大規模データに対するレイトレーシング

平成 14 年度

中丸 幸治

要 旨

レイトレーシングは強力かつ汎用的なレンダリング手法として知られる．これは非常に長い計算時間を必要とすることでも知られていたが, この問題についてはこれまでの研究で大きく改善された．また, 最近の改良により, 他の手法では扱うことが難しい複雑な光学現象をとらえることも可能となっている．

しかしながら, 通常のレイトレーシングでは, シーンのデータベースへのアクセスについてのコストが考えられておらず, 大量のデータを含むシーンはスラッシングを引き起こす可能性がある．一方, 最近のモデリングの進歩により, 非常に複雑なシーンを作ることや現実の詳細な形状を取り込むことが可能となっており, 要求されるデータ量は急速に増大している．通常のレイトレーシングはこうした要求に応えることができない．

本論文では, 大規模データに対するレイトレーシングを実現するためのアルゴリズムを提唱する．このアルゴリズムは幅優先レイトレーシングに基づくもので, 複数の光線をまとめて扱い, これらと各物体との比較をすることで処理を行なう．幅優先レイトレーシングの概念は以前に提案されたものであるが, その効率的なアルゴリズムは知られていなかった．本研究では幅優先レイトレーシングと様々な手法を組み合わせることにより, 効率のよいアルゴリズムを導出した．これらの手法には, ‘一様空間分割’, ‘バウンディングボックス階層’, ディスク上のデータへのアクセスを最小化するための新しい機構などが含まれる．このアルゴリズムはディスク上のデータへの逐次的なアクセスを常に維持するので, スラッシングが生じることはない．実験により, 提案アルゴリズムが任意の大きさのデータを効率よく扱えることが示された．本アルゴリズムにより, 任意の複雑さを持つシーンをレイトレーシングによってレンダリングすることが可能となる．

Abstract

Ray tracing is known to be a powerful, versatile rendering method. It is also known to require lengthy computation time, which has been greatly reduced by previous authors. Recent improvements also enable to efficiently capture complicated optical effects, which are difficult to handle with other methods.

There is, however, one serious problem in traditional ray tracing; the cost to access a scene database is not considered, so that accessing a scene with massive data may cause thrashing. Recent progress in modelling enables to create highly complex scenes and to aquire detailed geometry from the real world, thus the amount of data that users want to render increases rapidly. Traditional ray tracing does not fit for such requirements.

In this paper, we propose an efficient algorithm to realize ray tracing for massive data. The algorithm is based on breadth-first ray tracing, which forms a set of rays and compares each object in turn against this set. The concept of breadth-first ray tracing itself was previously proposed, but its efficient algorithm has not been known. We achieve the efficient algorithm by combining breadth-first ray tracing with various techniques, such as 'uniform spatial subdivision,' 'bounding box hierarchy,' and other new schemes to minimize accessing data on disk. The algorithm always keeps sequential access for data on disk, so that no thrashing occurs. Experimental analysis shows that the algorithm can efficiently handle any size of data. The algorithm makes it possible to render any complex scene by ray tracing.

謝 辞

本研究では多くの方々からのご助力を賜りました. ここにその方々の名前を挙げて, 感謝の意を表したいと思います.

まず, 大野義夫教授に心から感謝いたします. 大野先生は, 著者が学部生であった頃から, コンピュータグラフィックスを始めとするさまざまな事柄について教えてくださいました. そして, なによりも本研究の長く困難な道程を辛抱強く支えてくださいました.

また, 本論文を執筆するにあたり, 慶應義塾大学理工学部 萩原将文教授, 山本喜一助教授, 野寺隆助教授よりご意見, ご指導を賜りました. さまざまな分野を専門とされる方々によるご意見, ご指導は, 大変貴重なものでした. 厚く御礼を申し上げます.

最後に, 著者の我儘を許してくれ, 研究活動を影で支えてくれた父 和登, 母 美智子に深く感謝いたします.

平成 14 年 8 月

中丸 幸治

目 次

表 一 覧

図一覧

アルゴリズム一覧

第 1 章

序 論

本論文の目的は, 大規模データに対するレイトレーシングを実現することである. レイトレーシングは今日知られている手法の中で最も高い一般性を備え, 他の手法では扱うことが困難な状況も統一的に扱うことができる. 初期の主要な問題であった速度については, 特に 1980 年代に大きな改善がなされ, 今日では制限つきながらリアルタイムレイトレーシングの研究も行なわれている [Wald01]. また, さまざまな光学現象については, 1980 年代中期から 1990 年代を通して研究が行なわれ, 今日では多重散乱による間接光やレンズの焦光などの複雑な現象も扱うことができるようになった [Veac97, Jens98].

こうした状況にも拘わらず, 従来のレイトレーシングには重大な問題が存在する. それは, メモリに格納できないような大規模データをうまく処理できないことである [Cook87]. 従来のアルゴリズムはすべてのシーンデータをいつでも参照できることを仮定しており, 大規模データに対してはスラッシングが起こる可能性がある. この問題は 1987 年に初めて指摘されたが, 並列計算機など特殊な状況下以外では, 今日にいたるまで数える程の研究しかない. レイトレーシング以外の手法では, 異常ともいえるような大規模データさえ安定かつ高速に処理可能な手法が存在するが, レイトレーシングについていえば, そうした手法はなお存在しないのが現状である.

本研究の目的は, 大規模データに対するレイトレーシングを実現するための頑健かつ高速なアルゴリズムを構築することである. ここで, まず頑健であるという点が重要である. ちょうどレイトレーシングが他の手法で扱えない光学現象を扱えるのと同じように, 大規模データを扱えることと扱えないこと, 可能と不可能の差はきわめて大きい. このような頑健性を確立した上で可能な限り高速なアルゴリズムを得ることが, 最終的な目的である.

以下では, まず, レイトレーシングを含むレンダリング手法について概観する. 次に, 従来のレイトレーシングの高速化手法について述べる. そして, これらの知識を前提として, 扱われる問題と本論文の寄与を詳細に述べる. 最後に, 本論文で用いられるテスト用のデータについて簡単に解説し, 本論文の構成を述べる.

1.1 レンダリング

コンピュータグラフィックスは大きくモデリングとレンダリングに分けられる. 簡単にいうなら, モデリングは形を定義することであり, レンダリングはその定義に基づいて画像を生成することである. コンピュータグラフィックスの研究は 1950 年代に始まり, 当初, 3 次元画像のレンダリングにおいてはどのように線画を得るかに焦点があてられていた. その後, CRT ディスプレイの描画方式 (電子線をどのように走査するか) がベクトル型からラスタ型に移行するにつれ, 面画, つまり, 各ピクセルごとの可視面を決定し, 輝度を計算するアルゴリズムの研究が盛んになった.

線画であるか面画であるかを問わず, レイトレーシング以外のほとんどのレンダリング手法では, 次のような過程でレンダリングを行なう.

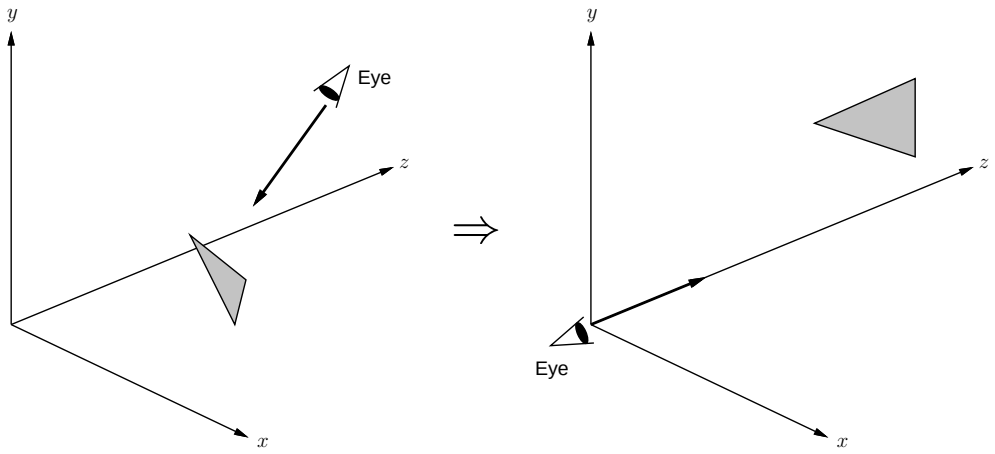
(1) 視野変換/透視投影
(2) 隠面除去/シェーディング

(1) の処理はスクリーンから見たとき, 物体の各面がどの位置と大きさで見えるかを決定する変換である. この変換により, 各面が見えるかどうかという問題は, スクリーン上での奥行きの比較に還元される. こうして, 元々は 3 次元の問題を 2 次元的な (スクリーン上とその奥行きの) 問題に簡略化した後, 実際に問題を解くのが (2) の処理である. (2) の処理のうち, 隠面除去は各ピクセルごとにどの面が見えるかを決定すること, シェーディングはその面の輝度を決定することを指す. これらのうち, 特に隠面除去の部分が手法間の主な違いであり, それぞれの手法を特徴付けている. 一方, シェーディングでは面の材質や面の上に貼られたテクスチャを参照しながら輝度を求めるが, この処理はどの手法においても同様である.

現在, 用いられているレンダリング手法として代表的なものとしては, スキャンライン法, Z ソート, Z バッファ, REYES, そしてレイトレーシングが挙げられる. 以下ではこれらについて, 本研究で扱うメモリ量の問題に触れながら述べる. なお, 簡単のため, ほとんどの部分でポリゴンのみを対象とした説明を行なうが, 実際には例えば曲面を扱う技術も含まれている.

1.1.1 視野変換/透視投影

レイトレーシング以外のレンダリング手法では, 隠面消去やシェーディングの前段階の処理として, 視野変換および透視投影を必要とする. そこで, 各手法について述べる前に, これらの処理について簡単に述べておく.



視線が z 軸に沿うように物体を移動する.

図 1.1 視野変換

まず, 視点を原点に置き, 視線を z 軸方向にとることを考える. 視野変換とは, 任意の視点および視線が与えられたとき, このように視点および視線を座標系に沿うようにする変換である (**図 1.1**). この変換は, 具体的には物体を回転および平行移動する変換になる.

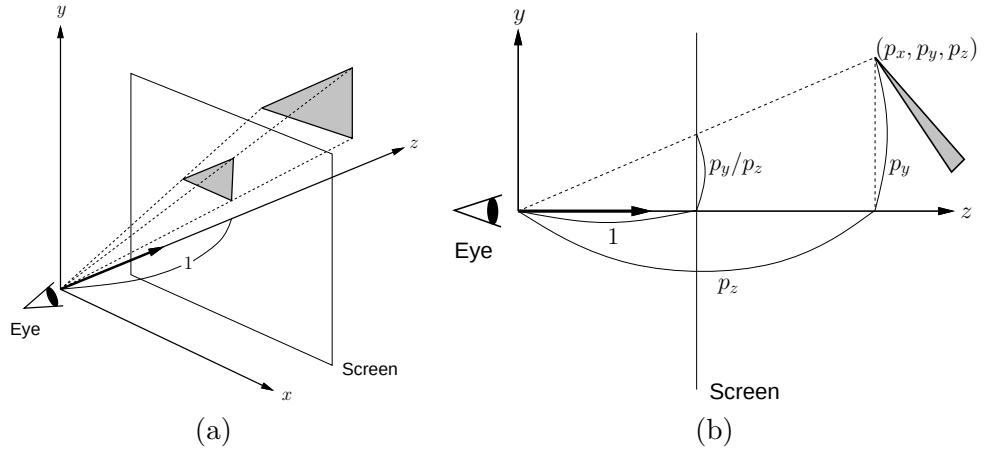
次に, 視野変換後の三角形ポリゴンを, $z = 1$ に置いた z 軸に垂直なスクリーンに写し取ることを考えよう (**図 1.2**-a). ある頂点の座標 (x, y, z) に対し, スクリーン上の座標 (x_p, y_p) は, 頂点と原点を結ぶ直線とスクリーンとの交点となるので (**図 1.2**-b), 次のようにして求めることができる. このような変換が透視投影である.

$$x_p = x/z,$$
$$y_p = y/z.$$

実際には透視投影の後, 隠面除去を行なうためには奥行きの情報が必要である. (x_p, y_p, z) を座標とする空間を考えると, 元の 3 次元空間で直線であったものが曲線に, 平面が曲面に変換されることになる. これに対し, もし直線が直線に変換されるようにすることができれば, ポリゴンの頂点のみを透視投影し, ポリゴン内では線形内挿によって奥行きを求めることができる. そこで, 透視投影では奥行き z_p を次のように定める.

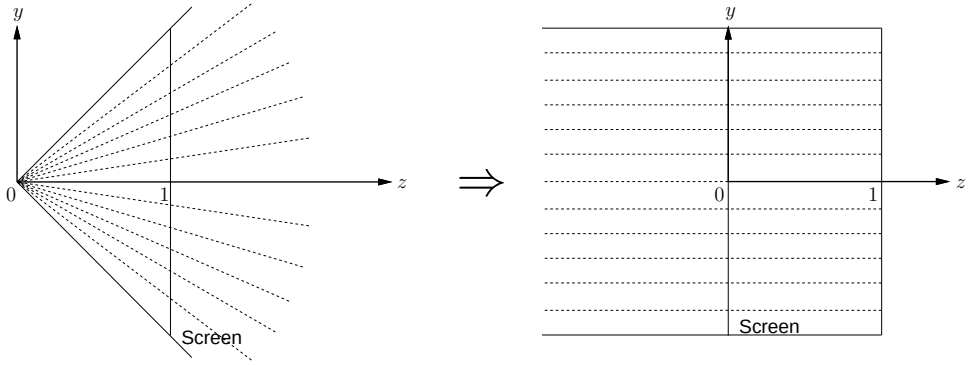
$$z_p = 1 - 1/z.$$

この変換により, $z = \infty$ は $z_p = 1$ に, スクリーンは $z_p = 0$ に, 視点は $z_p = -\infty$ に移り, 視点から各ピクセルを通る直線は z 軸に平行な直線になる (**図 1.3**). こうして, 物体の前後関係の問題が x_p–y_p 平面上での重なりとピクセルごとの z_p 値の大小という, より単純な問題に置き換わる.



(a) スクリーンへの投影. (b) 3 次元座標と投影座標の関係.

図 1.2 透視投影



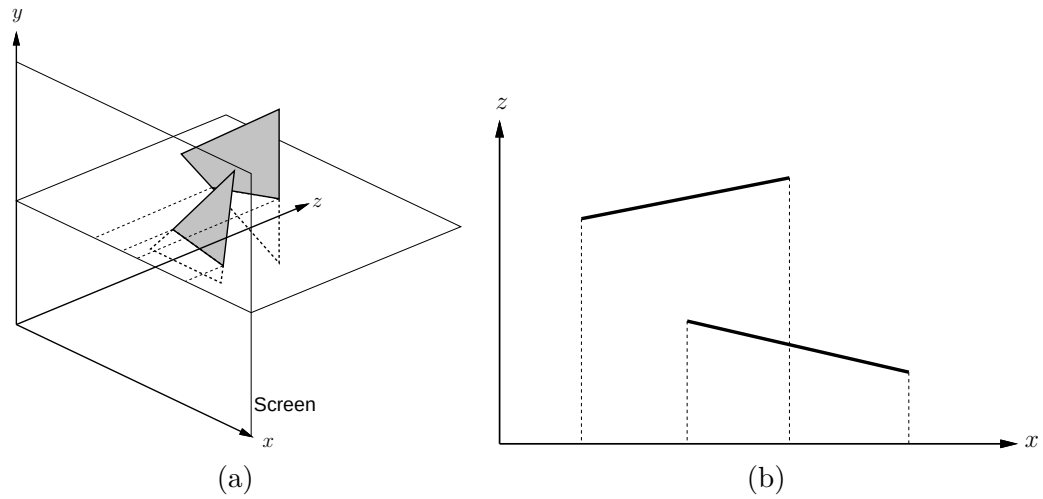
放射状の視線は z 軸に平行な半直線に変わる.

図 1.3 透視投影による空間の変化

1.1.2 スキャンライン法

スキャンライン法は, 実際には特定の手法というよりはスキャンライン単位に画像を生成する手法全般を指す.[*1] この手法では, スキャンライン平面と呼ばれるスキャンラインを通る x–z 平面に平行な平面を考え, この平面とポリゴンの交線を求める. そして, 各交線の前後関係を決定することで 1 本のスキャンライン上の隠面除去を行なう (**図 1.4**). つまり, 視野変換/透視投影によって単純化した問題を, さらにスキャンラインごとの問題に

[*1] 後述の Z バッファなども含んだ形で, レイトレーシングによらない手法をスキャンライン法と呼ぶこともあるが, より正確な定義はここに示したものである.



(a) スキャンライン平面. (b) スキャンライン平面上の線分を x 軸へ投影し, 前後関係を決定する.

図 1.4 スキャンライン法

分解して解く. このアルゴリズムの概要を**アルゴリズム 1.1**に示す. なお, アルゴリズム中の 'AET' は Active Edge Table の略で, 現在のスキャンラインと交差している辺を保持する.

アルゴリズム 1.1 スキャンライン法

```
AET を初期化
for all スキャンライン do
  AET を更新
  for all AET 中の辺 do
    スキャンラインとポリゴンとの交差線分を求める
  od
  交差線分の前後関係を求める
  for all 交差線分の可視となっている部分のピクセル do
    シェーディング
    ピクセルを描画
  od
od
```

ポリゴンをピクセル列に変換する処理を 'スキャンコンバージョン' と呼ぶが, この処理を複数のポリゴンに対して同期的に行なうようにしたのがスキャンライン法といえる. AET には現在のスキャンラインと交差するすべての辺が保持されるため, 状況によってはすべてのポリゴンに関連する辺が保持される可能性がある. したがって, 通常のスキャンライン法では, すべてのポリゴンをメモリに保持しなければならない. 一方, ポリゴンのデータ以外のために必要なメモリ領域はわずかであるため, ポリゴンの総数が少なければ

メモリの消費量は小さくなる.

1.1.3 Z ソート

視点から見て遠いポリゴンから順に描き, これまでに描いた画像に塗り重ねていけば, 最終的に隠面除去がなされた絵が得られる. このように絵描きと同様の処理をするのが Z ソートである. これもスキャンライン法と同様, 手法全般を指す名称である. このアルゴリズムの概要を**アルゴリズム 1.2**に示す.

