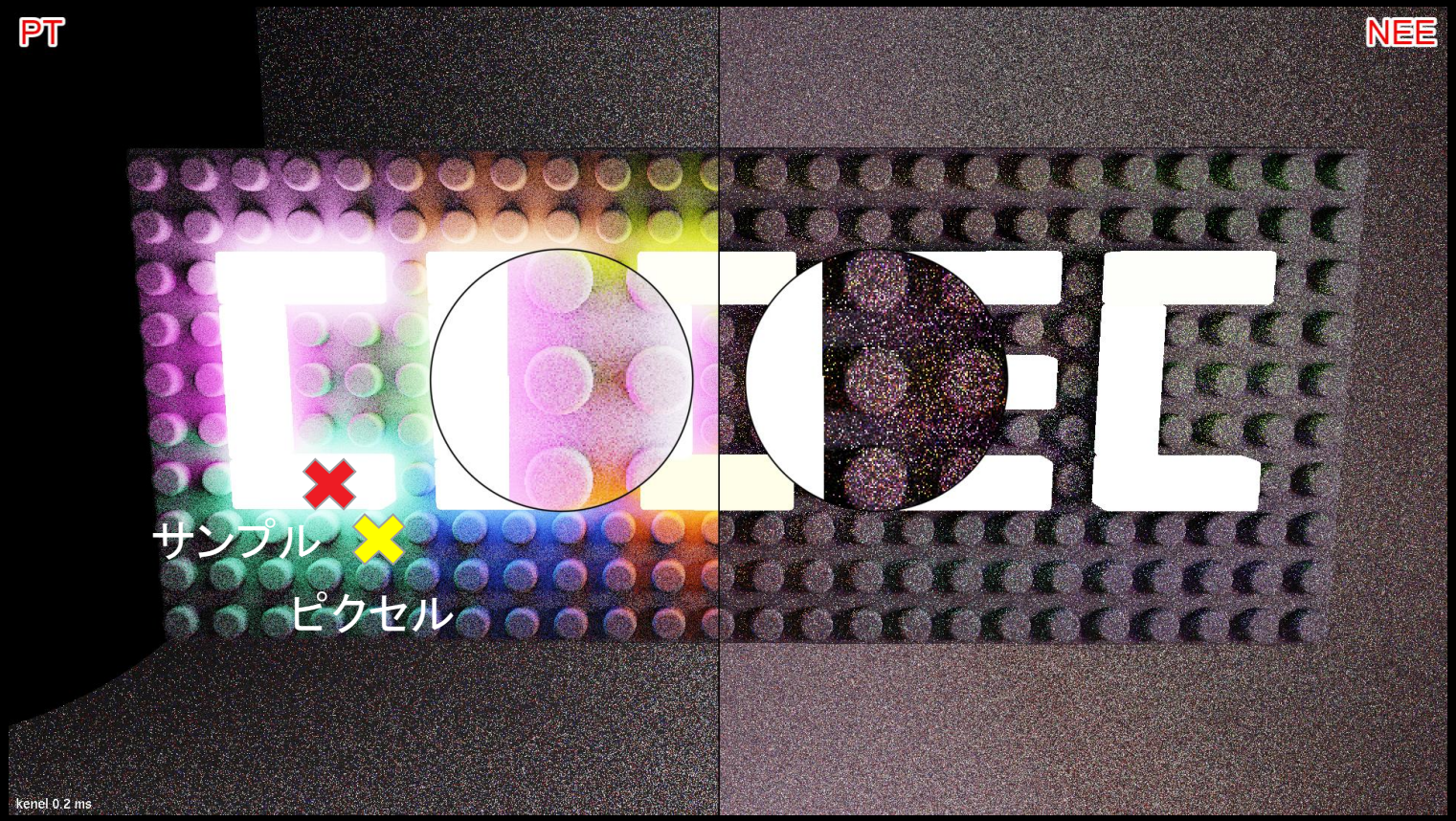





AMD

# PTはなぜノイズが少ない?

- PTでは自然と近い光源に当たる可能性が高い



# PTとNEEの違い

重点的サンプリング

$$\mathrm{PT}: \overbrace{BRDF \times \text{幾何項} \times \text{可視性}} \times \text{光源の明るさ}$$

重点的サンプリング

$$\mathrm{NEE}: BRDF \times \text{幾何項} \times \text{可視性} \times \overbrace{\text{光源の明るさ}}$$

# 理想の重点的サンプリング

重点的サンプリング

$$???: BRDF \times \text{幾何項} \times \text{可視性} \times \text{光源の明るさ}$$

# Resampled Importance Sampling (RIS)

# RISの概要

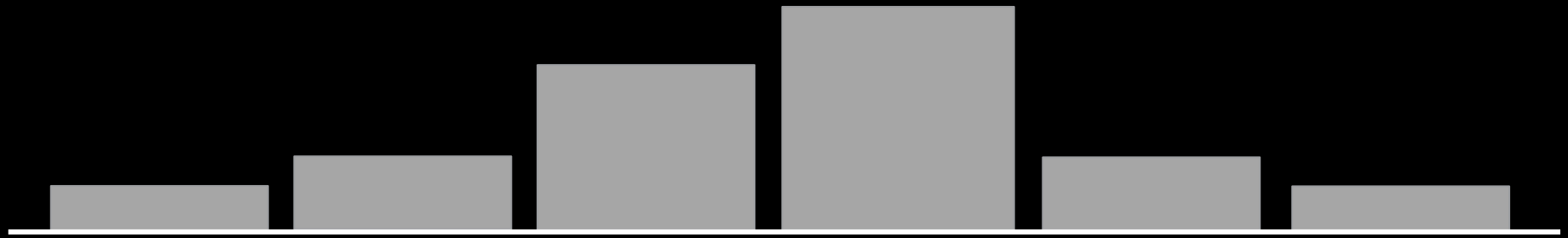
サンプル集合(M=6)

重み: 0.1　重み: 0.2　重み: 0.5　重み: 0.8　重み: 0.2　重み: 0.1

# RISの概要

サンプル集合(M=6)

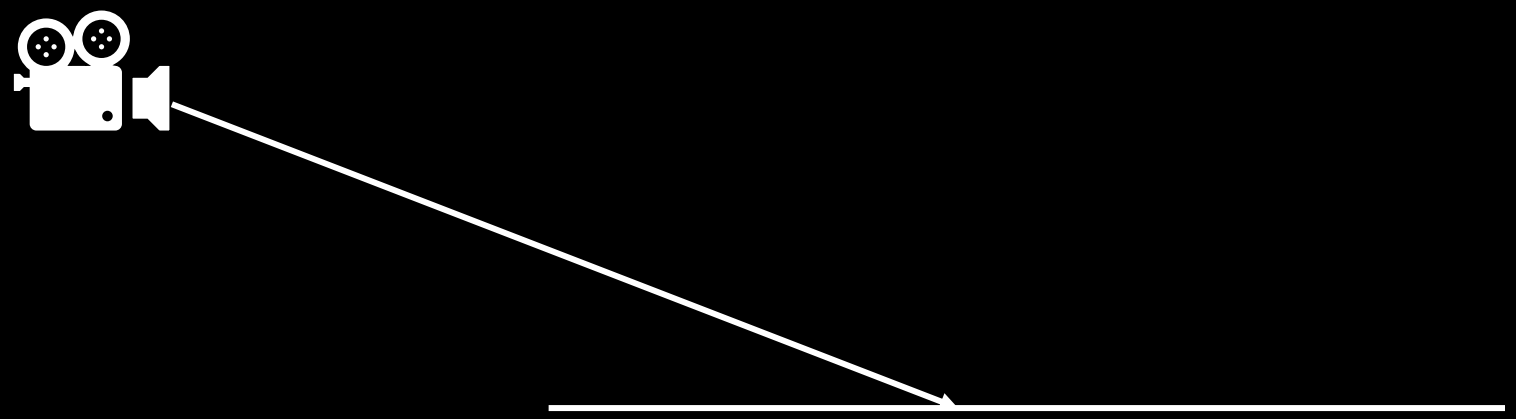

重みの分布



リサンプリング

**重みの分布に従った重点的サンプリング**

# Weighted Reservoir Sampling (WRS)



$$sample = null$$
$$w_{sum} = 0$$
$$M = 0$$

Reservoir (サンプル貯蔵庫)

