卒業論文

GPU 上での画面分割による
レイトレーシング法の高速化

氏　　名　：　増田　匠吾
学籍番号　：　2260090059-9
担当教員　：　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、視点に入ってくる光線を逆に追跡して、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描画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方法について検討し、画面分割の並列化について GTX580 においての処理速度を評価する。また、球の画像を得るために、球の反射の実装について検討する。

GPU を用いた画面分割の方法では、GPU で行なっている分割とブロック分割、サイクリック分割を比較し、高速化について検討する。

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、複数個の 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。

実験には、SPD（Standard Procedural Databases）のシーンデータから teapot、tetra、mount、nurbs を用い[3]、ブロック数とスレッド数を変化させ、5 通りの方法で比較を行った。その結果、teapot ではブロック数 64*64、スレッド数 8*8 のとき、実行時間が 1.52 秒で、CPU 1 個に比べて 83 倍の速度向上を得ることができた。

球の反射の実装についての検討では、現状の画像について、反射が得られない原因と改善案について検討を行う。

目次

図目次

表目次

1. はじめに

昨今、コンピュータの性能は急速に進歩しており、現在では非常にリアルな映像をコンピュータ・グラフィックスによって生成することも可能となっていて、映画やゲームなど様々な分野、用途で用いられている。3 次元コンピュータ・グラフィックス画像を生成する手法としてレイトレーシング法がある。レイトレーシングは、光の動きを計算することにより画像を描画する手法で、光の反射や屈折、透過なども簡単に実現ため、リアルな画像を得ることが可能である。

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を発生させ、すべての物体に対して交差する点を探索する。その交点で反射、屈折をして、光線がどの方向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再度探索する。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呼び、レイトレーシングの処理の中で計算量が大きい部分となっている。これは、交差判定の回数が物体の数だけ行われるため、描画する画像が複雑になればなるほど物体の数が増え、それだけ交差判定が繰り返されるというわけである。

また、コンピュータの性能向上の中でも GPU（Graphics Processing Unit）の性能向上が凄まじい。GPU は本来、画像処理用のものでビデオカードとしての使用が普及しているが、汎用 CPU に比べて桁違いの演算性能を持っており、高い演算を要求する問題への適用もされている。このように、GPU の演算資源をグラフィックス処理以外の汎用的な演算を行わせる「GPGPU（General-purpose computing on graphics processing units）」と呼ばれる技術が注目を集めている。GPGPU の適用により性能向上が期待されている分野は、流体計算、気候シミュレーション、天体シミュレーション、製薬、医療シミュレーションやセキュリティなど幅広い分野がある。

今までに、GPU を用いた画面分割の並列化を実現していて、現在、空間分割、スクリーンマッピング、新規シーンデータの追加について検討しているところである。

本研究では、GPU を用いてレイトレーシングの高速化を図る。本論文では、画面分割の並列化について検討し、高速化についての考察を行う。また、球を含むシーンデータにおいて球の反射について検討し、実現する。

画面分割の並列化では、GPU で行われているデフォルトの分割について、他の分割方法の検討やストリーミング・マルチプロセッサ（SM）への割り当てなどについて比較評価を行う。

本論文は 2 章でレイトレーシングのアルゴリズム、本研究で用いた GPU と CUDA について述べ、3 章では、本研究で用いるシーンデータ：SPD、ストリーミング・マルチプロセッサによる画面分割手法、GPU 上での処理方式、及び実験結果と考察について述べる。4 章では、球の反射の検討について述べる。5 章で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

2.1 レイトレーシングのアルゴリズム [2]

私達が普段見ている物体の多くは、それ自身が発光しているのではなく、光源によって照らされることにより見える。光源から出た光は、様々な物体に衝突して、反射や屈折を繰り返す。光線は物体に衝突するごとに物体の輝度をもち、その光線が目に入ることで人は物体を認識することができる。つまり、光源から出る光線を 1 本ずつ辿って行き、目に入ってくる光線を全て見つけ出せば、私達が見ている像を再現することができる。しかし、光源から出る光線は非常に膨大であり、また目に入ってくる光線はごく一部で、ほとんどの光線は物体に吸収されたり、目に見えない方向に行ったりしてしまう。これを光源から 1 本ずつ辿っていくのは非常に効率が悪い。そこでこの原理を逆に考え、レイトレーシング法では、目に入ってくる光線を逆に辿り像を生成するので、視線追跡法や逆方向レイトレーシングと呼ばれる。図 1 に示すように、レイトレーシングは次の手順で行う。

（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、光線と物体との交差判定を行う。
（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） 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、その方向を光線とみなして（2）、（3）の処理を行い、屈折、反射して見える物体を抽出する。

以上の処理をスクリーン上のすべての画素に対して行うことで、画像を生成する。

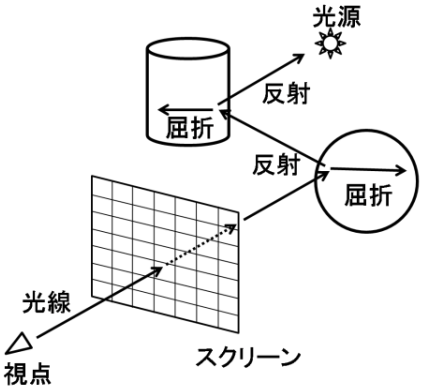


図 1 レイトレーシングの原理

2.2　GTX580 のアーキテクチャ

本研究で使用する GPU は、NVIDIA 社の Fermi アーキテクチャの Geforce GTX 580 である。図 2 に Geforce GTX 580 の構成を示す。Geforce GTX 580 は、Giga スレッドスケジューラと 4 個の GPC（Graphics Processing Cluster）から構成されている。Giga スレッドスケジューラはスレッドブロックを SM のスレッドスケジューラへと分配する機能を担当する。各 GPC には 4 個のストリーミング・マルチプロセッサ（SM）、及びポリゴンを画面上のピクセルに対応づけるラスターエンジンが搭載されている。つまり、GPU 全体では 16 個の SM をもつことになる。

Fermi アーキテクチャは第 3 世代の GPU で、メモリ階層もグローバルメモリ、L2 キャッシュ、L1 キャッシュ/共有メモリの 3 階層に強化されている。

