CUDAではじめるレイトレーシング

CEDEC 2010

株式会社スクウェア・エニックス
技術開発部　大垣真二

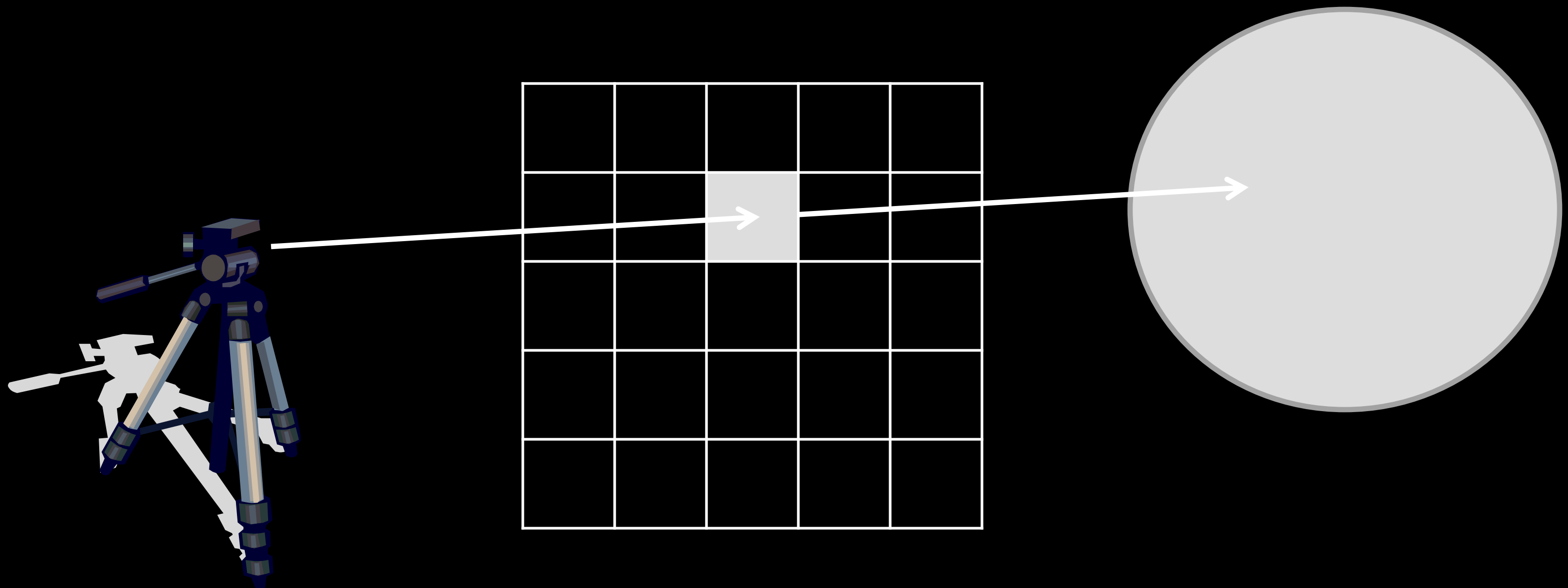
今日の内容

レイ・トレーシングの紹介
CUDAプログラミング
アルゴリズムとデータ構造
GPUを使うメリットとデメリット
実際に作成した画像の紹介

レイ・トレーシング

レイ･トレーシング

- スクリーンのある点を通るレイを視点から追跡してピクセルの色を決め画像を生成するアルゴリズム



レイ･トレーシング

- リアルな画像の生成に欠かせない大域照明の計算に向き、高画質化の一途をたどるゲームにも今後欠かせない要素になると思われる

- 大きなメリットは
  1. シャープな影(シャドウマップのようなアーティファクトがない)
  2. 反射・屈折を正確に表現できる
  3. 大域照明

プリミティブ

- レイ・トレーシングで使われるオブジェクトはプリミティブと呼ばれ、一般的にシーンはこれらの組み合わせで構成される

1. トライアングル(現在主流)
2. 2次曲面(球, 円柱 etc.)
3. パラメトリック・サーフェス(NURBS etc.)
4. ポイント

CUDAについて

CUDAとは?

- “CUDAはGPUの性能を利用することで、コンピューティング能力を劇的に増大させることができる、NVIDIAの並列コンピューティングアーキテクチャです”
http://www.nvidia.co.jp/object/cuda_home_new_jp.html
より引用

CUDAでのプログラムの流れ(1)

- CPUをHost、 GPUをDeviceと呼ぶ
- 以下のような4つのステップからなる

1. Prepare Data (on Host)
2. Transfer Data (from Host to Device)
3. Run Kernel , Synchronization(on Device)
4. Transfer Data from (from Device to Host)

CUDAでのプログラムの流れ(2)

- 2. Textureメモリの使い方

```
texture<float, 1, cudaReadModeElementType> texture;

void DataTransfer()
{
    int size = num_of_elements * sizeof(float);
    float* host = (float*)malloc(size);
    float* device;
    for(int i = 0; i < count; ++i) { host[i] = your_data[i]; }
    CUDA_SAFE_CALL( cudaMalloc ((void**)&device, size) );
    CUDA_SAFE_CALL( cudaBindTexture(0, texture, device, size) );
    CUDA_SAFE_CALL( cudaMemcpy(device, host, size, cudaMemcpyHostToDevice) );
    free(host);
}
```

CUDAでのプログラムの流れ(3)

- 3. Kernelプログラムの実行方法

```
// Device
__device__ inline bool Intersect(int3* color, Ray &ray) { ......... }

__global__ void CastRays( int3* frame_buffer, Ray* ray_stream, int count )
{
    const int thread_id  = blockDim.x * blockIdx.x + threadIdx.x;
    const int number_of_thread = blockDim.x * gridDim .x;
    for(int i = thread_id; i < count; i += number_of_thread)
        Intersect( frame_buffer, ray_stream[i], time );
}

// Host
Intersect <<<256, 256>>> ( frame_buffer, ray_stream, count );
```

CUDAでのプログラムの流れ(4)

- 4. Kernelプログラムの結果を取得方法 (ここではフレーム・バッファ)

```
CUDA_SAFE_CALL( cudaMemcpy(hst_frame_buffer, dev_frame_buffer,
sizeof(int3) * num_of_pixels, cudaMemcpyDeviceToHost) );
```

CUDAでのプログラムの流れ(5)

- カーネル・プログラムの中身はほぼCPU上でのプログラミングと変わらない
- 違いは配列の代わりにTextureをFetchするところ

```
__device__ inline void Intersect(Ray ray, object_id)
{
    int4 vertex_ids = tex1Dfetch( vertex_id_texture, object_id);
    ……..
}
```

書籍の紹介

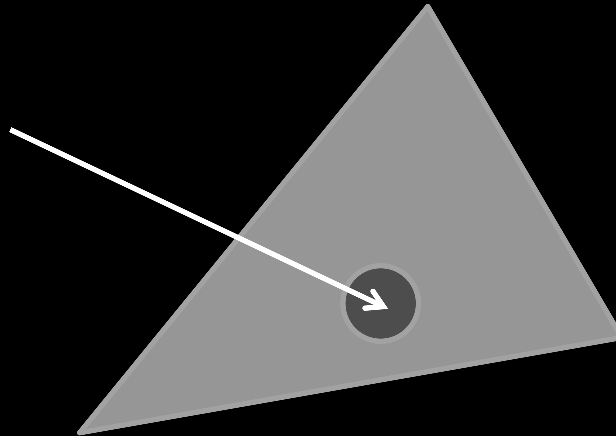
青木尊之・額田彰著
はじめてのCUDAプログラミング

交差判定アルゴリズム

アルゴリズムとデータ構造

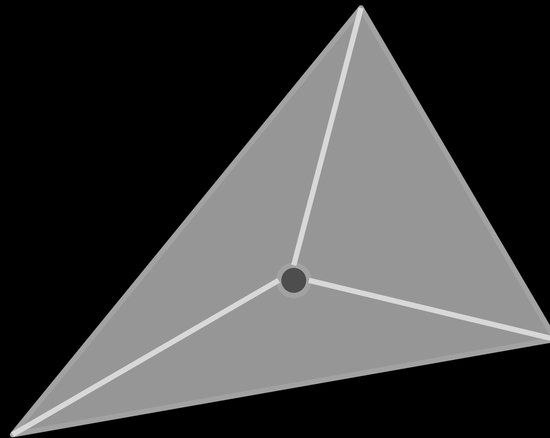
交差判定　トライアングル

1. 平面(ax+by+cz+d=0)とレイ(r=dt+p)の交点を計算する
2. 交点が三角形の中にあるかどうかテストする



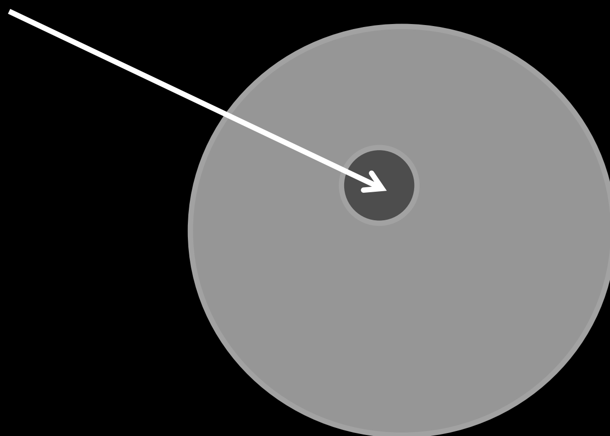
交差判定　トライアングル

- 内外判定は3つの小さい3角形の面積が大きい3角形の面積を超えなければ内、越えれば外とする
- 実際はXY、YZ、ZX平面のうち面積が最大となる面に投影して2Dで計算を行う