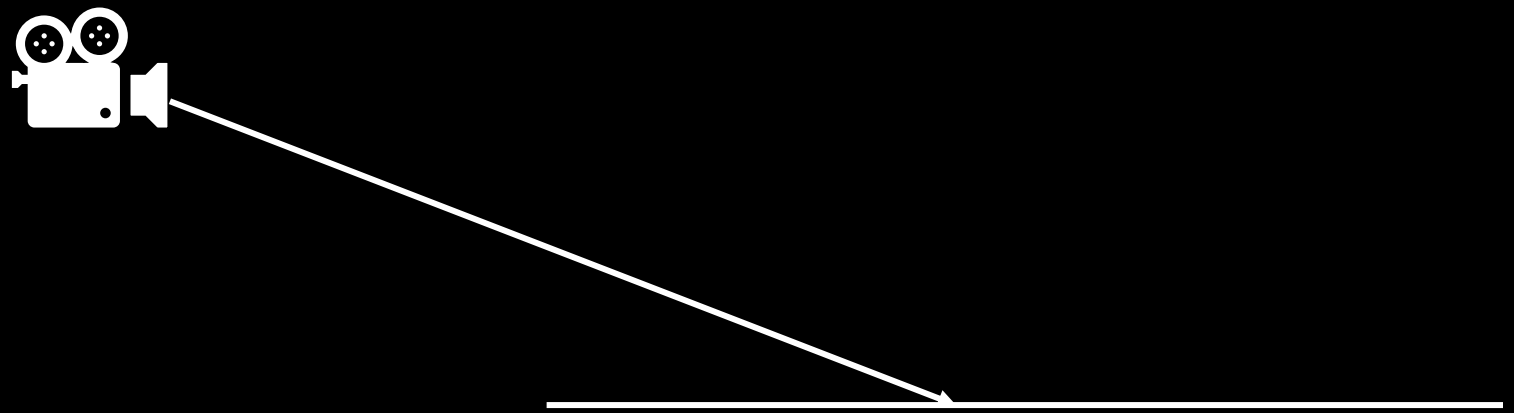
# Weighted Reservoir Sampling (WRS)

$p_1$

光源サンプリング

$$sample = null$$
$$w_{sum} = 0$$
$$M = 0$$

Reservoir (サンプル貯蔵庫)



# Weighted Reservoir Sampling (WRS)

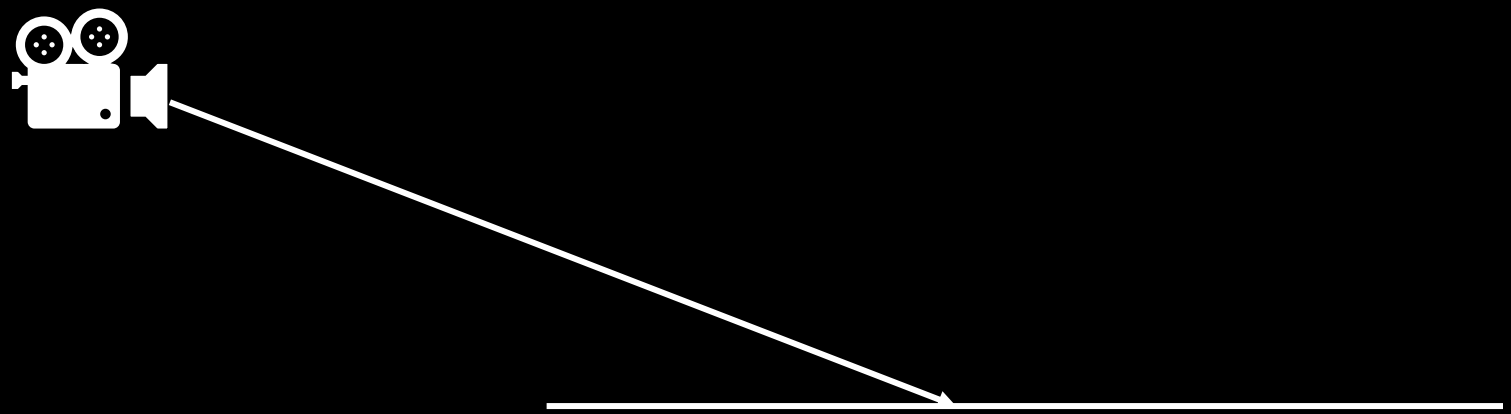
$p_1$

重み = 0.2

重み $\propto BRDF \times$ 幾何項 $\times$ 可視性 $\times$ 光源の明るさ

$$sample = null$$
$$w_{sum} = 0$$
$$M = 0$$

Reservoir (サンプル貯蔵庫)



# Weighted Reservoir Sampling (WRS)

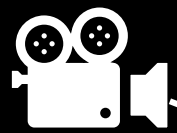

重み = 0.2

$sample = p_1$

$w_{sum} = 0.2$

$M = 1$

Reservoir (サンプル貯蔵庫)



# Weighted Reservoir Sampling (WRS)

$p_1$

重み = 0.2

重み $\propto BRDF \times$ 幾何項 $\times$ 可視性 $\times$ 光源の明るさ

$p_2$

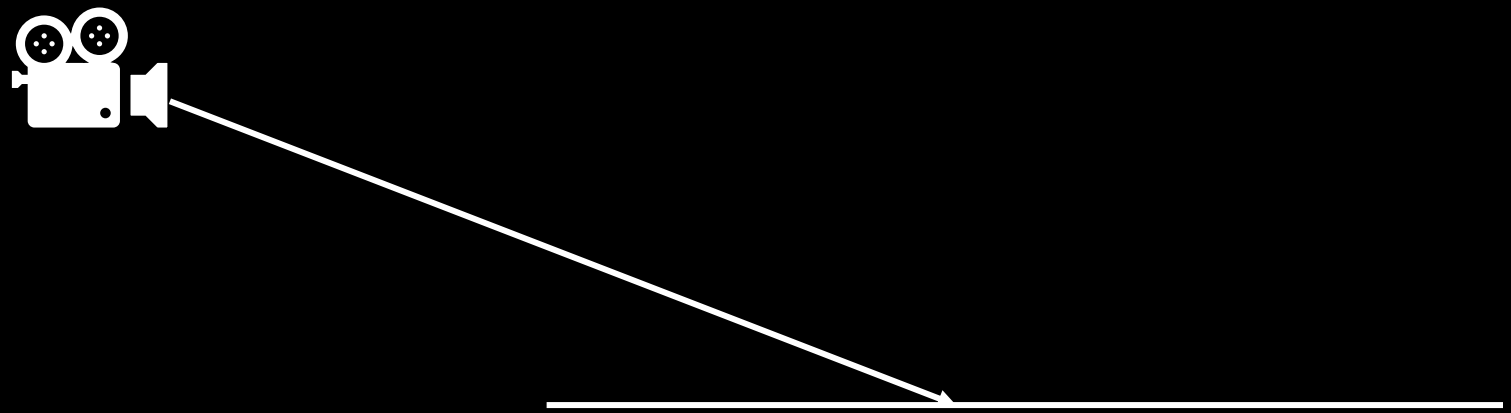
重み = 0.6

$sample = p_1$

$w_{sum} = 0.2$

$M = 1$

Reservoir (サンプル貯蔵庫)



