卒業論文

レイトレーシング法によるベンチマークシーンの表示と高速化

氏名　：藤川　佳之
学籍番号：2260080079-9
指導教員：山崎　勝弘　教授
提出日　：2013 年 2 月 20 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１つ１つの画素に視線をとばし、光線を追跡して、交差する物体を調べ、その情報をスクリーン上に投影させることによって画像を得る手法である。計算方法が単純であるが、多くの計算時間を必要とするため、処理の高速化が必要である。

本研究では、レイトレーシング法において使用するベンチマークシーンの正しい表示と追加、また GPU を使った処理時間の高速化について評価する。

実験には、SPD（Standard Procedural Databases）のシーンデータから teapot、gears、を用い、ブロック数とスレッド数を変化させて 5 通りの方法で、CPU との比較を行った。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 1.54 秒で、CPU 1 個に比べて 110 倍、gears ではブロック数 64*64、スレッド数 8*8 のときに、実行時間 9.573 秒で、826 倍の速度向上を得ることができた。

目次

図目次

表目次

1. はじめに

近年、映画、テレビ、またゲームなどの分野で使われるコンピュータグラフィックスの技術は急速に進歩し、普及してきている。そのグラフィックスの描写方法としてレイトレーシング法があげられる。レイトレーシング法は「光線追跡法」とも呼ばれ、一つ一つの画素に対して光線を飛ばし、その光線を追跡して画素の色・形を形成する。光線の一つ一つを細かく調べるため、非常にリアルな画像を生成することができる。最近では、フォトンマッピング法、ラジオシティ法などの新しい手法も開発されてきている。しかし、レイトレーシング法は、光線のすべてを追跡するため、計算処理・時間が非常に膨大になる欠点がある。

レイトレーシング法のアルゴリズムは、視線から光線をスクリーン上にある画素を通って物体方向に向かう。そして光線と物体が交差するのであれば、その交点で反射・屈折・透過などの処理を行う。そしてまた、その交点から変化した光線を追跡し直す。この処理を順次行い、最終的に得た情報を元に、スクリーン上に投影させ、画像を得る。物体の数が多いほど、また、物体の形が複雑なものほど、計算量が膨大になる。

この問題を改善するために、本研究室では GPU（Graphics Processing Unit）を用いてレイトレーシングの高速化を図る。GPU での研究では、画面が生成されるスクリーンをブロック分割する。これにより各スレッドが並列に演算を行うことで、高速化に繋がる。

本研究は、レイトレーシング用に開発されたシーンデータである SPD を用い、正しい画像の表示について行う。提案手法についても SPD シーンデータを用いて評価を行っている。現在、tetra、mount は既に表示されており、teapot、gears はバグが出ていたため、この二つの修正を行った。teapot は、光源の光の照り返しが強く、ヤカンの下部と床が正常に表示されていなかった。そこで反射の計算を行う calculate_reflection_ray 関数に減衰を与えることで解決することができた。gears は、床が正しく表示されてなく、また、ギアの１つ１つには黒い線が放射線状に表示されている。修正方法として、床は teapot と同じく反射を計算する際に、減衰を与えることにより解消した。黒い線の修正は、まず線が現れている箇所の特定から始めることにし、出力画像を５１２×５１２に番号付けを行った。そこからギア１つだけに注目し、実行時間を減らして修正を行う。

本論文は、2 章でレイトレーシング法のアルゴリズム、光線の種類、交差判定について述べ、3 章では本研究で用いるベンチマークシーンの概要と追加、4 章では CPU 上での実験と考察、5 章では GPU による実験と考察について述べる。6 章で本研究の成果と今後の課題を述べる。

2. レイトレーシングの各種手法

2.1 アルゴリズム

普段、私達が見ている物体は、それ自体が発光しているものもあるが、大部分は光源から発せられた光に照らされて見えている。その光は人体の目に届くまでに様々な物体に衝突し、反射や屈折を繰り返す。衝突した際に、光線は物体の輝度を持ち、その光線が目に入ることによって人は物体を認識する。つまり、その光線の１本１本を調べ、光線の情報を割り出すことで、私達が見ている映像を再現することができる。しかし、光源から発せられる光線はとても膨大である。また、光線のほとんどが物体に吸収されたり、物体を識別するのに必要ではない光線であったりと、目に入ってくる光線の数はごくわずかである。したがってこの多くの光線をすべて調べるには効率が悪い。そこで視点から目に入ってくる光線のみを辿り、物体を形成する方法がレイトレーシング法である。図１に示すように、レイトレーシング法は次の手順で行う。