各 SM は 32 個のストリーミング・プロセッサ（SP）、正弦関数や余弦関数を計算する SFU（Special Function Unit）4 個、ワープ・スケジューラとディスパッチ・ユニットがそれぞれ 2 個ある。また、64KB の L1 キャッシュ/共有メモリ、128KB のレジスタファイル、16 個のロード・ストアユニットをもつ。ロード・ストアユニットが 16 個あるため、1 クロックあたり 16 スレッド並列でソースアドレスと宛先アドレスを計算することができる。

CUDA 対応 GPU では、ある計算処理をスレッドと呼ばれる単位に分割して並列計算を行う。まず、スレッドスケジューラが GPU のプログラムをスレッドに分割して、SM の中の SP に割り当てる。SP はクロックサイクル毎に複数のスレッドに対して同一の命令を実行する。これを SIMT（Single Instruction Multiple Thread）と呼ぶ。

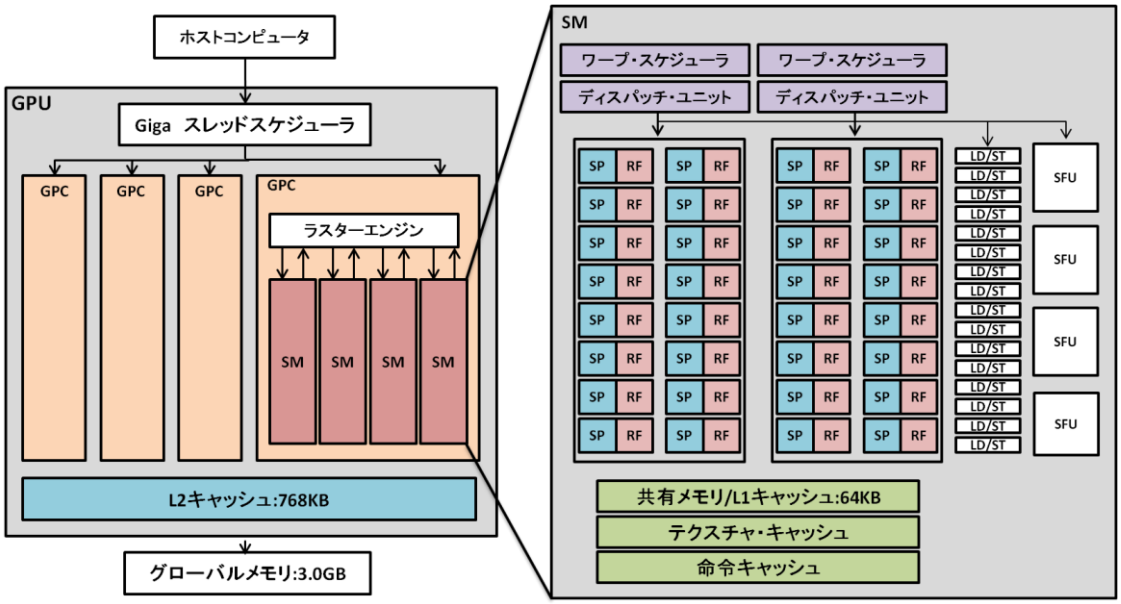


図 2　GPU の構成

SM は 32 個のスレッドをグループ化したワープを単位として、スレッドのスケジューリングを行う。SM にはワープ・スケジューラとディスパッチ・ユニットがそれぞれ 2 個ずつあるため、2 クロックで 2 ワープを実行することができる。各ワープ・スケジューラは、1 つのワープを選択し、1 命令で 16 スレッドを実行する。ディスパッチ先となるのは、16 個の SP、16 個のロード・ストアユニット、4 個の SFU のいずれかのグループとなる。Geforce GTX 580 には 16 個の SM があるため、最大 512 個の SP を並列動作させることができる。

2.3　並列プログラミング環境 [1]

CUDA（Compute Unified Device Architecture）とは、GPU 上で汎用的な演算を行わせるための統合開発環境で、NVIDIA 社により提供されている。CUDA でのプログラミングは C 及び C++がサポートされており、コンパイラとパフォーマンス向上のためのプロファイラも提供されている。

CUDA はコンピュータのマザーボード上にあるビデオカード内で動作を行う。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。通常のプログラムはこの方式で動作する。一方、デバイス側で実行されるプログラムをカーネルプログラムと呼び、デバイス上で動作し、GPU がプログラムの処理を行い、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 3 に示す。

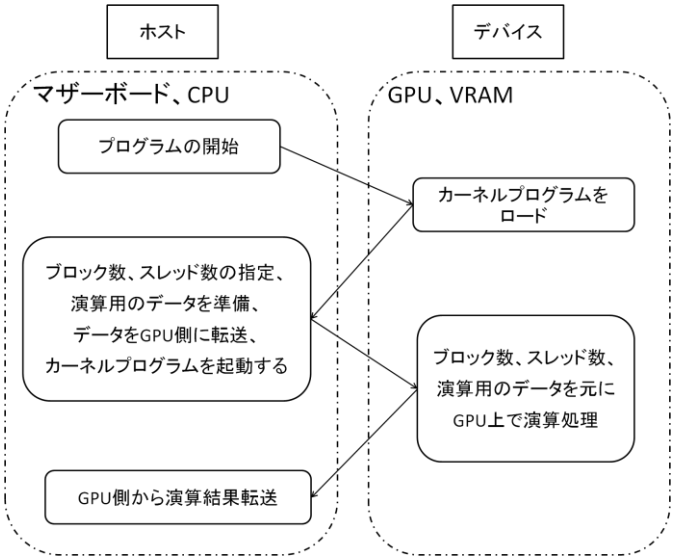


図 3　ホストとデバイスでの処理の流れ

CUDA では並列処理を行う概念として「グリッド」、「ブロック」、「スレッド」がある。これら 3 つは、図 4 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。スレッドは、カーネルで動作するプログラムの最小単位である。スレッド自体は、ホスト側から起動される。実行されるプログラムはすべて同じだが、タイミングは別、つまり非同期の動作を行なっている。ブロックは、スレッドをまとめたもので、1 つのブロックは最大 512 スレッドをまとめており、このスレッドの表現を最大 3 次元まで拡張することができる。グリッドは、ブロックをさらにまとめたもので、グリッド内のブロックは、2 次元で表現される。1 グリッド内の最大のブロック数は 65535 である。1 グリッド＝1 デバイスと認識できる。

