卒業論文

レイトレーシング法における

スクリーンマッピングの設計と実現

氏名　　：山田　遼
学籍番号：2260090067-0
指導教員：山崎　勝弘　教授
提出日　　：2013 年 2 月 19 日

立命館大学　理工学部　電子情報デザイン学科

内容梗概

レイトレーシング法は、視点から出た光線を追跡して、交差する物体を調べ、その物体の輝度値をスクリーン上に投影することで画像を得る手法である。レイトレーシング法では光線と物体の交差判定が全処理の大部分を占めるので、交差判定を高速化することが重要である。

本論文では、レイトレーシング法による画像生成を 100ms 以下のリアルタイムで実現するために、GPU を用いた画面分割の並列化、スクリーンマッピングによる光線数の削減について検討し、処理系を実現して、処理速度を評価する。

GPU を用いた画面分割の並列化では、画像が生成されるスクリーンを複数のブロックに分割し、各ブロックをストリーミング・マルチプロセッサ（SM）に割り当てる。各 SM ではブロック内のスレッドを 32 個単位のワープに分割し、32 個のストリーミング・プロセッサ（SP）を用いて並列実行する。すなわち、複数の SM によるブロックレベルの並列処理と、32 個の SP によるマルチスレッドの並列処理を併用して、高速化を実現している。

スクリーンマッピングとはあらかじめ物体のある場所をスクリーン上に予測することにより交差判定そのものを削減する手法である。物体があると思われる画素へと飛ばした光線は従来通りの演算を行い、物体が無いと思われる画素へと飛ばした光線は交差判定を行わず直接背景色とすることにより高速化を図る。

実験には、SPD（Standard Procedural Databases）のシーンデータから teapot、tetra、mount、nurbs を用い、スクリーンマッピングの有無とブロック数とスレッド数を変化させ、計 12 通りの方法で比較を行った。その結果、tetra ではスクリーンマッピング適用により、CPU 逐次の場合、実行時間が 25.7 秒で 2.52 倍、GPU 並列ではブロック数 64*64、スレッド数 8*8 の場合、実行時間が 0.88 秒で 75.3 倍の速度向上を達成することができた。

目次

図目次

表目次

1. はじめに

現在、コンピューターの性能は凄まじい速さで向上しており、現実と見まごうほどの写実的なコンピューターグラフィックスの生成も可能となっている。そのようなコンピューターグラフィックスの描画技術の一つとしてレイトレーシング手法がある。レイトレーシング法は、光の挙動を忠実に計算することにより、反射や屈折などの光学の表現を簡単に実現することができるので、非常にリアルな画像を生成することが可能である。近年ではフォトンマッピング、モンテカルロレイトレーシング手法などの新しい手法も開発されており、より高品質な画像を生成することが可能となっている。しかし、レイトレーシング法は 1 光線、1 画素ずつ光の挙動を正確に計算する必要が有るため、計算量が非常に膨大になるという欠点がある。

レイトレーシングのアルゴリズムは、視点からスクリーン上の各画素を通過する光線が物体と交差する点を探索し、その交点で反射、屈折など物体の表面の質感に応じて光線がどの方向に、どのように変化するのかを計算する。この変化した光線が、物体と交差する点を改めて探索し直す。この処理を順次行い、最終的に探索した点の輝度をスクリーン上に投影することで画像を生成する。光線と物体との交点を探索する処理を交差判定とよび、レイトレーシングの処理の中で特に計算量が多い部分である。交差判定は物体の数だけ行われ、特に複雑な立体においてはその演算量は膨大となる。さらに、物体が反射、屈折の質感を持つのであればさらに交差判定の回数が増える。

本研究では、この問題を改善するために GPU（Graphics Processing Unit）による画面分割を用いたレイトレーシング法において、スクリーンマッピングにより高速化を図る。GPU とは PC に搭載されている画像・動画処理用のハードウェアである。汎用的な計算を行うことを前提としている CPU と比べ、比較的単純な計算を同時並列的に処理することに特化している。GPU へのプログラミングは CUDA を用いて行う。CUDA は NVIDIA が提供する GPU 向けの総合開発環境である。

画面分割法とは、画面が生成されるスクリーンをブロック分割する。分割された各ブロックをストリーミング・マルチプロセッサ（SM）に割り当てる。各 SM）ではブロック内のスレッドを 32 個単位のワープに分割し、ストリーミング・プロセッサ（SP）を用いてワープ単位で並列実行する。スクリーンをブロック分割することで、各スレッドで並列処理を行い、レイトレーシングを高速で行う。

スクリーンマッピングとは、あらかじめ物体のおおまかな位置をスクリーン上に投影させておくことにより交差判定回数の削減を目指す手法である。画面中の全ての画素について、全ての物体との交差判定を行う必要があるのが通常のレイトレーシング法であるが、前述のとおりこれは非常に演算量が多く時間がかかるという欠点がある。

画素数はすなわち画面の大きさであり、交差判定は多くのポリゴンで構成される複雑な物体になるほど多く計算する必要がある。演算の大部分を占める交差判定をいかに減らすかがレイトレーシング法の高速化に肝要であるが、光線の先にどの物体が最も手前に来る

かは全ての物体との交差判定を行った上で最も近い物体を導出してやる他にない。そこで、少なくとも一つ以上物体がある範囲と、全く物体がなく交差判定を一度もする必要がない範囲とをあらかじめ調べておき、必要な範囲でのみ交差判定を行えば、物体を漏らすこと無く交差判定回数の削減ができる。

スクリーンマッピングのアルゴリズムは、まず視点の座標と物体の座標から方向ベクトルを求める。次に、スクリーン上の各画素に向けた全ての方向ベクトルと比べ、最も近いものとその近傍を”その先に物体があると予測される画素点”としてマッピングする。これを全ての物体に対して繰り返すことにより、スクリーン上の画素点全てについて物体の存在予測が求められる。レイトレーシングを行う際にこの予測を参照するようにすれば、物体が先に存在しない画素、マッピングされなかった画素へと飛ばした光線の交差判定は一切行う必要がなくなり高速化が図れる。一方、予測によって物体が先に存在する可能性があるとされた画素、マッピングされた画素へと飛ばした光線の場合には通常通りの交差判定を行えばよい。

本論文では 2 章でレイトレーシングのアルゴリズム、光線の種類、交差判定、及び本研究で用いた GPU の仕様について述べる。3 章では GPU による画面分割法、及びスクリーンマッピングのアルゴリズムについて述べる。4 章では CPU-GPU 間での動作とその実験結果、考察について述べる。5 章では本研究の成果と今後の課題を述べる。

2. GPU によるレイトレーシング法

2. 1　レイトレーシングのアルゴリズム

我々が日常目にしている物体の多くは自らが発光しているのではなく他の光源から照らされている。光源から出た光は様々な物体に衝突し、反射や屈折を繰り返す。光線は物体に衝突するごとにその物体の輝度に影響され、その光線が目に入ることにより人は物体を認識できることができる。すなわち、光源から出る光線を１つずつ調べていき、目に入る光線を全て見つけ出せば、我々が見ている像を再現することが出来るということである。しかし、光源から出る光線は膨大であり、また目に入ってくる光線数はそのごく一部に過ぎない。ほとんどの光線は物体に吸収されたりあらぬ方向へと飛んで行ってしまい、全ての光線について考慮していては非常に効率が悪くなってしまう。