交差判定　2次曲面

1. 2次曲面の方程式$axx+byy+czz+dxy+exz+fyz=0$にレイの式$r=dt+p$を代入
2. tについての2次方程式を解く
3. 交点が2つ得られるので近いほうを選択



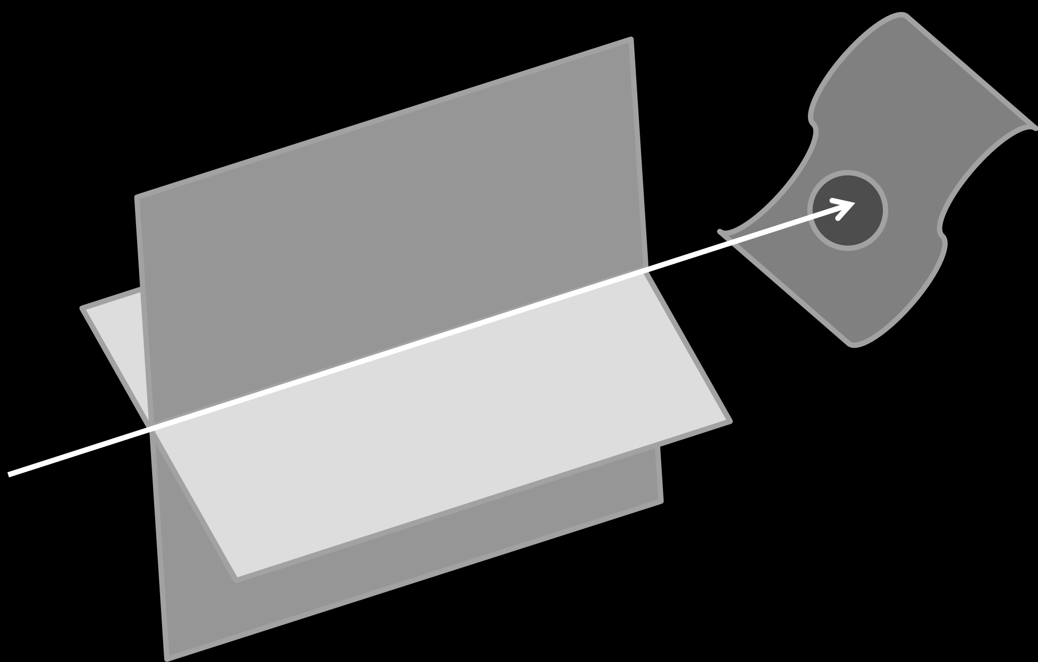
交差判定　2次曲面

- 実際は回転や平行移動といった変換が施されているため、以下の手順で交点を求める

1. レイをオブジェクトのローカル座標へ変換
2. 前ページの方法で交点を求める
3. 交点をローカル座標からワールド座標へ変換

交差判定 パラメトリック･サーフェス

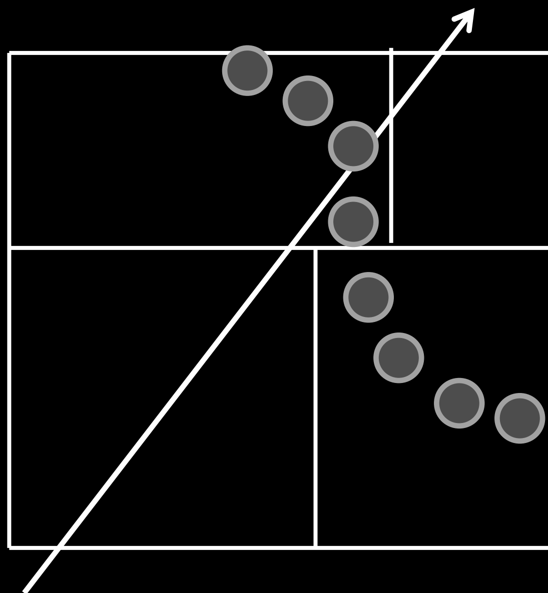
1. レイは2枚の平面の交線として表現できる
2. パラメトリック・サーフェスをf(x,y,z)=0としそこに2つの平面の方程式を代入して、2式を得る



3. 得られた2式を連立してNewton Raphson法で解く

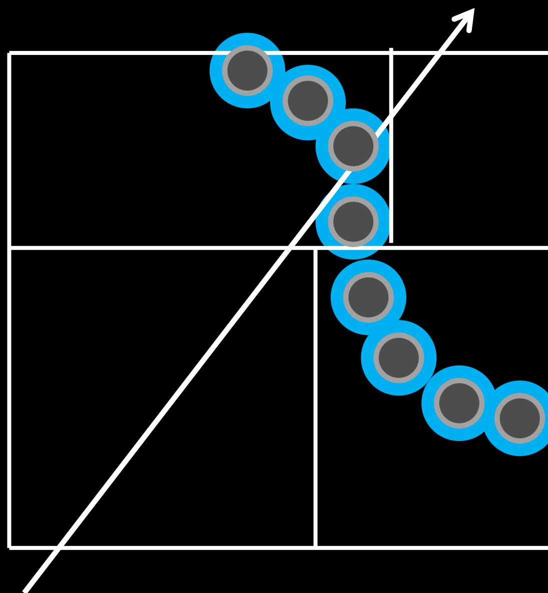
交差判定　ポイント

1. 複数の点からサーフェスを再構成する



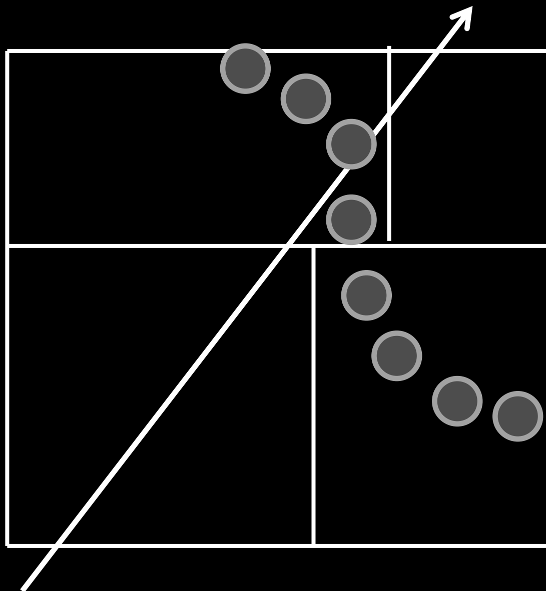
交差判定　ポイント

1. 複数の点からサーフェスを再構成する



交差判定　ポイント

1. 複数の点からサーフェスを再構成する



Markus Gross and
Hanspeter Pfister,
"Point-Based Graphics"

空間分割

アルゴリズムとデータ構造

空間分割

- レイとオブジェクトの当たり判定が必要だが、全てのオブジェクトとテストするわけにはいかないので空間分割を利用し、交差判定の処理を減らす

- 主だったものは次のとおり
  1. Uniform Grid(空間をLxMxN個の格子に分割)
  2. KD-Tree(空間を2分割していく)
  3. BVH(バウンディング・ボックスの階層)

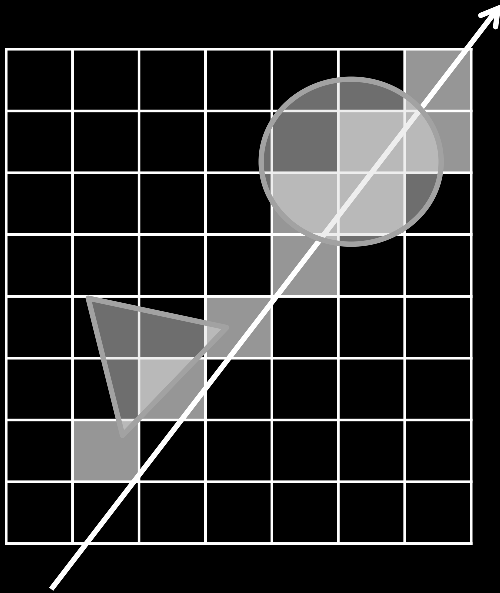
空間分割

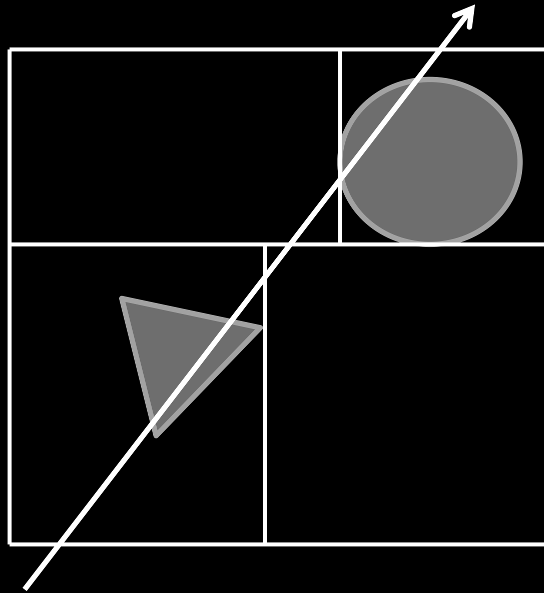
- それぞれの長所と短所

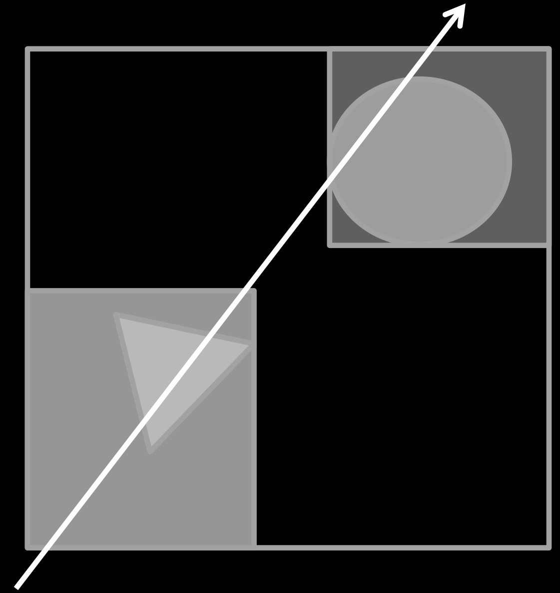
	Uniform Grid	KD-Tree	BVH
長所	構築がはやく簡単	とにかくはやい	構築がはやい まずまずはやい
短所	偏りのあるシーンが苦手	構築がおそい	
難易度	○	△	△