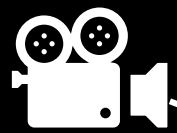
# Weighted Reservoir Sampling (WRS)

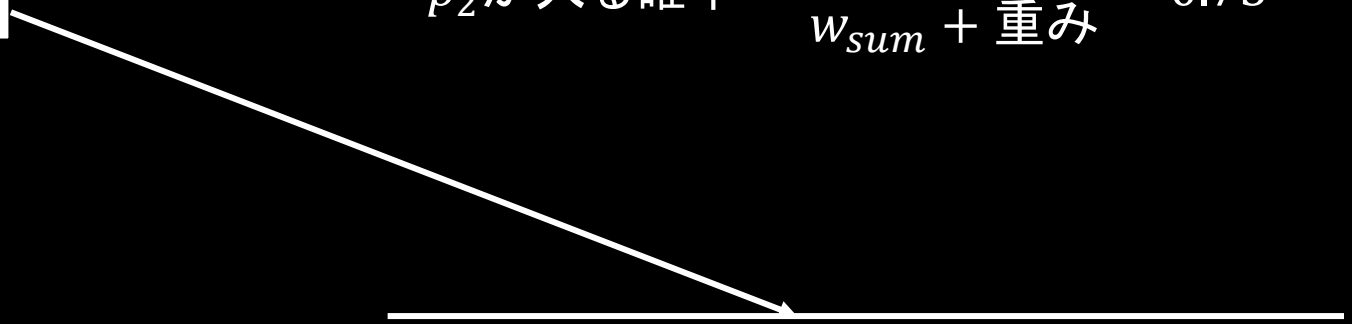
$p_1$

重み = 0.2

$p_2$

重み = 0.6





$$p_2\text{が入る確率} = \frac{\text{重み}}{w_{sum} + \text{重み}} = 0.75$$

$sample = p_1$

$w_{sum} = 0.2$

$M = 1$

Reservoir (サンプル貯蔵庫)

# Weighted Reservoir Sampling (WRS)

$p_1$

重み = 0.2

$p_2$

重み = 0.6

入る確率 = 0.75

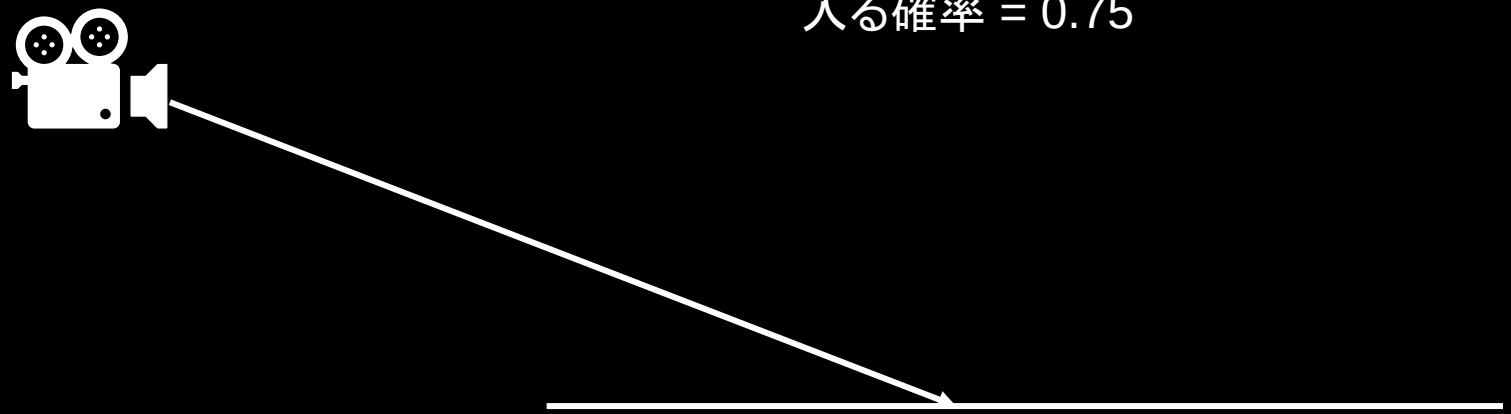
乱数 < 確率(0.75)なら入れる

$sample = p_2$

$w_{sum} = 0.8$

$M = 2$

Reservoir (サンプル貯蔵庫)