そこで、逆に目に入ってくるであろう光線のみを逆に辿っていき像を生成するというのがレイトレーシング手法である。レイトレーシング法の原理を図１に示す。

レイトレーシング法は光線追跡法とも呼ばれ、視点の先にスクリーンを仮定し、そのスクリーン上へと光線を飛ばす。次に光線と全ての物体について交点を求め、交点があるならば最も視点に近い物体を抽出、交点がないならば背景色とする。抽出物体がある場合には物体表面による輝度の計算を行うと共に反射、屈折が起きるならば新しい光線とみなし再度物体の抽出を行う。これを繰り返しスクリーン上の画素において輝度値を置くことにより画像の生成を行う。

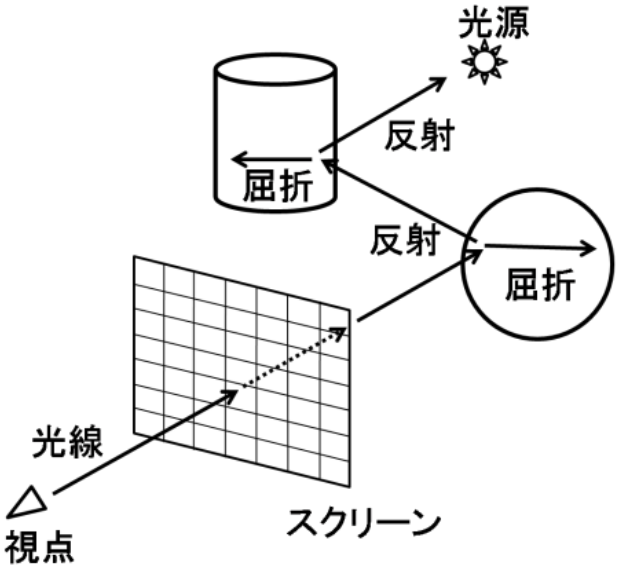


図 1　レイトレーシングの原理

交差判定のためにまず視点とスクリーンを結ぶ光線を求める。その手法を図 2 に示す。
直線の定義として視点を e、スクリーン上のあるがその点を q とする。視点 e から q までの方向ベクトルを d とすると、

$$d = q - e$$

このとき、視点 e からスクリーンを通り抜けるベクトル p(t)は、

$$P(t) = e + td$$

とあらわされる。t は視点から直線上のある点までの距離を示す。d は方向ベクトルなので正規化を行い、大きさを 1 としておく。

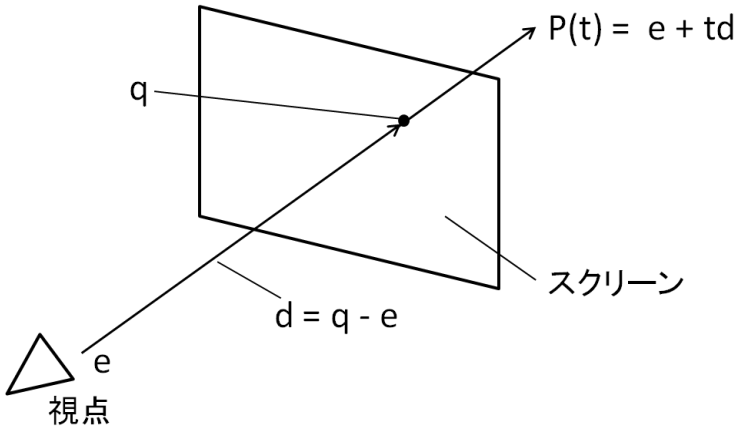


図 2　視点とスクリーン、物体の関係

次に、物体との交差判定の計算式を示す。ここでは三角形と球を示す。

図 3 に示すように、球の中心点を c とすると、視点 e から中心点 c までのベクトル v は、

$$v = c - e$$

t をある変数として、p(t) = e + td が指す点を光線と球の交点と仮定すると、球の中心点 c から交点までのベクトル l は、

$$l = td - v$$

となる。このベクトルの内積は、球の半径 r の 2 乗と等しいので、

$$l \cdot l = r^2$$

この式を変形すると、

$$t^2(d \cdot d) - 2t(d \cdot v) + (v \cdot v) = r^2$$

$$t^2 - 2t(d \cdot v) + (v \cdot v) - r^2 = 0$$

となる。この t についての方程式が実数解をもてば、光線と球は交差しているということになる。この方程式が実数解をもつ条件は、

$$D = (v \cdot d)^2 - (v \cdot v) + r^2 \geqq 0$$

となればよい。このとき t は

$$t = -(d \cdot v) \pm \sqrt{D}$$

と表せる。

このとき $t < 0$ ならその点は視点よりも後ろ側に存在することになり、交差点は見えないことになる。

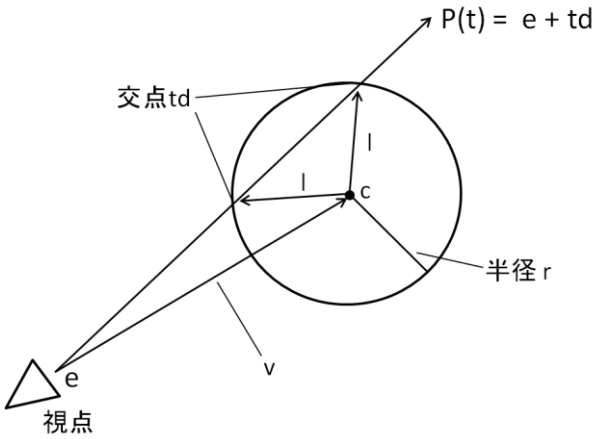


図 3　球の交差判定

次に光線と三角形の関係を図 4 に示す。三角形の頂点をそれぞれ A,B,C とする。このとき、三角形上のある点 T(x,y)は重心座標を用いて表すことができる。

$$T(x,y) = (1-x-y)A + xB + yC$$

このとき、T(x,y)は三角形内部に存在するので、

$$0 < x,\quad 0 < y,\quad x + y < 1$$

を満たす必要がある。

視線からの光線 p(t) = e + td と平面が交差する方程式は

$$e + td = (1-x-y)A + xB + yC$$

となる。この式を変形すると、

$$(-d \quad B-A \quad C-A)\begin{pmatrix} t \\ x \\ y \end{pmatrix} = e-A$$

ここで $O_1 = B-A$, $O_2 = C-A$, $T = e-A$ とすると、クラメールの公式より

$$\begin{pmatrix} t \\ x \\ y \end{pmatrix} = \frac{1}{|-d \quad O1 \quad O2|}\begin{pmatrix} |T \quad O1 \quad O2| \\ |-d \quad T \quad O2| \\ |-d \quad O1 \quad T| \end{pmatrix}$$

ここで 3 次元のベクトル L, M, N を考えると、

$$|L \quad M \quad N| = -(L \times N) \cdot M = -(N \times M) \cdot N$$

が成立するので、

$$\begin{pmatrix} t \\ x \\ y \end{pmatrix} = \frac{1}{(d \times O2) \cdot O1}\begin{pmatrix} (T \times O1) \cdot O2 \\ (d \times O2) \cdot T \\ (T \times O1) \cdot d \end{pmatrix}$$