（１） 視点からスクリーン上のある画素を通って、物体方向に向かい、光線と物体の交差判定を行う。
（２） 交差判定により、光線と交差する物体が存在するなら、光線と物体との交差判定を行う。交差する物体が複数ある場合は、すべての物体について交点を求める。交差する物体がない場合は、その画素を背景の輝度とする、
（３） 光線と交差する物体との距離を求め、最も視点に近い物体を抽出する。これは陰面消去するためである。
（４） 抽出した物体の輝度を求める。また、反射・屈折がある場合は、その方向を求め、その点を新たな光線とみなして（２）（３）の処理を行い、反射・屈折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い、物体をスクリーン上に形成する

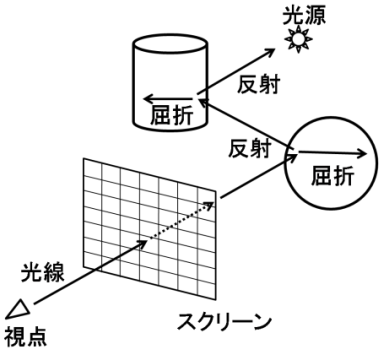


図１　レイトレーシングの原理

レイトレーシングでは画像に立体感を施すために、光源の種類や位置、物体の材質など

で定まる物体表面の色調の変化や影を求める。レイトレーシングでは図 2 に示すように、交点から光源が見えれば可視であるが、交点から光源を見たときに別の物体と交差すると、光源と反対側の物体表面には光が当たらない陰（shade）ができる。また物体はその形に対応した影（shade）を光源と反対側にある物体に投影する。この変化により物体は、より立体的になる。

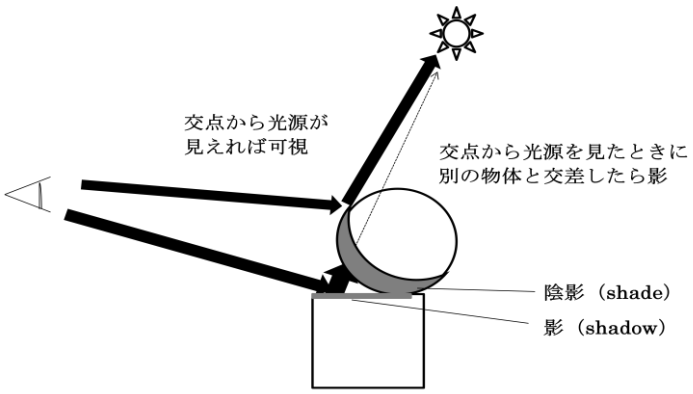


図 2　影と陰影

2.2　拡散反射光

拡散反射光とは、光源から入ってくる光が、すべての方向に均一に反射する光である。紙のようなざらざらとした質感のある物体でみられる。反射する光の量は入射する光の強さに依存する。入射する光の量は、光のあたる場所によって異なるため、拡散反射光を計算することにより物体の表面に陰影を得ることができる。図 3 に拡散反射光の原理を示す。

並行光源の場合、単位面積あたりの表面への入射光の強さは、入射角 θ の余弦（cosine）に比例する。入射角が 0 度の時に最大になり、大きくなるにつれて、$\cos\theta$ で光の強さが小さくなる。

拡散反射光の輝度 Id は以下の式で求めることができる。

・入射角：θ
・拡散反射係数：Kd
・入射光の強さ：Il
・光源への方向ベクトル：L
・面の法線ベクトル：N

$$Id = Kd \times Il \times \cos\theta = Kd \times Iq \times (L \cdot N)$$

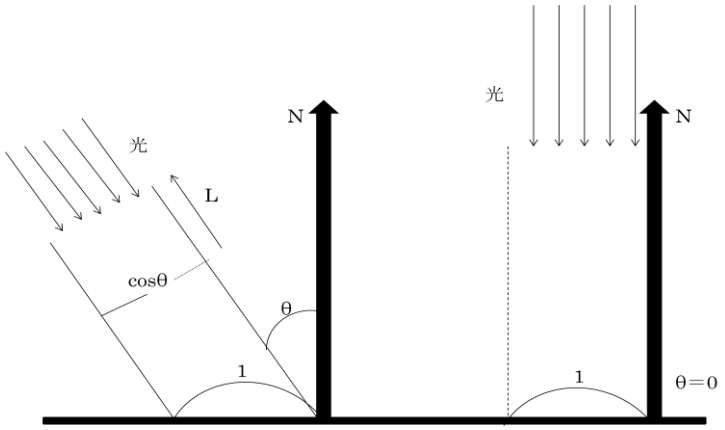


図 3　拡散反射

2.3　鏡面反射光

拡散反射光だけでは、滑らかな物体の表面や鋭く光るような光沢感のある画像は得られない。このような光沢の表現は、プラスチックや金属などの質感表現には必要不可欠である。入射した角度のちょうど反対側（反射角）に強く反射する光はハイライトと呼ばれる。ハイライトの位置は、正反射方向と視線が一致した場所に最も強く表われ、ずれるとハイライトは急激に減少する。図 4 に鏡面反射光の原理を示す。