- これらを階層的に組み合わせることも可能

空間分割







- 空間分割構造をたどる処理をトラバーサルといい、これが大きなボトルネック

トラバーサル

- トラバーサルの手法はGPUのレジスタの数の制限などから様々な方法が提案されてきた

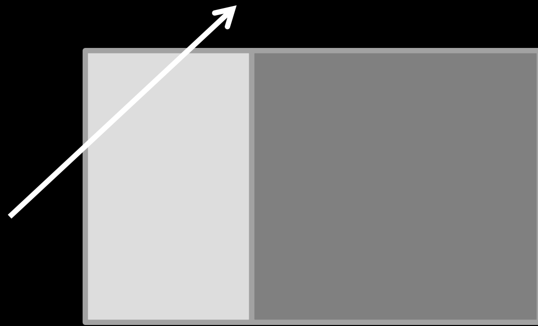
  1. Standard Stack
  2. Short Stack
  3. Stackless

  – Horn et al. Interactive k-D Tree GPU Ray Tracing
  – Stackless KD-Tree Traversal for High Performance GPU Ray Tracing

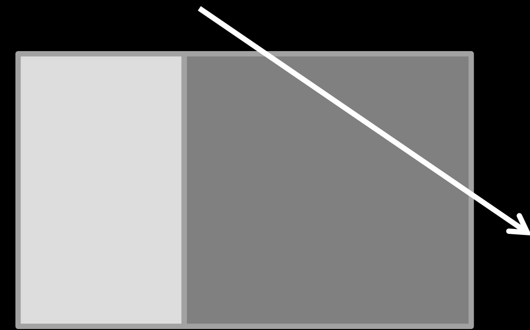
- CPUでは普通のスタックを使った実装でそれなりのパフォーマンスは得られる

トラバーサル

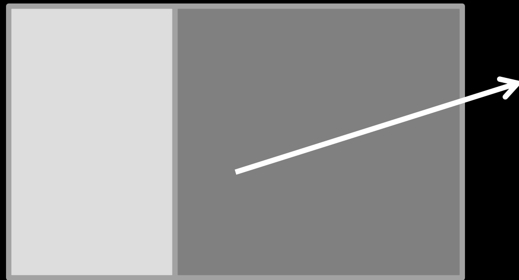
- KD-TreeのStackを使ったトラバーサル



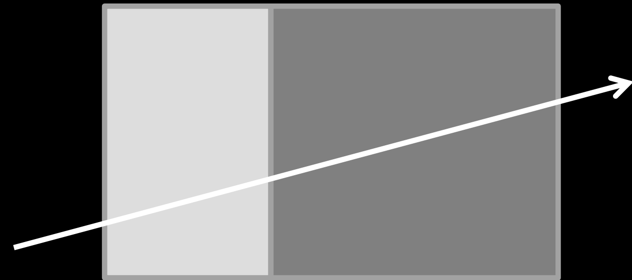
左の子のみ



右の子のみ

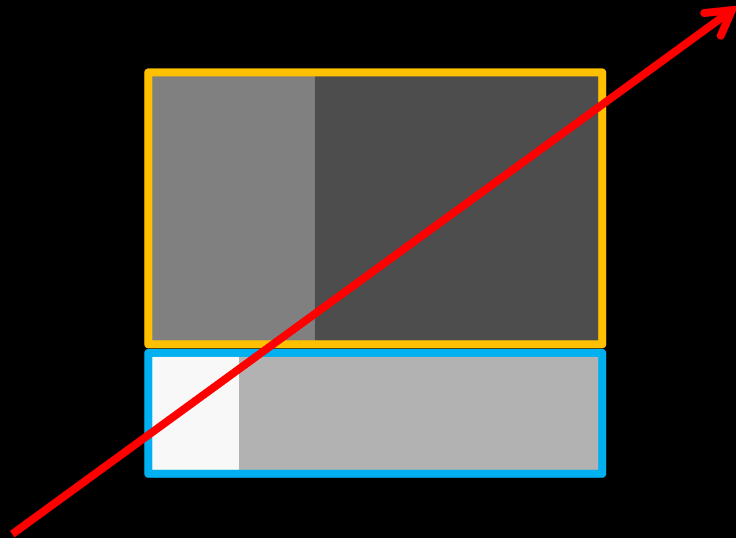


右の子のみ



両方！

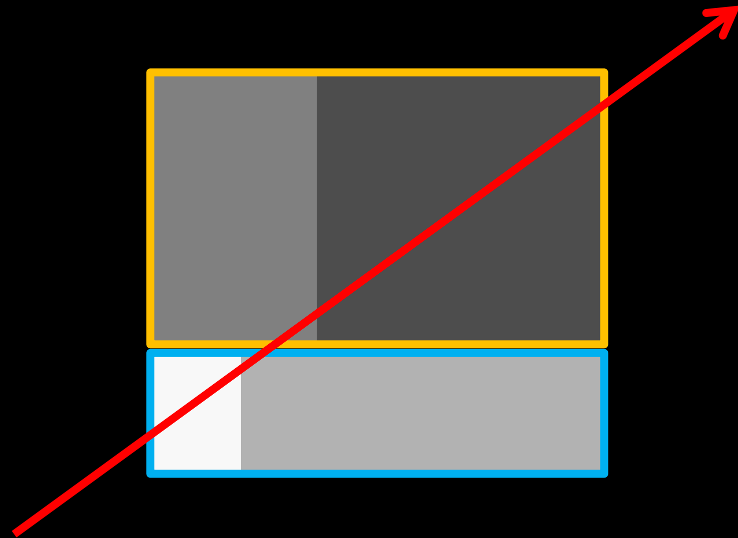
トラバーサル



処理しているノード

スタック

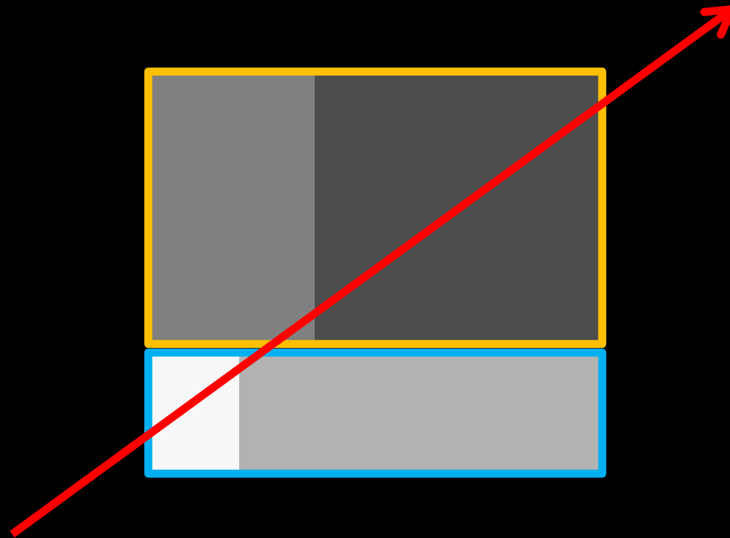
トラバーサル



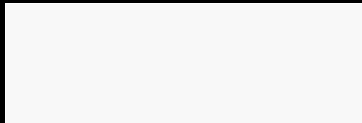