# Weighted Reservoir Sampling (WRS)

$p_1$

重み = 0.2

$p_2$

重み = 0.6

$p_3$

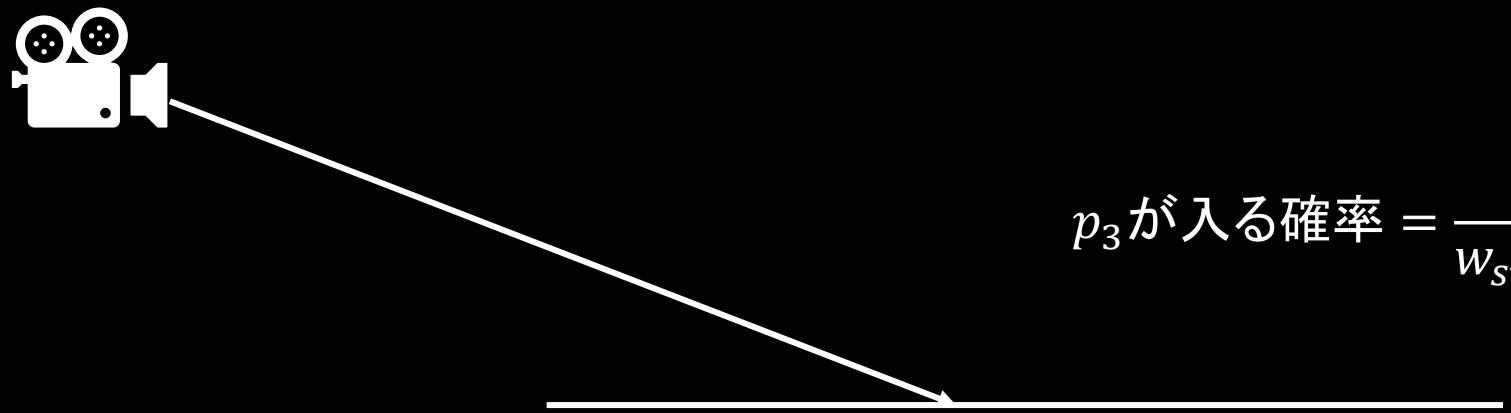
重み = 0.11

$sample = p_2$

$w_{sum} = 0.8$

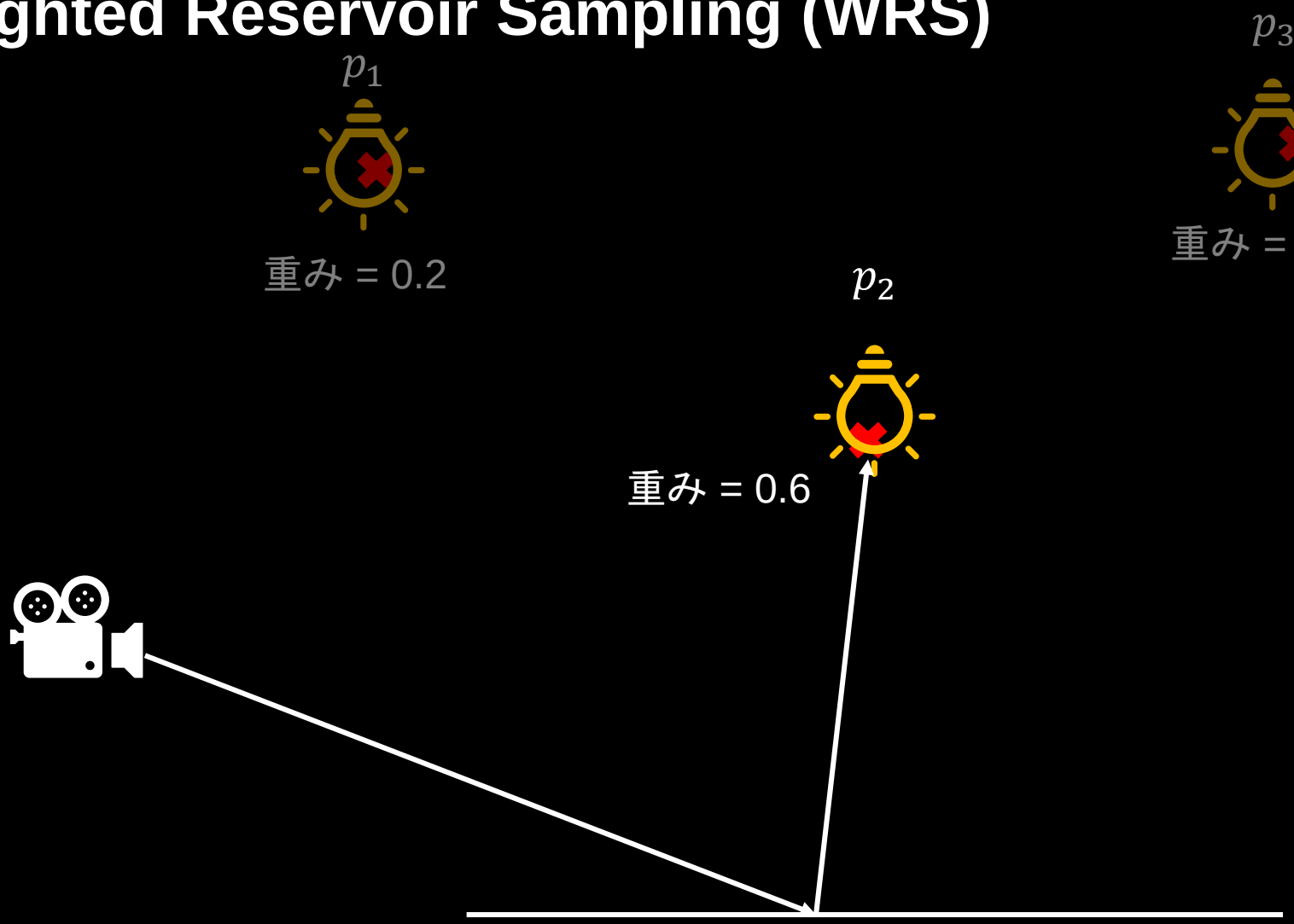
$M = 2$

Reservoir (サンプル貯蔵庫)



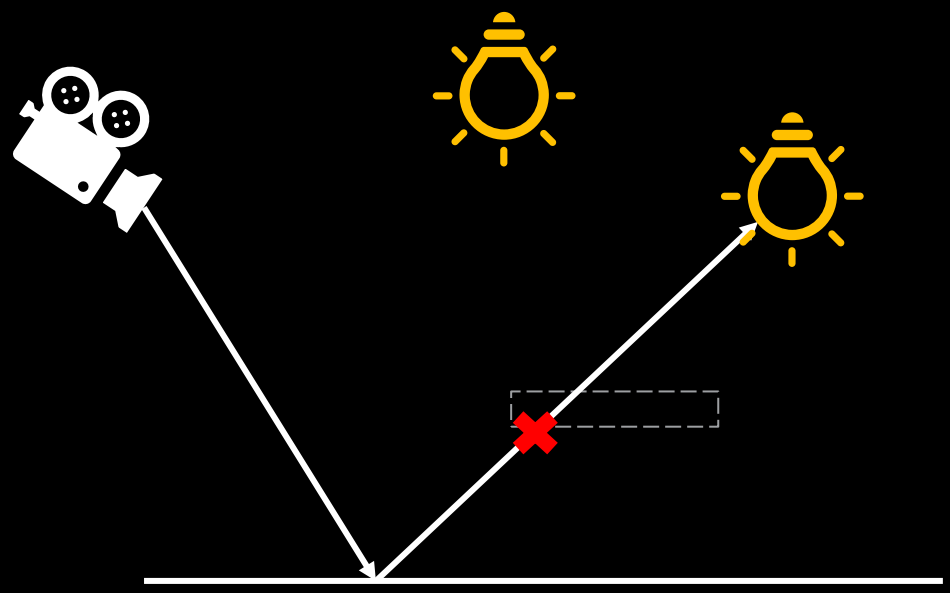
$$p_3\text{が入る確率} = \frac{\text{重み}}{w_{sum} + \text{重み}} = 0.11$$

# Weighted Reservoir Sampling (WRS)



# 実用的な重みの計算方法

$$\text{重み} \propto BRDF \times \text{幾何項} \times \text{可視性}(=1) \times \text{光源の明るさ}$$