鏡面反射光の輝度 Is は以下の式で求めることができる。

・入射角：θ
・鏡面反射係数：Ks
・入射光の強さ：Il
・光源からの反射光方向：R
・反射光の視点方向：V
・R と V のなす角：γ
・ハイライト係数：n
・面の法線ベクトル：N

$$Is = Ks \times Il \times \cos^n \gamma = Ks \times Il \times (R \cdot V)^n$$

このとき、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。

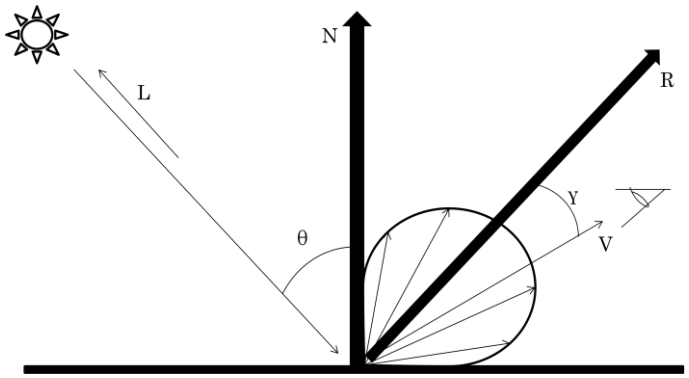


図 4　鏡面反射

2.4　透過光・屈折光

透過光・屈折光は、ガラスや透明な物体の質感表現を出すために必要なものであり、その物体を通過していく光のことである。光の強さが減衰するまで追跡して計算する。図 5 に透過光・反射光の原理になる、スネルの法則を示す。

光が屈折率の異なる物質に入射すると反射と屈折が生じる。屈折率は、物体によって固有の値をもっている。屈折率 n_1 の物体の中から屈折率 n_2 の物体の中へ光が入射すると、入射角 θ_1 と反射角 θ_2 の間にはスネルの法則が成立する。

$$n_1 \times \sin\theta_1 = n_2 \times \sin\theta_2$$

これより屈折光 T は以下の式で求めることができる。

・入射角：θ_1
・屈折角：θ_2
・入射光：L
・面の法線ベクトル：N
・物体 1 の屈折率：n_1
・物体 2 の屈折率：n_2

$$T = \frac{1}{n} \times \{L + (e - g) \times N\}$$

ここで $n = \frac{n2}{n1}$ は相対屈折率であり、

$$c = \cos\theta_1 = -(L \cdot N)$$
$$g = \sqrt{n^2 + c^2 - 1}$$

また、屈折光での輝度 I は次の式で求めることができる。

・透過率：kt

・透過した光線の明るさ：It

・反射率：kr

・反射した光線の明るさ：Ir

$$I = kt \times It + kr \times Ir$$

Kt と kr の間にはフレネルの法則が成立する。

$$Kt + Kr = 1$$

$$Kr = \frac{1}{2}\left\{\frac{\sin^2(\theta 1 - \theta 2)}{\sin^2(\theta 1 + \theta 2)} + \frac{\tan^2(\theta 1 - \theta 2)}{\tan^2(\theta 1 + \theta 2)}\right\}$$

これを変形すると

$$Kr = \frac{1}{2}\left\{\left(\frac{c-g}{c+g}\right)^2 + \left(\frac{n^2 \times c - g}{n^2 \times c + g}\right)^2\right\}$$

となる。

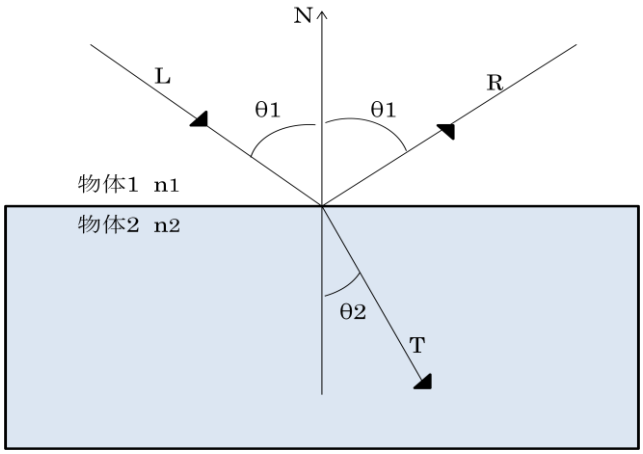


図 5　スネルの法則

2.5　交差判定

交差判定は、レイトレーシング法において最も重要な部分である。レイトレーシングの欠点でもある膨大な計算時間がこの交差判定になるが、この部分を効率化することによって、レイトレーシングの高速化に繋がる。

図 6 に視点とレイトレーシングの関係を示す。まず直線の定義として視線を e、スクリーン上のある画素の点を q とする。視点 e から画素 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