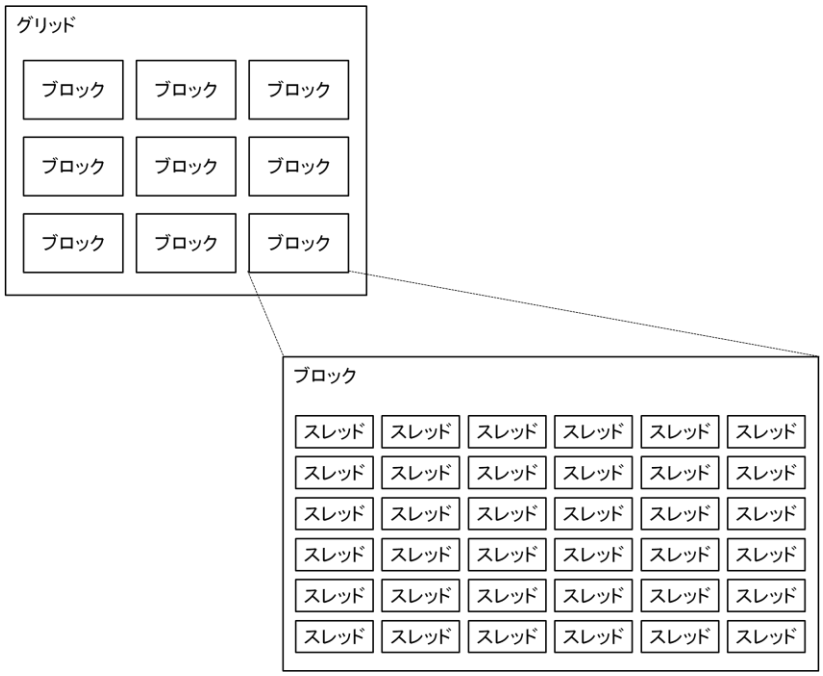


図 4　グリッド、ブロック、スレッドの構成

また、スレッドは 32 スレッドごとに動作し、この 32 スレッドを 1 ワープと呼ぶ。ここで重要なのは、スレッドの処理は 32 ずつだと区切りがいいということである。つまり、同時に動作するスレッド数は 32 の倍数がよい。

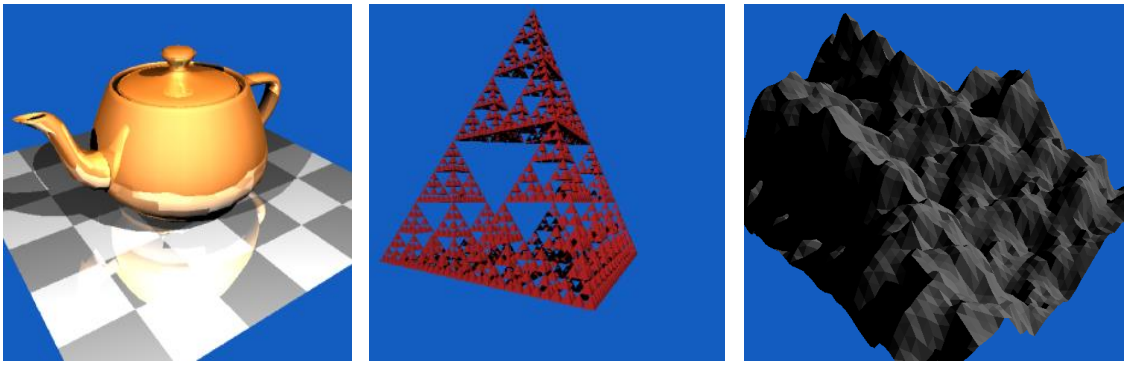
3.　画面分割による並列化の設計と実現

3.1　レイトレーシング用標準画像：SPD [3]

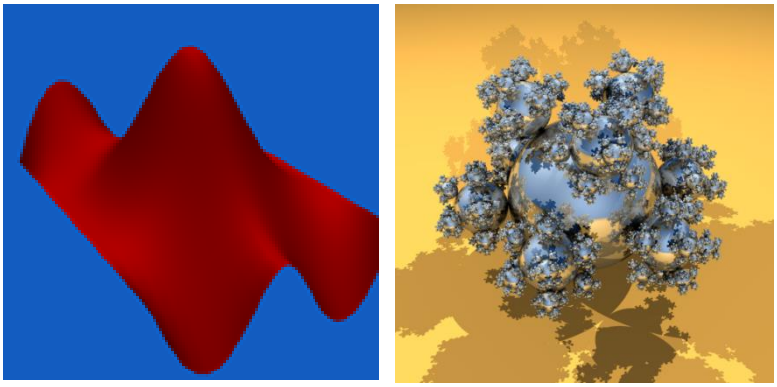
SPD（Standard Procedural Databases）は、レイトレーシング用に開発されたシーンデータプログラムである。この SPD を用いて出力されたシーンデータを使用することにより、様々な画像を生成することが可能となる。出力されたシーンデータは nff 形式のファイルとなっている。この nff 形式ファイルは以下の情報により構成されている。

・視線ベクトルと角度の定義
・スクリーンの背景色の定義
・照明の定義
・物体素材の性質の定義
・球の定義
・三角形の定義

図 5 に SPD を用いて生成した画像の例を、表 1 に各 SPD のポリゴン数とサイズを示す。



(a) teapot　　(b) tetra　　(c) mount



(d) nurbs　　(e) balls

図 5　SPD を用いた生成例

表 1　SPD シーンデータ

	teapot	tetra	mount	nurbs	balls
ポリゴン数	2328	4096	8192	1500	7381
サイズ(KB)	371	250	618	249	299

本研究ではこの SPD を用いてレイトレーシングの実験を行う。

3.2　ストリーミング・マルチプロセッサによる画面分割の並列化

レイトレーシングは実行処理時間が膨大であるが、その多くは交差判定処理に費やされている。つまり、交差判定の高速化が必要不可欠である。そのためには、交差判定の回数をアルゴリズムの改良により減らす、あるいは交差判定そのものを高速化することが必要である。ここでは交差判定を高速化するために GPU を用いた画面分割による並列化を行う。

レイトレーシングでは、スクリーン上のある画素の輝度値を求める処理が、他の画素の輝度値を求める処理に依存していないので、各画素の輝度値を独立に計算できる。そのため、GPU を用いた画面分割による並列化と相性が良い。