# Result (128サンプル)

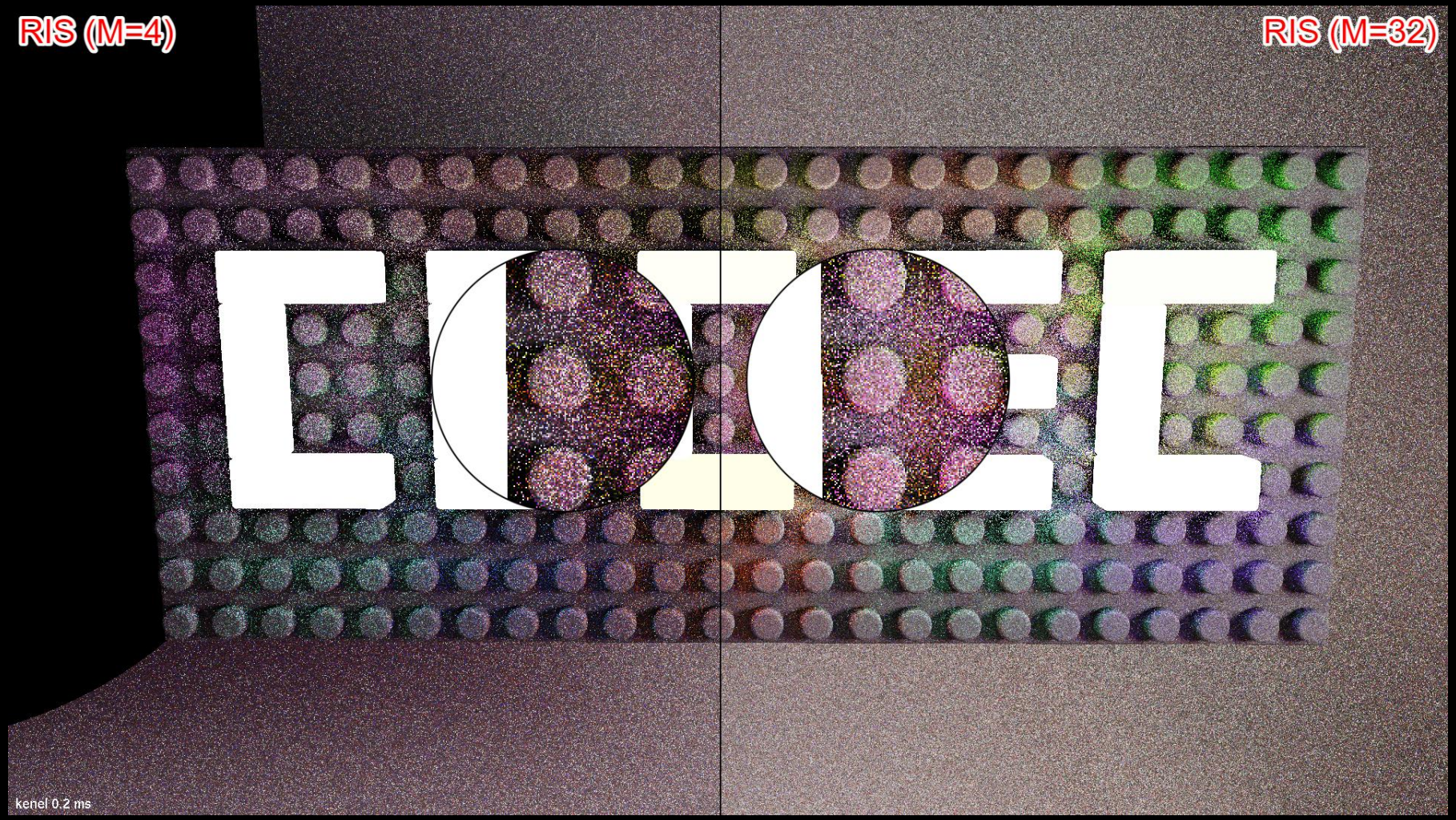
- RISによってノイズが減った




AMD

# Result (128サンプル)

- サンプル集合の数が少ないとノイズが増える



# ReSTIR DI

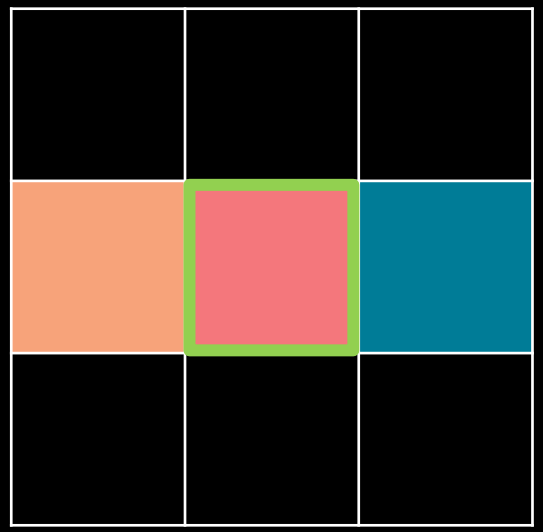
# RISの問題点

- Reservoirのサンプル集合(M)を計算量を抑えつつどうやって大きくするか?