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

となる。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。

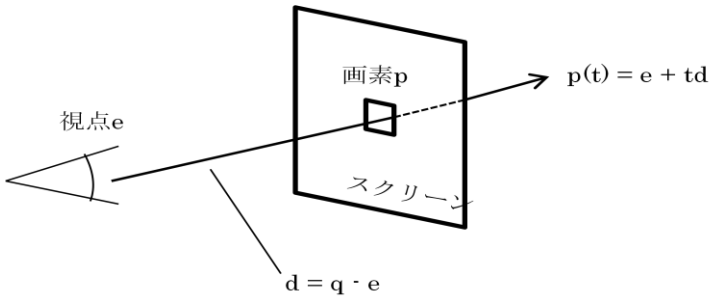


図 6　視点とスクリーン

(1) 球との交差判定

図 7 に光線と球との関係を示す。

・中心点：c
・半径：r
・視点から中心点までの距離：v
・視点から交点までの距離：t
・中心点から

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 が球と交差すると仮定すると、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、球の半径の２乗と等しいので、

$$l \cdot l = td - v \cdot td - v = | td - v |^2$$

この２式を連立して変形させると、

$$|td - v|^2 t^2 + 2v(x-c)t + |d-c|^2 - r^2 = 0$$

となる。この t についての方程式が実数解をもてば、光線は球と交差しているということになる。交差している際の t の値は、

$$t = \frac{-v(d-c) \pm \sqrt{D}}{|v|^2}$$

$$D = \{ v(d-c) \}^2 - |v|^2 (|d - c|^2 - r^2) \geqq 0$$

と表すことができる。このとき $t < 0$ ならばその点は、視線よりも後ろに存在することとなり、交差点は見えない。

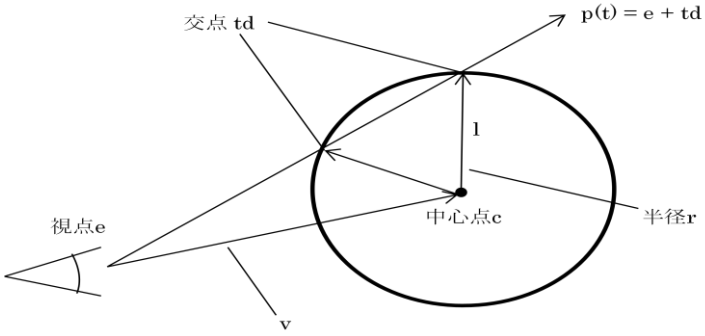


図７　球との交差判定

⑵ 三角形との交差判定

図 8 に光線と三角形との関係を示す。形成される物体は、各面が三角形からなる多面体で表現されることが多い。そのためレイトレーシング法では三角形の交差判定は特に重要である。

三角形の頂点を A、B、C とする。このときの三角形上のある点 P(x, y)は重心座標を用いて、

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

で表すことができる。ただし、点 P が三角形の内部にある条件は

$$0 < x < 1、0 < y < 1、0 < x + y < 1$$

である。

視線からの光線 $p(t) = e + td$ と三角形が交差する方程式は

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

となり、変形させると、

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

ここで $E_1 = B-A$, $E_2 = C-A$, $T = e-A$ とすると、クラメールの公式より

$$\begin{pmatrix} t \\ x \\ y \end{pmatrix} = \frac{1}{|-d \quad O1 \quad O2|}\begin{pmatrix} |T \quad E_1 \quad E_2| \\ |-d \quad T \quad E_2| \\ |-d \quad E_1 \quad T| \end{pmatrix}$$

ここで 3 次元のベクトル L, M, N を考えると、

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

なので、上式は以下のように書き換えることができる。

$$\begin{pmatrix} t \\ x \\ y \end{pmatrix} = \frac{1}{(d \times E2) \cdot O1}\begin{pmatrix} (T \times E_1) \cdot O2 \\ (d \times E_2) \cdot T \\ (T \times E_1) \cdot d \end{pmatrix}$$