処理しているノード

スタック

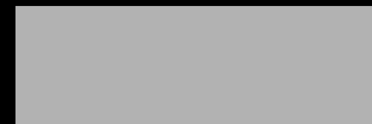
トラバーサル



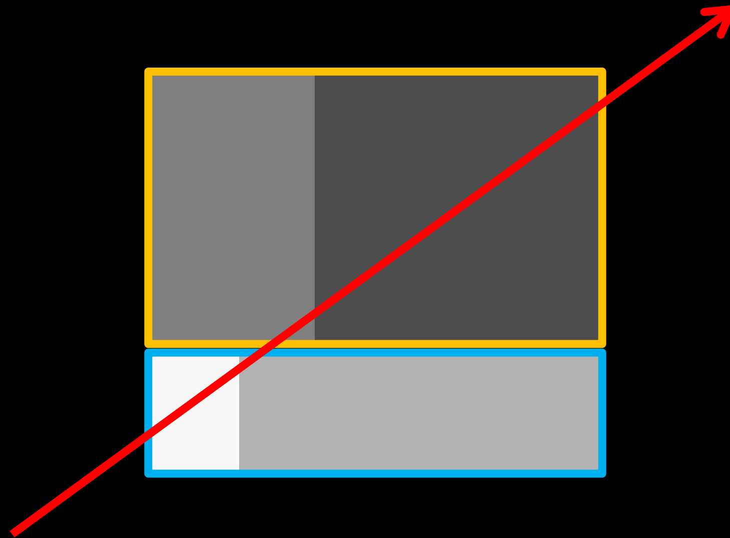
処理しているノード



スタック



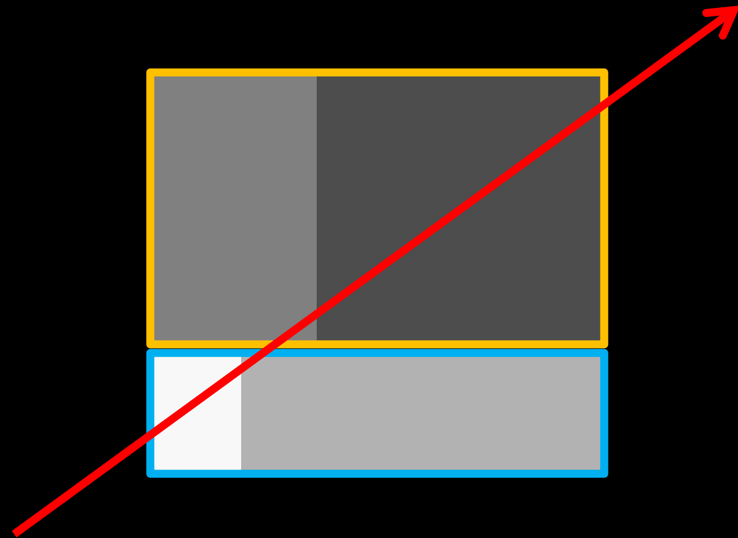
トラバーサル



処理しているノード

スタック

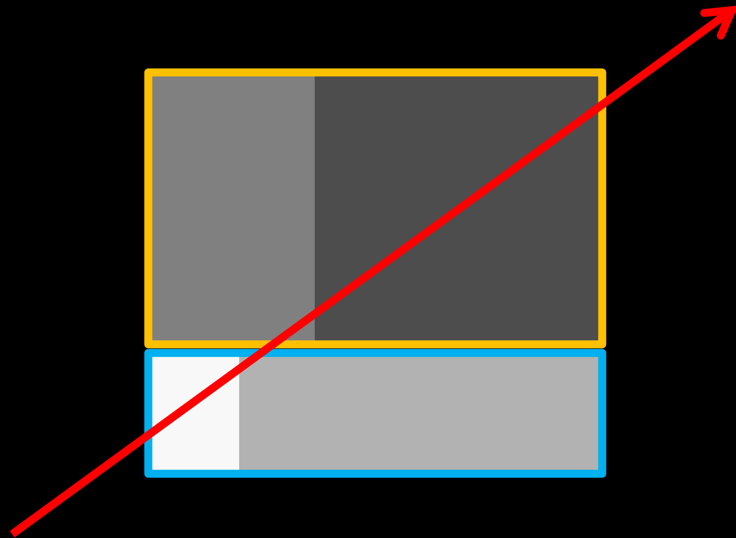
トラバーサル



処理しているノード

スタック

トラバーサル

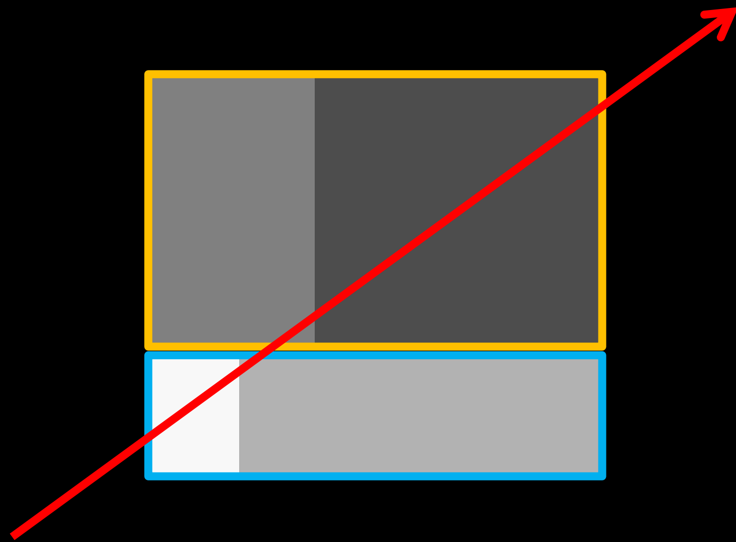


処理しているノード

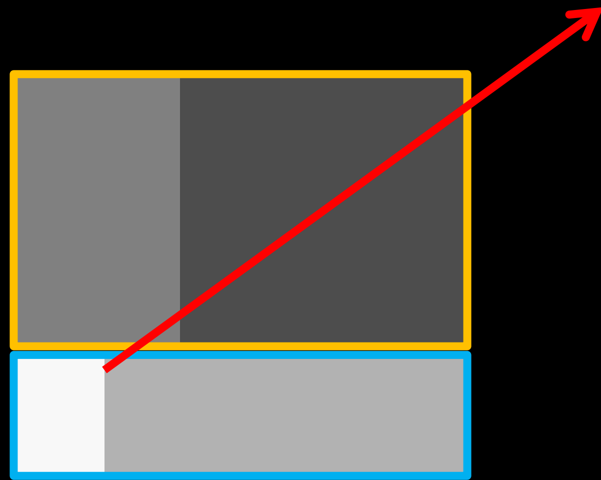
スタック

スタックレス・トラバーサル

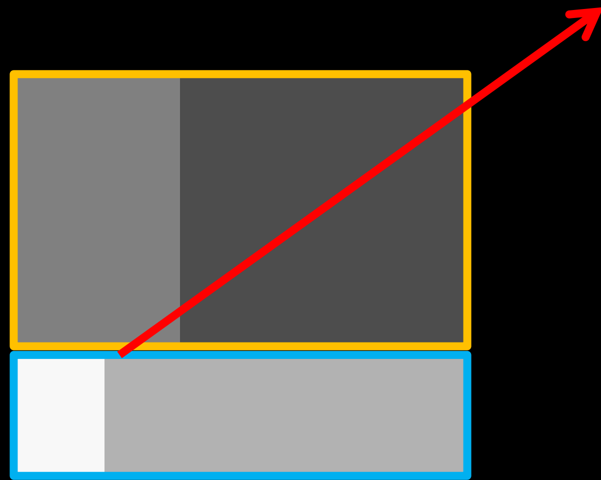
- KD-TreeのStackを使わないトラバーサル



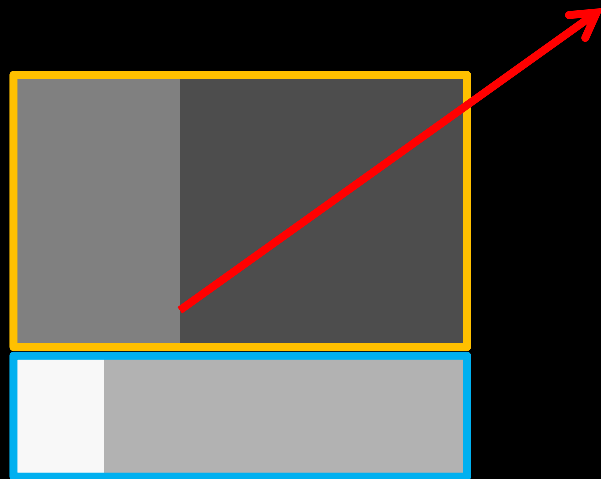
スタックレス・トラバーサル



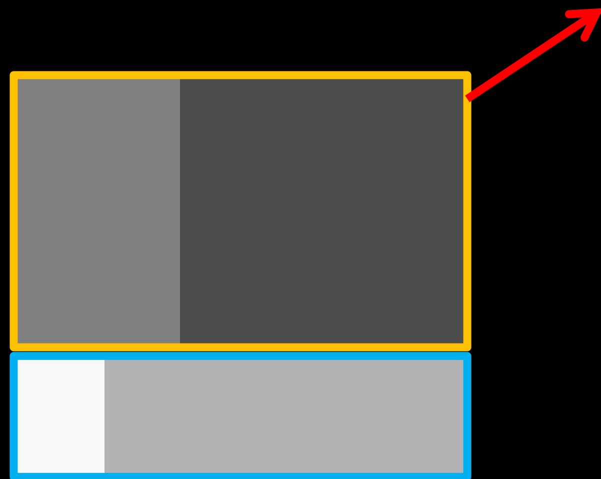
スタックレス･トラバーサル



スタックレス·トラバーサル



スタックレス･トラバーサル



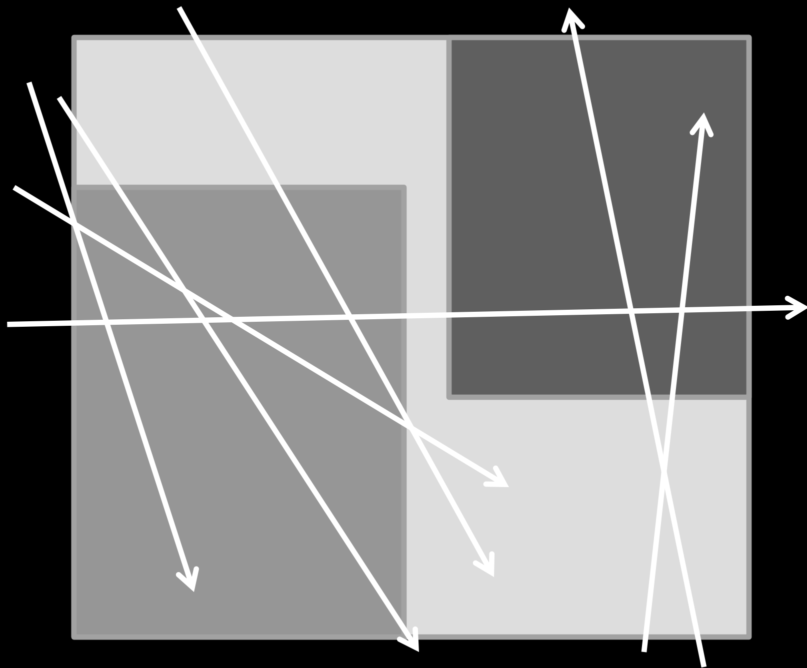
幅優先トラバーサル

- 従来トラバースはレイ1本や数本のパケットに対して行われてきたが、最近幅優先トラバーサルを使った方法が注目を集めている
- 幅優先トラバーサルはノードをたどるコストを償却できる

