GPU を用いた光シミュレーションのためのレイトレーシングの研究

蔵野 裕己
Yuki KURANO

1　はじめに

知的照明システムの研究において，実システムでの実験は不可欠であるが，必要な機器が多く，実システムの導入は容易ではない．また，知的照明システムを，実験スペースやオフィスなどに導入する際，事前に導入後の様子を再現できることが望ましい．そのため，計算機上で知的照明システムの環境を再現可能な，シミュレータの需要が大きい．そこで我々は，3DCG で照明環境を再現可能な，知的照明システムのシミュレータ開発に取り組んでいる．

レンダリング手法として，高品質な 3DCG が描画可能である，レイトレーシング法を採用した．また，知的照明システムの研究において，色は非常に重要であり，高い色表現能力が望まれる．そこで，色は光の波長成分によって表現されることを利用し，レイトレーシング法の色計算において，光の波長成分から色を算出する方法を採用した．

このような高い表現能力の実現には，大量の計算が必要となり，シミュレータの実行速度低下が懸念される．しかし，レイトレーシング法は，計算量が膨大な一方で，並列性の高いアルゴリズムであるため，並列計算を用いて高速化可能である．また波長成分を用いた色計算に関しても，光源毎に計算が独立しているため，並列化可能である．そこで，本研究報告では，我々が提案する光シミュレーション用レイトレーシング法の高速化に取り組み，その効果を評価する．

2　レイトレーシング法

2.1　レイトレーシング法のアルゴリズム

レイトレーシング法は，光線の振る舞いを再現することで，高品質な 3D 画像が生成可能なアルゴリズムである．照明シミュレータでは高い表現能力が必要となるため，レイトレーシング法を採用した．

Fig.1 にレイトレーシング法の概要図を示す．レイトレーシング法の計算手順は以下の通りである．まず図 1 のように，視点からスクリーン上の 1 つの画素に向けてレイ（光線）を発射する．このレイを視線ベクトルと呼ぶ．計算により，視線ベクトルと，空間中に定義される物体との交点を求める．交点が複数存在する場合，視点に最も近い交点を選択する．これらの計算を，交点計算と呼ぶ．

交点は，視線が到達した場所であるため，交点における物体の色，光の色や強さ，反射や屈折の有無などから視点に届く色を算出する．この色を，視線ベクトルが通過するスクリーン上の画素の色とする．これらの色を求める計算は，シェーディングと呼ばれる．

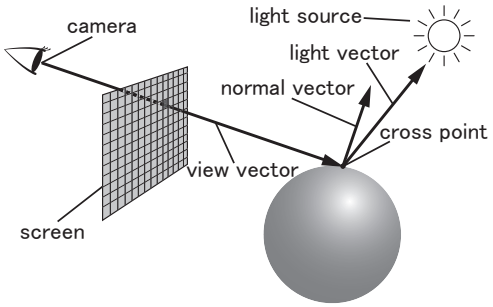

Fig.1　レイトレーシング法の概要

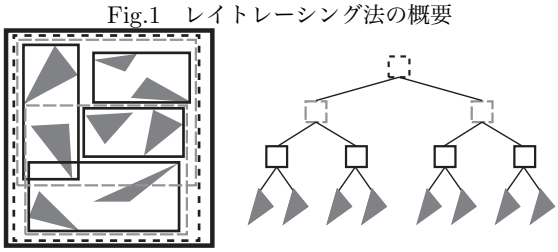

Fig.2　BVH 法による空間分割

より写実的な画像を生成するには，光の反射や屈折を表現する必要がある．反射や屈折は，物体と視線ベクトルの交点から，さらに新たなレイを飛ばすことで表現する．例えば，強く反射する面とレイが交差すると，レイが物体表面と成す正反射方向に新たなレイを発射する．このような 2 次的なレイの照射は，規定の回数だけ繰り返される，もしくは反射も屈折も生じない物体と交わるまで行われる．

2.2　空間分割手法

基本的なレイトレーシング法では，交点計算の際に，一つのレイにつき，空間に存在する全ての物体に対して交点を算出する．高品位な 3DCG では，物体の数は 1000 万を超え，これら全ての物体と交点を算出すると膨大な量の処理が必要になる．そこで，空間を分割して管理することにより，処理量を少なくすることが可能である．空間分割には複数の手法が存在する．その中でも，高速化の効果が高く，また今後の高速化で必須の GPU への適用がしやすい BVH 法を用いて高速化を図る．

2.3　BVH 法

BVH 法では，空間中の各物体を階層的に管理する．例えば，AABB(Axis Aligned Bounding Box) と呼ばれる，x,y,z 軸に平行な面で構成される特殊な直方体を用いて，各物体を階層的に包む．AABB だけでなく，球や直方体などを用いて管理する場合もあるが，ここでは広く用いられている AABB を採用する．