となる

ここで x, y が $0 < x,\quad 0 < y,\quad x + y < 1$ の条件を満たす場合、光線と三角形は交差す

る。

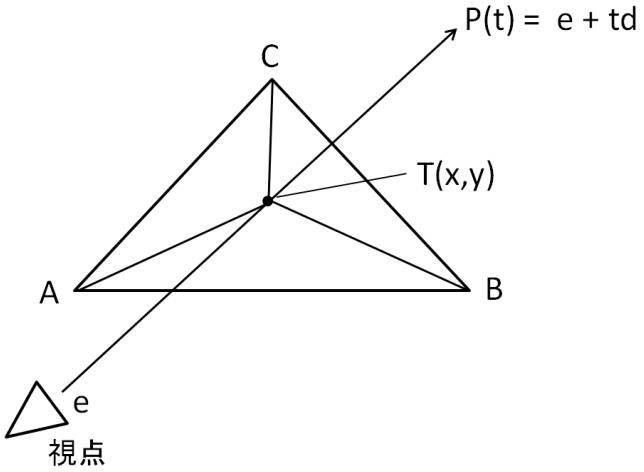


図 4　三角形の交差判定

2．2　GPU アーキテクチャ

GPU（Graphics Processing Unit）とは画像処理・動画処理を専門として搭載されている半導体チップである。GPU は本来、グラフィックス処理のみを行うものではあったが、近年では性能が向上し、汎用的な計算を並列処理することができる超並列プロセッサとなっている。

本研究で使用する GPU は、NVIDIA 社の fermi アーキテクチャの Geforce580 である。図 5 に GeforceGTX580 の構成を示す。動作ハードウェアの最小単位はストリーミングプロセッサ（SP：Streaming Processor）であり、これが 32 個集まってストリーミングマルチプロセッサ（SM：Streaming MultiProcessor）を構成している。SM は 4 個集まって GPC（Graphics Processing Cluster）を構成しており、GPU は GPC4 つから構成されている。すなわち、全体では SM が 16 個、SP は 512 個搭載しており、これらが同時並列で演算を行うことが出来る。

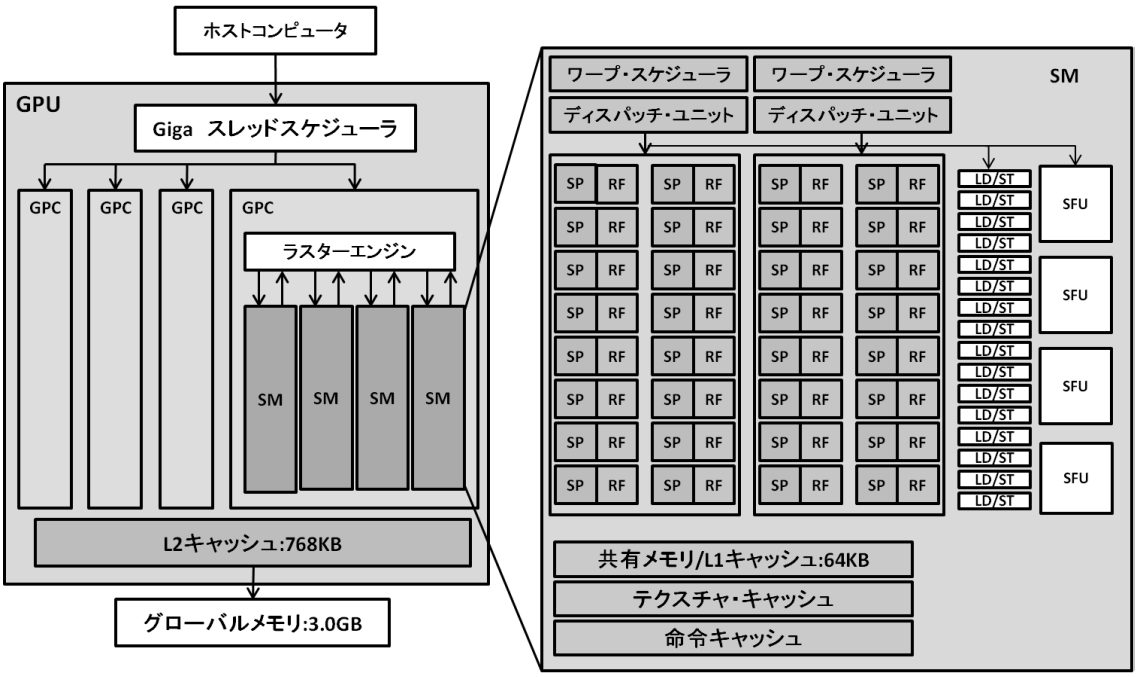


図 5　GTX580 ハードウェア構成

2．3　並列プログラミング環境

CUDA（Compute Undefined Device Architecture）とは、NVIDIA 社が自社のグラフィックカード上で、汎用コンピューティングを実現するために提供している総合開発環境である。CUDA でのプログラミングには C、及び C++がサポートされている。

CUDA はコンピュータのマザーボードに接続されたビデオカード上で動作を行う。このとき、マザーボードには CPU があり、メモリも装着されている。また、ビデオカードには GPU が搭載されており、VRAM（Video RAM）も搭載されている。マザーボード、及び CPU 側を「ホスト」、GPU、及び VRAM 側を「デバイス」と呼ぶ。

ホストプログラムと呼ばれるホスト側のプログラムは、ホスト上の CPU で動作し、ホスト上のメモリを利用する。一方、デバイス側で実行されるプログラムはカーネルプログラムと呼ばれ、デバイス上で動作し、VRAM のメモリを利用する。そのプログラムの一連の流れを図 6 に示す。

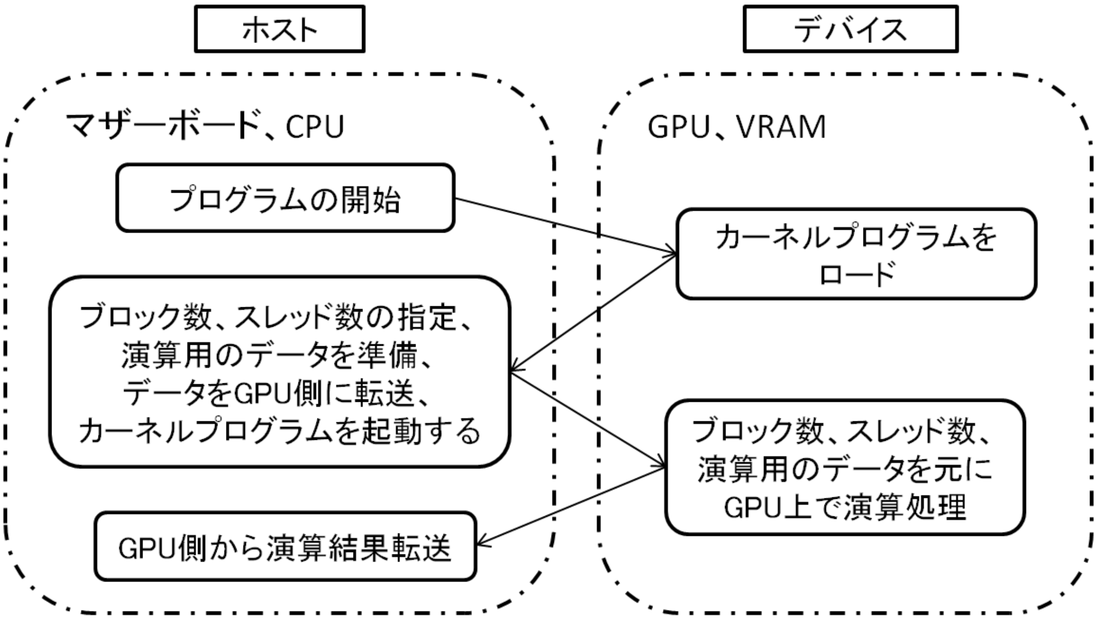


図 6　ホストとデバイスでの処理の流れ

CUDA には並列処理の実行単位として「グリッド」、「ブロック」、「スレッド」がある。これら 3 つは、図 7 に示すように階層構造の関係にあり、上からグリッド、ブロック、スレッドとなっている。グリッドとは、実行される 1 つのプログラムのことを指し、ブロックにより構成されている。ブロックとはプログラムを分割する並列処理の単位である。ブロックのサイズを指定することにより、グリッドを 1 次元や 2 次元に分割することができる。このブロックはスレッドにより構成されている。スレッドとはカーネルで動作するプログラムの最小単位である。各ブロックは各 SM 上で実行され、SM は 32 個の SP があるので、通常 32 スレッド単位で実行され、1 ワープ単位と呼ぶ。ハードウェア上にワープ・スケジューラーとディスパッチ・ユニットがそれぞれ 2 つずつあるため、2 クロックで 2 ワープを実行することが出来る。ブロック内の 32 個のスレッドを同時に実行することで、マルチスレッド並列処理が可能となる。