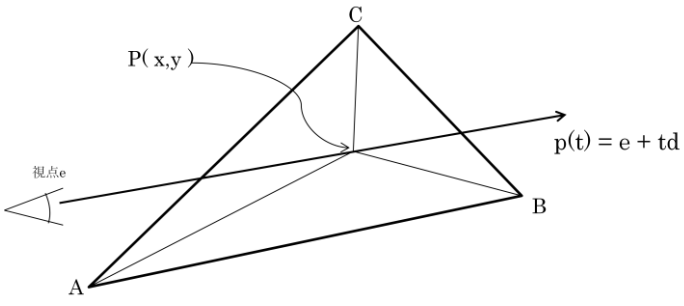


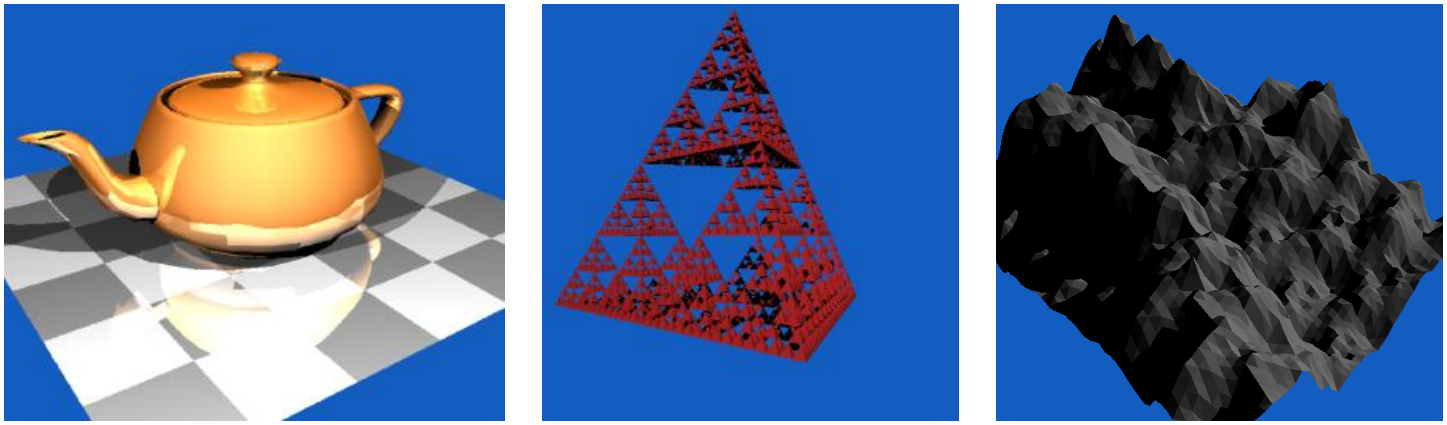
図 8　三角形との交差判定

3. ベンチマークシーン

3.1 レイトレーシング用標準画像：SPD

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。シーンデータは nff 形式のファイルとなっている。この nnf ファイルは以下の情報により構成されている。

・視点ベクトルの定義
・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
・多角形の定義



(a)teapot　　(b)tetra　　(c)mount

図 9　SPD を用いた生成例

3.2 ベンチマークシーンの追加

本研究は、今まで図 9 に表示した teapot、tetra、mount の 3 つのシーンデータで研究を行ってきた。また teapot に関しては実際の表示画像より光の照り返しが強く、うまく出力することができていなかった。そこで今回、teapot の正式な表示と gears の追加について行った。

teapot は 36 個の床を形成する四角形と 2256 個の多角形を形成する情報が nff ファイルに含まれている。表面が金属のような光沢をもっており、床の模様が反射して、ヤカンに映りこんでいる。

gears はギアの中身は空洞になっており、1 のギアが 144 個の上面と下面、144 個の突起の情報が全ギア 64 個分 nff ファイルに含まれている。そのため処理時間が膨大であり、CPU では、実行時間が約 7000 秒もかかってしまうため、処理の高速化が必要である。

4. CPU における実験

4.1 CPU 環境

OS：Windows 7 Ultimate 32bit

CPU(Hz)：Intel(R) Core(TM) i7 (3.07GHz)

メモリ：8.0GB

HDD：500GB

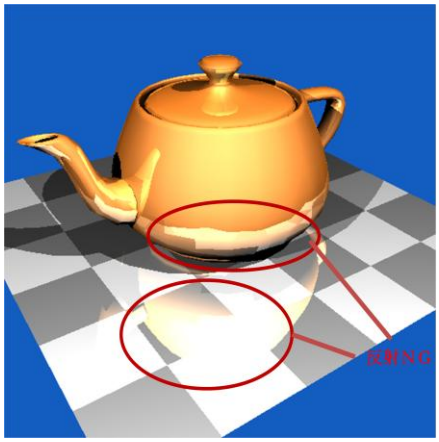
4.2 問題と解決方法

(1) teapot

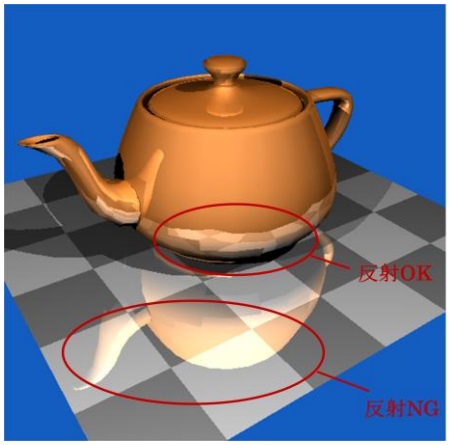
teapot は、以前より出力はされていたが、図 10(a)のように、光源の光の照り返しが強く、ヤカンの下部と床が正常に表示されていなかった。そこで反射の計算を行う calculate_reflection_ray 関数に減衰を与えることにした。鏡面反射係数:Ks>0 の場合、この関数内で、鏡面反射光を形成し、輝度値に足し合わせ色を決定する。ここで減衰を与えたところ図 10(b)の画像が出力された。ヤカンの下部はうまく出力できたが、床がまだ表示されていない。調べていくうちに、物体に一度反射し、その光が別の物体にあたり反射する際、光はさらに弱まらないといけないが、弱まることなく反射されていた。そこで拡散反射光、鏡面反射光、光源等の光を一つにまとめる関数である calculate_local_color 関数に、さらに減衰を与えたところ図 10(c)のように問題は解消された。

localColor = lightColor * (Kd * surfaceColor * diffuse + Ks * specular)