Fig.2 の左に，AABB を用いて空間を管理した場合の

様子を示す．なお，説明の簡便化のために 2 次元図を用いる．この空間には，8 つの物体が存在する．まず，これら全ての物体を包含する AABB を定義する．次に，その AABB 内の物体を 2 つのグループ分け，それぞれを包含する AABB を定義する．なお，このグループ分けは位置を基準にしており，近くに存在する物体同士を集める．さらに各 AABB 内の物体を 2 つのグループ分け，AABB を定義する．このように再帰的に処理を進め，AABB の含まれる物体の数が 1 つ，もしくは 2 つになったところで処理を終了する．このようにして完成する階層的な AABB の構造は，ツリー構造で表現可能である．Fig.2 の右のにその例を示す．ルートノードが最大の AABB に対応し，リーフノードは物体そのものに対応する．

BVH 法で管理される物体との，交点計算の手順について述べる．最初に最大の AABB と交差判定を行う．交差する場合，子ノードの AABB との交差判定を行うため，再帰的に交差判定を行う．リーフノードの親ノードに到達すると，子ノードが管理する物体との交差判定を行う．複数の交点が発見された場合，最も視点に近い交点を選択する．また，AABB と交差しない場合，その AABB が含む全ての物体と交差しないことになるため，交差判定の処理を打ち切る．この打ち切りにより，全ての物体との交差判定を行う必要性がなくなるため，処理量が減少する．

3 レイトレーシング法の高速化手法

3.1 BVH 法の並列計算手法

BVH 法を適用させたレイトレーシング法においてプロファイルを取得すると，トラバーサルが全処理時間の半分以上を占めることがわかった．また，トラバーサル時の，レイと AABB との交差判定の処理は，AABB 毎に独立した処理であるため，並列処理可能である．そこで，トラバーサルの演算を GPU，FPGA などの専用機器で並列処理することで，高速化を図ることが可能である．

3.2 交点計算の並列計算手法

レイトレーシング法では，画素ごとに独立して交点計算が行われるため，並列処理可能である．画素毎に並列に演算する手法は，画素並列法と呼ばれ，並列処理によるレイトレーシング法の高速化において広く利用されている [1)]．

4 GPGPU 実装

4.1 GPGPU

GPU は，画像処理専用のハードウェアであり，数百〜千数百ものコアを有する．近年，GPU の大量のコアを用いて汎用計算を並列実行する，GPGPU（General Purpus computing on GPU）のための開発環境が整備され，注目を集めている．本研究では，NVIDIA 社が提供する統合開発環境，CUDA を用いて GPU プログラムを記述した．言語としては，C/C++ の拡張言語である，CUDA C を用いた．CUDA C は，C/C++ の分法をもとに，CPU で行う処理，GPU で行う処理を記述できる．加えて，GPU とのデータ送受信，GPU 上のメモリ確保，解放などの機能が拡張されている．これらの機能を用い，一部の計算を GPU 上で高速に並列計算を行う．

4.2 CUDA における処理の流れ

CUDA では，CPU で演算を行うホスト，GPU で演算を行うデバイスの処理を分けて記述する．デバイスの処理は kernel 関数と呼ばれる特殊な関数に記述して行い，それ以外をホストで処理する．デバイスで処理を行う際の，大まかな処理の流れは以下の通りである．

- ホストでデバイスのメモリ確保を行う．
- ホストのメモリからデバイスのメモリへ，計算に用いるデータを転送する．
- デバイスで，転送されたデータを処理する．（kernel 関数）
- デバイスのメモリからホストのメモリへ，計算結果のデータを転送する．
- ホストでデバイスのメモリ解放を行う．

4.3 GPU の構造

GPU は複数の Streaming Multi Processor(以降 SM) から成り，SM は複数の Streaming Processor(以降 SP) から成る．計算資源は，grid，block，thread の 3 つの単位で管理され，1 つの grid は 1 つの GPU に対応する．また，thread は block と呼ばれる単位でまとめて管理される．1 つの SM は 1 つの block に対応する．1 つの SP は 1 つの thread に対応する．thread がホストから渡される処理の最小単位である．

即ち，同一 block に属する thread は，同一 SM に属する SP に割り当てられ，処理される．各 SP 上で thread が同時に処理されることで，並列計算が実現する．また，thread を 32 個まとめた warp という単位で同じ処理が発行され，同時に実行される．そのため，thread 数が 32 の倍数の場合に効率よく処理される．

4.4 実装

トラバーサル部分を kernel 関数として実装する．kernel 関数には木の情報とレイの情報を与える．そして，トラバーサルした結果，交差する候補となる物体のリストが返される．このリストに格納された物体との交差判定は，ホスト上で行う．

CUDA では，大量の thread を生成し，GPU 上の大量の SP を使って処理することで高い演算能力が得られる．また，warp の大きさである 32 個のスレッドをまとめた時に高い演算性能を発揮する．そのため，32 個の thread を生成し，同時に処理することが望ましい．

BVH 法で最も一般的な 2 分木では，各ノードは 2 つの AABB しか管理していない．そのため，複数のノードを辿り，交差判定を行う必要がある AABB を 32 個揃える必要が生じる．そこで，物体を 32 分木で管理することにより，32 個の AABB を一つのノードで揃えることが可能となる．