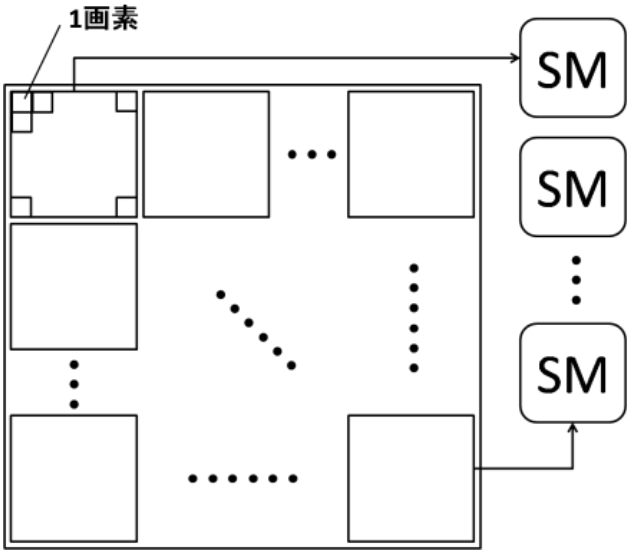


図 6　SM による画面分割

本研究では 512*512 の解像度のスクリーンを用いる。SM による画面分割では、図 6 に示すようにスクリーンを複数のブロックに分割し、各ブロックをそれぞれ 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する。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。例えば、ブロック数 64*64、スレッド数 8*8 の場合は、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 つ存在することになる。

3.3　GPU 上での並列処理方式

図 7 に作成した処理系の構成を示す。本プログラムは CPU 側と GPU 側の 2 つの処理から成り立っている。

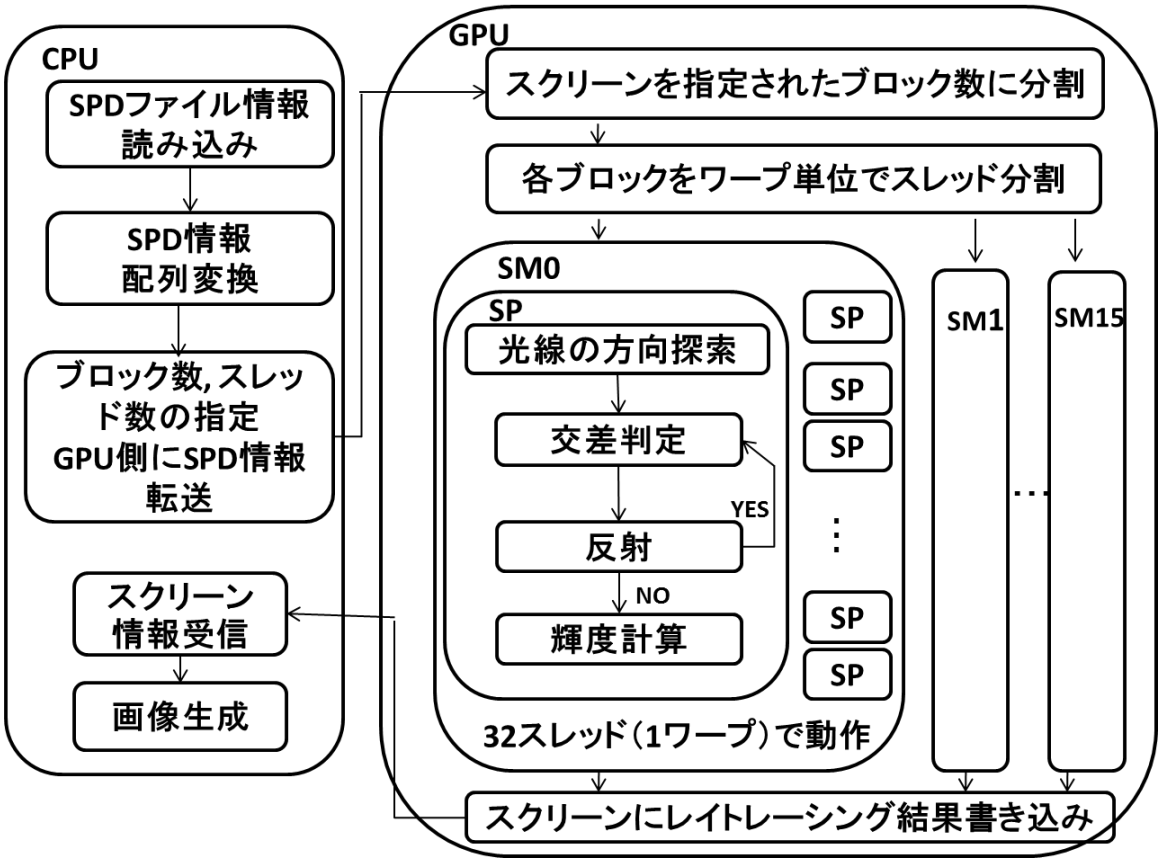


図 7　画面分割並列化の処理系

まず、CPU 側でレイトレーシングの実験用のシーンデータである SPD ファイルを読み

込む。次に、CPU 側でブロック数とスレッド数を指定し、SPD 情報を GPU 側に転送する。

GPU 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、屈折、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。

CPU 側でスクリーン情報を受信し、その結果を ppm 形式のファイルに変換することで画像を得る。

3.4　画面分割の実現方法

画面分割の方法は、図 6 のような単なるブロック分割や図 8 のようなサイクリック分割がある。

GTX580 と CUDA では、現状仕様として図 6 のようなブロック分割となっていてサイクリック分割のように任意の SM に任意のブロックを割り当てる事はできない。また、各 SM の動作は非同期であり、SM 内での処理が終了したら待機しているブロックに移行する。そのため、各 SM の動作回数は必ずしも一定ではなくサイクリック分割よりも効率的な動作を行なっていると言える。つまり、GPU においての分割方法による最適化はすでに図られているため、ユーザー側で手を加える必要はない。