アルゴリズム 1.2 Z ソート

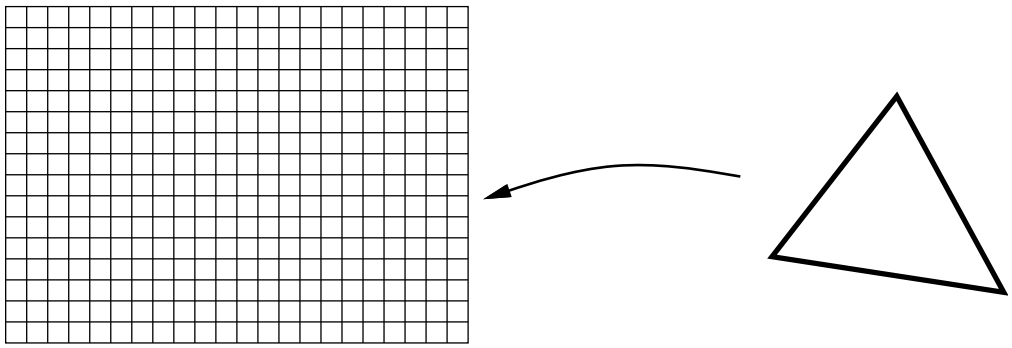
```
ポリゴンを奥行きによってソートする
for all ポリゴン do
  ポリゴンをスキャンコンバージョン
  for all ポリゴン上のピクセル do
    シェーディング
    ピクセルを描画
  od
od
```

どのように奥行き, つまりポリゴンの優先度を決めるかでさまざまな手法がある. 比較的よく用いられているのは, 単にポリゴンの重心の z_p 値など, 代表的な値を 1 つ定めてソートを行なうものと, BSP (Binary Space Partition) によるものである. 前者は必ずしも正しい結果を得ることはできないが, 高速な処理が可能であり, ハードウェアのサポートが弱かった時期にはゲームなどでよく用いられた. 一方, BSP は空間を平面で再帰的に分割する構造であるが, これを用いると, ある視点から見たときの各部分空間の前後関係を簡単に決定できる. 与えられたシーンデータに対して BSP を構成するにはある程度の時間が必要なため, 多数の物体が動く場合には適さない. しかし, 迷路を動き回るようなゲームや建築物のウォークスルーなどにはよく用いられている.

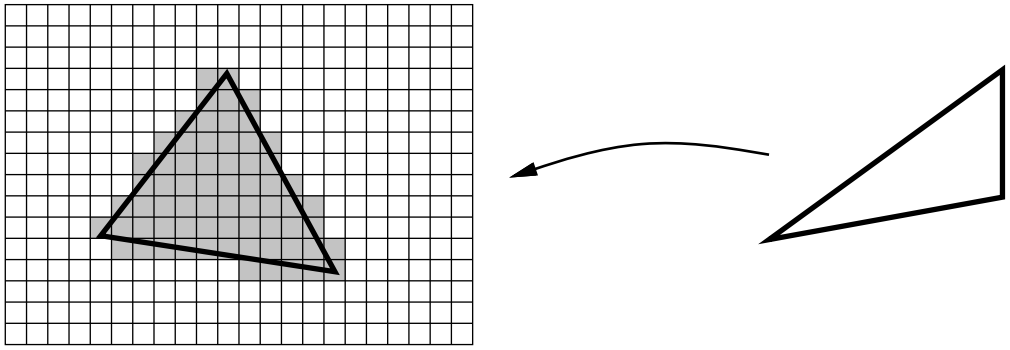
Z ソートはメモリ利用の観点から見ると, スキャンライン法と同様といえる. つまり, ポリゴンすべてや BSP (ポリゴン数に比例するメモリを消費する) を保持しなければならないが, ポリゴンの総数が少なければメモリの消費量は小さくなる.

1.1.4 Z バッファ

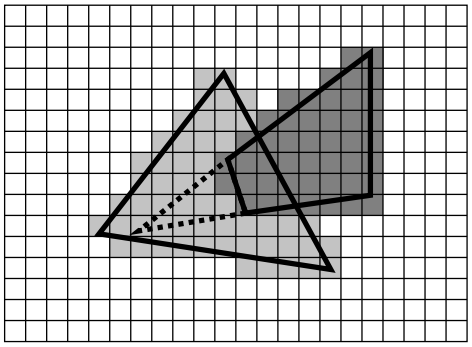
Z バッファは Catmull によって 1974 年に最初に提唱された [Catm74]. この手法はスキャンコンバージョンを行なう点は共通しているが, これまでの手法とはある点で大きく異なっている. アルゴリズムの概要を**アルゴリズム 1.3**に, 具体的な動作例を**図 1.5**に示す. ここで, アルゴリズム中の Z バッファは各ピクセルの z_p 値を保持する 2 次元配列である.



(a)



(b)



(c)

(a) 各ピクセルごとの距離を無限大に初期化する．この状態で最初のポリゴンを描き, ポリゴンと重なるピクセルの z_p 値を更新する．(b) 次のポリゴンを描く．今度は先に描いたポリゴンによって z_p 値が更新されているピクセルがあるため, ポリゴンと重なるピクセルの一部を更新することになる．(c) 最終結果.

図 1.5 Z バッファ

アルゴリズム 1.3 Z バッファ

```
Z バッファを無限遠に初期化
for all ポリゴン do
  ポリゴンをスキャンコンバージョン
  for all ポリゴン上のピクセル do
    if ポリゴンの z_p 値 < Z バッファ中の z_p 値 then
      シェーディング
      ピクセルを描画
      Z バッファをポリゴンの z_p 値で更新
    fi
  od
od
```

隠面除去はピクセルごとに ‘一番手前の面’, つまり, ‘一番小さな z_p 値’ を求める処理である. したがって, 各ピクセルごとの z_p 値を 1 つだけ保持すれば, どのような順序でポリゴンを処理しても, 最終的に一番手前の面が塗られた画像を得ることができる. これが Z バッファの原理である.

アルゴリズム 1.3では**アルゴリズム 1.2**にあったソートの処理がない. ポリゴンは単に 1 つずつ逐次的に処理されるのみである. このように Z バッファは複数のポリゴンを同時に処理したり並び替えたりする必要がないため, 多数のポリゴンを保持するためのメモリ領域を必要としない. z_p 値の格納に必要なメモリ量はスクリーンの解像度に比例して増加するが, この解像度はシーンデータのサイズと相関を持たない. このように限られたメモリしか必要としないため, ハードウェア化にも向いており, PC などに搭載されている 3D アクセラレータのほとんどは Z バッファを採用している.

1.1.5 REYES

映画 ‘スターウォーズ’ で有名な George Lucas は, 1980 年代前半に著名な研究者を集め, 映画製作に利用可能な高度なコンピュータグラフィックスの研究を行なった. フィルムのデジタル処理や自然物のモデリングなど多岐にわたる研究がなされたが, レンダリングについての成果がこの REYES である [Cook87]. この手法は 1987 年に Cook らによって発表され, それを実装した PIXAR 社の PRMan (Photo-realistic RenderMan) は, 今日でもハリウッド映画の製作で用いられるレンダラの事実上の標準となっている.

すでに 1980 年代前半には初期に必要とされたレンダリングの技術はほぼ確立していたが, Cook らはこれまでの技術と映画製作における要求を改めて入念に分析し, この手法を設計した. REYES は Render-Everything-You-Ever-Saw の略であるが, この名前に目指す意図が表れているといえる. アルゴリズムの概要を**アルゴリズム 1.4**に示す. なお, アルゴリズム中の ‘バウンド’ は物体などを覆う単純な形状 (たとえばバウンディングボック

ス) を, ‘ダイス’ は物体をマイクロポリゴンと呼ばれる微小なポリゴン集合に変換する処理を指す.

アルゴリズム 1.4 REYES

```
Z バッファを無限遠に初期化
for all 物体 do
  物体をファイルから読み込む
  if 物体のバウンドを計算可能 then
    バウンドを求める
    if バウンドが視野の外 then
      この物体の処理を終了
    fi
  fi
  if 物体がダイス可能か then
    物体をマイクロポリゴンへ変換
    マイクロポリゴンに対するシェーディングを行なう
    for all マイクロポリゴン do
      if マイクロポリゴンが視野の中 then
        マイクロポリゴンのバウンドを求める
        for all バウンドと重なるピクセル do
          if マイクロポリゴンの z_p 値 < Z バッファ中の z_p 値 then
            ピクセルを描画
            Z バッファをポリゴンの z_p 値で更新
          fi
        od
      fi
    od
  else
    物体を別の複数の物体に分割
    分割結果をファイルの未処理部分の先頭に追加
  fi
od
```

多様な形状の扱いなどを重視しているため, 異なる部分もあり, 複雑にもなっているが, REYES の基本的な構造は Z バッファのそれと同様である. つまり, 物体は逐次的に 1 つずつ処理され, 限られたメモリしか必要としない.

1.1.6 レイトレーシング

これまでに見てきたアルゴリズムは, いずれも 3 次元の問題を 2 次元的な問題に還元して処理を行なう. これは問題を簡略化し, 各種のコヒーレンスを利用した高速な処理を可能とする. たとえばポリゴンをスキャンコンバージョンする際, 隣接するピクセルにおける z_p 値などは, 簡単な増分計算で算出できる.

一方, これらの手法は, 投影によってうまく扱えない場合, つまり, 光の複雑な反射や屈

折などを含む場合には適用できない. 光の散乱は, 本来は任意の方向に起こりうるため, 視点に到達する光のように整然と連続的に並んでいる——投影によって扱える——ということはむしろ稀である. こうした現象を近似的に表現するさまざまな方法も開発されているが, これまでに述べた手法で完全に扱うことは極めて難しい.

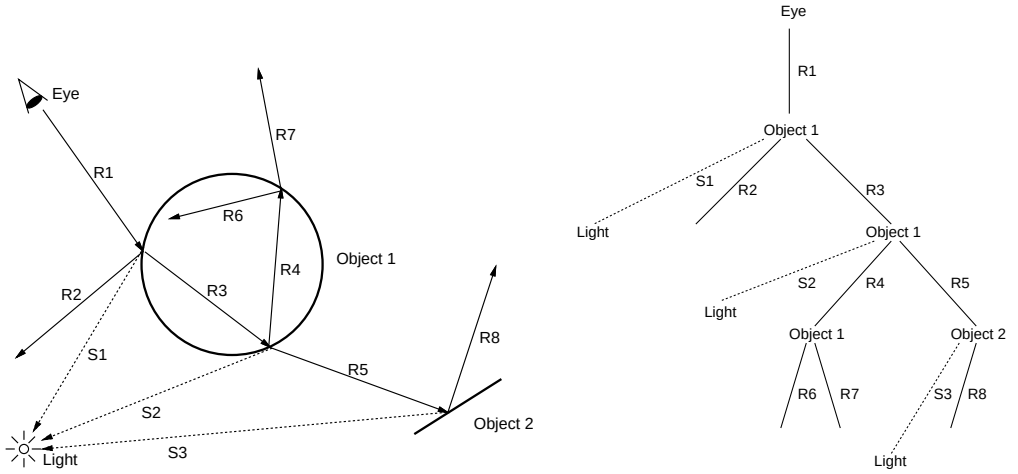
実は, スクリーンへの投影ということにこだわらなければ, もっと単純なアルゴリズムが考えられる. つまり, 幾何光学の考えに基づいて視点から各ピクセルを通る光線を考え, この光線を 1 本ずつ辿っていくのである. 物体表面で反射や屈折が生じている場合でも, 鏡面反射のように入射光に対して一意に方向が定まる場合であれば, 光がどのように反射または屈折してきたかを決定することも容易に行なえる. このような考えに基づき, 1980 年に Turner Whitted が提唱したのがレイトレーシングである. アルゴリズムの概要を**アルゴリズム 1.5**に示す.

アルゴリズム 1.5 レイトレーシング

```
for all ピクセル do
  { 隠面除去 }
  視点からピクセルを通る光線を求める
  光線に対する交点情報を初期化する
  for all 物体 do
    光線と物体の交点計算を行なう
    if 交点が存在 and 交点の距離 < 交点情報中の交点の距離 then
      交点情報を更新
    fi
  od
  { シェーディング }
  if 交点情報に交点が含まれる then
    交点での直接光による寄与をピクセルの輝度に加える
    if 交点で反射/屈折が生じている then
      反射/屈折方向の光線を求める
      これまでと同様の処理を再帰的に行なう
      反射/屈折方向の光線からの寄与をピクセルの輝度に加える
    fi
  fi
  ピクセルを描画
od
```

反射や屈折などが起きる面で再帰的な追跡が行なわれるため, 光線は交点で次々と分岐していく. このため, 全体の過程は光線木とよばれる木としてとらえられる (**図 1.6**). 実際にはこうした木がピクセルごとにあるが, **アルゴリズム 1.5**はこれらの木をまとめた全体の木を深さ優先で処理していることになる.

レイトレーシングによって複雑な光学現象を扱えるようになったが, この手法には遅いという難点があった. これは個々の光線を独立に処理するため, 他の手法のようにコヒーレンスを利用した高速な処理が行なえないためである. そのため, 開発された当初から高



レイトレーシングの過程は木として表現できる.

図 1.6 光線木

速化のための研究が数多く行なわれてきた. これらについては後述する.

各ピクセルごとにすべての物体を参照する処理となるため, 必要なメモリについてはスキャンライン法と似た傾向を持つ. つまり, シーンデータ全体の大きさに応じたメモリが必要になる一方, それ以外に必要となるメモリはわずかであるため, 物体の総数が少なければメモリをほとんど消費しない.

1.1.7 即時モードと保持モード

ここでは, 即時 (Immediate) モードと保持 (Retained) モードという 2 つの言葉について説明する. これらの言葉はシーンデータの処理形態を示す言葉で, 各物体を 1 つずつ処理しては破棄していくのが即時モード, シーンデータをすべて保持して処理を進めるのが保持モードである. これまで各手法の説明で必要とされるメモリ量について述べたが, この分類によると, Z バッファ, REYES だけが即時モード, それ以外のものは保持モードの手法といえる. 大規模データの処理について考えるときには, この即時/保持モードの区別が重要である.

1.2 レイトレーシングの高速化手法

レイトレーシングの高速化手法は今日でも研究されているが, 主流となっている手法の大半は 1980 年代後半に提案された [Arvo89]. レイトレーシングで最も時間がかかるのは交点計算なので, 研究はこれを如何に速くするかに集中している. 特によく用いられるのはバウンディングボリューム階層と空間分割の手法である. これらの概要を次に述べる.

1.2.1 バウンディングボリューム階層

バウンディングボリューム階層は Rubin および Whitted が最初に導入した最も古い高速化手法である [Rubi80]. この手法では, まず各物体を覆うバウンディングボリュームを定義する. ここで, バウンディングボリュームとは, 球や軸に沿った直方体 (Axis-Aligned Box) のように光線との交差判定が容易な形状である. そして, 交点計算の際, 光線とバウンディングボリュームが交差する場合のみ, 中に含まれる物体との実際の交点計算を行なう. 物体に対する交点計算は, 多くの場合バウンディングボリュームに対するそれよりも高価なので, バウンディングボリュームに対する交差判定のオーバヘッドを考慮した上でも, なお計算時間を短縮することができる.

このように単にバウンディングボリュームで各物体を覆うだけでも相応の効果を得ることができるが, 計算の効率は物体数 n に対して $O(n)$ のままである. これに対し, バウンディングボリューム階層では, さらに複数のバウンディングボリュームを覆う, より大きなバウンディングボリュームを考えていくことで, バウンディングボリュームの階層を構築する. そして, 交点計算の際には, このような階層をルートのバウンディングボリュームから処理していく. こうして計算の効率を $O(n)$ から $O(\log n)$ に変えることができる. この交点計算の概要を**アルゴリズム 1.6**, 概念図を**図 1.7**に示す.

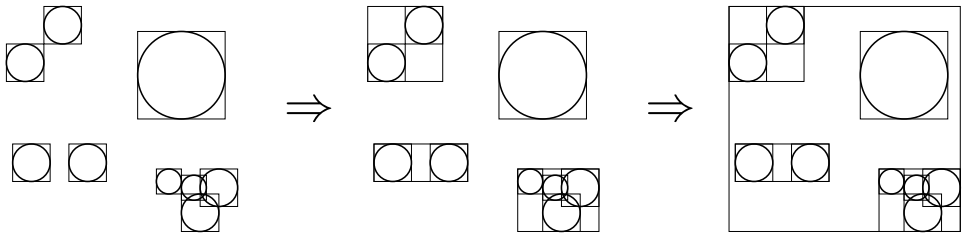
アルゴリズム 1.6 バウンディングボリューム階層による交点計算

```
procedure 交点計算 (光線, ノード)
  if ノードが葉 then
    物体との交点計算を行なう
  elif ノードのバウンディングボリュームと光線が交差 then
    for all 子供のノード do
      交点計算 (光線, 子供のノード)
    od
  fi
end
```

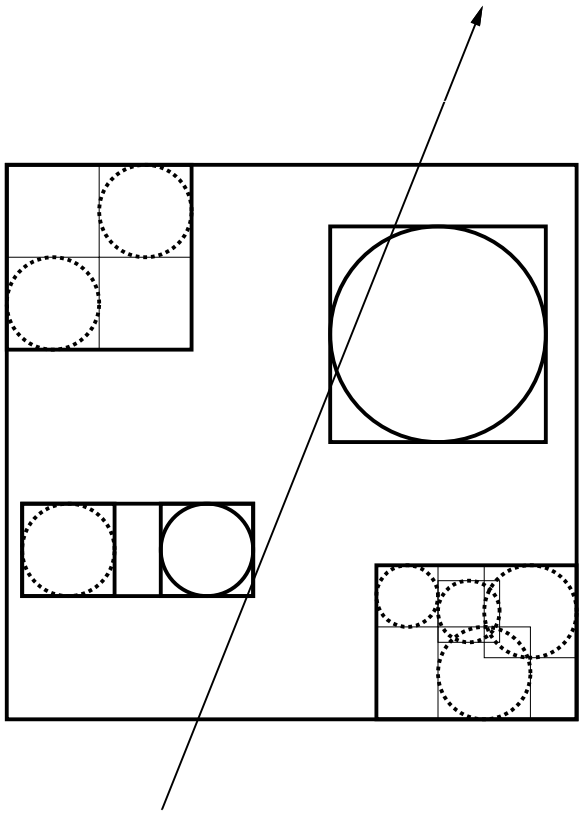
もちろん, この効率が実際に得られるかどうかは階層の構成に大きく依存する. Rubin らは人手によって階層を構築したが, 今日では Goldsmith らや Kay らの自動化された手法を利用することができる [Gold87, Kay86].