本稿では，このトラバーサルの GPU 実装は行ったが，

Fig.3　実験に用いたモデル

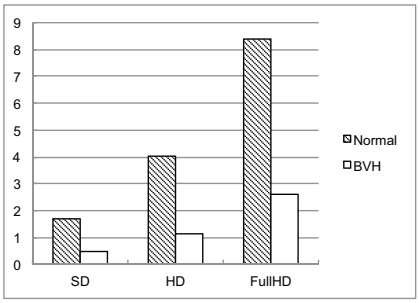

Fig.4　BVH 法と実行速度の関係-small

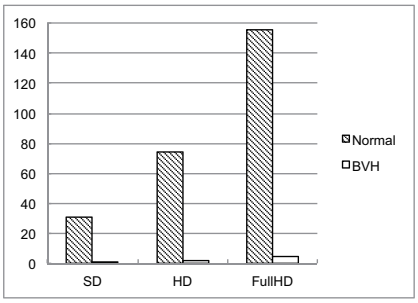

Fig.5　BVH 法と実行速度の関係-large

画素ごとの並列性を生かした並列実装は行うことができなかった．

5　実験

5.1　BVH 法による実行速度

まず，BVH 法を適用した場合としなかった場合について実行速度を調査した．レンダリングに用いたモデルは，Fig.3 に示すようなうさぎのモデルである．面の細かさにより，small と large の 2 種類存在する．それぞれ，small の面の数は 948，large の面の数は 16301 である．また，480 × 640 の SD，720 × 1280 の HD，1080 × 1920 の FullHD の 3 種類の解像度で速度を調査した．実験環境は 1 に示す．

small に対する実行結果を Fig.4 に，large に対する実行結果を Fig.5 に示す．結果より，BVH 法を適用することにより，大幅な実行速度向上が認められた．BVH 法を適用しない場合，画素数，モデルの面数の影響を大きく受けながら，比例して実行時間が大きくなった．一方で BVH 法を適用した場合．画素数と実行時間は強い比例関係を示したが，モデルの面数と実行時間の比例関係は，弱いことがわかる．これは，BVH 法のトラバーサルにおけるオーバーヘッドの支配率は，画素数の増減と関係なく一定である．しかし，モデルの面数が多くなると，オーバーヘッドの支配率が小さくなるためと考えられる．

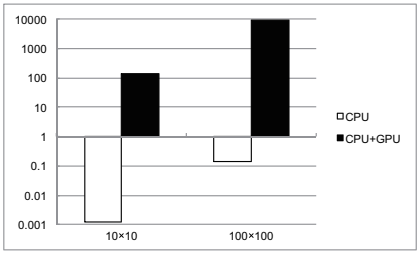

Fig.6　トラバーサルの GPU 処理化と実効速度の関係

5.2　トラバーサルの GPU 処理による実行速度

BVH 法を適用した実装，トラバーサルを GPU で並列化した実装を比較した．モデルは small を用い，画素数は 10 × 10，100 × 100 の場合の速度を比較した．

実行結果を Fig.6 に示す．GPU を用いた方が，実行速度が遅くなった．これは，GPU のメモリ確保，開放，データ転送を画素毎に行なっていたことが原因と考えられる．ツリーのデータは各画素によって共通なので，転送は一度だけ行って使いまわすことが可能と考えられる．また，各画素の処理で使用するメモリの領域は同じなので，確保，開放も一度だけ行えば良いと考えられる．

本稿では，画素毎の並列性を生かした実装は行わなかったが，この並列性を行うことでさらなる高速化が見込める．しかも，GPU へ処理を投げる際に，複数画素分の処理を投げることになるため，処理にかかるオーバーヘッドも削減可能と考えられる．

6　まとめと今後の展望

本稿では，レイトレーシング法の高速化に取り組んだ．BVH 法によるアルゴリズム面の高速化と，GPU による高速化の効果の検証を行った．結果として，BVH 法の有効性が確認できた．一方で，GPU 実装は，メモリの確保，および開放方法や転送方法に不備があり，高速化できなかった．今後は，このメモリ確保，開放および転送の最適化と，画素毎の並列性を生かした実装を行い，高速化に取り組む．また，光のスペクトルから色を計算する機能を実装し，色の再現性を高める．そして，色計算も並列性を持つため，GPU での高速化を検討する．

参考文献

1) 平田貴光, 安部毅, 大野義夫. レイトレーシングの並列化. 情報処理学会研究報告. グラフィクスと CAD 研究会報告, Vol. 94, No. 41, pp. 25–32, 1994-05-20.

Table1　実験環境

CPU	Core i5 2500
GPU	GTX 460
CPU code Compiler	gcc 4.4.7-1ubuntu2
GPU code Compiler	CUDA toolkit 5.0
OS	Ubuntu12.10 amd64
Compil Option	-O3, -lm