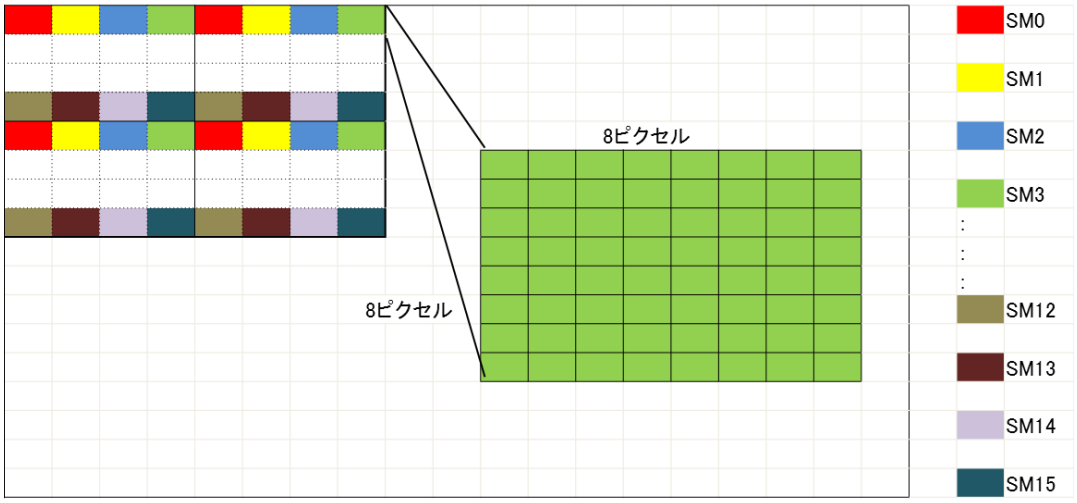


図 8　サイクリック分割

3.5 実験結果と考察

3.5.1 実験環境

図 5 で示した 4 種類のシーンデータ（teapot, tetra, mount, nurbs）に対して、GPU を用いた画面分割による並列化を行った。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。

C 言語逐次処理の実験環境は
OS：Windows7 Ultimate
プロセッサ：Intel(R) Core(TM) i7-2600K CPU @3.4GHz
メモリ容量：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 CPU @3.4GHz
メモリ容量：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：3.00GB
並列処理環境：CUDA

3.5.2　実験結果と考察

C 言語逐次処理と GPU での実行時間と速度向上比を表 2 に示す。

表 2　実行時間と速度向上比

ブロック数 スレッド数	teapot		tetra		mount		nurbs	
	実行時間(秒)	速度向上比	実行時間(秒)	速度向上比	実行時間(秒)	速度向上比	実行時間(秒)	速度向上比
256*256 2*2	13.6	9.27	7.34	9.03	16.2	12.0	3.27	9.44
128*128 4*4	3.65	34.5	1.90	34.90	4.06	48.1	0.83	37.2
64*64 8*8	1.52	83.0	0.76	86.8	1.79	109	0.32	96.6
32*32 16*16	1.82	69.7	0.86	76.3	1.81	107	0.34	90.9
16*16 32*32	2.33	54.1	1.04	63.1	1.93	102	0.39	79.2
CPU実行時間(秒)	126.1		66.3		195.2		30.9	

図 9 は teapot, tetra, mount, nurbs の 4 つのシーンの速度向上比を表したものである。このグラフからどのシーンも共通して、ブロック数が 64*64、32*32、16*16 の場合は速度向上比が大きく、特にブロック数が 64*64、スレッド数が 8*8 の場合が最も高速であることが確認できる。これは、ブロック数 64*64、32*32、16*16 の場合、ワープ内のスレッドロック数が少なくなると 1 ブロック内に含まれるスレッド数が増加し、スレッドあたりの使用レジスタ数が減少したためである。SM は 16 個が並列に動作しているので、64*64 の場合が、他の場合に比べて、使用効率が良い。これはブロック数が少なくなると 1 ブロック内に含まれるスレッド数が増加し、スレッドあたりの使用レジスタ数が減少したためである。SM は 16 個が並列に動作しているので、64*64 の場合が、他の場合に比べて、使用効率が良い。

また、ブロック数が 256*256、128*128 の場合は速度向上があまり得られていない。これは、1 ブロック内のスレッド数が 32 よりも小さいので、各 SP に均等に処理を割り当てることができず、速度向上比が大幅に低下している。

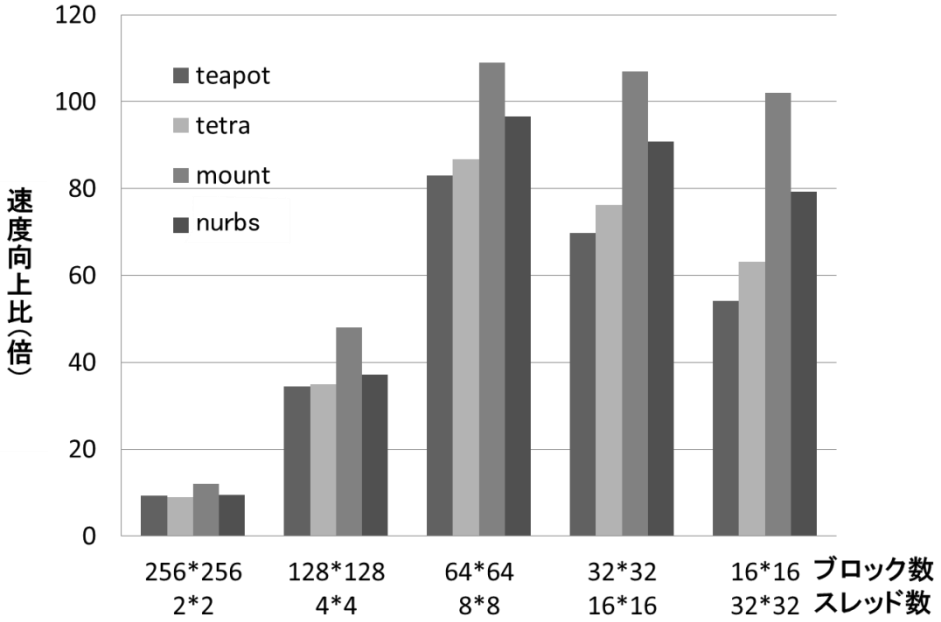


図 9　速度向上比

表 2 の実行時間は、表 1 の SPD シーンデータのサイズと相関が認められる。1 画素のレイトレーシング計算で SPD 全体を参照するので、SPD のサイズに比例した時間がかかっている。また、サイズがほぼ同じ tetra, nurbs ではポリゴン数に比例した時間がかかっていると考えられる。ほかに、mount が最も速度向上比が大きい。これはポリゴン数、サイズが大きいため、計算量が多く並列処理の効果が大きく表れたためだと考えられる。