1.2.2 空間分割

バウンディングボリューム階層における階層の構築技術がまだ未熟であったころ, 別の高速化手法として登場したのが空間分割である [Glas84, Fuji86]. この手法では, まず空間を一様または非一様に分割し, 個々の小領域, つまりボクセル (ピクセルのアナロジーから



(a)



(b)

(a) 前処理. バウンディングボリュームの階層を構築する. (b) 交点計算. 太い実線で描かれたバウンディングボリュームおよび物体のみと交点計算を行なう.

図 1.7 高速化手法 (バウンディングボリューム階層)

このように呼ばれる) に重なる物体を登録しておく. この構造に対して光線が与えられたとき, 光線が通過するボクセルのみを光線の始点に近いものから調べ, 物体があれば交点計算を行なう. このようにすれば, 光線と関わらない多数の物体を除外することができる. これが空間分割の基本的な考え方である. この交点計算の概要を**アルゴリズム 1.7**, 概念図を**図 1.8**に示す. なお, ここで示しているのは一様空間分割に対するものだが, 非一様な分割でも考え方は同様である.

アルゴリズム 1.7 空間分割による交点計算

```
procedure 交点計算 (光線)
  光線の始点を含むボクセルを決定する
  3DDDA を初期化する
  repeat
    for all ボクセル中の物体 do
      物体との交点計算を行なう
    od
    if 交点が見つかっていない then
      3DDDA を利用し, 次のボクセルへトラバースする
    fi
  until 交点が見つかる or 通過するボクセルすべてをトラバースし終える
end
```

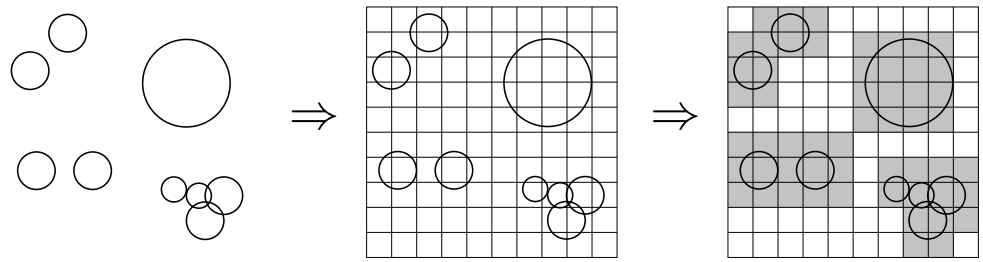
アルゴリズム中の '3DDDA' は Three Dimensional Digital Differential Analyzer の略で, 線分をスクリーンに描くアルゴリズムの 3 次元版に相当する. 3DDDA は藤本らによって初めて提案されたが [Fuji86], よく用いられているのは Amanatides らによるものである [Aman87]. これを次に示す. まず, 光線の始点を (o_x, o_y, o_z), 方向を (l_x, l_y, l_z) とすると, t をパラメータとして光線は以下のように表現できる.

$$x = l_x t + o_x,$$
$$y = l_y t + o_y,$$
$$z = l_z t + o_z.$$

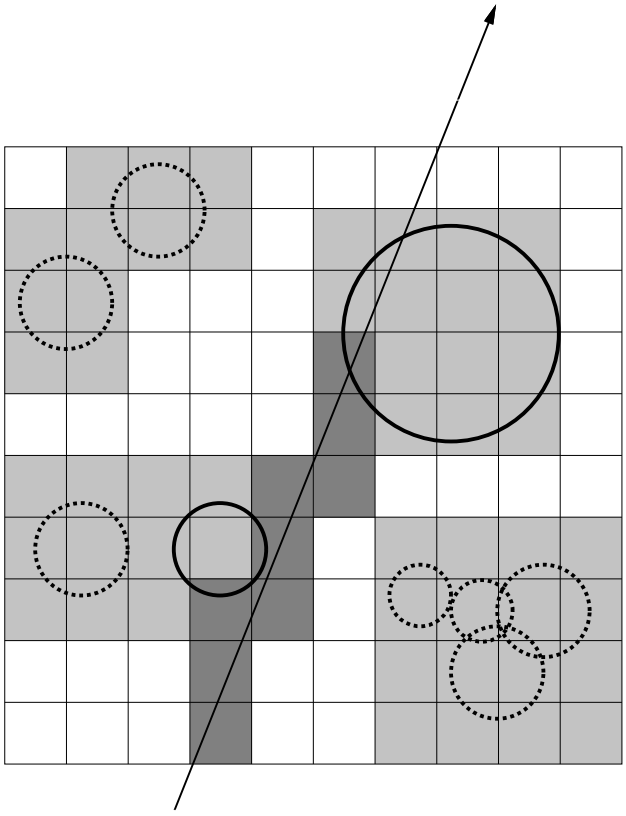
一方, ボクセルの各辺の長さを (v_x, v_y, v_z) とすると, 次の式によって各軸方向の t の変化 (dt_x, dt_y, dt_z) が定まる (**図 1.9**).

$$dt_x = v_x / l_x,$$
$$dt_y = v_y / l_y,$$
$$dt_z = v_z / l_z.$$

各軸方向ごとに次に通過するボクセルの境界平面を考えると, それぞれの平面までの距離 (t_x, t_y, t_z) の最小値を求めれば次に通過する平面が決まる. したがって, 次の処理を繰り返すことにより, 光線が通過するボクセルをトラバースすることができる.



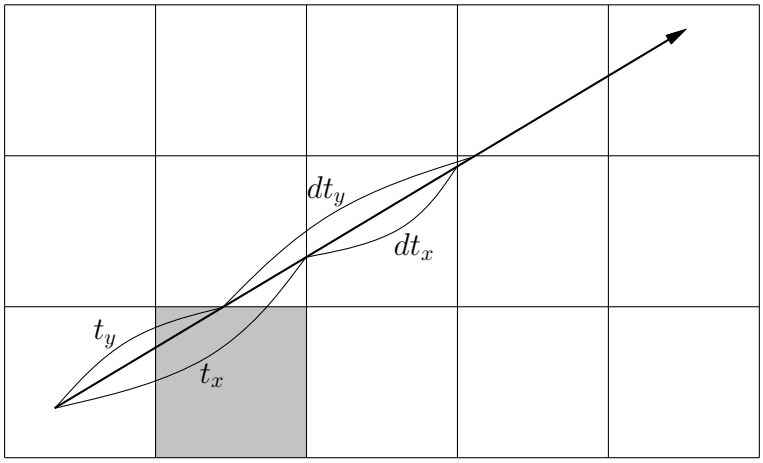
(a)



(b)

(a) 前処理. 空間を分割し, 各ボクセルについて重なっている物体を登録する. (b) 交点計算. 暗い灰色のボクセルのみを調べる. 太い実線で描かれた物体のみと交点計算を行なう.

図 1.8　高速化手法 (空間分割)



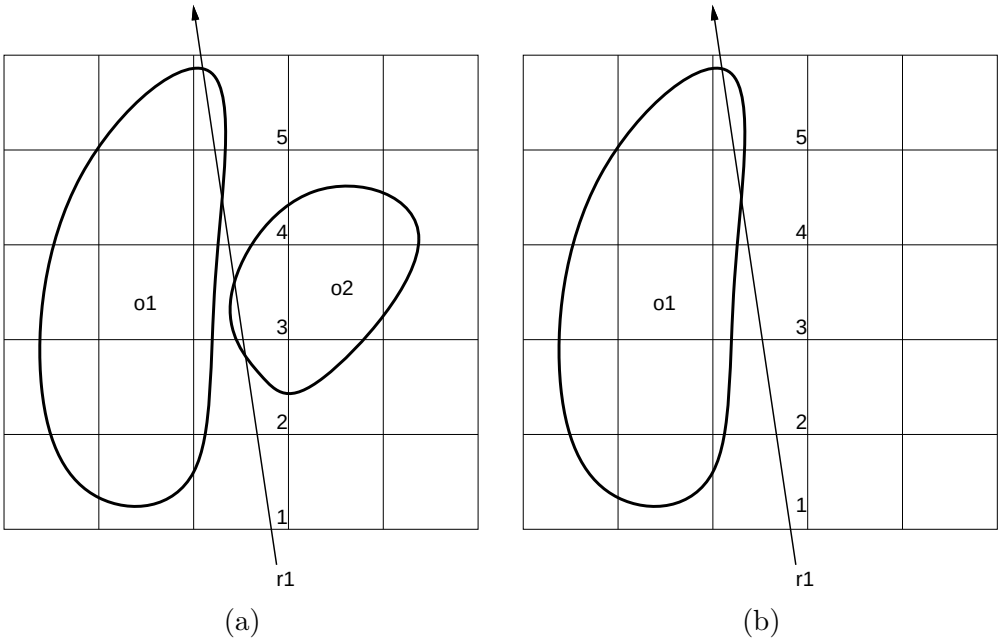
灰色のボクセルが現在のボクセル．この場合, t_y が t_x より小さいので, t_y を dt_y 増やし, 上のボクセルに移動することになる．

図 1.9 3DDDA

$t_m \leftarrow \min(\min(t_x, t_y), t_z)$
if $t_x = t_m$ **then**
　$t_x \leftarrow t_x + dt_x$
　ボクセル位置を x 軸方向に移動
fi
if $t_y = t_m$ **then**
　$t_y \leftarrow t_y + dt_y$
　ボクセル位置を y 軸方向に移動
fi
if $t_z = t_m$ **then**
　$t_z \leftarrow t_z + dt_z$
　ボクセル位置を z 軸方向に移動
fi

空間分割の効率は一様な空間分割, すなわちグリッドの場合, その一辺の分割数を r とすれば, 物体数 n に対して $O(n/r^{2\sim3})$ 程度となる．実際に最適な r を決定するのは必ずしも容易ではないが, バウンディングボリューム階層の場合と異なり, 大まかな設定でもかなりの効率を得ることができるのが空間分割の利点である．

原理はわかりやすいが, 実際に空間分割を実装する際には, 注意しなければならない点が 2 つある．1 つは交点が求まった際, その交点が含まれるボクセルに到達するまでは交点を確定してはならないということである (**図 1.10**-a)．これは, まだトラバースしていないボクセルに含まれている別の物体によって, 現在求まっている交点が隠されてしまうかもしれないからである．もう 1 つは複数のボクセルにまたがる物体に対して交点計算を繰り返す問題である (**図 1.10**-b)．これについては Arnaldi らが提唱したメイルボックス [Arna87] により解決できる．つまり, 各物体にメイルボックスとよばれる変数を用意して



(a) ボクセル 1 において交点計算を行なうと, ボクセル 4 にある o1 に対する交点が求まる. しかし, ここでトラバースを終了してしまうと, ボクセル 2 にある o2 に対する交点を見逃してしまう. (b) トラバースをボクセル 4 まで続けると, ボクセル 1 からボクセル 4 まで o1 に対する交点計算が繰り返される.

図 1.10 空間分割における注意点

おき, 交点計算を行なった際に現在の光線の ID を格納するのである. 実際の交点計算を行なう前にこのメイルボックスを調べれば, 現在の光線とその物体との交点計算を既に行なったかどうかがわかる.

1.3 扱われる問題

前節の高速化手法により, 初期に大きな問題となっていた計算時間については大きく改善された. これで大量の物体を含む画像を高速に生成することが可能となったが, 1987 年, 先に挙げた REYES の論文で次のような指摘がなされた [Cook87].

> 2.1. Geometric Locality.
>
> When ray tracing arbitrary surfaces that reflect or refract, a ray in any pixel on the screen might generate a secondary ray to any object in the model. The object hit by the secondary ray can be determined quickly [20,21,34], but that object must then be accessed from the database. As models become more complex, the ability to access any part of the model at any time becomes more expensive; model and texture paging can dominate the rendering time. For this reason, we consider ray tracing algorithms poorly suited for rendering extremely

> complex environments.
>
> ...

つまり, レイトレーシングのアルゴリズムは保持モード型でシーンデータへの参照のコストを考慮せず, メモリに入りきらないような大規模データに対してはスラッシングを生じてしまうということである. Cook らは映画製作に必要なレンダラの要件を詳しく分析し, 最も重要なものとしてデータ参照の局所性を挙げた. REYES が任意の大きさのデータを処理可能な Z バッファと同様の構造をもっているのはこのためである. このような構造にすることで, ユーザが与えた任意のデータを安定して処理することが可能となる. もちろん, レイトレーシングを採用しないことで, 反射や屈折などの複雑な光学現象は扱えないことになるが, Cook らはそうした現象は擬似的な表現によって代用できるとした. この選択は, 映画のために本物らしく見える画像を生成するという目的には十分適合する.

Cook らがレイトレーシングを採用しなかったことの意味は大きい. なぜなら, 彼ら自身が 1984 年に分散レイトレーシング (Distribution Ray Tracing) を発表し, モンテカルロ法の導入によりレイトレーシングを劇的に進歩させたからである [Cook84]. 最もレイトレーシングを進歩させた人々がレイトレーシングを捨てたことに, この問題の重大さが表れているといえよう.

REYES を実装した PIXAR 社の PRMan が, 映画などのためのハイエンドレンダラの事実上の標準になっていることは先に触れたが, この PRMan を含めたハイエンドグラフィックスに関するニュースグループが comp.graphics.renderman である. このグループでも, しばしば大規模データに関連した議論が行なわれている. それらの議論から, なぜ PRMan が使われるのか, なぜレイトレーシングは使われないのかという質問に対し, 著名な研究者が答えたものを以下に抜粋する.[*2]

> Larry Gritz:
>
> As for the original question, why one renderer should be preferred over another for film production, several answers have been given: flexibility, feature set, speed. Nobody has mentioned two equally important (probably *more* important) properties: robustness and the ability to handle massive complexity with grace. Studios will, generally speaking, take robustness over speed any day (at least up to a certain integer factor difference in speed).
>
> ...
>
> Tom Duff:
>
> Second, ray tracers usually use a lot more memory than scan-line renderers because they have to keep the entire geometric data base around for the whole

[*2] 抜粋中にある 'Matt Pharr's toro' は Pharr らのメモリコヒーレントレイトレーシングの実装の名前である. Pharr らの手法については後述する.

> frame. This seriously limits the scene complexity that they can handle. There are shots in A Bug's Life in which PRMan's memory footprint approaches two gigabytes. I don't believe there's a raytracer on the planet that can handle such scenes...
> ...
> On scenes for which PRMan occupies 2GB, a raytracer (unless it's Matt Pharr's toro) will likely occupy 40GB. PRMan, as we run it (using highly proceduralized models), never loads the complete database into memory. Even without proceduralization it never stores the complete tesselated data base.
> ...

このように, ハイエンドのレンダラにおいて重視されなければならないのは, 速度だけでなく安定性, 頑健性であり, 大規模データを処理する能力である. 通常のレイトレーシングは, すべてのシーンデータをメモリに保持しなければならないため, PRMan では安定して扱えるような状況を処理することが出来ない.

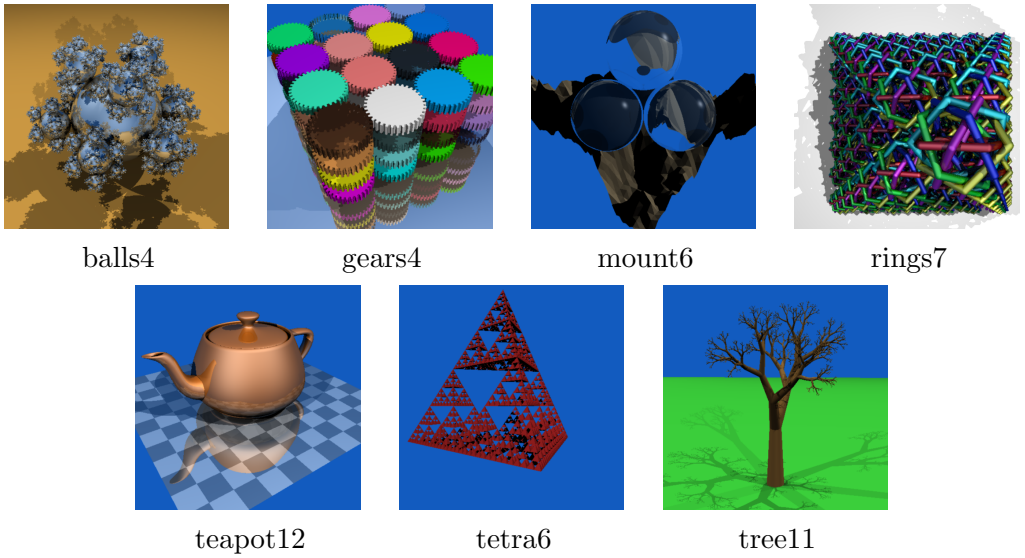
1.4 本論文の寄与

本研究の目的は, 任意の規模のシーンデータを効率よく処理可能なレイトレーシングを実現し, 前節で述べた問題を解決することである. このために本論文では大きく分けて以下の 2 つの手法を示す. 各手法ごとに記述してある項目が具体的な寄与である.

- 基本手法 [Naka97]
  - 幅優先レイトレーシングと一様空間分割を組み合わせた交点計算アルゴリズム
  - 光線および交点情報のソートに基づいたシェーディングアルゴリズム
  - 公開されたプログラムとの比較実験
- 改善手法 [Naka01]
  - 基本手法の主要な問題点の検討
  - 遅延処理の導入
  - シーンの階層化によるカリング[*3]
  - 高解像度グリッドの実現
  - 最大 10^9 個/87GB の大規模データを対象とした基本手法との比較実験

なお, これら 2 つの手法は別個のものではない. 名前が示す通り, 基本手法を改善/強化したものが改善手法である.

[*3] Culling. レンダリングの場合, シーンデータ中の各部分のうち, 生成画像に寄与しないものを刈り取る処理のことを指す. 様々な手法については Möller および Haines による名著 "Real-Time Rendering" [Möll99] に手際よくまとめられている.