AMD

# Spatial resampling



$sample = p_1$
$w_{sum} = 0.9$
$M = 3$

Reservoir 1

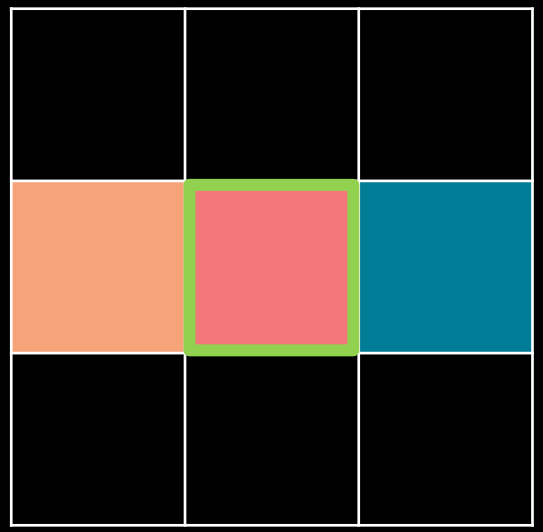
$sample = p_2$
$w_{sum} = 0.9$
$M = 3$

Reservoir 2

$sample = p_3$
$w_{sum} = 0.9$
$M = 3$

Reservoir 3

# Spatial resampling



$sample = p_1$
$w_{sum} = 0.9$
$M = 3$

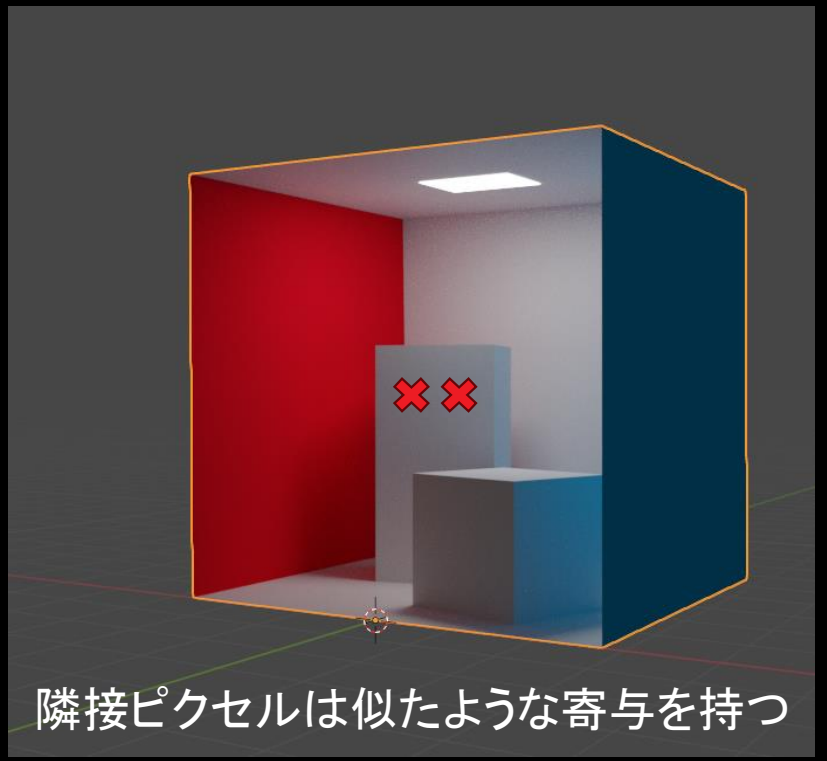
Reservoir 1

$sample = p_2$
$w_{sum} = 0.9$
$M = 3$

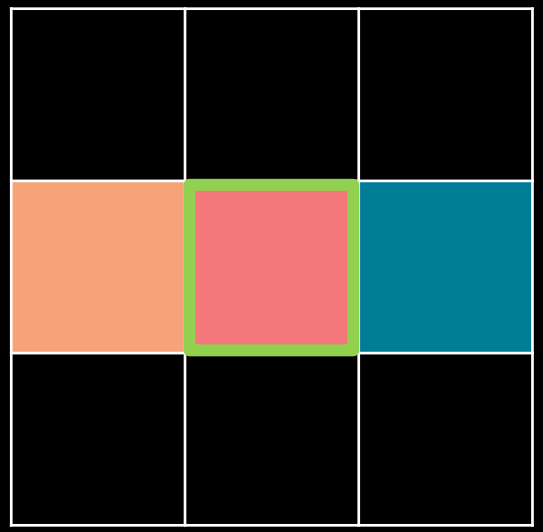
Reservoir 2

$sample = p_3$
$w_{sum} = 0.9$
$M = 3$

Reservoir 3



# Spatial resampling



$sample = p_1$
$w_{sum} = 0.9$
$M = 3$

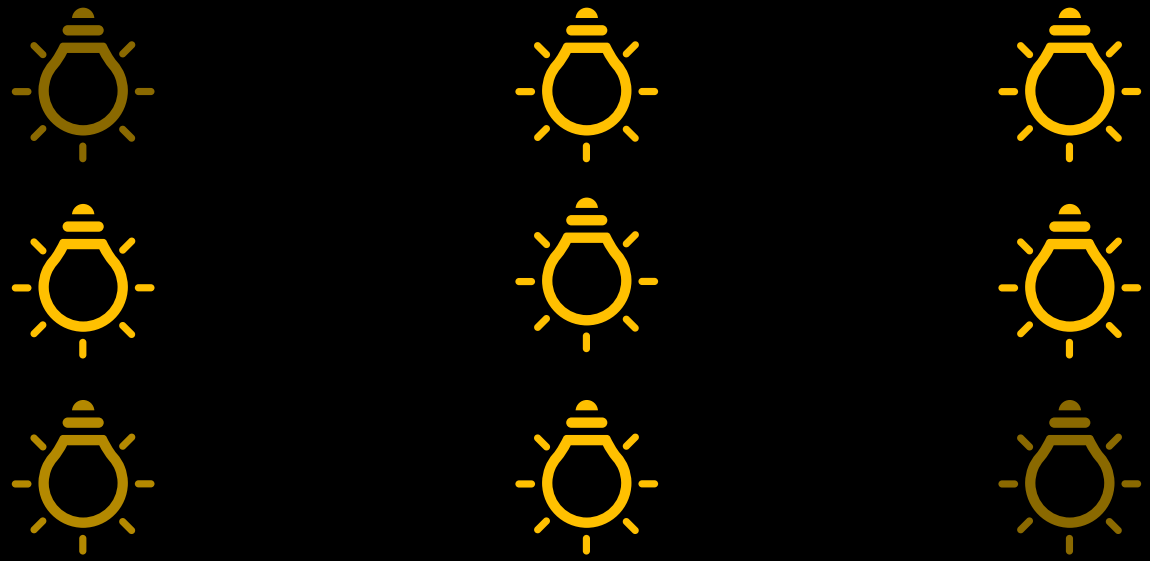
Reservoir 1

$sample = p_2$
$w_{sum} = 0.9$
$M = 3$

Reservoir 2

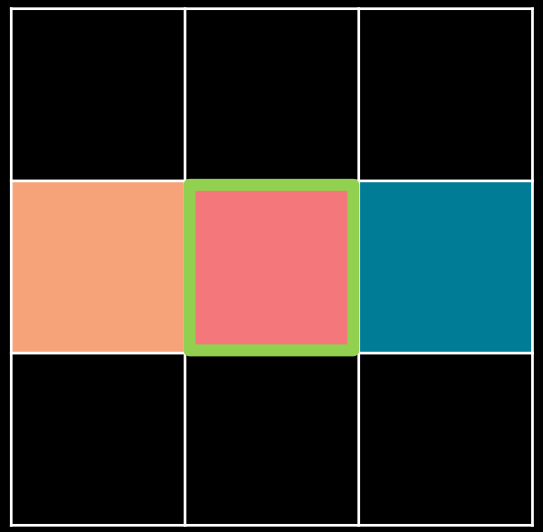
$sample = p_3$
$w_{sum} = 0.9$
$M = 3$

Reservoir 3



寄与が大きいサンプルが周辺に含まれる

# Spatial resampling



$sample = p_1$
$w_{sum} = 0.6$
$M = 3$

Reservoir 1

$sample = p_2$
$w_{sum} = 0.9$
$M = 3$

Reservoir 2

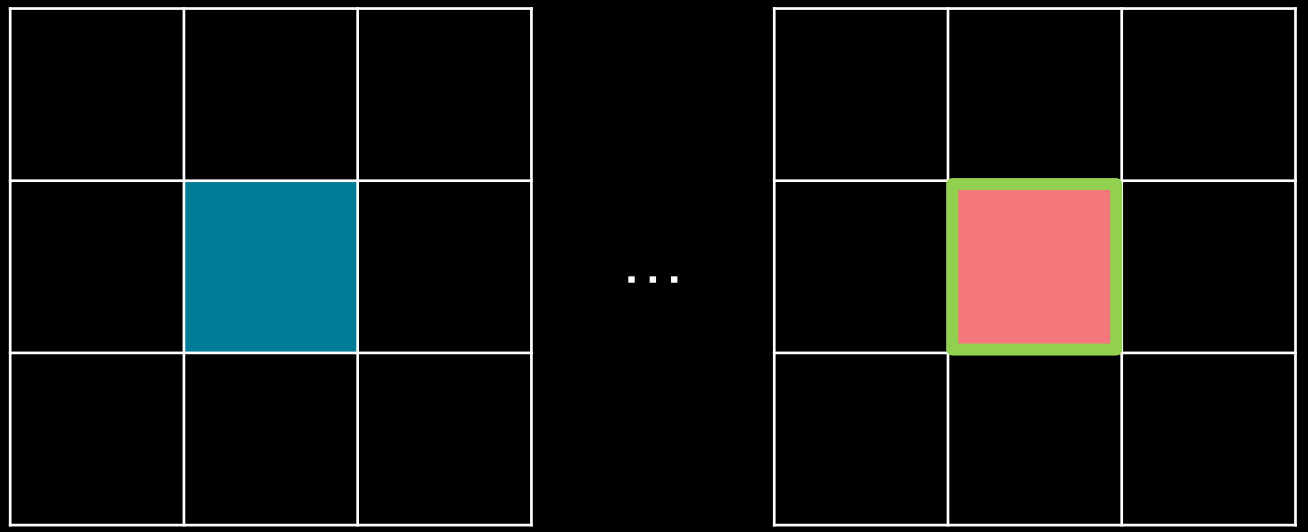
$sample = p_3$
$w_{sum} = 0.5$
$M = 3$

Reservoir 3

$sample = ?$
$w_{sum} = 2.0$
$M = 9$

Merge

# Temporal resampling



1時刻前のフレーム

…

現在のフレーム

$sample = p_1$
$w_{sum} = 0.2$
$M = 3$

Reservoir 1

$sample = p_2$
$w_{sum} = 0.9$
$M = 3$

Reservoir 2

$sample = ?$
$w_{sum} = 1.1$
$M = 6$

Merge

# ReservoirのMerge

- WRSが適用できる