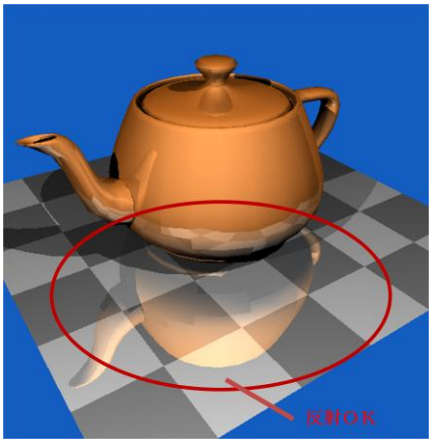
色の輝度値の生成は、上記の式で表すことができ、lightColor は光源、Kd・Ks はそれぞれ拡散反射係数・鏡面反射係数、diffuse・specular は拡散反射光・鏡面反射光、surfaceColor は物体の反射時の輝度値を示している。光は物体に反射すればするほど、減衰していくが、既存のプログラムでは減衰せずに一定の光量が保たれていた。そこでいくらかの減衰を与えた結果、0.78 倍することで正しい出力画像に近づけることができた。



(a) 初期図



(b) ヤカン下部修正後



(c) 床修正後

図 10　teapot 修正比較

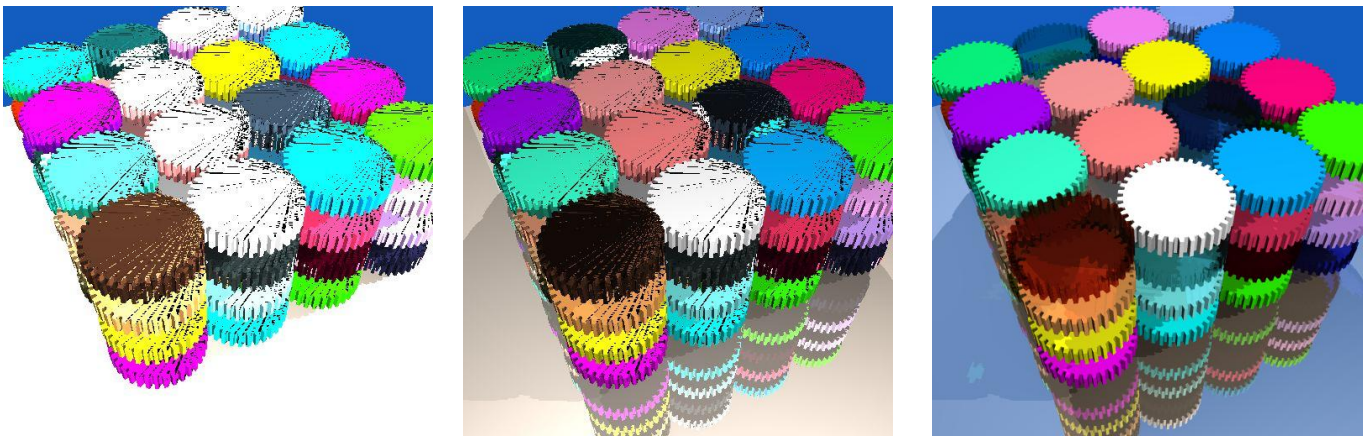
(2) gears

gears を生成させると図 11(a)のように出力された。図 11(c)はネットからの引用であるが、比較してみると、問題点としては、床が白く表示されている点と、ギアの一つ一つに黒い線がある点から放射線状に現れている点である。

まず床の表示だが、teapot 同様、反射が強すぎるのではないかと考え、減衰を与えることにした。すると図 11(b)のようにうまく表示させることができた。次にギアに現れている黒い線の修正だが、CPU では一回実行させ、画像を表示させるのに約 7000 秒もかかってしまうため、ギアの数を減らして修正を行う。出力画像を 512×512 に座標番号を付け、画素の表示が間違っている箇所の割り出しを行った。ここで色を生成する ray_color 関数に注目する。この関数では、