ラベルはそれぞれシーン名とサイズファクタを示す.

図 1.11　SPD のシーン

1.5　実験用シーンデータ

1980 年代後半にレイトレーシングの高速化手法が次々に現れていたころ, 各研究者は独自に作成したモデルを対象に実験を行なっていた. しかし, このような評価方法では研究間での公平な比較ができない. そこで, Haines が実験データの共通化を目指して提唱したのが SPD (Standard Procedural Database) である [Hain87]. 本論文では, この SPD を実験用シーンデータとして用いている.

SPD はシーンデータを生成する複数のプログラムからなり, サイズファクタと呼ばれる数値を指定することで, 各シーンごとにさまざまな規模のデータを生成することができる. デフォルトのサイズファクタ (物体数 10,000 個以下) による画像を**図 1.11**に示す. なお, 実験の条件を揃えるため, SPD では形状以外の要素 (各シーンのプログラムが生成するデータの処理方法, 画面の解像度, 再帰的追跡の最大の回数など) についても細かく定めている.

SPD は多くの高速化手法の論文で用いられている. また, 本論文では RayShade [Kolb89] という有名な実装との比較を, SPD の実験手続きに従って厳密に行なっている. したがって, 本論文で提示する手法を, 既存の高速化手法や将来の研究と比較することも容易である.

なお, 後述の実験の節において, mount のコンパイルオプションについての記述がある

が, これはフラクタルの山を生成するハッシュ関数の種類の指定である. 初期の SPD において使われていたハッシュ関数は, あるサイズファクタ以上になると不自然さが際立つことが判明したため, 後のバージョンの SPD では改良されたハッシュ関数が含まれている. 本論文では実験の時点で最新の SPD を使用し, ハッシュ関数についても最もよいものを利用している.

1.6 本論文の構成

以下の章の構成は, 次のとおりである. まず, 本章で述べた問題について, これまでに行なわれてきた研究や関連する研究を**第 2 章**で述べる. 次の**第 3 章**では, これまでの手法の問題を解決する新しい手法である基本手法について述べる. この基本手法によって, 他の手法では扱うことが難しかったり, 安定して扱うことは可能でも計算時間がかかるようなデータの効率的な処理が実現する. **第 4 章**では, この基本手法自体の問題点を検討し, さらに改善した手法について述べる. この改善手法は, 基本手法のよい性質は崩さずに強化を行なった手法である. 最後に**第 5 章**で結論と今後の発展について述べる.

第 2 章

従来の研究

本章では, 前章で述べた問題に関連した従来の研究について述べる. これらの研究の基本的な考え方は, 次のようなものである.

- レイトレーシングの処理は, 複数の光線と複数の物体との積を行なう処理としてとらえられる. また, 光線の量は物体数などシーンデータの大きさとは直接の相関を持たない. そこで, 計算の順序を変更して複数の光線をまとめて扱えば, シーンデータへの参照を連続的にしたり参照の量を減らしたりすることができる.

複数の光線をまとめて扱う考え方は, 高速化 [Hanr86, Müll86] やベクトル計算機上のレイトレーシング [Max81, Plun85] のためにも利用されているが, これらは必ずしも一般的とはいえない. これは通常のレイトレーシングと比べて, 複数の光線をまとめて扱う処理そのものがオーバーヘッドになるからである. しかし, 大規模なデータについては, このような考え方をとることが不可欠となる.

実際には, この考え方の実現の仕方によってさまざまな手法が考えられるが, これまでに問題に直接答えようとした研究は少ない. これには通常のアルゴリズムの基本的な流れを変更しなければならないという困難さに加え, 次のような理由が考えられる.

- もし, テクスチャなどを大量に使用しないのであれば, シーンデータ中の各物体は, ごく少ないメモリしか消費しない. たとえば, 球の幾何形状を格納するのに必要なのは 4 個の実数だけである.
- シーンデータの複雑さは, インスタンシング, つまり物体の複製をつくることによって仮想的に増大させることができる [Snyd87, Kirk88, Deus98]. これは強力な手法であり, 照明シミュレーションを行なうような高度なレイトレーシングソフトウェアにおいても用いられている [Ward94]. また, 明確な幾何形状として保持するのではなく, ボリュームデータなどの別の形態で近似することによりデータの総量を減らす手法もある [Kaji89, Hart91, Neyr98].

- シーンデータへの参照パターンをある程度改善する手法がいくつか存在する．たとえば, 簡単なキャッシング [Hain88, Hain91] や, 空間充填曲線によるコヒーレントなピクセル参照 [Voor91] などがある．

これらのうち, 最初の点は, かつてのスキャンライン法の状況に似ている．1980 年代以前にはメモリは極めて高価であったため, 物体がそれほど多くないうちは, 少ないメモリしか消費しないスキャンライン法がよく用いられた．スクリーンを区切るなど特別な処理をしない限り, 大量のピクセルに対する z_p 値の保持が必要な Z バッファは——Z バッファさえ確保すれば物体数には制限がないにもかかわらず——あまり用いられなかったのである．また, 他の点で挙げた各種の技術は, それぞれ非常に有効ではあるが, 問題を先送りした影響もあったと考えられる．

これに対し, 個々のプロセッサあたりのメモリが少ないために問題が深刻であった並列計算機上では, 比較的早くから研究が行なわれていた．そこで, まず次節において, 並列計算機上のレイトレーシングの研究について概観する．次に, これを前提として, ワークステーションなどのより一般的な計算機を対象にした研究について述べる．

2.1 並列計算機上のシーンデータの扱い

並列計算機上のレイトレーシングとして初期に試みられ, 今日でも主流となっているアプローチは, キャッシュに基づいたものである．Green らは粗い解像度での画像生成によってプロファイリングを行い, 参照頻度の高いデータを検出することでキャッシュの管理を最適化する手法を提唱した [Gree89]．この手法では, 参照頻度の高いデータをメモリの固定領域に保持し, 参照頻度の低いデータはディスクから動的にキャッシュに読み込む．その後, より大規模な並列計算機が現れるにつれて, 各プロセッサのメモリにシーンデータを分散して保管し, 処理に必要なデータを他のプロセッサから各プロセッサのキャッシュに読み込む手法が開発されるようになった．代表的な研究としては Badouel らのものが挙げられる [Bado95]．また, 高度な光学現象を扱えるように拡張された手法も Reinhard らが提案している [Rein99]．

一方, キャッシュによらないアプローチとしては, Gaudet らのデータをプロセッサ間で共有しているバスにブロードキャストする手法 [Gaud88] や, Law らの各プロセッサのメモリに分散して保持したシーンデータを逐次的に参照する手法 [Law96] などがある．前者はブロードキャストゆえに, データやプロセッサが増えるにつれて, 速度が頭打ちになるという問題点があり, 後者は平行光線しか扱えないという制限がある．しかし, いずれもデータ参照を逐次化する手法としてとらえることができ, 後述の幅優先レイトレーシングに最も近いアプローチとして興味深い．

また, 安部 [安部 95] や加藤ら [Kato01] のように, シーンデータを各プロセッサのメモリに分散して保持しておき, それぞれを独立に処理した後に結果を併合する手法もある. つまり, プロセッサ間を頻繁に行き来するのは光線や交点の情報だけで, 各物体を処理するのは, あくまでそれを保持しているプロセッサということになる. シーンデータの分散の度合いが高くなると, 無駄な交点計算が増えるために並列化の効率は下がるが, スラッシングのような極端な性能低下を起こすことはない.

ここで特に注目すべきなのは, 加藤らが当初キャッシュを基本とする手法を採用しながら, 最終的にはこのような選択をしたことであろう. このシステムは並列/分散計算機上のレイトレーシングシステムとしては最新のものであるが, 映画製作を目指して考えられたため, 大規模なシーンデータの扱いが非常に重要であった. そこで, 多種多様な形態をとり, 量も大きくなるシーンデータではなく, 単一の形態をとり, 扱いの容易な光線や交点情報を通信で扱うようにしたのである.

以上をまとめると, 大きく分けてキャッシュに基づく手法と, そうでない手法があり, 前者は計算過程で動的にシーンデータの授受を行なうもの, 後者はあらかじめ定められた, 静的なシーンデータの授受を行なうものであるといえる. この対比は, 次節のワークステーションに対して行なわれた研究においても有効である.

2.2 ワークステーション上のシーンデータの扱い

最初に述べたように, ワークステーションなど, 一般的に用いられている計算機上で問題に取り組んだ研究はほとんどない. 後述する基本手法の原型となった筆者の研究 [中丸 92] を除けば, 直接的な研究といえるのは Pharr らのメモリコヒーレントレイトレーシング [Phar97] と Müller らの幅優先レイトレーシング [Lamp90, Müll92] のみである. これらをデータの参照方法で分類すると, Pharr らの手法がキャッシュによる動的な参照を行なうもの, Müller らの手法が逐次的, 静的な参照を行なうものとして理解することができる. 先の並列計算機上の対比と合わせ, 以下ではまず Pharr らの手法について述べ, 次に Müller らの手法について述べる.

2.2.1 メモリコヒーレントレイトレーシング

Pharr らの手法 [Phar97] は 1997 年に発表されたものである. この手法ではシーンデータや光線をディスクに格納し, キャッシュへ動的に読み込んだりディスクに書き出したりする. 変位マップ[*1] などによって手続き的に生成されるシーンデータについては, メモリ

[*1] Displacement Map. 物体表面をテクスチャとして与えられた変位によって変形する技術. たとえば球に岩肌の変位マップを指定すれば, 全体としては丸い, ごつごつした岩の形を作り出すことができる.

中のみでキャッシュに加えられ, 破棄されるものもある. このようなキャッシュを中心とした処理は, Pharr らが初期に行なった変位マップをレイトレーシングで扱うためのの研究 [Phar96] にその原型を見ることができる. また, 先に述べた並列計算機のための研究や, Funkhouser, Teller らのラジオシティ[*2] における研究 [Funk96, Tell94] の影響も大きい.

この手法では, 高速化のために空間分割を利用する. また, これとは別に, より粗い分割の空間分割をキャッシュ管理のために行なう. 後者のボクセルはキャッシュ管理の基本的な単位となり, 処理を進めた際に得られる利益が大きいボクセルが優先して処理されることになる. つまり, キャッシュに読み込まれていない大量の物体を含むようなボクセルを処理する利益は低くなり, 一方, 多数の光線を含むようなボクセルを処理する利益は高くなる. 全体の処理は**アルゴリズム 2.1**のように行なわれるが, **while** の直後のボクセルの決定が今述べた部分である.

アルゴリズム 2.1 メモリコヒーレントレイトレーシング

```
視点からの光線を生成する
while 処理の完了していない光線が残っている do
  処理するボクセルを決定する
  for all ボクセル中の光線 do
    ボクセル中の物体がキャッシュになければ読み込む
    ボクセル中の物体との交点計算を行なう
    if 交点が存在 then
      シェーディングを行なう
      新しく生成された光線をボクセルに追加
      現在の光線を破棄する
    else
      現在の光線を次のボクセルに加える
    fi
  od
od
```

Pharr らは, この利益の大きいボクセルの優先的処理により, キャッシュのヒット率を大きく上げることに成功している. 彼らが論文で示している全体で 440MB になるシーンの場合, 優先的処理を使わないとキャッシュサイズが 440MB 以下に制限されたところで計算時間が急激に上昇していくが, 優先的処理を使うとキャッシュサイズが 50MB 程度までは計算時間がそれほど上昇することはない.

以上のように, Pharr らの手法により, メモリの限界を超えた大きなデータを扱うことが可能となるが, 逆に考えるとキャッシュの能力を超えたところでは, やはりスラッシングが生じてしまうともいえる. また, キャッシュがうまく動作するかどうかは, 物体や視点

[*2] Radiosity. 拡散反射面の間の多重散乱を考慮して各面の照度を決定する, レイトレーシングとは別の照明計算手法.

の配置, 追跡される光線などのシーンの具体的な状況に依存するため, データサイズなどの単純な指標だけでは判断できない. つまり, Pharr らの手法は元々の問題を大きく軽減するが, 完全には回避できないことが難点である.

2.2.2 幅優先レイトレーシング

1990 年に発表された Müller らの手法 [Lamp90, Müll92] は, おそらく, REYES の論文で提起された問題に取り組んだ最初の研究と言えるだろう.[*3] この手法では, その名前の通り, 光線木を幅優先に処理することになる. つまり, まず視点から各ピクセルを通る 1 次光線すべてに対して各物体との交点計算を行い, 得られた結果を元にシェーディングを行なう. そして, その結果, 新たに生成された反射/屈折/影のための光線, つまり 2 次光線すべてに対して同様の処理を行い, ということを繰り返していく. このように世代ごとに光線をまとめて処理することにより, シーンデータを逐次的に処理し, しかも参照の回数を抑えることができる. 全体の処理を**アルゴリズム 2.2**に示す. なお, 図中の Ray-Z-Buffer は高速化のために使われるデータ構造である.

アルゴリズム 2.2 幅優先レイトレーシング

```
視点からの光線を生成する
while 処理の完了していない光線が残っている do
  光線をメモリに読み込み, Ray-Z-buffer を構成する
  for all ディスク上の物体 do
    Ray-Z-buffer を利用して物体と光線との交点計算を行なう
  od
  for all メモリ中の光線 do
    if 交点が存在 then
      シェーディングを行なう
      新しく生成された光線を未処理の光線として追加
    fi
  od
od
```

アルゴリズム 2.2を見ればわかるように, 幅優先レイトレーシングの構造は Z バッファのそれによく似ている. つまり, スクリーンの奥行き方向の深さを保持する Z バッファではなく, 各光線とそれらの交点の情報を保持する Ray-Z-Buffer を使い, 物体を逐次的に処理する. こうすることで, スラッシングを生じることなく, 任意の大きさのシーンデータを扱うことが可能となる. このように, 幅優先レイトレーシングは Z バッファの構造をレイ

[*3] 最初に挙げた論文 [Lamp90] の著者は Lamparter となっているが, この論文は Müller が以前から研究していた手法を英語で書き記したものであり, ドイツ語の文献に限れば 1988 年にはこの手法を記述しているものがある. 実際, この手法は Müller がレイトレーシングの高速化手法として開発した研究 [Müll86] に類似している.

トレーシングに導入したものといえる．

実際に処理速度を決める Ray-Z-Buffer は, 次のような発想に基づいたものである．以下, z 軸を基準に考えるが, 他の軸についても同様である．まず, 光線を次の式で表す．

$$x = az + b,$$
$$y = cz + d.$$

ここで, a, b, c, d は光線ごとに決まる定数である．つまり, 各光線は 4 次元の点となる．一方, z 軸に垂直で x, y 軸に平行な辺を持つ矩形, つまり, 軸に沿ったバウンディングボックスの面を成す矩形は次のように表現される．

$$z = c_0,$$
$$c_{11} \leq x \leq c_{12},$$
$$c_{21} \leq y \leq c_{22}.$$

この矩形が光線と交差するときには, 次の式が満たされなければならない．

$$c_{11} \leq ac_0 + b \leq c_{12},$$
$$c_{21} \leq cc_0 + d \leq c_{22}.$$

したがって,

$$-ac_0 + c_{11} \leq b \leq -ac_0 + c_{12},$$
$$-cc_0 + c_{21} \leq d \leq -cc_0 + c_{22}.$$

つまり, a–b, c–d それぞれの平面で 2 本の直線によって囲まれた帯状の領域に含まれる点 (光線) が, 交差する光線となる．この判定を行なうため, a–b, c–d それぞれの平面上の光線を 4 分木で保持し, これらの木を入れ子にした構造を作る．このような構造をさらに各軸に対して用意したものが Ray-Z-Buffer である．

Ray-Z-Buffer の効率は, 光線数 n に対して $O(n^{0.5\sim0.8})$ であることが理論および実験で確認されている．これは比較的良好といえる効率であるが, かかる係数は大きく, 主要な高速化手法と比較すると絶対時間で大きく劣っている．この理由には実装や利用可能なメモリ量も挙げられるが, 帯状の領域に含まれる光線を得るための処理に多数の掛算が必要になることが最も大きな理由である．Müller らは, この絶対時間の問題を克服するため, 分散計算機で動作するアルゴリズムを開発している [Müll92].

第 3 章

基本手法

本章では, 本研究における基本手法 [Naka97] の詳細について述べる. 次章の改善手法は, この基本手法を土台に幾つかの重要な改善を行なったものである. 最終的に用いられるべき手法は改善手法であるが, 基本的な考え方は基本手法にあり, アルゴリズムの構成要素も多いため, 章をわけて説明する.

本研究での基本的な考えは, 頑健性を得るために幅優先レイトレーシングを土台としながら, 交点計算の高速化のために一様空間分割を利用する, というものである. 前述のように, Müller らは高速化のために特殊なデータ構造を利用したが, 幅優先レイトレーシングにこのような構成が必ずしも必要というわけではない. 本質的な点は “光線とシーンデータの役割を入れ替える” ということであり, 通常のレイトレーシングで用いられている高速化手法の応用が可能である.

“光線とシーンデータの役割を入れ替える” ということは, 一様空間分割の場合では “物体ではなく, 光線をボクセルに登録する” ということである. 非一様な物体分布については, シーンデータを幾つかのグループにわけ, それぞれに対応するグリッドを用意する. 各グリッドについての処理結果を併合すれば, 最終的な結果が得られる. 全体の処理の流れは次のようになる.