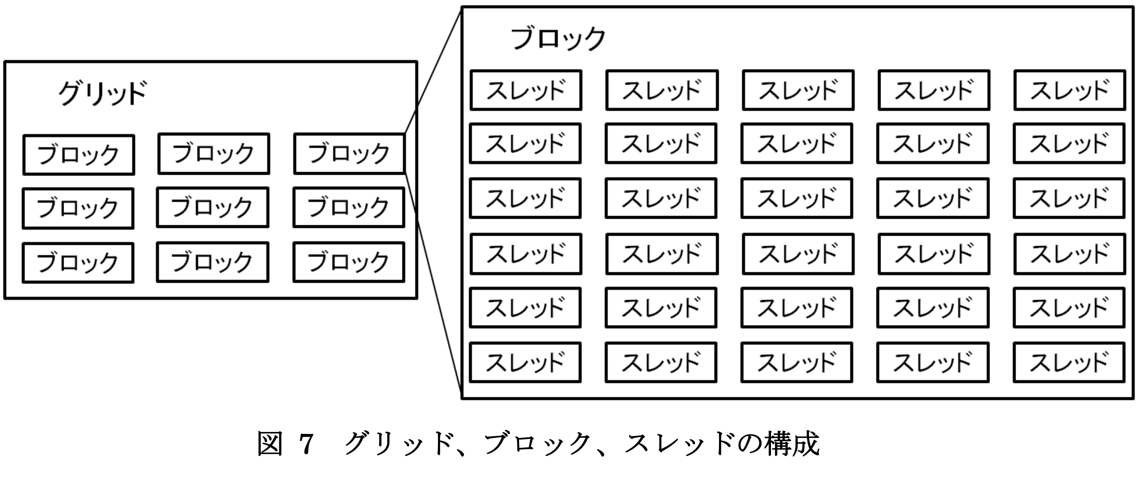


図 7　グリッド、ブロック、スレッドの構成

3. 画面分割とスクリーンマッピングによるレイトレーシング法の高速化

3. 1　画面分割によるレイトレーシング法の高速化

図 8 に GPU を用いた画面分割による並列高速化の処理系を示す。本プログラムはホスト側：CPU とデバイス側：GPU の二つの処理で成り立っている。

まず、ホスト側でレイトレーシングで用いるシーンデータである SPD ファイルを読み込む。通常、シーン情報はリスト構造で管理されているが、このままではデバイス側に正しくデータを渡すことが出来ない。これは、ホスト側とデバイス側のメモリが分離しているためである。そこで、SPD 情報をリスト構造から配列構造に変換して管理する。

デバイス側では受け取ったブロック、スレッド情報をもとに、スクリーンを SM 数分のブロックに分割する。次に、各ブロックをワープ単位の 32 スレッドに分割する。このとき、1 スレッドはスクリーンの画素 1 つの処理を行う。

各スレッドでは 1 画素分のレイトレーシング結果を得るために、光線の方向探索、交差判定、反射、及び輝度計算を行い、スクリーンにレイトレーシング結果を書き込む。

ホスト側ではスクリーン情報を受信する。受信した結果を ppm 形式のファイルに変換することで画像を得る。

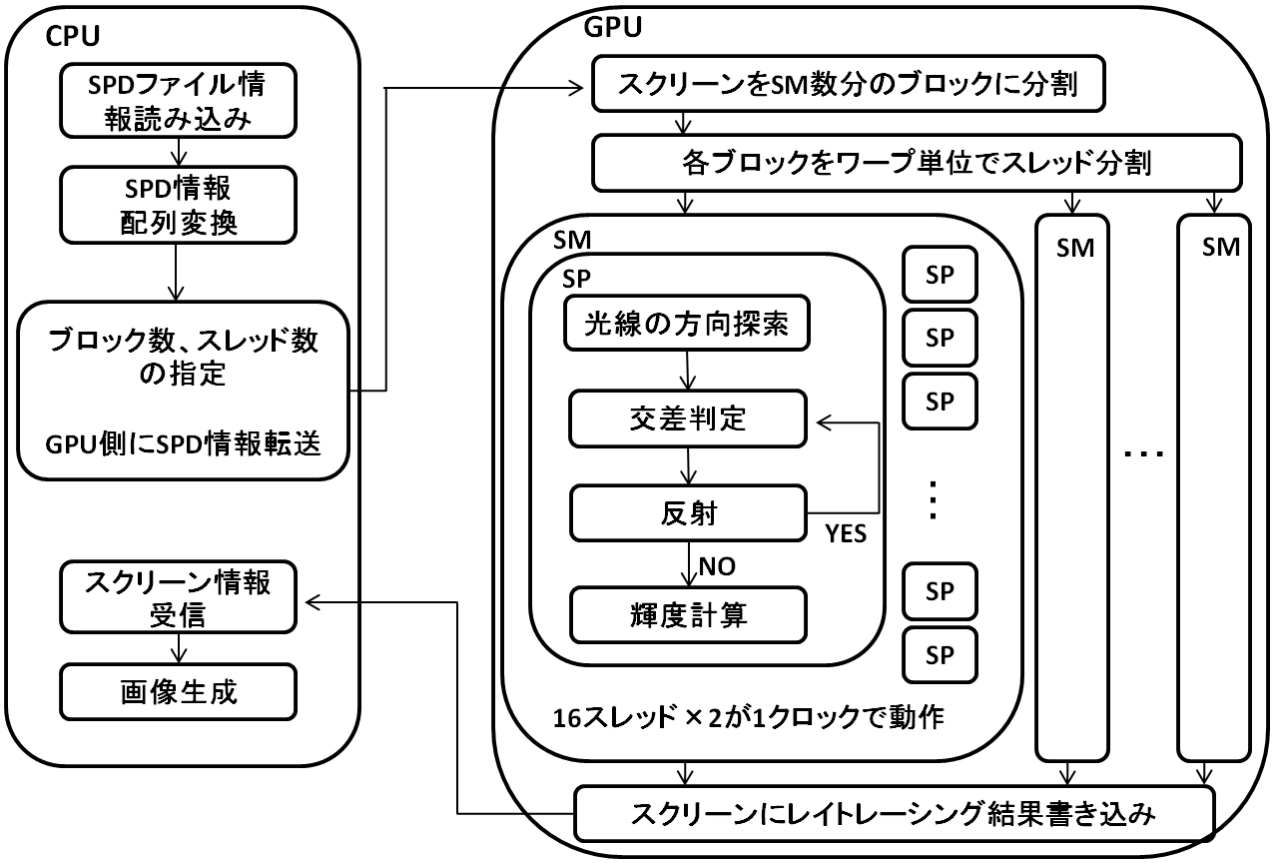


図 8　画面分割並列化の処理

3．2　スクリーンマッピング法とは

レイトレーシング法は非常に演算量が多く処理に時間がかかるアルゴリズムであるが、その演算の多くは光線と物体の交差判定が占めている。一つの光線について存在しうる全ての物体と交差判定を行うため、複雑な物体になるほどその演算量は膨大になってしまう。レイトレーシング法の高速化として有効なのはこの交差判定部の処理を如何に軽くするかが肝要である。