① 色の初期化
② 交差判定
③ 拡散反射光、鏡面反射光の計算
④ 全体の輝度値の計算
⑤ 色の確定

以上の５つのステップで色の輝度値を生成している。問題のある箇所では、④の計算では、前回の繰り返した計算結果が次の画素生成に使われているはずが、前回の値が保持されてなく、ミスが生じていた。これは初期化されていたためではないかと考え、調べている。



(a) 初期図　　(b) 床修正後　　(c) 正規出力画像

図 11　gears の修正比較

5. GPU による実験

5.1 GPU のアーキテクチャ

GPU（Graphics Processing Unit）とは、画像処理のためのハードウエアであり、具体的には画像処理用の LSI チップのことを指す。GPU は従来、グラフィックス処理のみを行うものであったが、近年では性能が向上し、汎用的な計算を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 Geforce GTX 480 である。図 12 に Geforce GTX 480 の構成を示す。Geforce GTX 480 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。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）を 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、64K バイトの共有メモリ/L1 キャッシュ、128K バイト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 スレッド分の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。SM は、32 のスレッドをグループ化したワープを単位に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。2 個のワープ・スケジューラは、2 つのワープを選択し、1 ワープあたり 1 つの命令を実行できる。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。図 13 に SM の構成を示す。Geforce GTX 480 には 16 個の SM があるが、NVIDIA 社の仕様により 1 個の SM を無効にしている。そのため、GPU 全体での SP 数は 32×15 となる。

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。

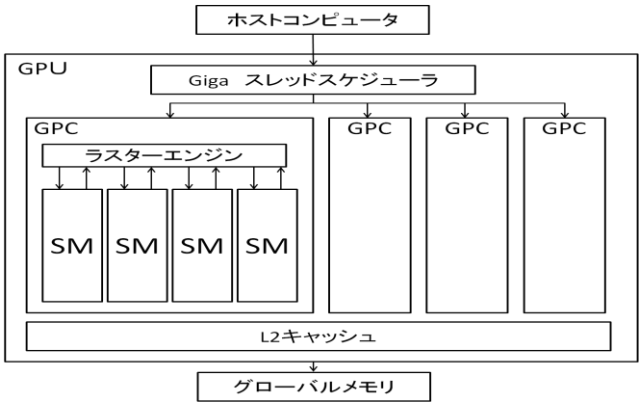


図 12　GTX480 の構成

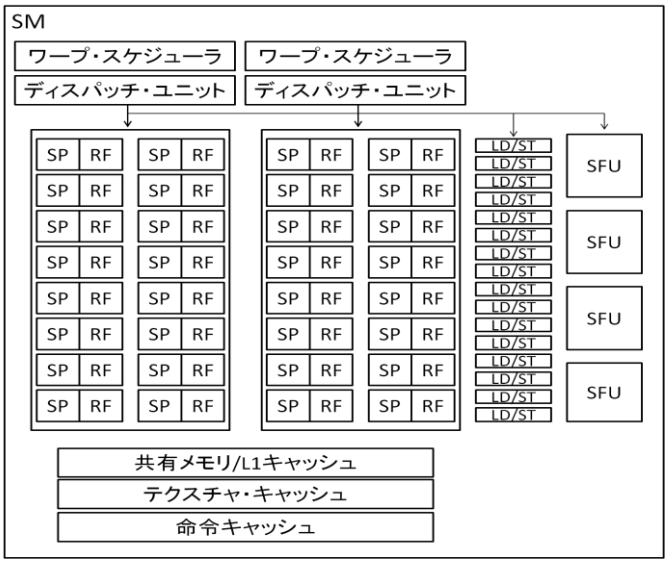


図 13　SM の構成

5.2　並列プログラミング環境

CUDA（Compute Undefined Device Architecture）とは、NVVIDIA 社が提供しているグラフィックチップを、汎用的なプログラミングを実行できるようにした総合開発環境である。CUDA でのプログラミングは C 及び C++がサポートされています。

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、メモリも搭載されている。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、CPU 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。

また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、カーネルプログ

ラムは、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。プログラムの一通りの流れは図 14 に示す。

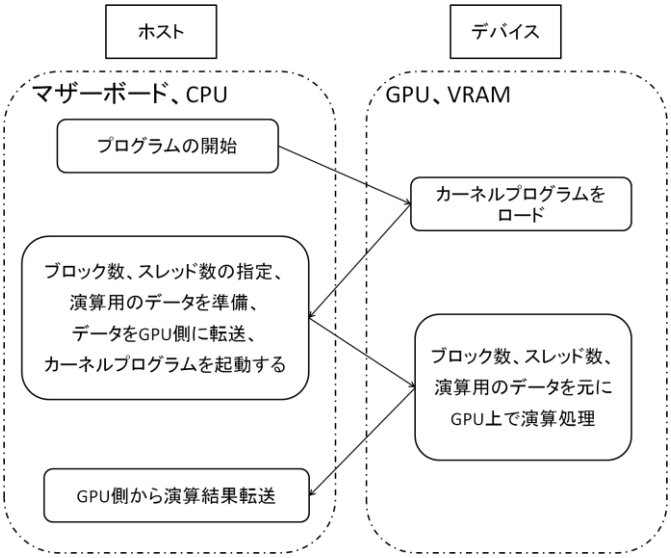


図 14　ホストとデバイスの処理の流れ

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。図 14 に示すように、これら 3 つは階層構造の関係にあり、上からグリッド、ブロック、スレッドの構造となっている。スレッドはスカラプロセッサ上で動作するプログラムの単位である。