1. Garanzha et al. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing
2. Tsakok, Faster Incoherent Rays: Multi-BVH Ray Stream Tracing

幅優先トラバーサル

- レイ一本やパケットに対ではなくレイのストリームをまとめて処理する



レイIDのストリームを入力し、ヒットしたノードで2つに分類する

1	2	3	4	5	6 / 6	…	M	…	N

ストリームを2つに分け,一方をスタックに積む

3	5	6	…	N

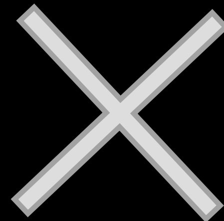
1	2	4	6	…	M	…

あとはこれを再帰的に行う

トラバーサル手法の分類

- フル・スタック+レイ・ストリームがよい

レイ
シングル
パケット
ストリーム



スタック
フル
ショート
なし

コヒーレント・レイトレーシング

パフォーマンスの改善

コヒーレント・レイ・トレーシング

- レイの相関を利用する
- 相関とは
  - レイが同じ方向である
  - レイが同じ始点をもつ
- なぜ効率が良いか?
  - メモリアクセス (キャッシュのヒット率)
  - コードの分岐 (Warp Divergence)

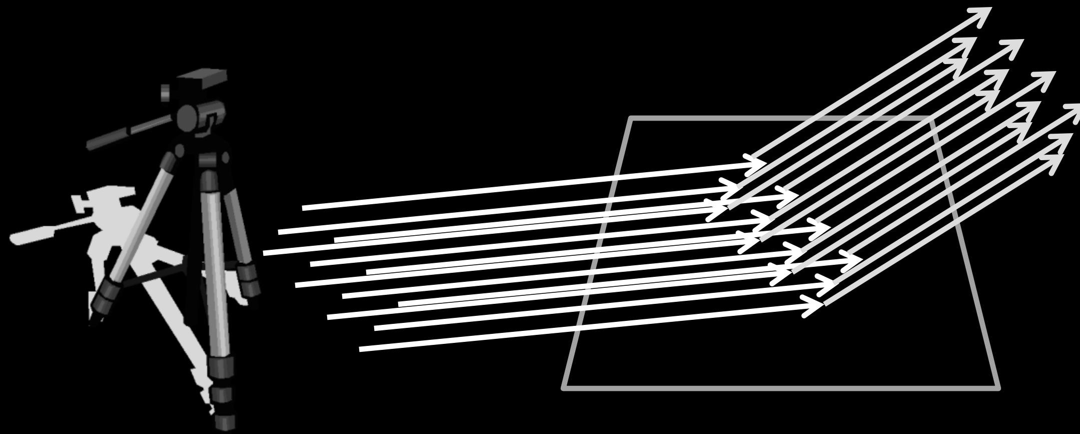
コヒーレント･レイ･トレーシング

- 相関の利用の仕方は一般的に次の2通りである

1. 計算順序を工夫する方法
2. データの並びを工夫する方法

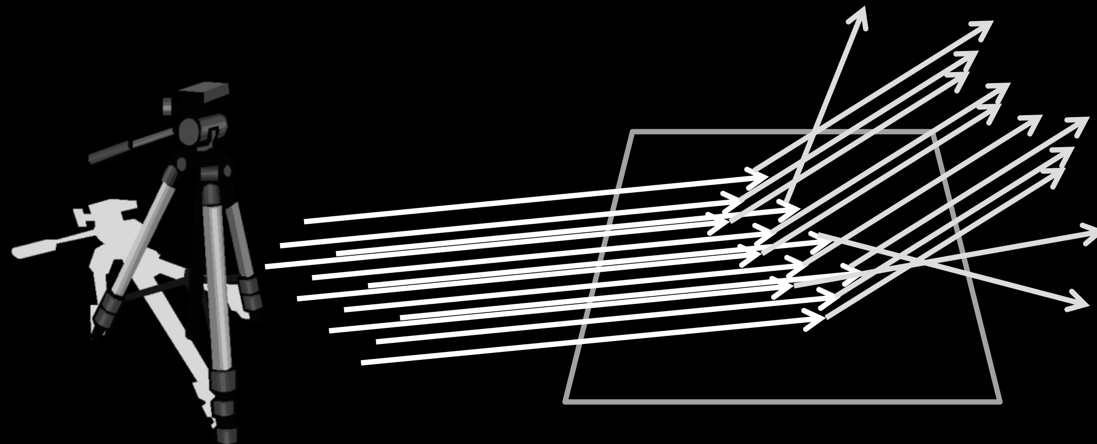
コヒーレント・レイ・トレーシング

- 効率のよい例
  同じ向きから来たレイが同じ方向に飛んでいく



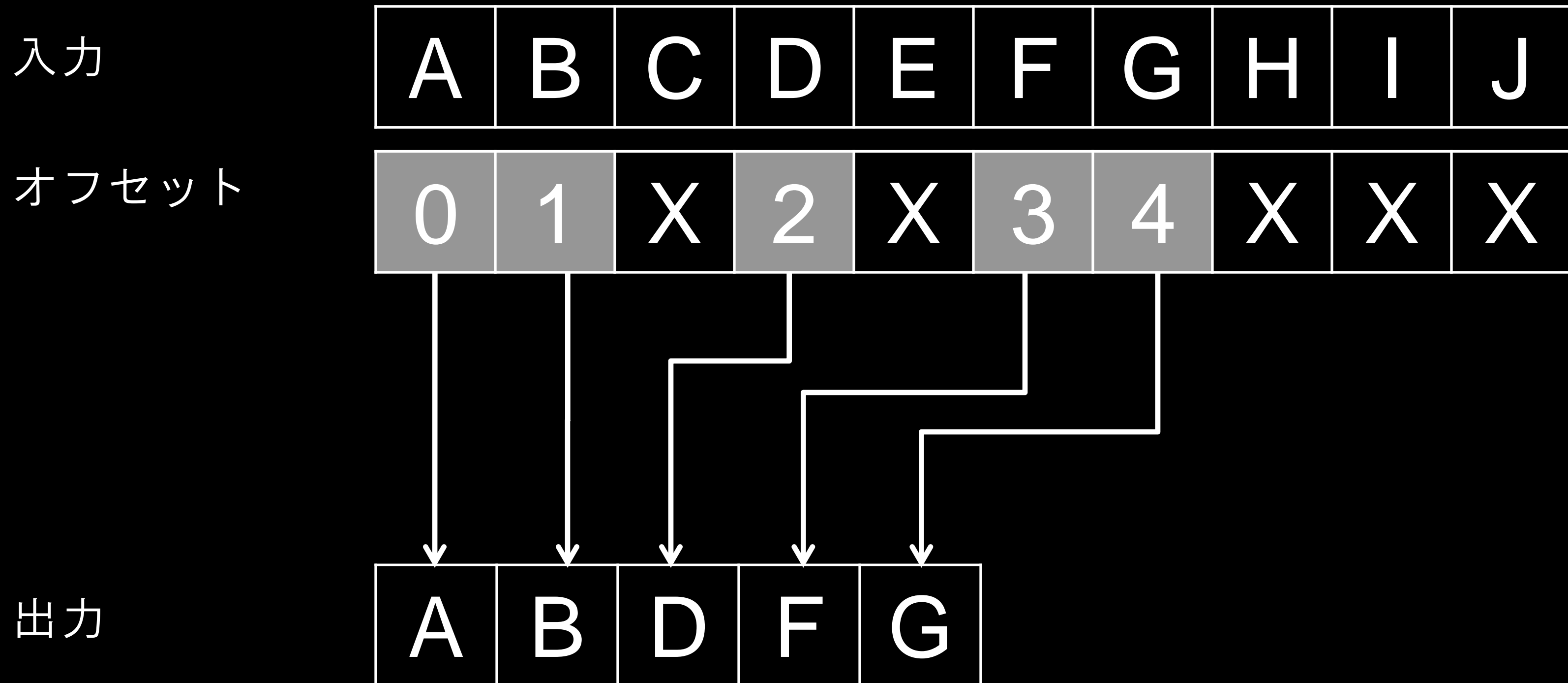
コヒーレント・レイ・トレーシング

- 効率の悪い例
  反射や屈折を経てレイがばらばらの向きに
- 解決法
  1. ストリーム・フィルタを使う(簡単)
  2. ソート+幅優先トラバーサル(比較的難しい)