- **前処理:** すべてのシーンデータをバイナリ形式に変換し, 各物体をバウンディングボックスと共に格納する. 次に各グリッドごとにボクセル構造を用意し, シーンデータを逐次的に読み込みながらボクセルに印をつける.
- **交点計算:** グリッドごとに光線をボクセルに登録し, シーンデータを逐次的に読み込みながら各光線の交点情報を更新する. 得られた結果は併合され, 最終的には現在の世代のすべての光線に対する交点を出力する.
- **シェーディング:** 影光線の結果からその影光線の前世代にあたる光線への輝度の寄与を求める. 影光線以外の光線については, それらの交点において必要であれば次

世代の反射/屈折/影光線を生成する．これらの処理もシーンデータを逐次的に読み込みながら行なわれる．

後の 2 つの処理は, 処理すべき光線が無くなるまで繰り返される．つまり, 擬似コードとして記述すると, 全体の処理は**アルゴリズム 3.1**のようになる．なお, 3 つの段階いずれにおいてもシーンデータは常に逐次的に処理されている．このようにすることで, スラッシングを生じないアルゴリズムを得ることができる．

アルゴリズム 3.1 全体の処理

```
前処理
視点から全ピクセルを通る光線 (第 1 世代の光線) を生成
repeat
  交点計算
  シェーディング
until 次世代の光線が存在しない
```

このアルゴリズムの考えは一様空間分割のみならず, 他の高速化手法においても適用可能である．特に, 8 分木などによる非一様空間分割には直接的に適用できる．ただし, “一様空間分割には特定の領域を定数オーダで参照できる” という, 他の手法にはない優れた特性があり, このことが多数の物体が存在する領域を参照する際の大きな利点となる．

以下では, まず, 前処理から交点計算までの基本となるアルゴリズムを示す．次に, このアルゴリズムに伴う問題とそれらの問題を解決するための処理について述べる．こうして交点計算までのアルゴリズムが得られた後, シェーディングのアルゴリズムを示す．最後に, これらのアルゴリズムの実装による実験結果について述べる．

3.1 前処理と交点計算

先に述べたように, 一様空間分割において光線とシーンデータの役割を入れ替えるということは, ボクセルに物体ではなく光線を登録するということである．このように光線が登録されているグリッドに対し, 物体を 1 つずつ読み込み, 対応するボクセルを決定する．これらのボクセルのうち, 光線を含んでいるものがあれば, それらの光線と物体との交点計算を行ない, 各光線の交点情報を更新する．こうしてすべての物体の処理が終われば, 最終的な交点情報が得られることになる．擬似コードとして書くと, **アルゴリズム 3.2**のようになる．なお, コード中のメイルボックスは通常の空間分割におけるメイルボックスに対応し, 交点計算の重複を防ぐためのものである．物体ではなく光線にメイルボックスがあり, 光線ではなく物体の ID を保持する点が, 通常の空間分割と異なっている．

アルゴリズム 3.2の考えで処理を行なうことは可能だが, 幾つかの問題が存在する．まず, 光線の始点に近い順に交点計算を行なうことができない．通常の空間分割では, 光線の

アルゴリズム 3.2 交点計算 (基本手法/初期版)

```
ボクセルを空に初期化
while ディスク上に未処理の光線が存在する do
  while 光線でメモリが満たされていない do
    光線をディスクから読み込む
    3DDDA を初期化
    ボクセルをトラバースし, 各ボクセルに光線を登録
  od
  for all ディスク上の物体 do
    物体をディスクから読み込む
    物体と重なるボクセルを決定
    for all 重なるボクセル do
      for all ボクセルに登録されている光線 do
        if 光線のメイルボックス ≠ 物体の ID then
          if 光線とバウンディングボックスが交差 then
            光線と物体との交点計算を行なう
            光線の交点情報を更新
            光線のメイルボックス ← 物体の ID
          fi
        fi
      od
    od
  od
  for all メモリ中の光線 do
    交点情報をディスクに書き出す
  od
od
```

始点に近い物体から交点計算を始め, 見つかっている交点がトラバースしたボクセルに含まれた時点で交点を確定し, 処理を終了することができる. 他の問題は光線を通過する全ボクセルに登録するため, 大量のメモリが必要になることである. これは深刻な問題である. なぜなら, 必要なメモリはグリッドの解像度に比例して増大するからである.

あきらかに物体と重ならないボクセルに光線を登録する必要はない. したがって, どのボクセルが物体と重なるかを調べておけば, 必要なメモリを減らすことができる. ただし, すべてのボクセルがなんらかの物体と重なるような状況を考えれば, これだけでは十分とはいえない. そこで, 必要なメモリ量を限定するため, 次の処理を行なう. つまり, 光線をすべての通過ボクセルに登録するのではなく, 物体と重なるボクセルのうちで光線の始点に最も近い数個のみに登録し, すべての物体を処理する. そして, 再び次の数個のボクセルに登録を行い, 同様にすべての物体を処理する. これを繰り返すことで, 最終的に正しい交点を得ることができる. また, この方法により, 物体との交点計算を光線の始点に近い順に行なうこともできるようになる.

この他, 物体と重なるボクセルの決定のコストや情報の格納方法を考慮し, 物体と重な

るボクセルの決定を, 座標軸に沿ったバウンディングボックス (Axis-Aligned Bounding Box) と重なるボクセルの決定に置き換える. これにより, 関連するボクセルを簡単な計算で求めることができ, ディスクに格納するシーンデータとしては, 物体ごとにバウンディングボックスの 6 つの実数を付帯させるだけで済むことになる. このようなデータ形式はグリッドの解像度にも依存しない. よりタイトなボクセルを前処理で求めて, ディスクに格納することも可能であるが, グリッドの解像度を変えたとき, ボクセルの決定をやり直す大量の処理を行ない, さらにディスク上のデータを再構成しなければならない. 後述のように, バウンディングボックスは, その凸包性を利用し, 他の無駄な処理を省くためにも利用できる.

バウンディングボックスによる決定は, より厳密な決定方法と比べて関連するボクセルを増加させるため, 無駄な交点計算を生じることになる. しかし, この点は実際にはそれほど大きな影響を持たないことが多い. たとえば, 有名なレイトレーシングプログラムである Rayshade [Kolb89] では, このようなバウンディングボックスによる決定を行なっている. また, もし無駄な交点計算が大きな問題となる場合には, 1 つの物体を仮想的に分割して扱い, 複数のよりタイトなバウンディングボックスを対応させることで問題を解決できる.

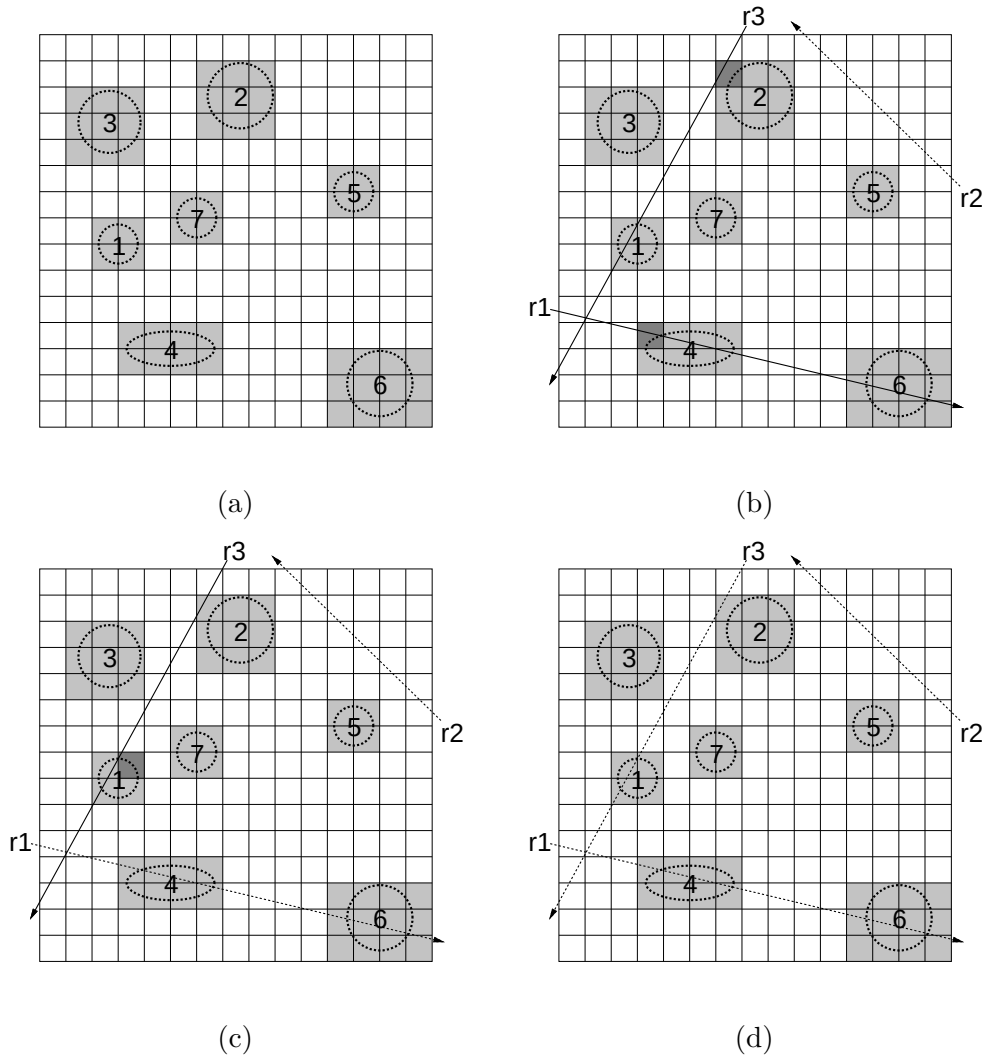
上述の修正を適用すると, 新しい擬似コードは**アルゴリズム 3.3**のようになる (**図 3.1**参照). なお, コード中で “交点情報が完全である” とは, (1) 交点が既にトラバースしたボクセル中に含まれている場合か, (2) 光線がグリッド外に出た場合を指す.

以上により, 始点に近い物体から順に交点計算を行なえない問題と, メモリ量の問題は解決した. しかしながら, 最初に挙げた方法と異なり, 完全な交点情報が得られるまでシーンデータが繰り返し参照されることになる. また, よりわかりにくい問題として, 交点計算の無駄も存在する. これらの問題と解法については, 次の **3.2 節**と **3.3 節**で詳しく述べる.

ここで説明のため, 幾つかの言葉を定義する. まず, ‘パス’ は**アルゴリズム 3.3**中の内側の **while** ループ本体の処理を表す. ‘候補ボクセル’ は光線を登録すべきボクセルを表す. これは, これまでの説明では, 印のついているボクセルということになる. ‘登録数’ はパスあたりの各光線に対する候補ボクセル数を表す. この数は光線を登録するのに必要なメモリ量を制御する. ‘ボクセル境界’ はバウンディングボックスと重なるボクセルからなる領域の境界を表す.

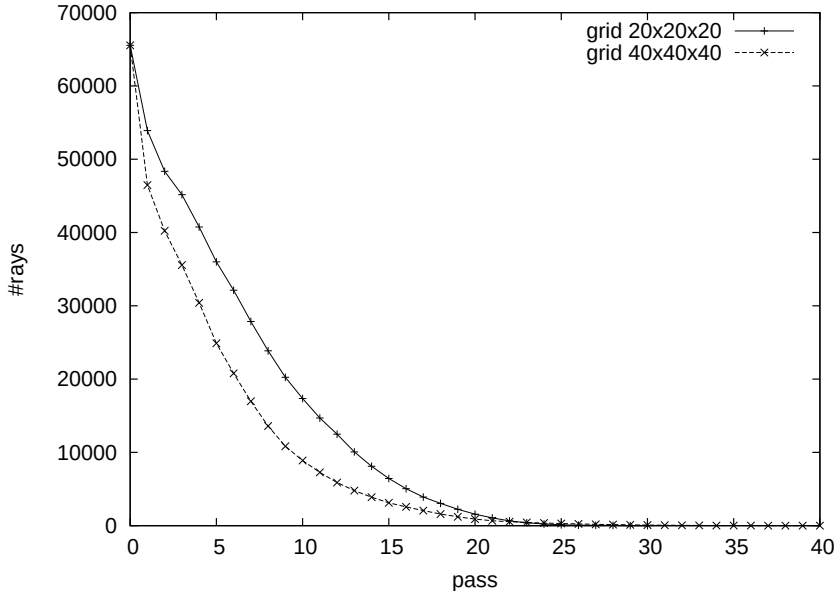
3.2 シーンデータ参照回数の削減

登録数を固定とした場合, シーンデータの参照回数, つまり, パスの回数はどの程度必要になるだろうか. 本研究では, 実験的に完全な交点情報を得ていない光線の本数が指数的

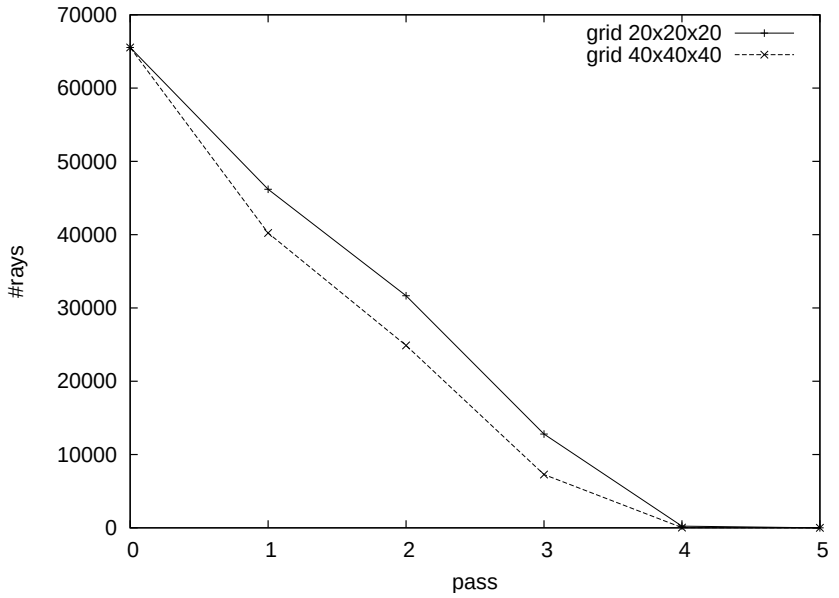


物体は 1 から 7 までの 7 個, 光線は r1, r2, r3 の 3 本, 一度に登録するボクセルは各光線ごとに最大 1 個であるとする. (a) バウンディングボックスと重なるボクセルに印がつけられる (明るい灰色). (b) r2 に対する処理は即座に終了し, r1 と r3 は候補となる最初のボクセルに登録される (暗い灰色). すべての物体についての処理が終わると r1 に対する交点が求まり, r1 についての処理が終了する. (c) r3 が候補となる次のボクセルに登録され, 再びすべての物体についての処理が行なわれる. (d) r3 の交点が求まり, すべての処理が終了する.

図 3.1 交点計算 (基本手法/修正版) の例



(a)



(b)

初期の光線は 65,536 本である．(a) 登録数を 1 に固定．(b) 登録数を動的に増加．初期の登録数は, 解像度 20 のグリッドに対して 1, 解像度 40 のグリッドに対して 2 である．

図 3.2　残存光線の本数

アルゴリズム 3.3 交点計算 (基本手法/修正版)

```
ボクセルを空に初期化
while ディスク上に未処理の光線が存在する do
  while 光線でメモリが満たされていない do
    光線をディスクから読み込む
    3DDDA を初期化
  od
  while メモリ中に交点情報が不完全な光線が存在する do
    for all メモリ中の光線 do
      if 光線の交点情報が不完全 then
        ボクセルをトラバースし, 印のついたボクセルの数個に光線を登録
        { 途中で交点情報が完全になった場合には中断する }
      fi
    od
    for all ディスク上の物体 do
      物体をディスクから読み込む
      物体と重なるボクセルを決定
      for all 重なるボクセル do
        for all ボクセルに登録されている光線 do
          if 光線のメイルボックス ≠ 物体の ID then
            if 光線とバウンディングボックスが交差 then
              光線と物体との交点計算を行なう
              光線の交点情報を更新
              光線のメイルボックス ← 物体の ID
            fi
          fi
        od
      od
    od
  od
  for all メモリ中の光線 do
    交点情報をディスクに書き出す
  od
od
```

な減少を示すことを見出した. **図 3.2**-a はある実験データに対して常に登録数を 1 とした場合の例である. このような指数的減少について完全な説明を与えることは困難であるが, 確率的に考えれば次のようになる. つまり, 各パスの後, 各光線の処理が確率 $(1-p)$ で終了すると (各光線が確率 p で生き残るとすると), n 回のパスの後, 残っている光線の割合は p^n となる.

この他, このグラフからグリッドの解像度が高いほどパスの回数が増大することもわかる. これは相対的にボクセルが小さくなるため, 一時にトラバースできる空間的な距離が小さくなるためである.

物体数が少ない場合は問題ないが, 多い場合はもちろんパスの回数の増大を無視するこ

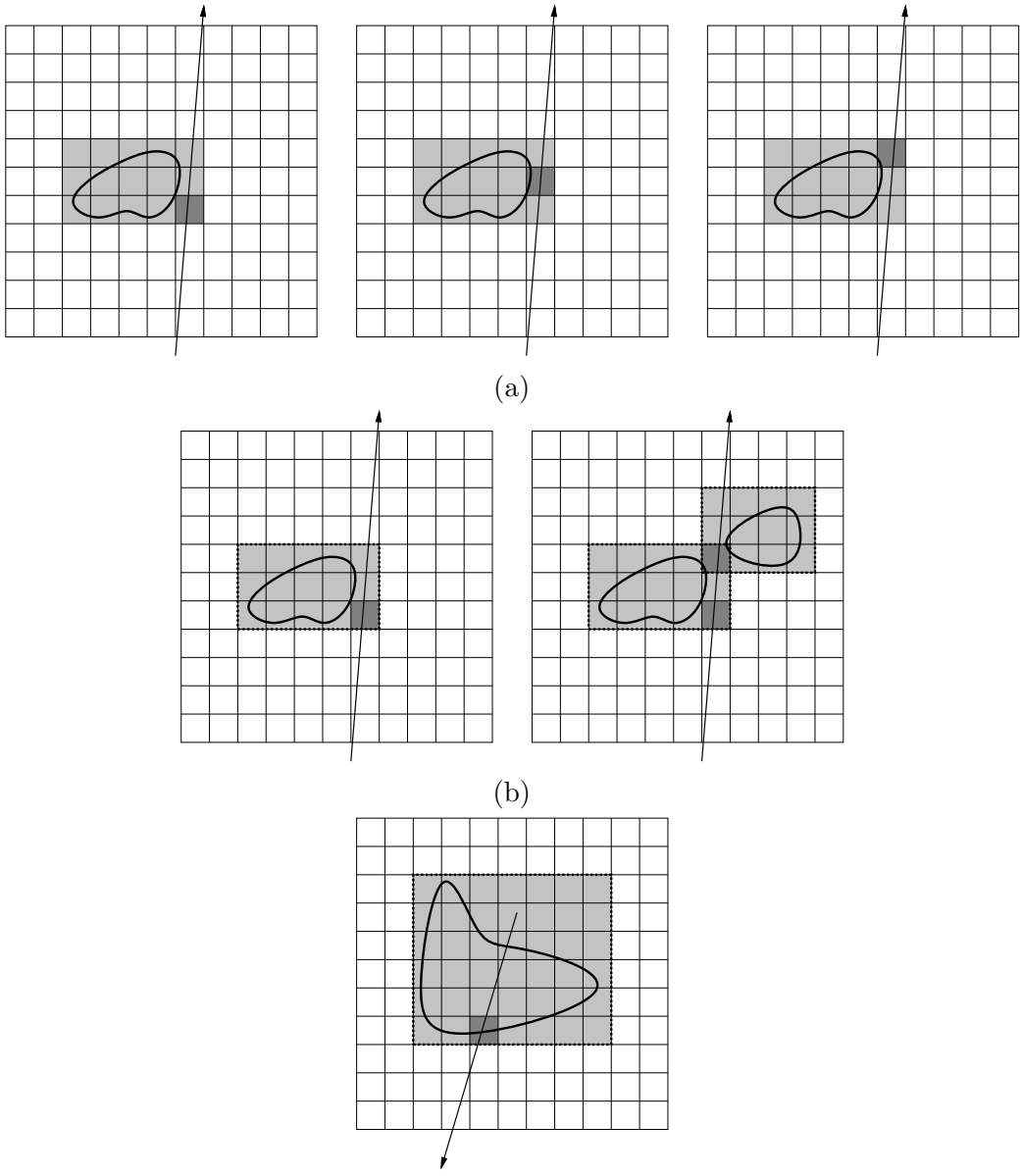
とはできない. そこで, 次のように登録数を変化させることにより, パスの回数の増大を抑える.