実行時間は、ポリゴン数や反射回数などがシーンデータに含まれているため、シーンデータのサイズとほぼ比例する。

4.　球の反射の検討

4.1　球との交差判定 [13]

球の中心のベクトルを c、視点ベクトルを e とすると、視点から球の中心 c へののベクトル 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、

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

となればよい。このとき t は

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

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。

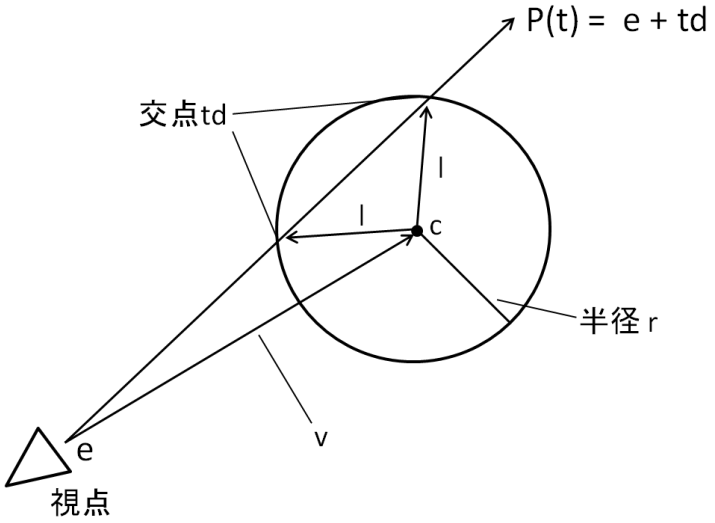


図 10　球の交差判定

4.2　Phong のモデル [2]

輝度を計算するモデルを陰影付けモデル（shading model）という。
陰影付けモデルでは、環境光、拡散反射光、鏡面反射光を考慮している。
環境光、拡散反射光、鏡面反射光について以下に述べる。

（1）　環境光

環境光とは光源からの光ではなく、光が壁や床などにあたって反射したり、それらの相互反射により到達した光を仮定している。環境光による反射光の強さI_aは以下の式で求めることができる。

・面の反射率をk_a
・環境光の強さをI_{amb}

とすると環境光による反射光の強さI_aは

$$I_a = k_a \times I_{amb}$$

と表すことができる。

（2）　拡散反射光

拡散反射光は、どの方向にも均等に反射する光で、光が物体表面に入り込み、もう一度放射される光である。
拡散反射光の輝度I_{d}は以下の式で求めることができる。
・拡散反射光を求める点への入射角を θ
・入射光の強さをI_l
・拡散反射係数をk_d
・光源への方向ベクトルを L
・面に対する単位法線ベクトルを N
とすると拡散反射光の輝度I_{d}は

$$I_{\mathrm{d}} = k_d \times I_l \times \cos\theta = k_d \times I_{\mathrm{l}} \times (\mathrm{L} \cdot \mathrm{N})$$

と表すことができる。

（3）　鏡面反射光

表面の滑らかなプラスチックや金属などの光沢感を持つ物体は、光源に照らされることで、鏡面反射によって表面の一部にハイライトが生じる。このような、ハイライト効果を出すために鏡面反射の式が必要である。鏡面反射光の輝度I_sは以下の式で求めることができる。

・鏡面反射光を求める点への入射光をI_l
・入射光の正反射光と視線のなす角を α
・鏡面反射係数をk_s
・光源からの反射方向ベクトルを R
・視線ベクトル E の逆ベクトル V
・ハイライト係数を n
とすると鏡面反射光の輝度 Is は

$$I_s \quad = \quad k_s \times I_l \times \cos^n \alpha \quad = \quad k_s \times I_l \times (\mathrm{R} \cdot \mathrm{V})^n$$

と表すことができる。このとき、n の値によってハイライトの広がりを制御することができ、n の値が大きいほど光沢の広がりが小さくなり、シャープなハイライトを得ることができる。

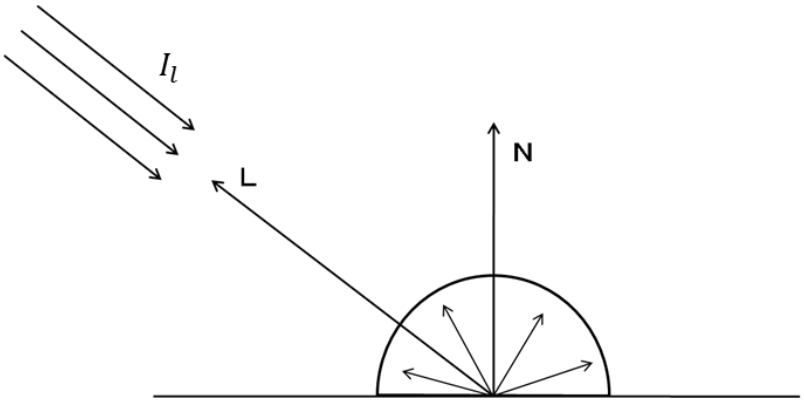


図 11　拡散反射光の原理

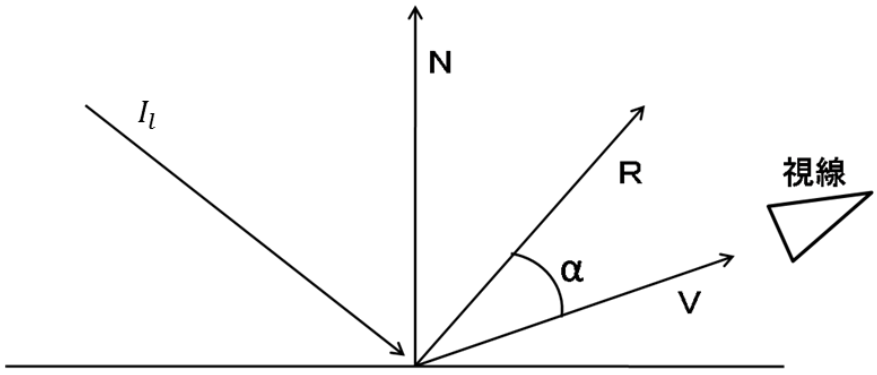


図 12　鏡面反射光の原理

環境光、拡散反射光、鏡面反射光を考慮した陰影付けモデルは、次式のようになる。

$$I = k_a \times I_a + k_d \times I_l \times \cos\theta + k_s \times I_l \times \cos^n \alpha$$