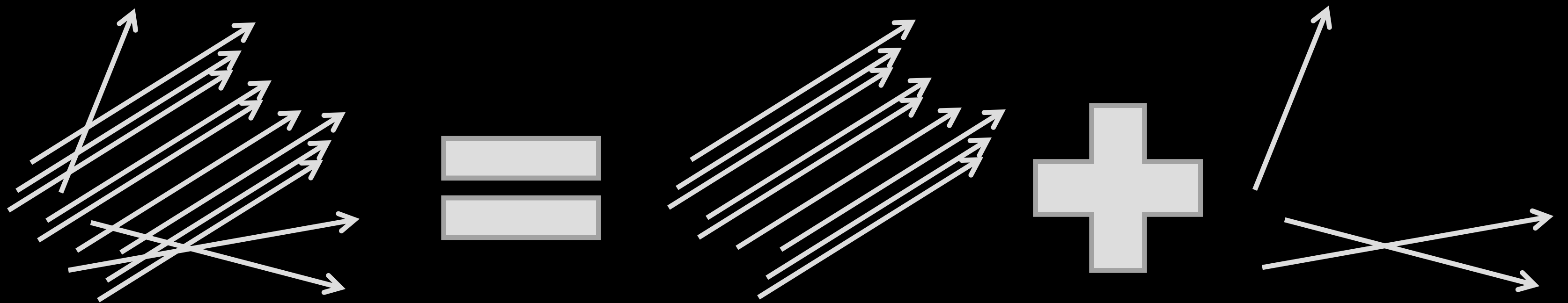
ストリーム・フィルタ

- GPUがもつScatter/Gather Operationをつかう



ストリーム・フィルタ

- レイがヒットしたマテリアルの種類でふるいにかけることで、比較的容易に相関の高いレイを取り出せる



レイのソート

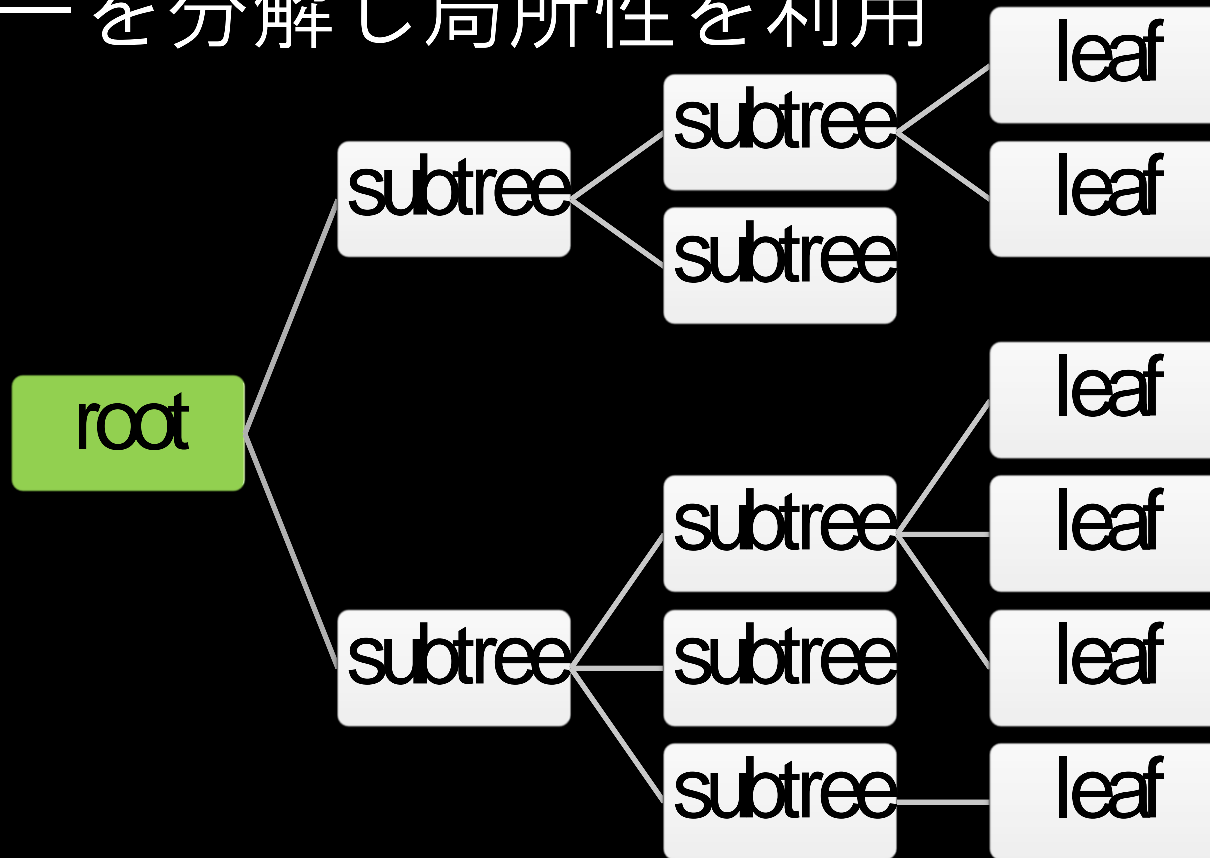
- 5D Morton Codeを使う
- レイの始点と方向でソートする
- ソート後、隣接するレイは相関をもつ
- ソートにコストがかかる