- 処理が完了した光線のためのメモリ領域を残っている光線のために利用する. たとえば, 最初に光線の本数が 10,000, 登録数が 1 であったとき, 残っている光線が 5,000 以下になったら登録数を 2 に, 3,333 以下になったら登録数を 3 に, というように登録数を変化させていく. つまり, 残っている光線の本数に反比例する形で登録数を増加させるのである. このようにして減少の仕方が線形となるようにできる.
- 初期の登録数をグリッドの解像度 r に比例させる. たとえば, $r = 20$ に対して初期の登録数が 1 であった場合, $r = 100$ に対しては初期の登録数を 5 とする. この方法に必要なメモリ領域は $O(r)$ で増大するが, 係数が小さいため, 先のような深刻な問題とはならない. このようにしてパスの回数を一定に保つことができる.

図 3.2-b にこれらを適用した結果を示す. このようにパスの回数を大きく減少させ, またグリッドの解像度に関係なく, ほぼ一定に保つことができる.

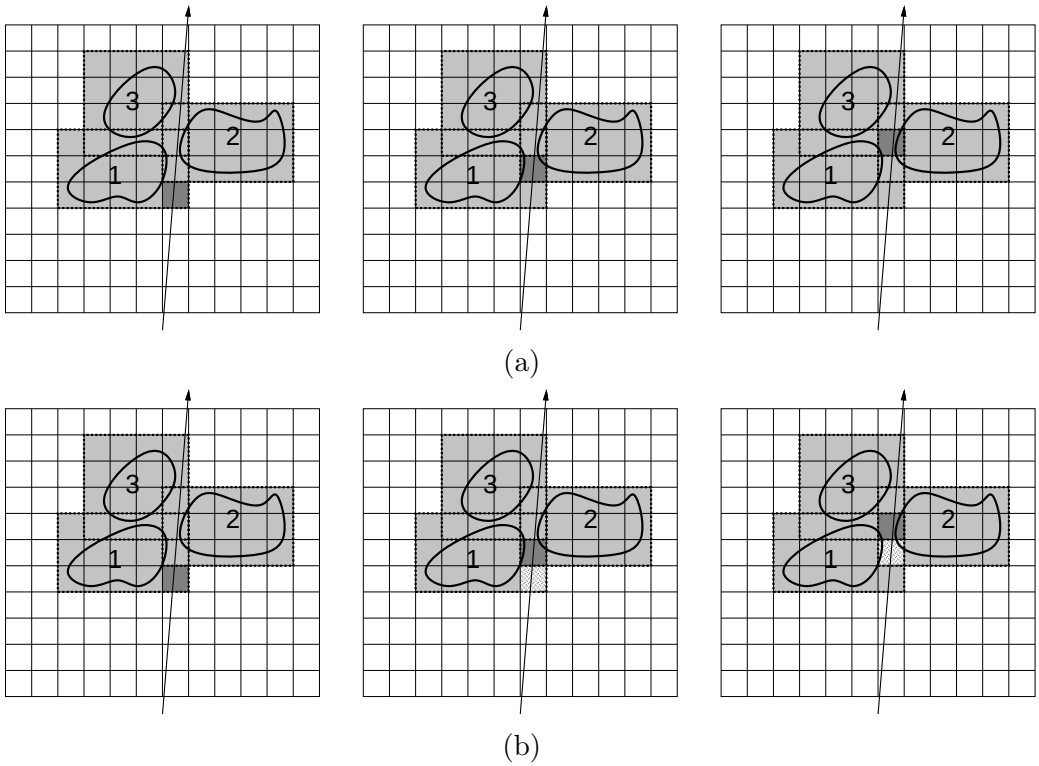
ところで, パスの回数の増大を抑えるということは, 別の見方をするなら如何に各光線を速くトラバースさせるかということである. この点から考えるとトラバースを遅くする別の問題があることがわかる. たとえば, 登録数を 1 として**図 3.3**-a の状況を考えよう. この場合, 光線がバウンディングボックスと重なるボクセルを通過するためには 3 つのパスが必要である. 実際には光線がバウンディングボックスに対するボクセル群に '入った' 時点でのみ, 各物体に対する交点計算を行なえばよい. そこで, 候補ボクセルの定義を以下のように変えて無駄を省く.

- バウンディングボックスの重なりを示すフラグに加え, 各ボクセルのどの面が (いずれかのバウンディングボックスに対する) ボクセル境界となっているかを示す 6 つのフラグを定義する. これらのフラグは前処理で設定される.
- いずれかのバウンディングボックスと重なるボクセルのうち, (1) 光線がそれと相対する (光線の側を向いた) ボクセル境界を通過した直後のものか (**図 3.3**-b), (2) 光線の始点がボクセル境界内 (バウンディングボックスと重なっているボクセル) にある場合は任意のボクセル境界を通過する直前のもの (**図 3.3**-c) を候補ボクセルとする. ここで, 後者はそのような光線がボクセル群に '入る' ことが起きないために必要になる.



登録数を 1 とする．(a) 印のついたボクセルをトラバースするために, 3 つのパスが必要になる．(b) 左: 同じ状況において 1 つのパスのみで済む．右: 複数の物体が存在する場合．この場合, 2 つのパスが必要になるが, 以前の方法なら 5 つのパスが必要である．(c) 光線の始点がボクセル境界内にある場合．ボクセル境界内の物体との交点計算を行なうため, ボクセル境界を通過する前に光線を登録しなければならない．

図 3.3 ボクセル境界での登録



登録数を 1 とする. (a) 物体 1 に対する交点計算は各パスで繰り返される. (b) 最後に登録されたボクセル (斑模様) の位置を保持する場合. 2 回目および 3 回目のパスでは, このボクセルが物体 1 に対するボクセルに含まれているため, 物体 1 に対する交点計算を行なわない.

図 3.4 交点計算の削減

3.3 交点計算の削減

通常の一様空間分割にはない無駄な交点計算は, 2 つの理由で生じる. 第 1 の理由は, **アルゴリズム 3.3**において導入した通過する一部のボクセルへの登録である. 登録数を 1 として**図 3.4**-a のような状況を考えると, たとえメイルボックスを使ったとしても, 物体 1 に対する交点計算は各パスで行なわれてしまう. これは物体 1 以外の他の物体により, メイルボックスが上書きされてしまうためである. つまり, メイルボックスをパスを超えて動作させることはできない.

しかし, 各物体に対するボクセルは, バウンディングボックスと重なるボクセルである. この定義によるボクセル群の領域は凸となるため, 光線は領域を最大で 1 度しか通過しない. このことを利用すると, 次のように問題を解決できる (**図 3.4**-b).

- 各光線について, 前回のパスで最後に登録された (光線の始点から最も遠い) ボクセ

ルの位置を保持する．もし, そのボクセルがある物体に対するボクセル群に含まれているなら, 前回のパスで交点計算は行なわれているので, あらためて交点計算をする必要はない．ボクセルが含まれているかどうかの判定は, バウンディングボックスと重なるボクセルに含まれるかどうかを判定するだけなので, 容易に行なうことができる．

無駄な交点計算が生じるもう 1 つの理由は, **3.2 節**で導入した登録数を動的に変化させる処理である．登録数が 1 より大きい場合, 最終的に求まる交点によって隠れてしまう, 光線の始点からより遠くにある交点の計算まで行なってしまう可能性がある．これは登録数を変化させる限り不可避であるが, もしメモリに入りきらない大量の光線がある場合は, 次のように削減できる．つまり, **アルゴリズム 3.3**では単に光線を複数のグループに分けて別々に処理する簡単な方法をとっているが, これを次のように変えて登録数の増加を抑える．

- まず, パスの最後で処理が完了した光線の交点情報を出力する．次に, ディスクに残っている未処理の光線を完了した光線が占めていたメモリに読み込み, それらの 3DDDA を初期化する．このように可能な限りメモリが満たされた後, 登録数がメモリ中の光線の本数から決定される．

この方法により, メモリが光線で満たされている限り, 登録数は可能な限り小さくなり, 無駄な交点計算も抑えられる．一方, 交点情報の出力の順序が光線の入力のそれとは異なってしまうため, 交点情報のソートが新たに必要となる．しかし, これはアルゴリズムの他の部分と比べれば極めて容易であり, 短時間しか必要としない．

上述の修正を含んだ前処理および交点計算のアルゴリズムを, **アルゴリズム 3.4**および**アルゴリズム 3.5**に示す．なお, パスの最後の処理に注目してほしい．次のパスにおける候補ボクセルの手前までに求まっている交点が含まれるかどうかを調べるため, メモリ中の光線に対するトラバースを行なっている．このように, 先にメモリ中の光線が次回のパスまで生き残るかどうかを確定しないと, 新たにどれだけの光線を読み込めるかがわからない．

3.4 シェーディング

最初に述べたように, シェーディングは 1 つの世代に対する交点計算が終了した後に実行される．一見すると, 既に**アルゴリズム 3.4**のように光線をメモリに加えていく処理を知っているので, 交点が求まった時点でシェーディングを行ない, 新たな光線をメモリに加えてもよいように思われる．実際, これはシェーディングのために必要なシーンデータ

アルゴリズム 3.4 前処理 (基本手法/最終版)

```
ボクセルを空に初期化
for all ディスク上の物体 do
  物体を読み込む
  バウンディングボックスと重なるボクセルを決定
  for all 重なるボクセル do
    ボクセルに印をつける
    if ボクセルの面がボクセル境界に含まれる then
      対応する面に印をつける
    fi
  od
od
```

の参照のコストが問題にならなければ可能である. しかし, シェーディングでは, まず交点での法線ベクトルの計算を行なわなければならず, これには物体の幾何形状が必要である. また, 交点での色を決定するため, 大きなテクスチャマップを参照することもある. これらのデータ参照は, もちろんスラッシングを起こさないように行なわれなければならない.

また, Müller らは, おそらくはシェーディングを行なう部分の実装を単純化するため, 法線ベクトルや物体表面上の座標 (テクスチャを参照するためなどに必要) を交点が求まる度に計算しているが, このようにすると, 最終的に交点が確定するまでに相当に無駄な計算が行なわれてしまう. 法線ベクトルや座標の計算は物体によっては高価なものとなるので, こうした処理にも注意を払うべきである.

これらの点から, シェーディングは交点計算から分離し, 法線ベクトルなどの計算はシェーディングまで行なわないようにする. 交点計算の結果として得られる交点情報は, 光線 ID, 物体 ID, 光線の始点から交点までの距離, の 3 つの要素からなるが, これらの交点情報と光線, そしてシーンデータをどのように参照し, 処理を進めるかが残りの問題である.

シーンデータは, 物体の幾何形状やシェーディングの種類ごとに構造が大きく異なるが, 光線や交点情報の構造は一定しており, メモリ割り当てなどの各種の処理が容易である. そこで, 交点計算と同様にシーンデータ以外の情報をメモリに保持し, それらを物体順にソートすることでシーンデータへの逐次的な参照を実現する. **アルゴリズム 3.6**に具体的なアルゴリズムを示す.

要約すると, (1) 光線と交点情報をメモリに格納可能な複数のグループに分割し, (2) それぞれのグループとシーンデータとの結合を行なうということである. なお, **アルゴリズム 3.6**内の “局所的な輝度への寄与と次世代の光線をディスクに書き出す” という部分は, 入力とは異なる順序で結果を出力することを意味している. このため, ソートを行なう必要があるが, これも交点計算の場合と同様に極めて容易である.

アルゴリズム 3.5 交点計算 (基本手法/最終版)

```
while ディスク上に未処理の光線が存在する do
  while 光線でメモリが満たされていない do
    光線をディスクから読み込む
    3DDDA を初期化
    ボクセルをトラバースし, 最初の候補ボクセルに光線を登録
    { 途中で交点情報が完全になった場合には中断する }
    if 交点情報が完全 then
      交点情報をディスクに書き出す
    fi
  od
  メモリ中の光線の本数から登録数を決定
  for all メモリ中の光線 do
    ボクセルをトラバースし, 残りの候補ボクセルに光線を登録
  od
  for all ディスク上の物体 do
    物体をディスクから読み込む
    バウンディングボックスと重なるボクセルを決定
    for all 重なるボクセル do
      for all ボクセルに登録されている光線 do
        if 光線のメイルボックス ≠ 物体の ID then
          if 前回登録されたボクセルが範囲外 then
            if 光線とバウンディングボックスが交差 then
              光線と物体との交点計算を行なう
              光線の交点情報を更新
              光線のメイルボックス ← 物体の ID
            fi
          fi
        fi
      od
    od
  od
  for all メモリ中の光線 do
    ボクセルをトラバースし, 次のパスの最初の候補ボクセルに光線を登録
    { 途中で交点情報が完全になった場合には中断する }
    if 交点情報が完全 then
      交点情報をディスクに書き出す
    fi
  od
od
```

アルゴリズム 3.6 シェーディング (基本手法)

```
while ディスク上に未処理の光線が存在する do
  while 光線でメモリが満たされていない do
    光線をディスクから読み込む
  od
  メモリ中の光線に対応する交点情報をディスクから読み込む
  光線と交点情報を交点情報内の物体 ID でソートする
  シーンデータの先頭に移動する
  for all メモリ中の交点情報 do
    while 現在の物体 ID < 交点情報中の物体 ID do
      物体をディスクから読み込む
    od
    シェーディングの計算を行なう
    局所的な輝度への寄与と次世代の光線をディスクに書き出す
  od
od
```

3.5 実験結果

本節では, これまでに述べたアルゴリズムの実装による実験結果を示す. 実装は個々の処理ごとに C で記述されたサブプログラムからなり, sray という名前のシェルスクリプトがこれらを統合する. 実行環境は Linux を稼動している Pentium 90MHz, メモリ 32MB の PC である. 生成画像の解像度など, 各種実験条件は SPD の指定に沿っている. 使用した SPD のバージョンは 3.6 で, シーン mount のコンパイルには `NEW_HASH` を指定した.

実験では, まず, Rayshade [Kolb89] と sray との比較を行なった. Rayshade は, メモリ中で動作する一様空間分割を実装したレイトレーシングプログラムで, 作られた時期は古いものの, 今日でも比較的速いプログラムとして知られている. 条件を厳格に揃えるため, sray の物体に対する交点計算ルーチンは RayShade のそれに基づいたものを使い, 一方, Rayshade についても幾つかの修正を加えた. 次に示す結果で, rayshade-ss は SPD で指定された条件の光線を追跡するように, 1 次光線の生成部分やシェーディングの処理に修正を加えたもの, rayshade-ss-r はさらに影のキャッシング [Hain88] およびボクセル中の光線ボックスによるカリング [Snyd87] の機能を外したものを示す. ほとんどのシーンでは, すべての物体は単一のグリッドで処理され, 物体数 n に対し, グリッドの一辺の解像度を $\lfloor n^{1/3} + 0.5 \rfloor$ とした. シーン balls および tree については, 土台の板の部分を除いた残りの物体を同様に処理した.

表 3.1はデフォルトのサイズファクタのシーンについての比較である. デフォルトのサイズファクタでは, いずれのシーンも数 MB 程度のメモリしか消費しないため, Rayshade は問題なく動作する. この状態で Rayshade と sray を比較することにより, 基本手法の

表 3.1 小規模データに対する計算時間

	計算時間 (秒)			比		
	rss	rss-r	sray	rss-r/rss	sray/rss	sray/rss-r
	全体					
balls4	137.59	214.94	710.01	1.56	5.16	3.30
gears4	289.18	545.32	1045.31	1.89	3.61	1.92
mount6	171.05	223.81	816.13	1.31	4.77	3.65
rings7	361.80	607.32	1204.18	1.68	3.33	1.98
tetra6	44.88	71.23	111.04	1.59	2.47	1.56
tree11	102.42	143.20	480.11	1.40	4.69	3.35
	前処理					
balls4	2.31	2.36	3.12	1.00	1.35	1.35
gears4	9.02	9.28	15.18	1.00	1.68	1.68
mount6	4.78	4.90	7.51	1.00	1.57	1.57
rings7	4.95	4.99	5.63	1.00	1.14	1.14
tetra6	2.35	2.36	3.68	1.00	1.57	1.57
tree11	3.78	3.73	5.48	1.00	1.47	1.47
	交点計算					
balls4	81.14	163.52	303.40	2.02	3.74	1.86
gears4	201.72	444.37	563.35	2.20	2.79	1.27
mount6	94.43	159.35	280.44	1.69	2.97	1.76
rings7	213.74	490.66	763.91	2.30	3.57	1.56
tetra6	24.89	55.61	52.66	2.23	2.12	0.95
tree11	63.15	108.36	202.92	1.72	3.21	1.87
	シェーディング					
balls4	54.14	49.06	403.49	0.91	7.45	8.22
gears4	78.44	91.67	466.78	1.17	5.95	5.09
mount6	71.84	59.56	528.18	0.83	7.35	8.87
rings7	143.11	111.67	434.64	0.78	3.04	3.89
tetra6	17.64	13.26	54.70	0.75	3.10	4.13
tree11	35.49	31.11	271.71	0.88	7.66	8.73

‘rss’ は rayshade-ss を, ‘rss-r’ は rayshade-ss-r を表す. Rayshade の交点計算の時間は, プロファイリングの結果から推定された値である. Rayshade のシェーディングの時間は, 全体の時間から前処理および交点計算の時間を差し引いたものとして定義される.