高速化手法の一つであるスクリーンマッピングとは、視点からの光線数の削減を目的とした手法である。図 9 にスクリーンマッピングのアルゴリズムを示す。あらかじめスクリーンに、視線の先に物体があるかどうかの情報を記述しておき、その先に物体がないのならば光線と物体との交差判定を一切すること無く背景色を描画するというものである。一つの光線を減らすごとに物体数*交差判定の分だけ演算量を減らすことができるため、特に緻密なポリゴンで描かれている物体や、スクリーンに占める物体像の範囲が狭いほど大きな効果がある。一方、ほぼ画面全体に物体が描画されるような場合には削減できる光線数が限られるため、その効率は悪くなる。

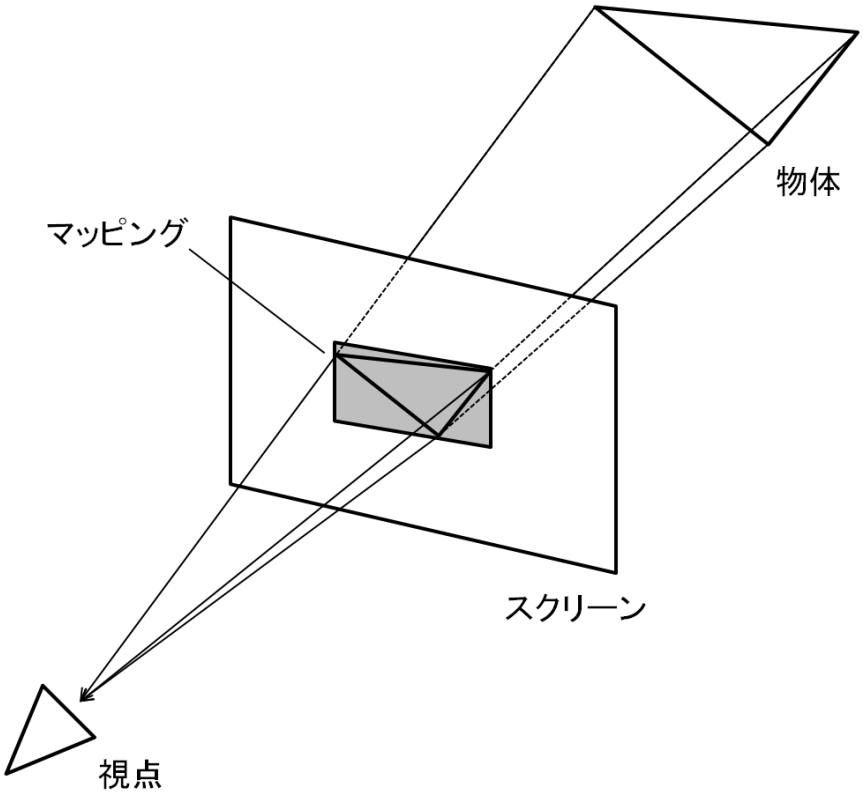


図 9　スクリーンマッピングのアルゴリズム

3．3　スクリーンマッピングのアルゴリズム

スクリーンマッピング法は光線数を削減する手法である。物体の頂点座標、視点座標からスクリーン上の座標を求め、その点を通る視線以外は交差判定を省略するというものである。すなわち、レイトレーシング法においては通常、視点からスクリーン、物体という順序で演算を行うが、その前に視線のふるい分けを行うということである。

以下にスクリーンマッピングを用いたレイトレーシング法の手順を示す。

（A）　視点と物体を結ぶ直線を、視点からスクリーン上への方向ベクトルへ変換する。
（B）　方向ベクトルからスクリーン上の交点を求め、マッピングする。
（C）　全ての物体について（A)、（B）を繰り返す。
（1）　視点からスクリーン上の画素を通過する光線を発生させる。
（D）　マッピングの結果を参照し、物体が存在するならば光線と物体との交差判定を行う。存在しない場合にはその画素を背景の輝度とする。
（2）　交差判定により、光線と交差する物体が存在するなら、光線と物体との交点を求める。交差する物体が複数ある場合には、すべての物体について交点を求める。交点がない場合は、その画素を背景の輝度とする。
（3）　交差する物体が複数ある場合には、光線と交差する物体の交点までの距離を求め、最も近い物体を抽出する。
（4）　抽出物体の輝度の計算をする。また、反射、屈折があるならその方向を求め、その方向を新たな光線とみなして（2)、（3）の処理を行い、屈折、反射してみえる物体を抽出する。

（1）～（4）はレイトレーシングの手順である。（A）～（D）はスクリーンマッピングの手順である。交差判定が省略できる画素の場合、（2）以下が削減され高速化することが出来る。レイトレーシング法全体において最も計算量が多い交差判定は（2）で行われるため、スクリーンマッピングによって増えた演算量よりも全体の演算量は少なくできる見込みが大きい。

具体的に三角形ポリゴンをスクリーンマッピングによって予測、マッピング、光線を削減する場合について図 10 以下で述べる。

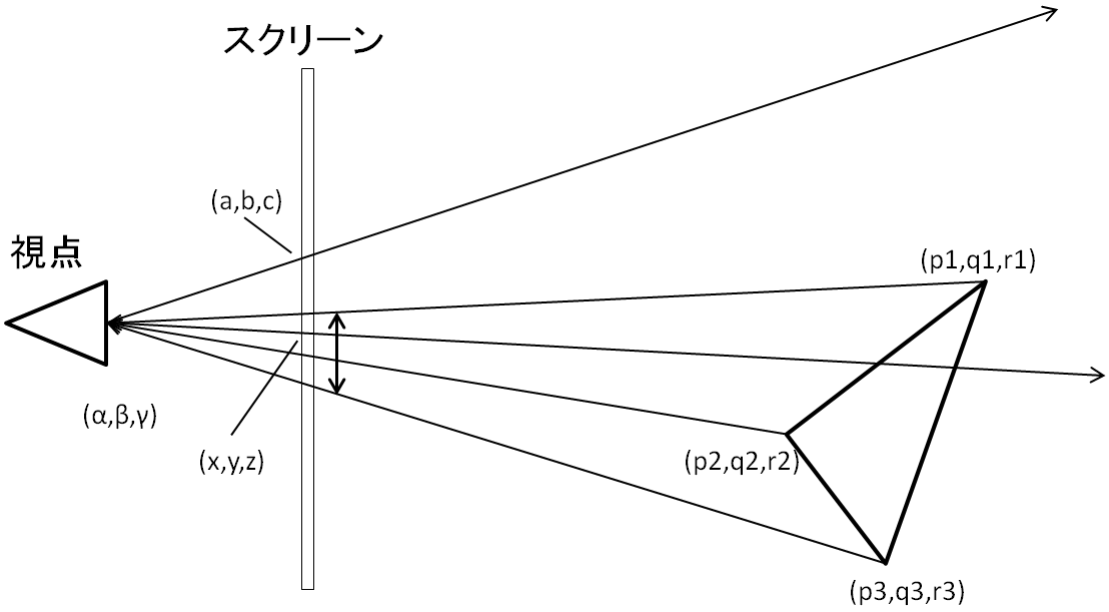


図 10　スクリーンマッピングを用いた交差判定の削除

通常のレイトレーシング法においてはスクリーンを通る全ての視線について交差判定を行う。視線のベクトルは視点の座標（α, β, γ）とスクリーン上の座標（a, b, c）、（x, y, z）で求められる。スクリーン上の全座標に対して順次ベクトルを算出して視線とし、これら全ての視線は全ての物体との交差判定を行う。交差判定は一定距離進むまで行い、衝突しなかった場合には物体無しと判断し背景色をスクリーンに反映する。