新しい手法

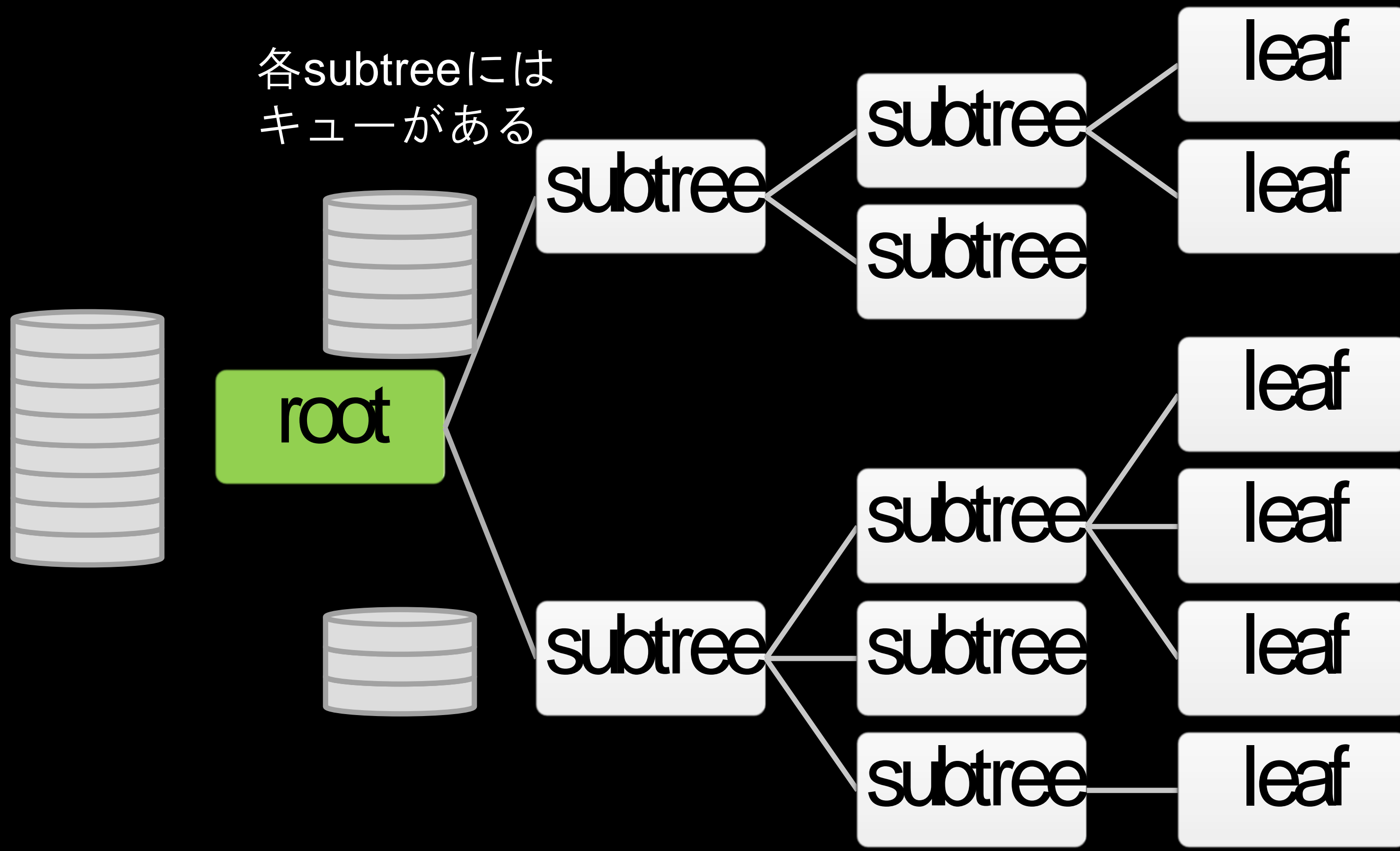
パフォーマンスの改善

新しい手法

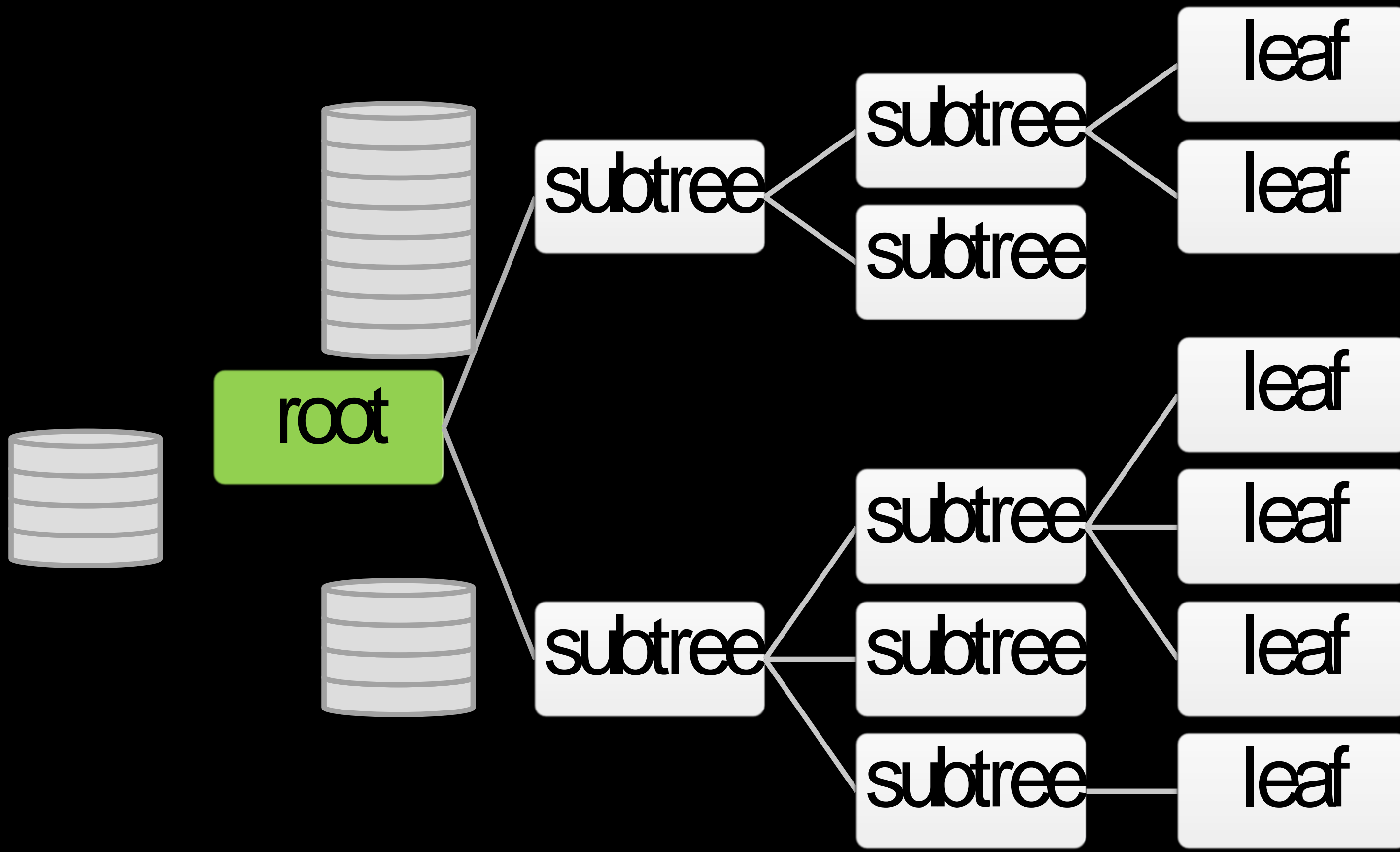
- ツリーを分解し局所性を利用



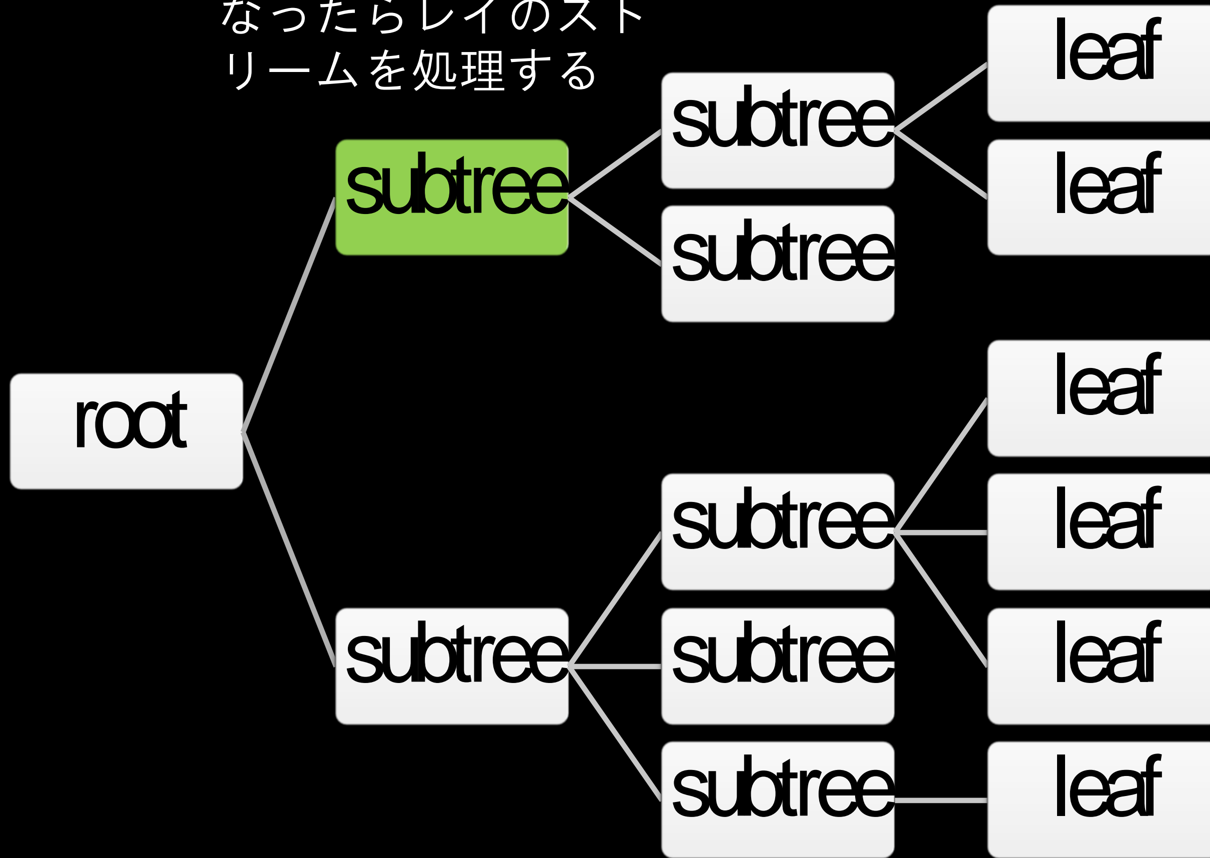
新しい手法



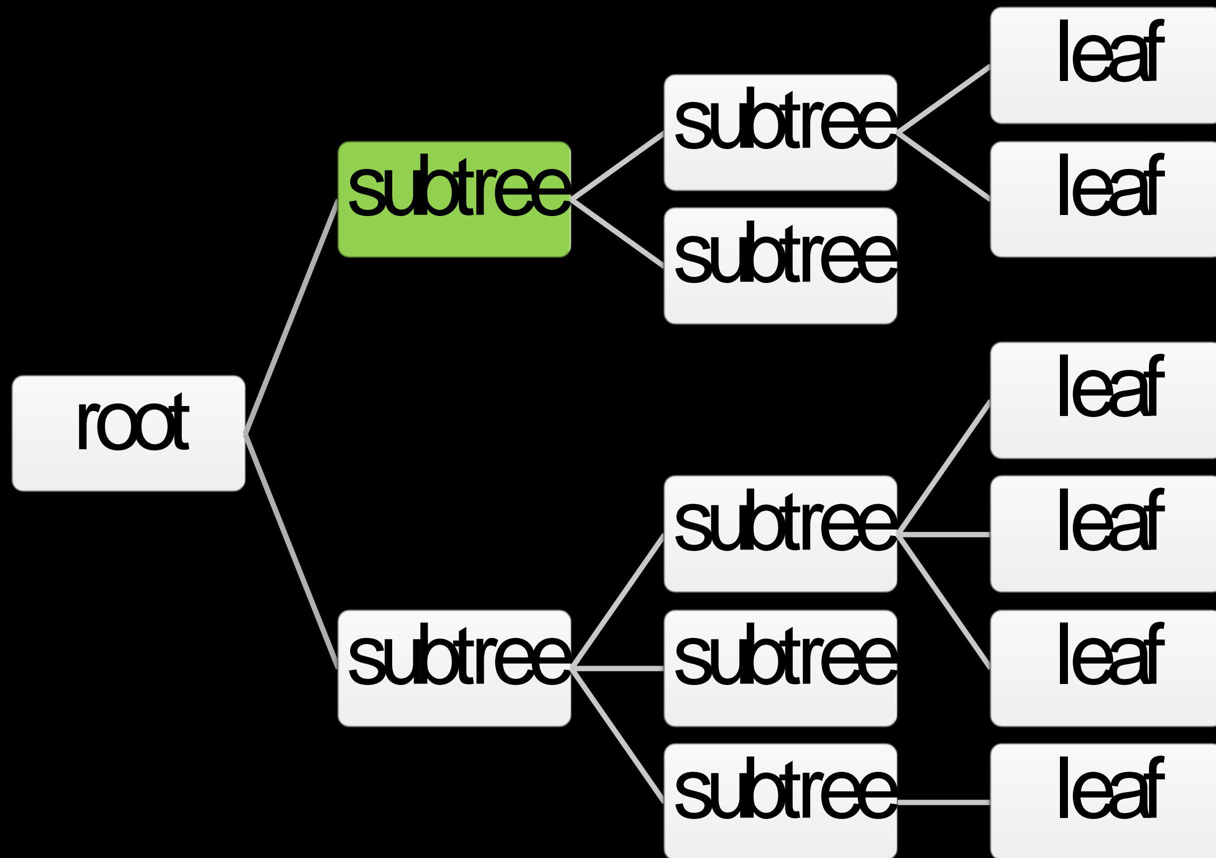
新しい手法



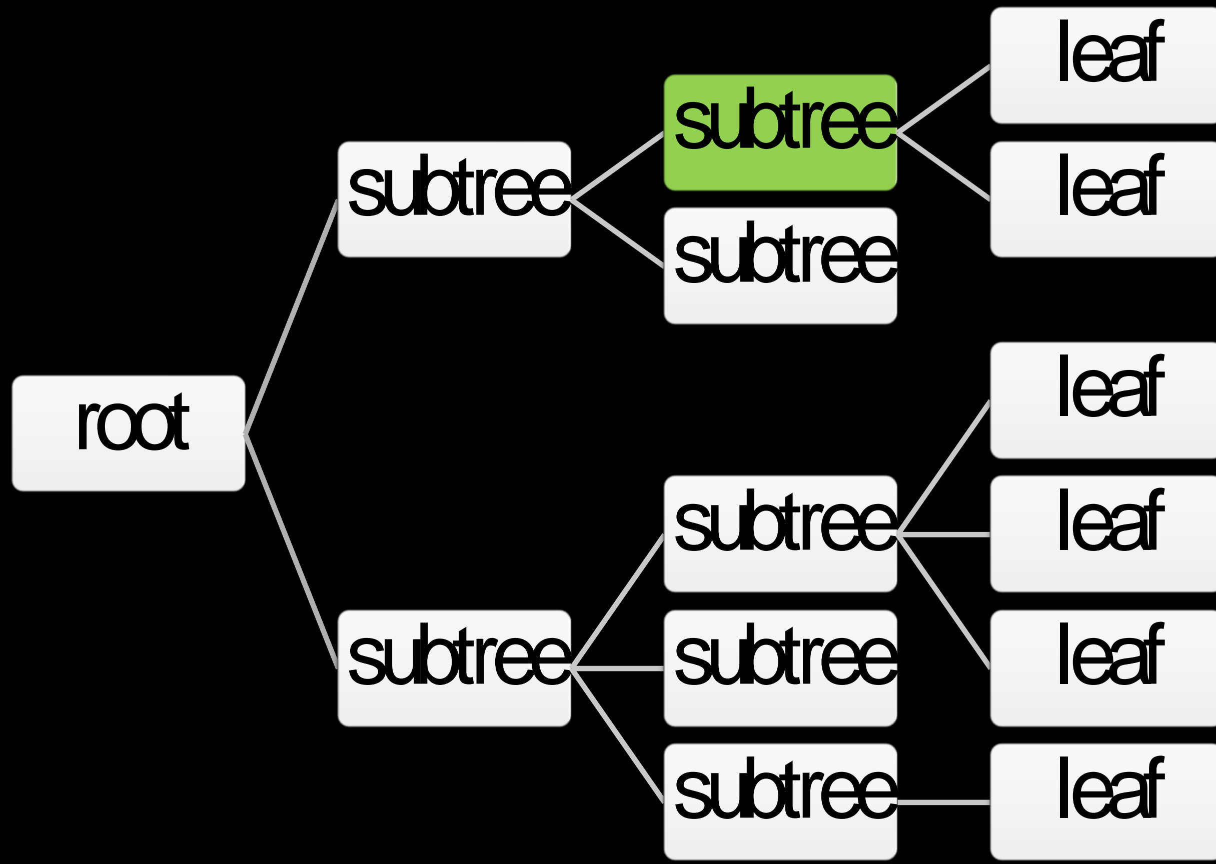
新しい手法



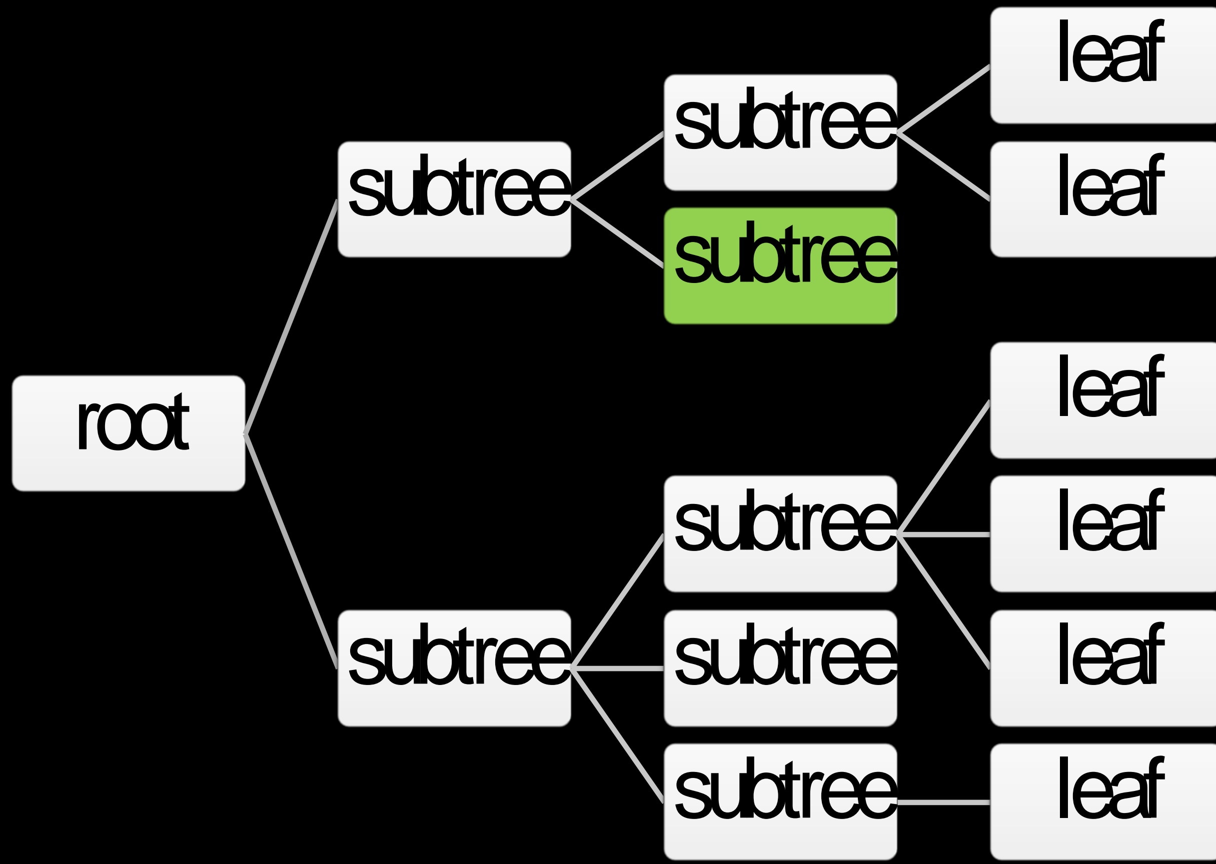
新しい手法



新しい手法



新しい手法



GPUを使うメリット・デメリット

GPUの利点