オーバーヘッドが明らかになる. なお, 登録数は交点計算の回数を Rayshade と一致させるため, 常に 1 としている.

この結果から, 全体の時間では rayshade-ss に対して 3 から 5 倍程度, rayshade-ss-r に対して 2 から 3 倍程度の時間がかかっていることがわかる. 各部分について, sray の処理時間が長い理由を以下に示す.

- **前処理:** Rayshade と sray で前処理の内容はまったく異なるが, sray の処理時間が長い基本的な理由はディスク I/O である.

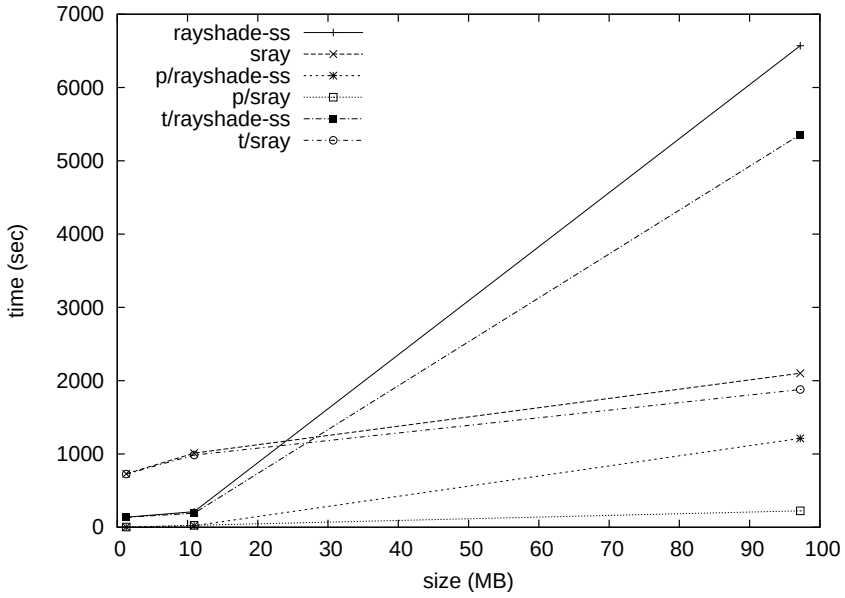
- **交点計算:** sray の実装は大規模なシーンを効率よく扱うため, 可能な限り多くの光線を保持するようになっており, 光線ボックスを実装していない. このことが rayshade-ss に対して sray の処理時間が長くなっている理由の 1 つである. 影のキャッシングについても同様の影響がみられる. ただし, これらの機能を抑制した rayshade-ss-r についての比較でも, 交点計算に 1.2 から 1.8 倍の時間がかかっている. この理由を次に列挙する.
  - 光線や交点情報に対するディスク I/O
  - ボクセルに格納された光線の参照
  - 光線のボクセルへの登録
  - 光線が最後に登録されたボクセルについてのテスト

  なお, gears4 および tetra6 については, sray の処理時間が比較的短くなっている. これは非常に薄いバウンディングボックスが多数存在するためである. これらのバウンディングボックスについて Rayshade が誤って交差していると判定する場合でも, sray では正しく交差を避けることができる. 先の 1.2 から 1.8 倍という数値は, プロファイリングでバウンディングボックスの無駄なテストをカウントし, 調整を行なった上での値である.
- **シェーディング:** sray は 3 から 9 倍の時間がかかっている. この原因は, ディスク I/O と光線および交点情報のソートである.

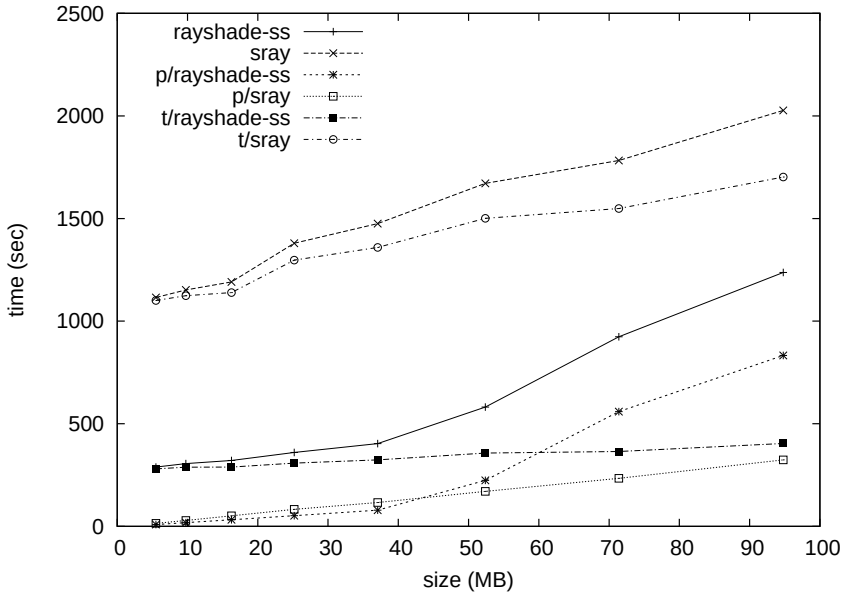
なお, ここで使用しているシーンでは, 光線数が物体数に比べてずっと大きい. このため, 光線に対する各種の処理が速度低下の大きな理由となる.

図 3.5から**図 3.7**はシーンデータの増加に対する時間の変化を示したグラフである. **表 3.2**は, 各シーンごとに使用したサイズファクタとそれに対する物体数, さらに rayshade-ss が割り当てたメモリ量を示している. sray について, 登録数はグリッドの解像度 r に対して $\lfloor r/10 + 0.5 \rfloor$ と定めた. 解像度 r は Rayshade および sray で同じ値を使っている. なお, 幾つかののシーンではサイズファクタに対してデータが非多項式オーダで増加する. このため, サイズファクタを 1 ステップずつ変えても, グラフ上のサンプル点の間隔は広くなる.

これらのグラフから, ほぼ期待通りの結果が得られていることがわかる. つまり, メモリが十分であるうちは rayshade-ss が速く, rayshade-ss がスラッシングを生じ始めると, sray が速くなる. スラッシングが始まると, rayshade-ss の計算時間は急速に増大するが, 各処理について見てみると, 前処理の時間よりトレーシング (全体の時間から前処理の時間を除いた処理) の時間の方がより増大の割合が大きかったり, 逆に前処理の時間が増加しているにも拘わらず, トレーシングの時間はなお sray よりも速い場合などが見られる. これらの興味深い現象は, それぞれのシーンについて次のように説明できる.



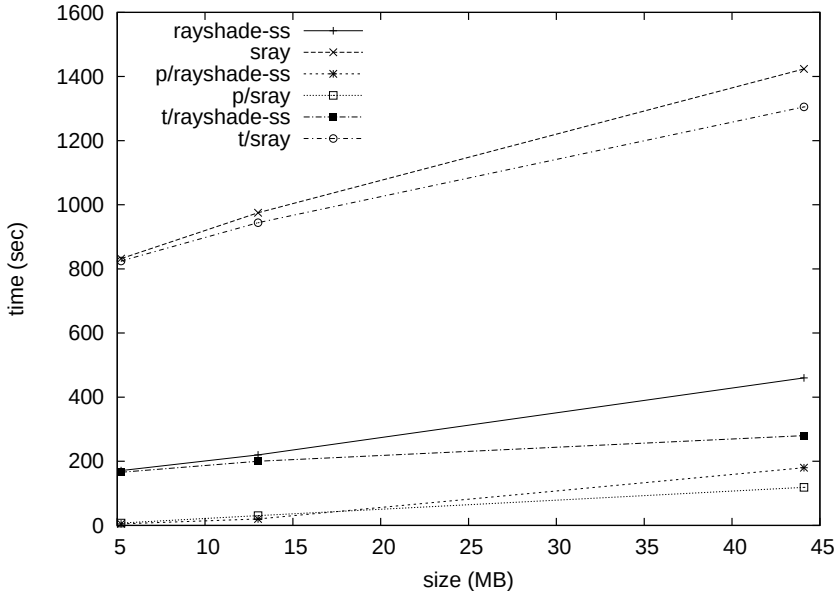
balls



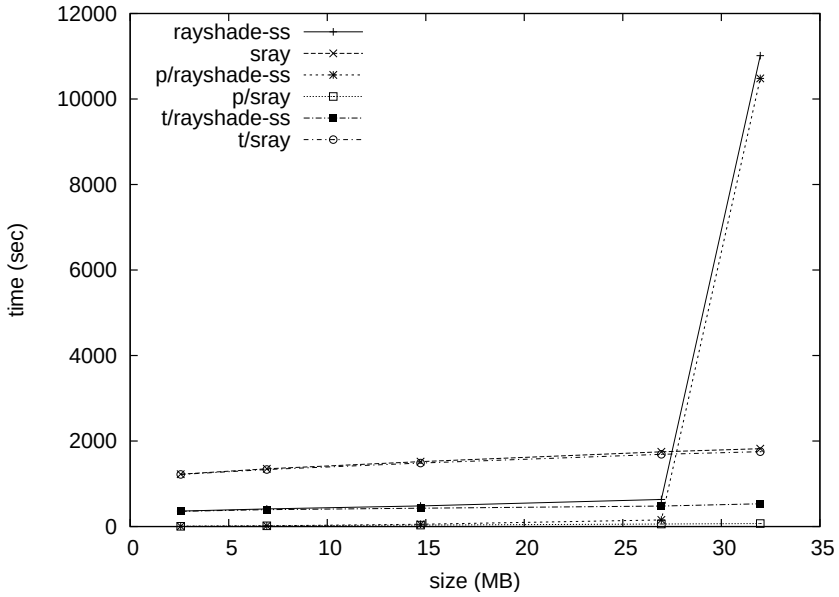
gears

シーンデータのサイズは rayshade-ss において割り当てられたメモリ量を示している．プレフィックス ‘p/’ は前処理を, ‘t/’ はトレーシング (前処理後の実際のレイトレーシングの処理) を表す．プレフィックスがないものが全体の時間である．

図 3.5 データサイズに対する計算時間の変化



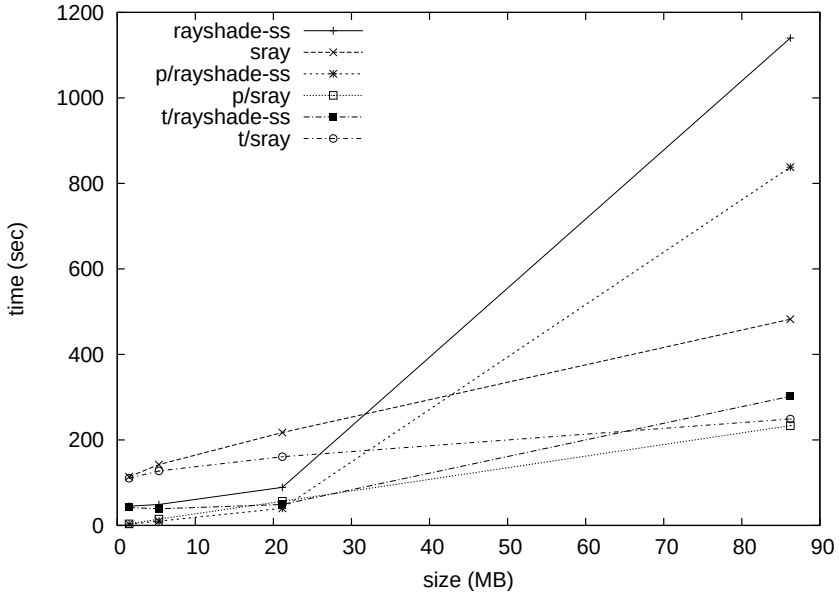
mount



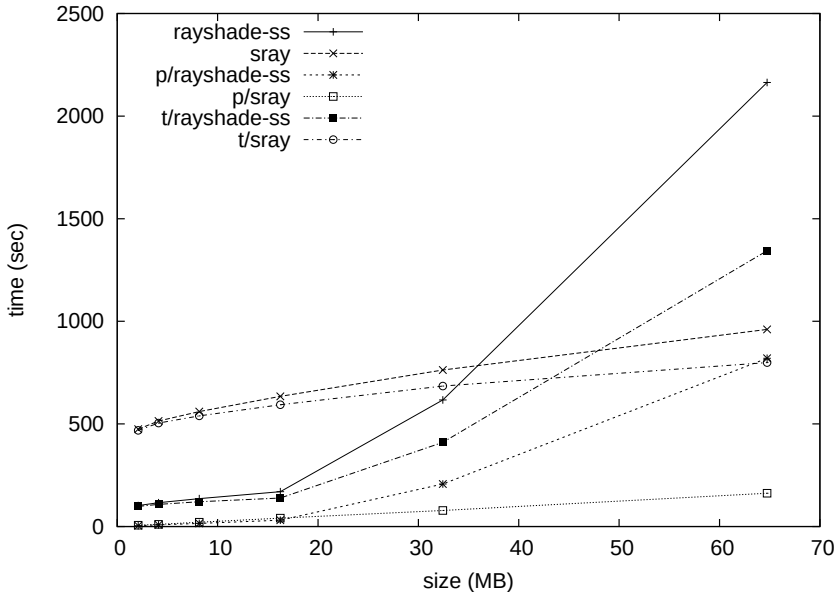
rings

表記については**図 3.5**を参照.

図 3.6 データサイズに対する計算時間の変化 (続き 1)



tetra



tree

表記については**図 3.5**を参照.

図 3.7 データサイズに対する計算時間の変化 (続き 2)

表 3.2 rayshade-ss によるメモリ使用量

	サイズファクタ	物体数	メモリ使用量 (MB)
balls	4	7,382	1.3
	5	66,431	10.9
	6	597,872	97.2
gears	4	9,345	5.5
	5	18,251	9.8
	6	31,537	16.3
	7	50,079	25.2
	8	74,753	37.1
	9	106,435	52.4
	10	146,001	71.4
	11	194,327	94.8
mount	6	8,196	5.2
	7	32,772	13.0
	8	131,076	44.1
rings	7	8,401	2.6
	10	23,101	6.9
	13	49,141	14.7
	16	89,761	27.0
	17	107,101	32.0
tetra	6	4,096	1.5
	7	16,384	5.3
	8	65,536	21.1
	9	262,144	86.2
tree	11	8,191	2.1
	12	16,383	4.1
	13	32,767	8.2
	14	65,535	16.3
	15	131,071	32.4
	16	262,143	64.7

- **balls/tree:** 多数の非常に小さな物体が一箇所に集中した状態になっている．このため, 光線がそうした領域にはいると, 多数のメモリ例外を生じることになる．
- **gears:** 手前の物体によって隠れていて, 参照が生じない物体が多数存在する．このため, 比較的大きなシーンデータについても, rayshade-ss の実質的なワーキングセットは小さくなる．
- **mount/tetra:** balls や tree のような物体の集中はないが, 多数の物体が最終的には多くのメモリ例外を生じる．mount については次のサイズファクタがスワップ領域を超えるサイズのデータとなってしまうため, 実験を行なっていないが, シーンの状況からほぼ tetra と同様の過程となると考えられる．
- **rings:** サイズファクタに対して物体の大きさが比較的緩やかに減少するため, 他

表 3.3 圧縮された大規模データに対する結果

	物体数	非圧縮データ (MB)	圧縮データ (MB)	計算時間 (時間)
mount11	8,388,612	904	380	8.35
rings80	10,432,801	1,522	384	6.30
tetra12	16,777,216	1,808	255	2.37

計算時間にはデータの生成/変換の時間を含まない.

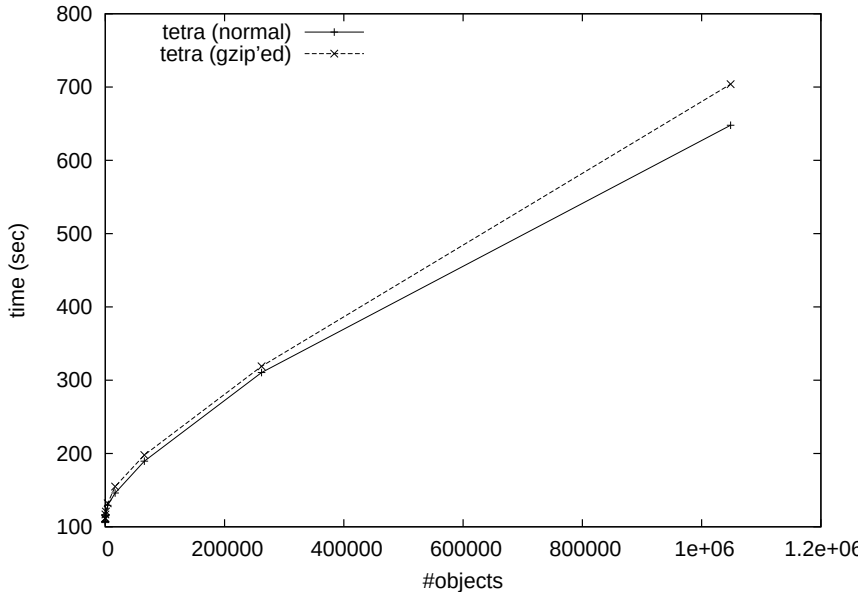
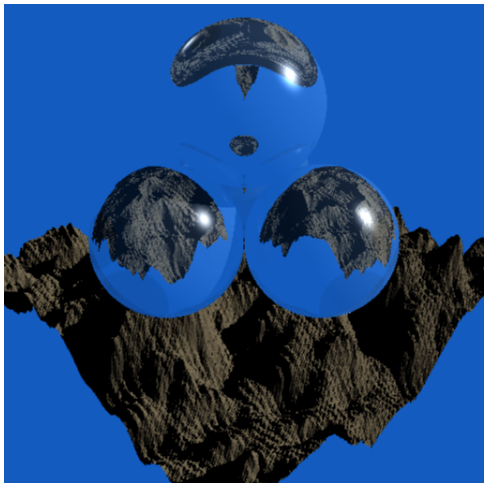


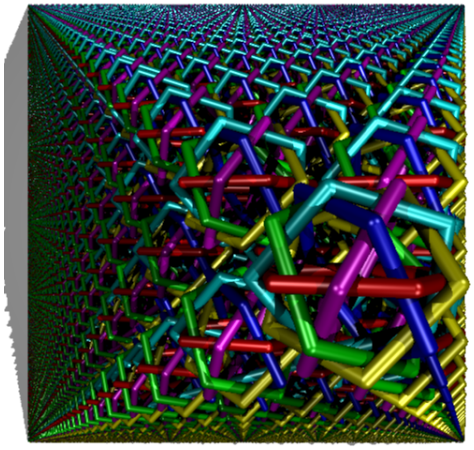
図 3.8 圧縮の有無による計算時間の変化

のシーンと比べると各物体が重なるボクセルが多い. このことが物理メモリの容量をわずかに超えた時点での非常に長い前処理時間の原因である. 一方, トレーシングについては, gears と同様の傾向を示している.