Reservoir 1

$sample = p_1$

$w_{sum} = 0.2$

$M = 3$

重み = 0.4

Reservoir 1のサンプル集合

Reservoir 2

$sample = p_2$

$w_{sum} = 0.9$

$M = 3$

Reservoir 2のサンプル集合

# ReservoirのMerge

Reservoir 1

$sample = p_1$

$w_{sum} = 0.2$

$M = 3$

重み = 0.4

確率 = 0.26

乱数 < 確率なら入れる

Reservoir 2

$sample = p_2$

$w_{sum} = 1.1$

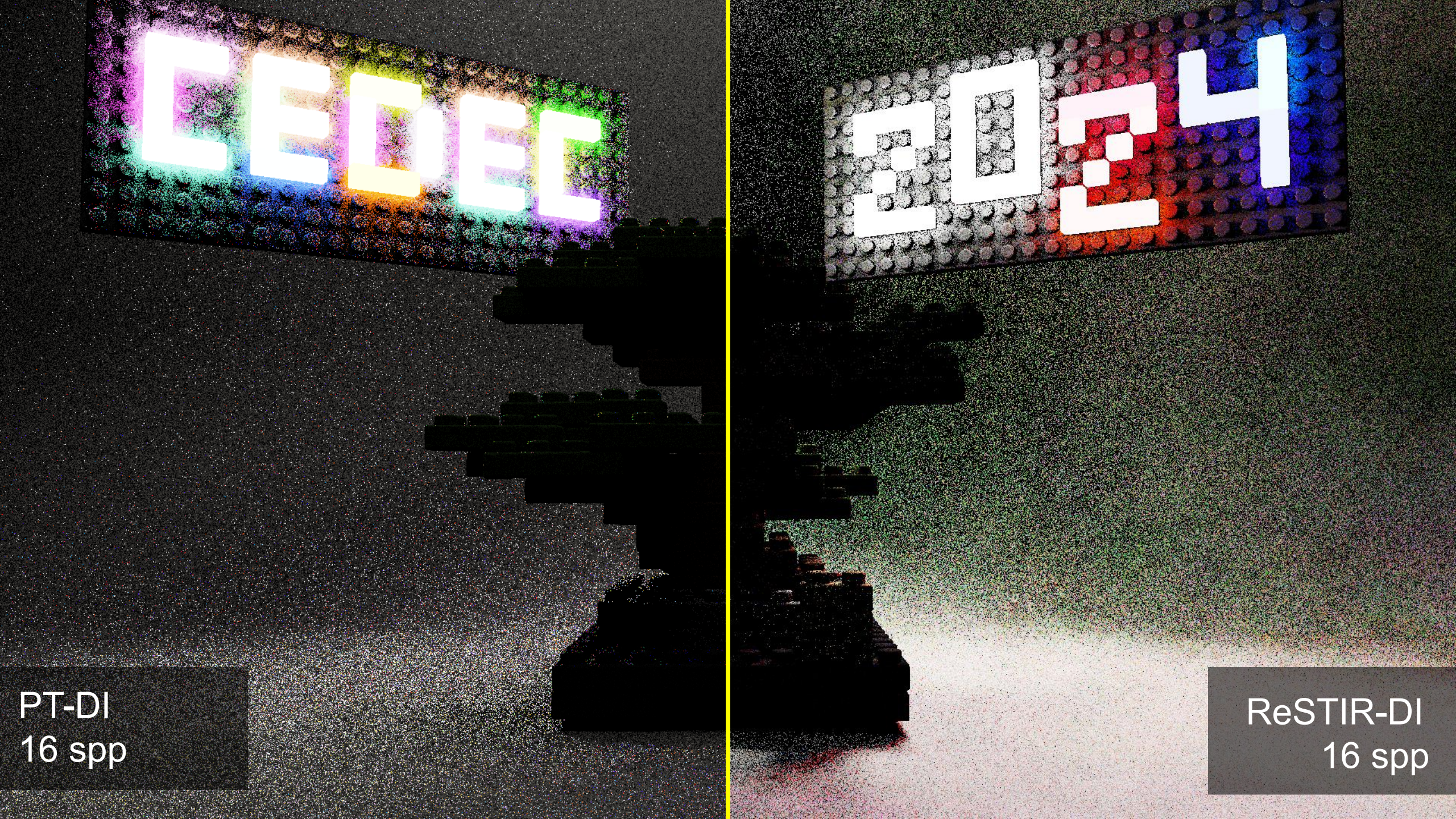
$M = 6$

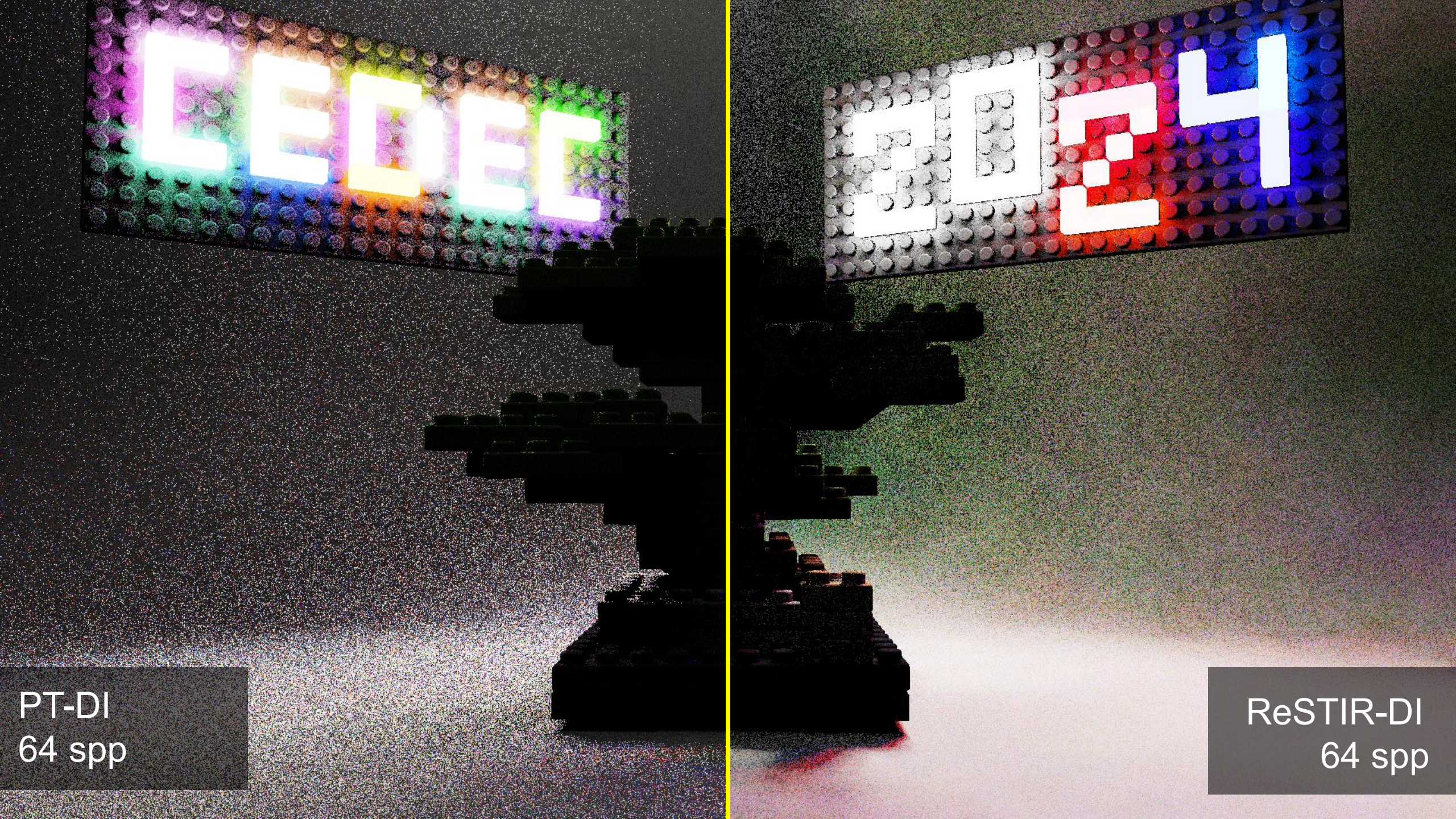
Reservoir 1のサンプル集合

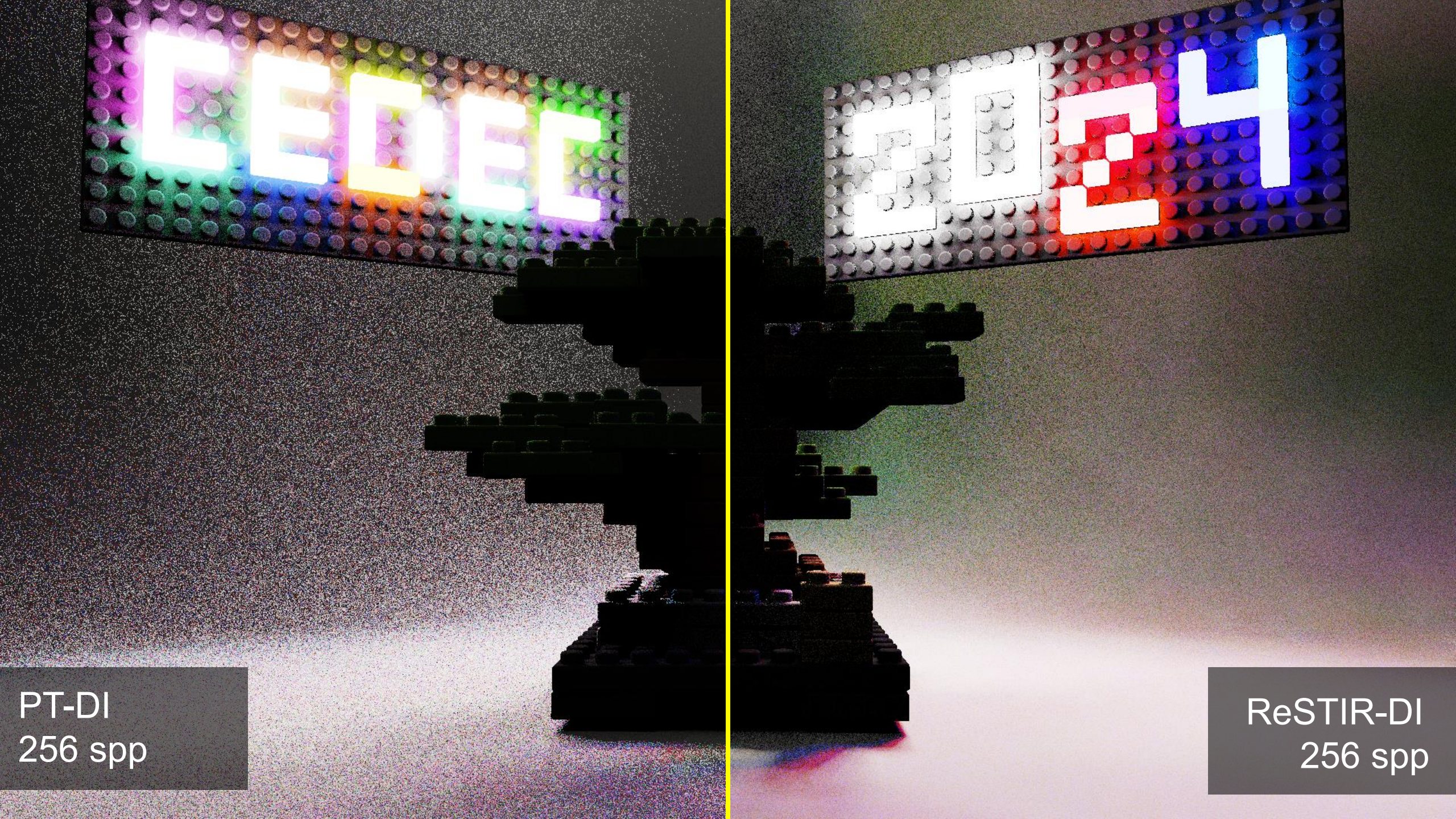
Reservoir 2のサンプル集合

# Result (1サンプル)

- ReSTIRによって寄与を持つサンプルが増えた


PT-DI
16 spp
ReSTIR-DI
16 spp


PT-DI
64 spp
ReSTIR-DI
64 spp


PT-DI
256 spp
ReSTIR-DI
256 spp

# ReSTIRの問題点

- 似たようなピクセル同士でリサンプリングしないとノイズが増える
  - → 寄与の類似度に応じてReservoirをmergeの仕方を変える [Tokuyoshi 2023]
- カメラから見えない点ではリサンプリングできない
  - → ワールドスペースのデータ構造の導入 [Guillaume 2021, Zhang and Wang 2023]
- サンプル同士が強い相関を持つ
  - → MCMCの導入 [Sawhney et al. 2024]


CEDEC
2024
AMD
ARR
Advanced Rendering Research Group
kenel 6.0 ms

# 参考資料

Ray-Triangle Intersection

- Tomas Möller and Ben Trumbore, “Fast Minimum Storage Ray-Triangle Intersection” a.k.a Möller-Trumbore
- Woop et al. “Watertight Ray/Triangle Intersection”

HIPRT

- The project webpage, https://gpuopen.com/hiprt/
- Source code repository, https://github.com/GPUOpen-LibrariesAndSDKs/HIPRT