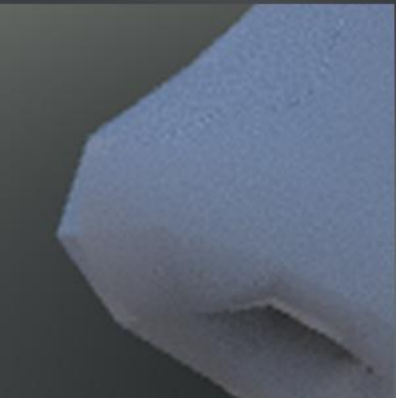
- 相関が高いレイには非常に効果的である(レイの密度が上がるほどパフォーマンス[ray/sec]が良くなる)

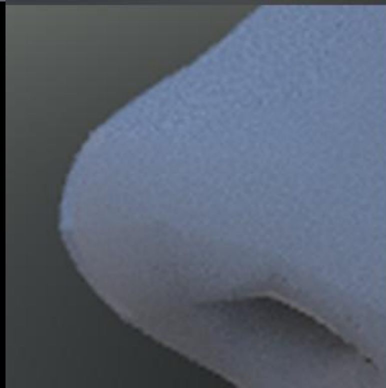
1. アンチ・エイリアス
2. モーション・ブラー
3. 半影
4. アンビエント・オクルージョン

CUDAの問題点

1. パフォーマンスを引き出すにはばらばらになったレイをどう扱うかが課題
2. 反射や屈折で分岐したレイの取り扱いがパフォーマンスの低下につながる
3. カーネルプログラムの同時実行の数は任意ではなく上限がある(GTX 470/480)
4. キャッシュが小さくL2 まで (GTX 470/480)
5. テクスチャの配列が作れない(3次元テクスチャ)

画像の紹介





ご静聴ありがとうございました