一方、スクリーンマッピング法では先に視点の座標（α, β, γ）と物体の頂点座標（p1, q1, r1）～（p3, q3, r3）を用いて、スクリーン上に物体が投影される範囲を予測する。図中矢印で囲まれた範囲が予測された部分である。（x, y, z）はこの範囲内であるため、物体との交差判定が必要な視線であると判断し、通常通りの交差判定が必要な視線としてレイトレーシングを行う。一方、（a, b, c）はこの範囲外であるため、視線上の交差判定を行う必要がない。そのため、交差判定を削減できる。この場合、スクリーン上には背景色が適用される。このようにして通常のレイトレーシング法では行なっていた交差判定を行う光線を減らすことにより高速化を実現する。

4. スクリーンマッピングの実装

4. 1　CPU 上での事前処理

スクリーンマッピングを行うためにはあらかじめスクリーン上に物体のおおまかな位置を記述しておく必要が有る。

スクリーン上の画素のパラメーターに色情報：RGB 以外に物体の存在判定を埋め込む（スクリーンマッピング)。視点から光線を伸ばす前に、この予測情報を演算しておく。

スクリーン上への光線ベクトルは、視点からの方向ベクトル＋スクリーン座標情報によって求められている。スクリーン上の左上を開始点とし、スクリーンの大きさを最大値とするパラメーターによってスクリーン上の画素の座標情報は示される。

視点と物体を結び、方向ベクトルを算出し、スクリーン上へのベクトルへと変換することによって、その光線がスクリーン上のどの点を通るのか、その座標パラメーターはどうなのかを導出し、マッピングする。

4. 2　GPU 上での処理演算

CPU から送られてきた視点座標、スクリーン座標、スクリーンマッピング結果、及び物体座標を用いて交差判定を行う。視点から光線を飛ばすためにスクリーン座標を参照する際にマッピング結果を参照し、マッピングされていれば通常通りの演算を行い輝度計算を、マッピングがなければ交差判定を行わず背景色を適用する。後者の場合に速度向上が見込める。

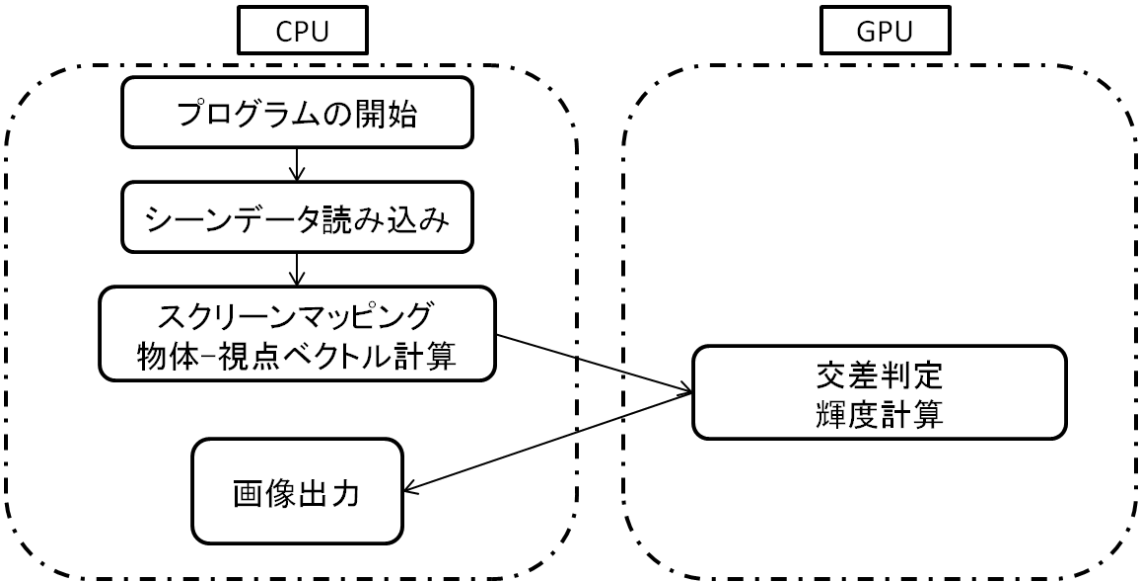


図 11　CPU と GPU における処理分担

4．3　実験結果と考察

4 種類のシーンデータ（teapot, tetra, mount, nurbs）に対して、レイトレーシング法について GPU を用いた画面分割並列化及びスクリーンマッピングを行った。各シーンデータに対して、ブロック数とスレッド数を変化させて画面分割を行い、最適な分割パターンを検討した。実験は以下に示す 5 通りのブロック数とスレッド数で行った。

・ブロック数 256*256、スレッド数 2*2
・ブロック数 128*128、スレッド数 4*4
・ブロック数 64*64、スレッド数 8*8
・ブロック数 32*32、スレッド数 16*16
・ブロック数 16*16、スレッド数 32*32

例えば、ブロック数 64*64、スレッド数 8*8 の場合には、スクリーンを 64*64 ブロックに分割し、各ブロックは 8*8 の画素で構成されている。このとき 1 ブロック内には 64 スレッドがあるため、ワープが 2 つ（32*2）存在することになる。1 つのブロックは 1 つの SM で処理され、1 つのスレッドは 1 つの SP で処理される。

CPU 逐次処理の実験環境は

OS：windows7 Ultimate
プロセッサ：Intel(R) Core(TM) i7-2600K @3.4GHz
実装メモリ：8.00GB

GPU を用いた画面分割の並列化の実験環境は

OS：windows7 Ultimate
プロセッサ：Intel(R) Core(TM) i7-2600K @3.4GHz
実装メモリ：8.00GB
GPU：Geforce GTX580
グラフィッククロック：772MHz
メモリクロック：2004MHz
メモリ容量：3.0GB
並列処理環境：CUDA

実験で用いた SPD は表 1 のような構成である。

表 1　SPD 情報

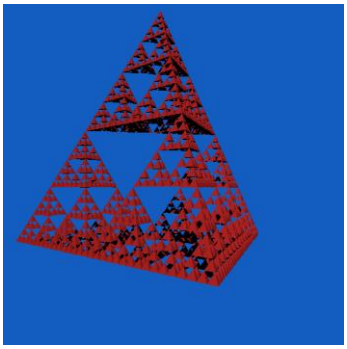
	teapot	tetra	mount	nurbs
ポリゴン数	2328	4096	8192	1500
スクリーンサイズ	512*512	512*512	512*512	512*512