Others

- The bonsai toy, https://jp.daisonet.com/collections/party0226/products/4550480233550
- Radeon™ ProRender, https://www.amd.com/en/products/graphics/software/radeon-prorender.html

# 参考資料


- Rendering
  - Pharr, Matt, Wenzel Jakob, and Greg Humphreys. Physically based rendering: From theory to implementation. MIT Press, 2023.
  - Bitterli, Benedikt, et al. "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting." ACM Transactions on Graphics (TOG) 39.4 (2020): 148-1.
  - Ouyang, Yaobin, et al. "ReSTIR GI: Path resampling for real-time path tracing." Computer Graphics Forum. Vol. 40. No. 8. 2021.
  - Boissé, Guillaume. "World-space spatiotemporal reservoir reuse for ray-traced global illumination." SIGGRAPH Asia 2021 Technical Communications. 2021. 1-4.
  - Lin, Daqi, et al. "Generalized resampled importance sampling: Foundations of restir." ACM Transactions on Graphics (TOG) 41.4 (2022): 1-23.
  - Tokuyoshi, Yusuke. "Efficient spatial resampling using the pdf similarity." Proceedings of the ACM on Computer Graphics and Interactive Techniques 6.1 (2023): 1-19.
  - Wyman, Chris, et al. "A Gentle Introduction to ReSTIR." (2023).
  - Sawhney, Rohan, et al. "Decorrelating restir samplers via mcmc mutations." ACM Transactions on Graphics 43.1 (2024): 1-15.