ベクトル L,N,R,V を用いて表すと、

$$I = k_a \times I_a + k_d \times I_l \times (\mathrm{L} \cdot \mathrm{N}) + k_s \times I_l \times (\mathrm{R} \cdot \mathrm{V})^n$$

となり、光源が複数個(n_L個)ある場合は、

$$I = k_a \times I_a + \sum_{l=1}^{n_L} I_l \times (k_d \times (L_l \cdot \mathrm{N}) + k_s \times (R_l \cdot \mathrm{V})^n)$$

となる。

しかし、Phong のモデルでは映り込みの生じる鏡面反射を表現できない。

4.3 大域照明モデル [2]

レイトレーシング法では、Phong のモデルに加えて、反射光成分と透過光成分がある。反射光成分は、映り込みを起こす反射光で、透過光成分は、交差した物体が透明な場合に、その物体を透過してくる光のことである。反射光成分と透過光成分は光線の強さが減衰するまで追跡して計算されるため、Phong モデルと区別して、大域照明モデルと呼ばれる。

レイトレーシング法のシェーディング計算は 2 次光に対する反射係数をk_r、透過係数をk_tとして次式で表せる。

$$I = k_a \times I_a + \sum_{i=1}^{n_L} I_i \times \left[\left(k_d \times (\mathrm{L} \cdot \mathrm{N}) + k_s \times (\mathrm{R} \cdot \mathrm{V})^n \right) \right] + k_r I_r + k_t I_t$$

この式の各係数は、次式を満たすように設定する。

$$k_a + n_L(k_d + k_s) + k_r + k_t = 1$$

4.4 実装方法の検討

実装前の状況として、図 5(a)と図 13(a)より三角形ポリゴンから三角形ポリゴン、球ポリゴンへの鏡面反射は映っていたが、図 13(b)より球ポリゴンから三角形ポリゴン、球ポリゴンへの鏡面反射が映っていなかった。

4.3 で示しているように、レイトレーシング法では、2 次光に対する反射、透過を考えているが、球のシーンデータにおいてはその部分がうまくできていないために図 13(b)のような画像が生成されると思われる。

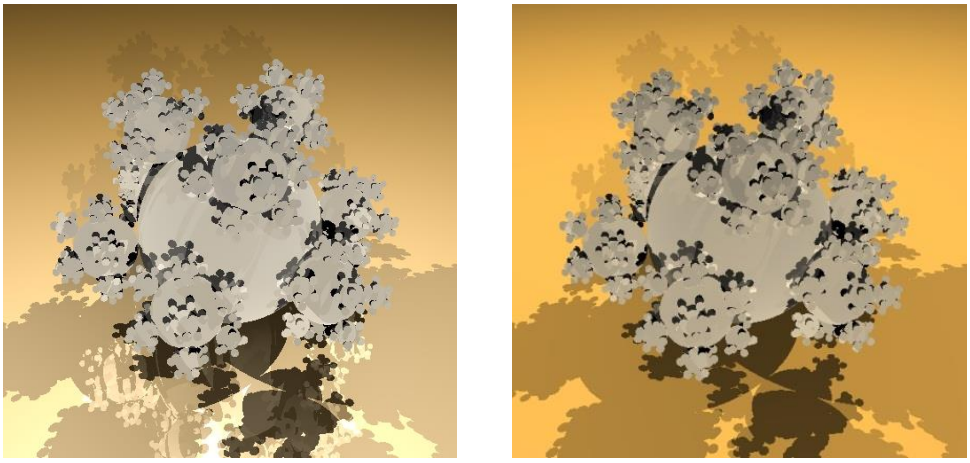
球の反射ができていない原因として、球面にあたった光線の反射先がおかしい、光線との交点が球の内部になっているなどが可能性としてあげられる。

この問題への対策としては、反射した光線ベクトルの修正、プログラムの構造の見直しなどが考えられる。具体的には、光線の動きについての演算の動きを見ていくことが必要である。

また、図 5(e)では、青く空の色が反射しているが、その部分の出力ができていない。反射

した光線が無限遠点に到達した場合その点の色情報がうまく読み込めていないことが演算の処理の出力よりわかった。

この部分については、無限遠点に到達した場合の条件を定義し、背景色を反映させることで正しい出力結果が得られると思われる。



(a)壁と床に反射を追加した結果　　(b) 通常の実行結果

図 13　現状の balls 生成画像

5. おわりに

本研究では、GPU を用いた画面分割の手段の検討、球の反射の実装について述べた。

画面分割の並列化では、他の分割方法について検討を行った。実験では Geforce GTX 580 を用い、SPD の 4 つのシーンデータ（teapot, tetra, mount, nurbs）に対して、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。その中でも teapot はブロック数 64*64、スレッド数 8*8 のときに、実行時間 1.52 秒で、CPU 1 個に比べて 83 倍の速度向上を得ることができた。これはスレッド数が 8*8=64 で、ワープ数 32 の倍数になっているため SP を効率よく使用できているためである。

球の反射の検討では、現在の状況と対策について述べた。

謝辞

本研究の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深く感謝します。また、本研究に助言をして頂いた高性能計算研究室の皆様に心より感謝いたします。

最後に、本研究において多くのご協力をいただきました孟助手、岡崎氏、藤川氏、山田氏に深く感謝いたします。

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 CUDA™ 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 11 回目支部大会, D-05, 2012

[12] GDEP：高速演算記 第 3 回 「チューニング技法その１　ＣＵＤＡプログラミングガイドからピックアップ」http://www.gdep.jp/column/view/3

[13] Digital Matrix：http://www.not-enough.org/abe/manual/index.html

[14] 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

[15] 山田遼 : 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, 立命館大学理工学部電子情報デザイン学科学士論文, 2013