以下に 4 通りの実験結果を示す。

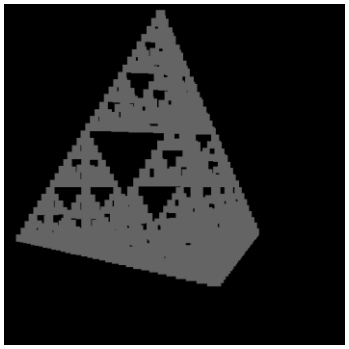
(a) 生成画像

(b) マッピング画像

図 12　teapot

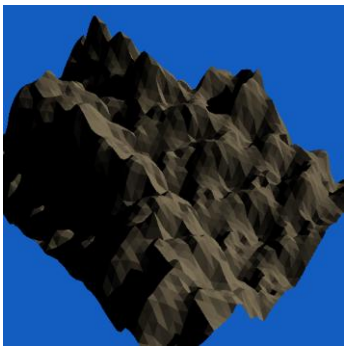


(a) 生成画像



(b) マッピング画像

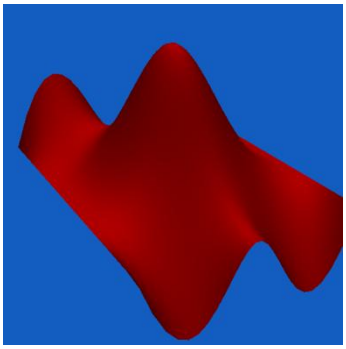
図 13　tetra



(a) 生成画像

(b) マッピング画像

図 14　mount



(a) 生成画像

(b) マッピング画像

図 15　nurbs

それぞれ、実行時間は以下の表 2、表 3 のようになった。

表 2　CPU 逐次処理の実行時間

	CPU 逐次	CPU＋マッピング				
	実行時間(秒)	実行時間(秒)	速度向上比(倍)	全体画素数	削減画素数	全体比(%)
teapot	126.1	123.64	1.02	262144	87061	33.2
tetra	66.3	25.72	2.58	262144	196681	75.0
mount	195.2	123.48	1.58	262144	91341	34.8
nurbs	30.9	5.56	5.56	262144	143284	54.7

表 3　GPU 並列処理の実行時間　　単位：秒

ブロック数 スレッド数	teapot				tetra			
	GPU		GPU+マッピング		GPU		GPU+マッピング	
	実行時間	速度向上比	実行時間	速度向上比	実行時間	速度向上比	実行時間	速度向上比
256*256 : 2*2	13.60	9.27	14.91	8.46	7.34	9.03	8.02	8.27
128*128 : 4*4	3.65	34.55	3.72	33.90	1.90	34.89	2.04	32.50
64*64 : 8*8	1.52	82.96	1.59	79.31	0.76	87.24	0.88	75.34
32*32 : 16*16	1.82	69.29	1.90	66.37	0.86	77.09	0.93	71.29
16*16 : 32*32	2.33	54.12	2.42	52.11	1.04	63.75	1.17	56.67
ブロック数 スレッド数	mount				nurbs			
	GPU		GPU+マッピング		GPU		GPU+マッピング	
	実行時間	速度向上比	実行時間	速度向上比	実行時間	速度向上比	実行時間	速度向上比
256*256 : 2*2	16.20	12.05	17.49	11.16	3.27	9.45	3.83	8.07
128*128 : 4*4	4.06	48.08	4.21	46.37	0.83	37.23	0.97	31.86
64*64 : 8*8	1.79	109.05	1.86	104.95	0.32	96.56	0.39	79.23
32*32 : 16*16	1.81	107.8	1.90	102.74	0.34	90.88	0.44	70.23
16*16 : 32*32	1.93	101.14	2.04	95.69	0.39	79.23	0.48	64.38

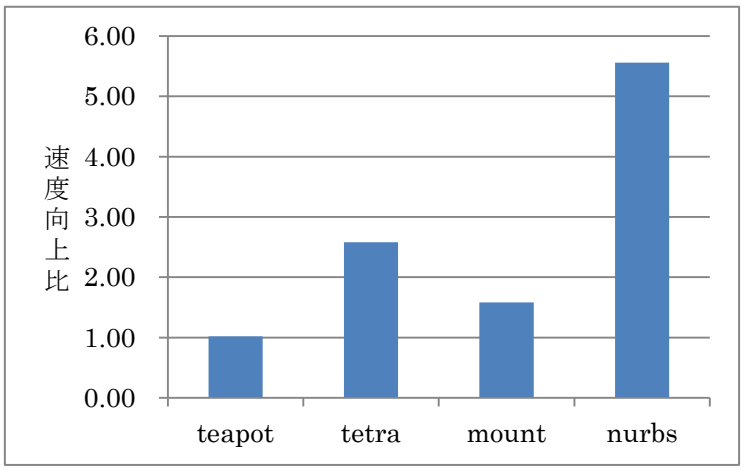


図 16　CPU 逐次速度向上比

図 16 は CPU 逐次処理にスクリーンマッピングを適用した結果の速度向上比である。特に tetra、nurbs で高速化の効果が見られた。ともに表 1 での削減画素全体比が 75.0、54.7（%）とスクリーン全体の半分以上について交差判定演算を削減できたためである。tetra では画素削減率に対して速度向上がほぼ理想的に得られた。一方、nurbs では 5 倍以上の速度向上が得られたが、これは tetra に比べ、物体の影が少ないことより衝突してから再生成された光線数が少ないからだと考えられる。

一方、mount や teapot については高速化の効果が鈍い。mount は画素削減率と速度向上比がほぼ反比例であるため、スクリーンマッピングによって削減された画素数が少ないためである。画面に占める物体量が多いほどスクリーンマッピングによる高速化は効率が悪くなる。teapot はほぼ速度向上が見込めなかったが、これは一つの光線について再生成された光線数が多いからであると思われる。物体同士や床との映り込みはスクリーンマッピングによる高速化の恩恵を受けないだけでなく、マッピングによる演算量の増加によって通常のレイトレーシングよりも遅くなってしまうからである。