5.3　CPU と GPU との比較

2 種類のシーンデータ（teapot, gears）に対して、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 CPU と GPU との比較を行った。

GPU の実験環境と、ブロック数・スレッド数は以下に示す。

OS：Windows 7 Ultimate 64bit
CPU(Hz)：Intel(R) Core(TM) i7 (3.07GHz)
メモリ：6.0GB
HDD：500GB
GPU：Geforce GTX480
メモリクロック：1.40GHz
並列処理環境：CUDA

- 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

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。各 SM ではストリーミング・プロセッサ（SP）を用いてワープ単位でスレッドを実行する。スクリーンをブロック分割することで、各スレッドが並列で演算処理を行い、レイトレーシングを高速化する。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。

(1) teapot

CPU での実行時間は 170.6020 秒である。

表 1　teapot 実行時間

ブロック数,スレッド数	実行時間（秒）	速度向上比（倍）
256*256, 2*2	13.709	12
128*128, 4*4	3.557	48
64*64, 8*8	1.540	110
32*32, 16*16	1.904	90
16*16, 32*32	2.397	71

(2) gears

CPU での実行時間は 7904.4870 秒である。

表 2　gears 実行時間

ブロック数,スレッド数	実行時間（秒）	速度向上比（倍）
256*256, 2*2	81.459	97
128*128, 4*4	20.915	378
64*64, 8*8	9.573	826
32*32, 16*16	9.704	815
16*16, 32*32	10.573	748

表 1、表 2 は teapot、gears の実行時間と CPU との速度向上比を示す。２つ共に、ブロック数 64*64・スレッド数 8*8 の場合、実行時間が短く、速度向上比が多く得られた。これはスレッド数が 8×8=64 でワープ数 32 の 2 倍になっていることから、効率よく処理できたためと思われる。

6. おわりに

本研究では、CPU と GPU を用いて、レイトレーシング法によるベンチマークシーンの表示と、画面分割の並列化による高速化について述べた。画面分割の並列化については、レイトレーシングの一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て、ストリーミング・マルチプロセッサによるブロック分割を行った。

実験では、Geforce GTX 480 を用い、SPD の 2 つのシーンデータ（teapot、gears）につて、正しい画像出力を得るための修正と、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。正しいベンチマークシーンの表示では、画像生成時に色の輝度値を生成する際の反射計算が間違っていたために、正しく表示されていなかった。これは減衰を与えることで解消することができた。実行時間については、teapot ではブロック数 64*64、スレッド数 8*8 のときに、実行時間 1.54 秒で、CPU 1 個に比べて 110 倍の速度向上を得ることが確認できた。また gears ではブロック数 64*64、スレッド数 8*8 のときに、実行時間 9.573 秒で、CPU 1 個に比べて 826 倍の速度向上を得ることが確認できた。これはスレッド数が 8×8=64 でワープ数 32 の 2 倍になっていることから、効率よく処理できたため、ブロック数 64*64、スレッド数 8*8 のときが 1 番よくなっていると思われる。

本研究の最終目的は、正しいベンチマークシーンの画像生成の高速化である。そのためには現段階では行っていない適応型空間分割、スクリーンマッピング等の適用が必要である。今後の課題としては、まだ作成していない円柱・円錐等の他の形状のシーンデータの実験、適応型空間分割・スクリーンマッピング等の実現である。

謝辞

本研究の機会を与えてくださり、ご指導いただきました山崎勝弘教授、孟林助手に深く感謝します。また、本研究に関して貴重な助言、ご意見をいただきました、岡崎大輔氏、山田遼氏、増田匠吾氏、高性能計算研究室の皆様に心より感謝いたします。

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： 3 次元 CG の基礎と応用, サイエンス社, 2002.

[3] Eric Haines et al：Standard Procedural Databases, http://www.csee.umbc.edu/~olano/635s03/spd.html

[4] 上野謙二郎：GPU を用いたリアルタイムレイトレーシングの並列化の検討と実現,立命館大学理工学研究科修士論文, 創造理工学専攻, 2012

[5] Jason Sanders, Edward Kandrot 著, 株式会社クイーブ訳：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.

[6] 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, IZB-1, 2012.

[7] 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.

[8] 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.

[9] 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大会 D-05, 2012.

[10] 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, 2012.

[11] 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卒業論文, 2012.

[12] 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卒業論文, 2012.