付録

以下に実験で使用したプログラムの一部を記す。

main 関数(CPU)

```
int main(int argc, char **argv)
{
        unsigned char *pixels;
        color background_color;
        triangle_node triangles = NULL;
        sphere_node spheres = NULL;
        light_node lights = NULL;
        viewpoint view;
        unsigned int do_close;
        FILE *in = NULL;
        FILE *output = NULL;

        unsigned int cudaTimer = 0;

        /* check how we were called */
        do_close = check_args(argc, argv, &output);

        /* read our input */
        in = fopen("mount.nff","r");
        parse_nff(in, &view, background_color, &triangles, &spheres,
                        &lights);



        /* allocate by the given resolution */

        pixels = (unsigned char*)mem_alloc(sizeof(unsigned char) * view.resolution[0]
                        * view.resolution[1] * 3);

        char c;
        int sizeof_triangle = sizeof(triangle) + sizeof(triangle_node);

printf("sizeof verticles %d¥n", sizeof(triangles->element.vertices));
```

```
	int lcount = 0;
	light_node l;
	for(l=lights; l!=NULL; l=l->next){
		light *lig = &(l->element);
		lcount++;
	}
	printf("lcount %d¥n", lcount);

	light *Darray_light;

	cutilSafeCall(cudaMalloc((void **)&Darray_light, sizeof(light) * lcount));
	printf("sizeof larray %d¥n", sizeof(light) * lcount);

	int li=0;
	for(l=lights; l!=NULL; l=l->next){
		light *lig = &(l->element);

		cutilSafeCall(
			cudaMemcpy(&(Darray_light[li]), lig, sizeof(light),
cudaMemcpyHostToDevice)
		);
		li++;
	}

	int tcount = 0;
	triangle_node t;
	for(t=triangles; t!=NULL; t=t->next){
		triangle *tri = &(t->element);
		printf("is_patch %f¥n", tri->is_patch);

		tcount++;
	}
	printf("tcount %d¥n", tcount);

	triangle *Darray_triangle;
```

```
        cutilSafeCall(cudaMalloc((void **)&Darray_triangle, sizeof(triangle) * tcount));
        printf("sizeof tarray %d¥n", sizeof(triangle) * tcount);

        int ti=0;
        for(t=triangles; t!=NULL; t=t->next){
                triangle *tri = &(t->element);

                cutilSafeCall(
                        cudaMemcpy(&(Darray_triangle[ti]), tri, sizeof(triangle),
cudaMemcpyHostToDevice)
                );

                ti++;
        }


unsigned char *Dpixels;

        cutilSafeCall(cudaMalloc((void**) &Dpixels, sizeof(unsigned
char)*512*512*3));
        cutilSafeCall(cudaMemcpy(Dpixels, pixels,sizeof(unsigned char)*512*512*3,
cudaMemcpyHostToDevice));


cutilCheckError(cutCreateTimer(&cudaTimer));
cutilCheckError(cutStartTimer(cudaTimer));

//dim3 block(2,2);   dim3 grid(256,256);
//dim3 block(4,4);   dim3 grid(128,128);
dim3 block(8,8);   dim3 grid(64,64);
//dim3 block(16,16);   dim3 grid(32,32);
//dim3 block(32,32);   dim3 grid(16,16);

cudaFuncSetCacheConfig(ray_trace, cudaFuncCachePreferL1);

        ray_trace<<<grid,block>>>(Dpixels, Darray_triangle, Darray_light);   …①
```

```
        cutilSafeCall(cudaMemcpy(pixels, Dpixels,sizeof(unsigned char)*512*512*3,
cudaMemcpyDeviceToHost));

cutilCheckError(cutStopTimer(cudaTimer));
printf("second %f¥n", cutGetTimerValue(cudaTimer));
cutilCheckError(cutDeleteTimer(cudaTimer));

        /* write the results */
        write_ppm_file(output, pixels, view.resolution[1], view.resolution[0]);

        if (do_close)
                fclose(output);

        /* free the linked lists */
        delete_triangle_list(&triangles);
        delete_light_list(&lights);

        printf("pixels[0]=%d¥n", pixels[0]);

cutilExit(argc, argv);

        return 0;
}
```

```
ray_trace 関数(GPU)
__global__
void ray_trace(unsigned char *pixels,    triangle triangles[], light lights[])
{
        int i, j, k;
        int width, height;
        width = 512;
        height = 512;
        point3 u, v, w, d;
        color object_color = {0.0, 0.0, 0.0};
        k =0;

        typedef struct {
        point3 from;
        point3 at;
        point3 up;
        double angle;
        double hither;
        double aspect_ratio;
        double resolution[2];
        double t;
        double b;
        double r;
        double l;
} viewpoint;

viewpoint Dview;

Dview.from[0] = -1.60;
Dview.from[1] = 1.60;
Dview.from[2] = 1.7;


        calculate_basis_vectors(u, v, w);

        multiply_vector(u, -1.0, u);
```

```
i = blockIdx.x * blockDim.x + threadIdx.x;       …②
j = blockIdx.y * blockDim.y + threadIdx.y;

                        compute_viewing_ray(d, u, v, w, i, j);



                        if (ray_color(Dview.from, 0.0, d, triangles, lights, object_color,
                                                        MAX_REFLECTION_BOUNCES,
NULL)) {
                                pixels[((i+(j*width))*3)+0] = object_color[0]*255;
                                pixels[((i+(j*width))*3)+1] = object_color[1]*255;
                                pixels[((i+(j*width))*3)+2] = object_color[2]*255;
                        } else {
                                pixels[((i+(j*width))*3)+0] = 0.078*255;
                                pixels[((i+(j*width))*3)+1] = 0.361*255;
                                pixels[((i+(j*width))*3)+2] = 0.753*255;
                        }
}
```

CUDA では、ブロック、スレッドと各画素の割り当ては式①、式②で表されている。

式①は GPU の起動命令で、式②は演算を行う画素の座標(i, j)の指定である。

式②の blockIdx.x, blockIdx.y は各ブロックの x 座標, y 座標で blockDim.x, blockDim.y は各ブロックの大きさ、threadIdx.x, threadIdx.y は各スレッドの x 座標, y 座標を表している。

式②の座標は SP に直接割り当てているため、実行する SM の指定は行えない。このため、サイクリック分割を CUDA で実装することはできない。

式①の GPU 起動命令を複数個並べることによって、擬似的にサイクリック分割を生成することは可能であるが SM の動作が一定数終わるまで次の部分に移ることができない。また、GPU 起動命令を複数回行うことになるのでタイムラグが発生する。これらの理由により、遅くなるのでこの方法は、現実的ではない。