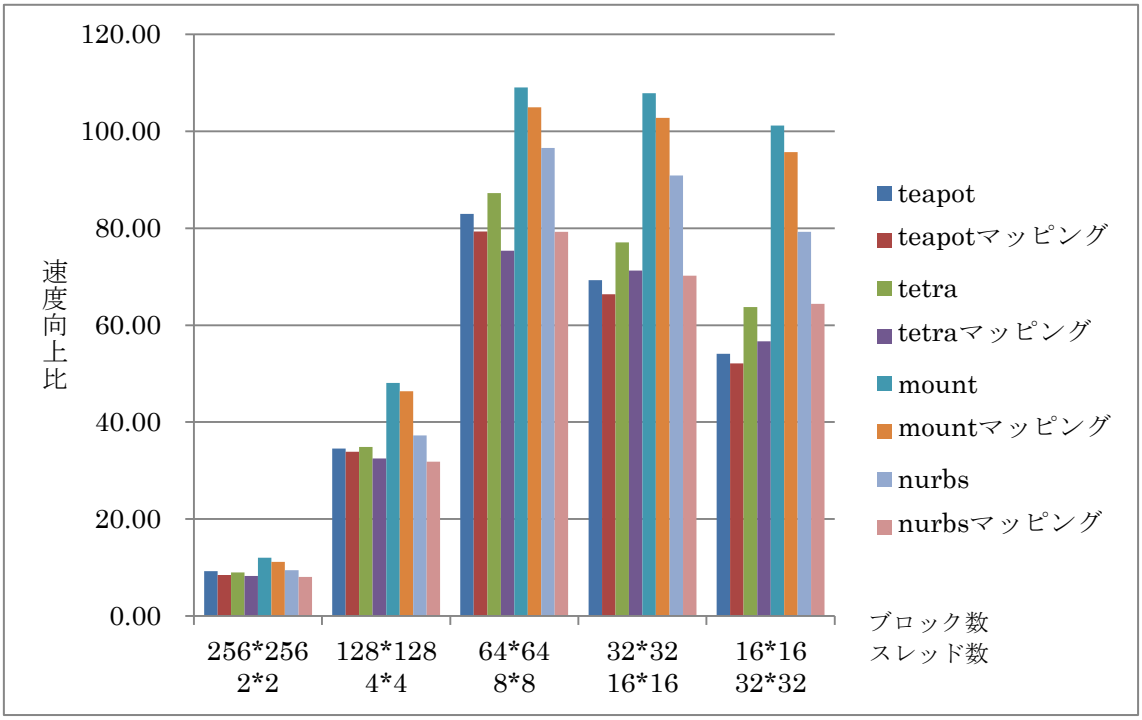


図 17　GPU 並列速度向上比

図 17 は GPU での画面並列化にスクリーンマッピングを適用した結果の速度向上費である。グラフからどのシーンも共通して、ブロック数 64*64、32*32、16*16 の場合に速度向上比が大きく、特にブロック数 64*64、スレッド数 8*8 の場合が最も高速であると確認できる。一方、ブロック数 256*256、128*128 の場合は速度向上があまり見られない。

ブロック数が 64*64、32*32、16*16 の場合、1 ブロック内のスレッド数がワープ数 32 の倍数となるため、各 SP が常に動作することが出来る。そのため、速度向上比が大きくなっている。また、ブロック数が小さくなるほど速度向上比が小さくなっているが、これはブロックあたりのスレッド数が増えたことによりスレッドあたりの使用可能レジスタ数が減ったためである。ブロック内のスレッド数とレジスタ数はトレードオフ関係にあり、最もバランスが取れているのがブロック数 64*64 であると考えられる。ブロック数 256*256、128*128 の場合、1 ブロック内のスレッド数がワープ数よりも小さいため、各 SP に均等に処理を割り振ることが出来ず、空きのある SP を作ってしまっている。そのため、速度向上が鈍い。

GPU による画面分割処理にスクリーンマッピングを適応した結果全ての結果について適用しなかった場合よりも速度向上比は低くなっている。これは SP 一つ一つがスクリーン上の 1 画素を担っているが、マッピングによってある一部の画素の処理が速くなったとしてもブロック単位では全ての SP の動作が終了するまでは次のワープ処理が行えないからだと思われる。また、交差判定をする SP に着目すると、マッピングの可否判定により従来の

レイトレーシングよりも処理が多くなり時間がかかる。そのため、スクリーンマッピングによって画素へと飛ばした一部の光線の演算量が減ったとしても、並列単位ではその恩恵をほとんど受けられない。

マッピングによって並列処理環境下でも確実な速度向上が得られる状況とは、ブロックに割り当てられた画素全てがマッピングされていない、すなわち物体が存在しないと予想される場合のみである。解決策としては、ブロック内のマッピング状況によって SP への割り当てを意図的にふるい分ける手法が考えられる。また、ほとんどの画素がマッピングされている場合には、あえてマッピング結果を参照せずに交差判定を行ったほうが処理を増やさずに済む可能性もある。これらは今後の研究として実現、実現していきたい

5. おわりに

本論文では、GPU を用いた画面分割の並列処理及びスクリーンマッピングによる高速化について述べた。画面分割の並列化では、レイトレーシングの 1 画素の輝度値を求める処理が他の輝度値を求める処理と依存関係にない特徴を利用し、ストリーミング・マルチプロセッサによる画面分割を行った。一方、スクリーンマッピングでは、非常に計算量の多い交差判定を効率的に削減するため、視点と物体の位置関係から視点からの物体位置を大まかに予測した。実験では Geforce GTX580 を用い、4 つのシーンデータ（teapot, tetra, mount, nurbs）に対して、スクリーンマッピングによって交差判定を減らした上で、ブロック数とスレッド数を変化させて実行時間を測定した。

画面中の全ての点に対して光線を飛ばすというレイトレーシング法に対して、その光線数を減らすというスクリーンマッピング、視点から物体へと遡っていく従来手法に対して、物体から視点へと演算してスクリーン上の座標位置を算出するマッピングが実現でき、またその効果を実証できた。ただし速度向上率は SPD によって大きく左右されてしまったため、改善の余地がある。また、GPU による画面分割並列化とスクリーンマッピングとの併用による更なる速度向上は現時点では実現できなかったばかりか遅くなってしまった。GPU 並列による速度向上は大きいためスクリーンマッピングの応用を継続して考えたい。

今後の課題として、安定的な速度向上を目指すと共に GPU 並列でのスクリーンマッピングによる高速化、レイトレーシング法以外の画像生成手法へのスクリーンマッピング適用である。

謝辞

本研究に機会を与えてくださり、ご指導を頂きました山崎勝弘教授に心より深く感謝いたします。さらに、本研究に助言をして頂いた山崎研究室の皆さまに感謝いたします。

最後に、本研究において格別のご協力を承りました孟助手、岡崎氏、藤川氏、増田氏に改めて感謝の意を表します。

参考文献

[1] 小山田耕二, 岡田賢治：CUDA 高速 GPU プログラミング入門, 秀和システム, 2010.
[2] 千葉則茂, 土井章男： 3 次元 CG の基礎と応用[新訂版], サイエンス社, 2004.
[3] Eric Haines et al：Standard Procedural Databases
http://tog.acm.org/resources/SPD/
[4] Patrickomatic.com：http://patrickomatic.com/c-ray-tracer
[5] ホワイトペーパー　NVIDIA の次世代 CUDATM アーキテクチャ：
http://www.nvidia.co.jp/docs/IO/81860/NVIDIA_Fermi_Architecture_Whitepaper_FINAL_J.pdf
[6] Jason Sanders, Edward Kandrot 著, 株式会社クイープ訳：CUDA BY EXANPLE 汎用 GPU プログラミング入門, インプレスジャパン, 2011. .
[7] CUDA テクニカルトレーニング　Vol. I CUDA プログラミング入門：
http://www.nvidia.co.jp/docs/IO/59373/VolumeI.pdf
[8] 上野謙二郎, 孟林, 山崎勝弘：GPU を用いたリアルタイムレイトレーシングの検討, 情報処理学会第 74 回全国大会, 1ZB-1, 2012.
[9] 孟林, 上野謙二郎, 山崎勝弘：GPU を用いたリアルタイムレイトレーシングの並列化, 第 11 回情報科学技術フォーラム, FIT2012, 2C-1, 2012.
[10] 吉谷崇史, 山崎勝弘：適応型空間分割による並列レイトレーシング法, 情報処理学会第 54 回全国大会, 4Q-6, 1997.
[11] 増田匠吾, 山田遼, 孟林, 山崎勝弘：GPU を用いたレイトレーシングの高速化, 情報処理学会関西支部第支部大会, D-05, 2012
[12] Digital Matrix：http://www.not-enough.org/abe/manual/index.html
[13] 岡崎大輔：GPU 上でのレイトレーシング法の適応型空間分割の検討と実現, 立命館大学大学院理工学研究科創造理工学専攻電子システムコース修士論文, 2013
[14] 増田匠吾：GPU 上での画面分割によるレイトレーシング法の高速化, 立命館大学理工学部電子情報デザイン学科学士論文, 2013