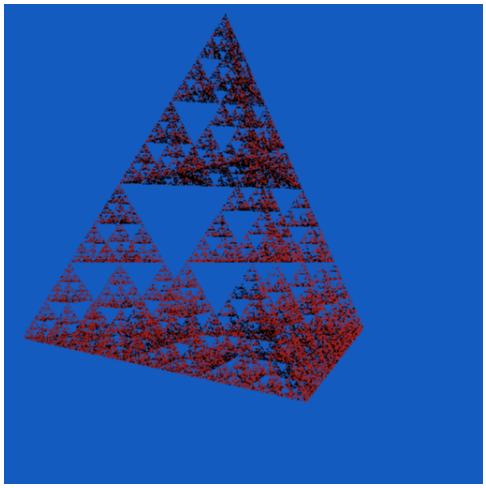
最後に, より大規模なデータに対する興味深い例を紹介する. sray ではシーンデータに対する参照が完全に逐次的であるため, 圧縮された状態のシーンデータを扱うことは極めて容易である. **図 3.8**は, tetra を対象として `gzip` によって圧縮されたデータと非圧縮のデータについて比較した結果である. 圧縮データを展開するコストはデータサイズに比例して増大するが, 圧縮によってディスクサイズを超えるデータさえ扱うことが可能である (本実験ではシーンデータの格納のために 700MB のディスクを使用している). **図 3.9**に圧縮データを利用して生成した画像を, **表 3.3**にこれらの画像についてのデータサイズや計算時間を示す.



mount11



rings80



tetra12

rings80 については視点からの光線と影のみについて計算した.

図 3.9 圧縮された大規模データに対する生成画像

第 4 章

改善手法

本章では, 前章で述べた基本手法の問題点を改めて検討し, 改善を加えた手法 [Naka01] について述べる. これらの改善は, 基本手法の根本的な構造は変えない. そのため補完的ではあるが, 得られる効果は極めて大きく重要な改善といえる.

近年の重要な成果である Pharr らの手法 [Phar97] と基本手法を改めて比較すると, 前者はキャッシュ主体, 後者はストリーム主体のアルゴリズムといえる. つまり, Pharr らの手法は通常のレイトレーシングをキャッシュによって直接拡張したものであり, ディスク上のデータをキャッシュを利用しながら動的に参照する. これに対し, 基本手法ではディスク上のデータを常に逐次的, つまり静的に参照する. このような静的な参照による処理の利点は, 任意のデータに対して安定して動作することが保証されることである. これは先に述べたように Z バッファや REYES など即時モード型アルゴリズムの最も重要な特性であり, 保持モード型アルゴリズムにはないものである. もちろん, Pharr らが行なったように保持モード型アルゴリズムを改良し, 扱えるワーキングセットを拡大することはできるが, 何らかのデータに対して突然プログラムが適用できなくなる可能性は残る. 特に, 単にデータサイズだけではなく, 物体や光線の空間的配置など, データの構成によって安定性が大きく変化しうることが問題である.

一方, 基本手法の Pharr らの手法に対する欠点は, 最終的に必要かどうかにかかわらず, すべてのデータを処理することにある. Pharr らの手法はキャッシュ主体という性質上, 本質的に不要なデータは処理しない. 不要なデータを処理する無駄は, 個々の物体など処理単位ごとでは非常に小さいが, シーンデータが大きくなると無視できない.

また, Pharr らの手法に対するものとは別の欠点として, グリッドの解像度がメモリサイズによって制限されることがあげられる. 一般にグリッドの解像度は物体数に対して緩やかに増加し, また, 制限された解像度においてもアルゴリズムは安定して動作するが, 一様空間分割の効率は物体数 n, 解像度 r に対して $O(n/r^{2\sim3})$ であるため, 制限された解像度に達した後は $O(n)$ の効率となってしまう.

以上に挙げた 2 点を改善するのが本章の目的である．これらの改善によって異常ともいえる大規模なデータをも安定かつ高速に処理することができるアルゴリズムが得られる．改善の多くは基本的な考えに基づいており, 特に前者の問題に対するものは, レイトレーシング以外の即時モード型アルゴリズム, たとえば強化された REYES [Apod00] や各種のカリング手法 [Möll99] と関連が深い．実際, このことは基本手法の性質のよさを示している．つまり, 全体の構造が即時モード型となっているため, 他の即時モード型アルゴリズムでの改良を応用しやすいのである．また, ややわかりにくい部分だが, 即時モード型であるということ自体も, アルゴリズムの構造, 特にメモリ管理に関する部分を単純化し, さまざまな改善を加えやすいことに繋がっている．

以下の節では, まず, 上に挙げた欠点について具体的に述べる．次に, 無駄なデータ処理の問題に対する改善を行なう．そして, これらの改善を利用しながら, グリッドの解像度の制限に対する改善を行なう．最後に, 以上に基づいて実装したプログラムによる実験結果について述べる．

4.1 基本手法とその欠点

基本手法の詳細については前章で述べたとおりであるが, 具体的にどの部分にどのような問題があるのだろうか．この点を検討するため, 3 つの主要部分——前処理, 交点計算, シェーディング——についての擬似コードを改めて**アルゴリズム 4.1**および**アルゴリズム 4.2**に示す．後に続く節においても, これらのコードを適宜参照されたい．

アルゴリズム 4.1および**アルゴリズム 4.2**を見ると, まず, すべてのデータが最終的に必要かどうかによらず, 読み込まれていることがわかる．実際には, たとえば交点計算では, 以下の条件がすべて満たされるとき, 初めて物体の幾何形状が必要になる．

(1) 物体のバウンディングボックスが重なるボクセルに光線が含まれている．
(2) (1) のいずれかの光線に対し, 前回登録されたボクセルがボクセル境界の外にある．
(3) (2) のいずれかの光線のうち, バウンディングボックスと交差するものがある．

シェーディングについても, 物体の幾何形状や材質, テクスチャなどは, 交点情報に格納された物体 ID に対応するものだけが必要である．

一方, 前処理や交点計算に関する部分の暗黙の前提が, “グリッドはメモリ中に保持される” ということである．グリッド全体をメモリに保持しているからこそ, ボクセルに対するランダムアクセスが可能となり, 印をつけたりトラバースしたりする処理が簡単に行なえるわけだが, このことがグリッドの解像度を制限することになる．

アルゴリズム 4.1 前処理および交点計算 (基本手法)

```
{ 前処理 }
ボクセルを空に初期化
for all ディスク上の物体 do
  物体を読み込む
  バウンディングボックスと重なるボクセルを決定
  for all 重なるボクセル do
    ボクセルに印をつける
    if ボクセルの面がボクセル境界に含まれる then
      対応する面に印をつける
    fi
  od
od
{ 交点計算 }
while ディスク上に未処理の光線が存在する do
  while 光線でメモリが満たされていない do
    光線をディスクから読み込む
    3DDDA を初期化
    ボクセルをトラバースし, 最初の候補ボクセルに光線を登録
    if 交点情報が完全 then
      交点情報をディスクに書き出す
    fi
  od
  メモリ中の光線の本数から登録数を決定
  for all メモリ中の光線 do
    ボクセルをトラバースし, 残りの候補ボクセルに光線を登録
  od
  for all ディスク上の物体 do
    物体をディスクから読み込む
    バウンディングボックスと重なるボクセルを決定
    for all 重なるボクセル do
      for all ボクセルに登録されている光線 do
        if 光線のメイルボックス ≠ 物体の ID then
          if 前回登録されたボクセルが範囲外 then
            if 光線とバウンディングボックスが交差 then
              光線と物体との交点計算を行なう
              光線の交点情報を更新
              光線のメイルボックス ← 物体の ID
            fi
          fi
        fi
      od
    od
  od
  for all メモリ中の光線 do
    ボクセルをトラバースし, 次のパスの最初の候補ボクセルに光線を登録
    if 交点情報が完全 then
      交点情報をディスクに書き出す
    fi
  od
od
```

アルゴリズム 4.2 シェーディング (基本手法)

```
while ディスク上に未処理の光線が存在する do
  while 光線でメモリが満たされていない do
    光線をディスクから読み込む
  od
  メモリ中の光線に対応する交点情報をディスクから読み込む
  光線と交点情報を交点情報内の物体 ID でソートする
  シーンデータの先頭にファイルポインタを移動する
  for all メモリ中の交点情報 do
    while 現在の物体 ID < 交点情報中の物体 ID do
      物体をディスクから読み込む
    od
    シェーディングの計算を行なう
    局所的な輝度への寄与と次世代の光線をディスクに書き出す
  od
od
```

4.2 無駄なデータ処理に対する改善

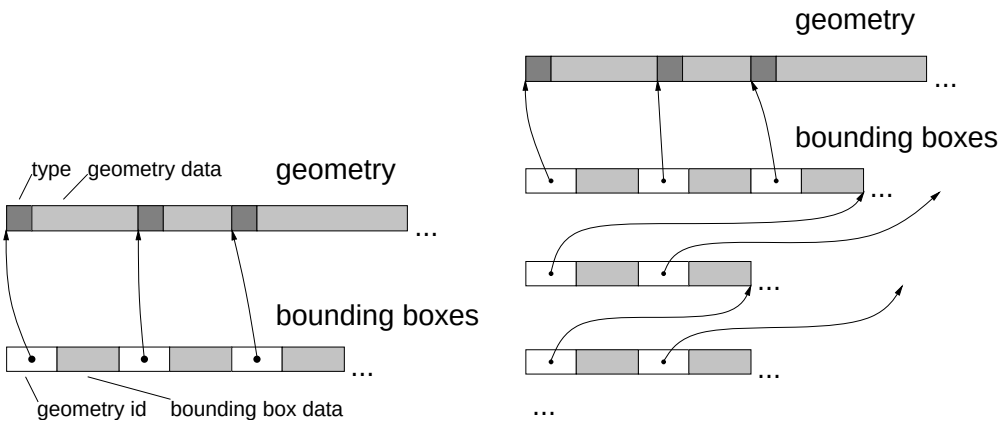
4.2.1 幾何形状とバウンディングボックスの分離

処理のほとんどの部分は, 幾何形状とバウンディングボックスの両方は必要としない. たとえば前処理にはバウンディングボックスだけが必要であり, シェーディングについては幾何形状だけが必要である. 交点計算は両方のデータを必要とするが, 先に見たように幾何形状はバウンディングボックスといずれかの光線が交差したときのみ必要である. これらを考慮して幾何形状とバウンディングボックスを分離し, **図 4.1**-a のように別々のファイルとなるようにディスクに格納する.

なお, このようにファイルを分離しなくても, 単一ファイル内でシークを行なえばよいように思えるかもしれないが, このような処理は OS のシステムコールという, 別のコストを要求する. 個々のシークは極めて短い時間しか必要としないが, 何百万, 何千万というシステムコールのコストは無視できない. 実際, 本研究では大規模データ格納のためのコードを作成した際, 可能な限りシークの呼び出しを避けることによる効率の改善を経験している.

4.2.2 遅延処理

次に, データを常に読み込んで処理するのではなく, '前方向のみ' のシークを導入し, 可能な限り処理を遅延する. これはごく基本的な遅延評価の例であり, 多くの論文であまり強調されていない. しかし, 後述する結果で示されるように大規模なデータを処理する場



ここには幾何形状とバウンディングボックスについてのみ示している．(a) 入力データは, 幾何形状およびバウンディングボックスそれぞれのファイルに変換される．幾何形状のファイルのレコードは, 形状の種類を示す ID と実際のデータからなる．バウンディングボックスのレコードは, 対応する幾何形状データのオフセットと, バウンディングボックス自体のデータからなる．(b) バウンディングボックスが逐次的に併合され, バウンディングボックス階層を構成する．上位のファイルの各レコードは, 子供に対する終端オフセットを含む.

図 4.1 ディスク上の格納形式

合には大きな影響があり, より重視されるべき点である.

ここで導入する遅延処理には, 各物体ごとのものとバウンディングボックス階層を利用した 2 つがある．次節でこれらについて述べる.

■幾何形状に対する遅延処理 先に述べたように, 交点計算においてある物体の幾何形状が必要とされるのは, その物体のバウンディングボックスがいずれかの光線と交差する場合のみに限られる．そこで, 交点計算を**アルゴリズム 4.3**のように変更する．一番内側の **if** 文の処理に注目してほしい．このように本当に幾何形状が必要となるまで読み込みを遅延している.

シェーディングについても同様に**アルゴリズム 4.4**のように変更する．この変更により, シェーディングの効率は物体数ではなく, ほぼ光線数のみに依存するようになる.

■バウンディングボックス階層による遅延処理 幾何形状とバウンディングボックスを分離して扱うとしても, バウンディングボックス自体に対する処理の効率は, 物体数 n に対して $O(n)$ のままである．この効率を改善するため, バウンディングボックスを逐次的に併合することによって得られるバウンディングボックス階層を導入する．もし, ある階層レベルにあるバウンディングボックスを何らかの処理において省くことができれば, その

アルゴリズム 4.3 交点計算 (改善手法/遅延処理)

```
while ディスク上に未処理の光線が存在する do
  ...
  for all ディスク上の物体 do
    バウンディングボックスをディスクから読み込む
    バウンディングボックスと重なるボクセルを決定
    for all 重なるボクセル do
      for all ボクセルに登録されている光線 do
        if 光線のメイルボックス ≠ 物体の ID then
          if 前回登録されたボクセルが範囲外 then
            if 光線とバウンディングボックスが交差 then
              if 幾何形状が読み込まれていない then
                幾何形状ファイル上でシーク
                幾何形状をディスクから読み込む
              fi
              光線と物体との交点計算を行なう
              光線の交点情報を更新
              光線のメイルボックス ← 物体の ID
            fi
          fi
        fi
      od
    od
  od
  ...
od
```

アルゴリズム 4.4 シェーディング (改善手法/遅延処理)

```
while ディスク上に未処理の光線が存在する do
  ...
  for all メモリ中の交点情報 do
    if 交点情報中の物体 ID に対するデータが読み込まれていない then
      幾何形状やシェーディングなど, 各種データファイル上でシーク
      データをディスクから読み込む
    fi
    シェーディングの計算を行なう
    局所的な輝度への寄与と次世代の光線をディスクに書き出す
  od
od
```

子孫にあたるすべてのバウンディングボックスや幾何形状についての処理も省くことができる. 逆に省くことができない場合, 先祖にあたるバウンディングボックスに対する処理は無駄となるが, 全体の物体数に比べ, 先祖にあたるバウンディングボックスの総数はずっと小さいため, このオーバーヘッドもごく小さく抑えられる. 以上に基づいた新しいデータ格納形式が, **図 4.1**-b である.

1 つの階層レベルに対する併合は, 次のように行なう.

(1) '面積比閾値' T を $\left(s/M^{1/3}\right)^2$ とする. ここで, s はユーザが指定する定数, M は現在の階層レベルにおけるバウンディングボックス数を表す. グリッドの解像度は物体数 n に対して $n^{1/3}$ で決まるため, 最下層の併合を行なう際には, T はボクセルの各辺を s 倍した直方体に相当することになる.

(2) 各バウンディングボックスを読み込み, もし, $A/A_{all} \leq T$ であるなら併合する. ここで, A は併合の結果得られるバウンディングボックスの表面積, A_{all} はシーン全体に対するバウンディングボックスの表面積を表す.

上の手続きを繰り返すことで, バウンディングボックス階層を構成する. さらに, 次のようにして最上層のバウンディングボックス数を制限する.

- $M \leq m$ か $L \geq l$ である場合, T ではなく t を用いて上の手続きを 1 回だけ実行し, 階層の構築を終了する. ここで, m, l, t はユーザが指定する定数, L は次に (最終的に) 得られる階層レベルを表す.

上述の簡単な処理により, 小さなバウンディングボックスが空間的に近い集団ごとにまとめられ, 大きなバウンディングボックスが選り分けられる. また, 基本手法ではユーザによって指定されていた最上層のグルーピング (各グループが 1 つのグリッドに対応する) が自動化されることも優れている点の 1 つである. 一方, この処理で得られる階層は, あくまで逐次的な併合を行なっているだけなので最適なものではない. しかし, 物体のストリーム, つまり, モデリングプログラムの出力は, 特に大量の物体が存在する場合, 高い空間的局所性をもった分布をしていることが多い. また, 上のような処理を採用することで, アルゴリズムを即時モード型のままに保つことができる.

もちろん, 上の処理のさらに前段において物体を並べ替えることもできる. 大量の物体の再配置は余分な時間と空間を必要とするが, 空間的局所性を増大させ, 最終的な処理時間を短縮する可能性もある. 再配置のためには, たとえば, Pharr らの手法で記述されているようなキャッシュシステムのためのデータベース [Phar97], R 木 [Gutt84, Leut97] などの応用が考えられる. また, メモリ上で動作する通常のレイトレーシングのための高速化手法の多くも, データをメモリに格納可能な大きさごとにグループ化し, 各グループを個別に処理することで応用できる.

4.3 グリッド解像度の制限に対する改善

一様空間分割の効率はグリッドの解像度に依存しており, 効率的な分割に必要なメモリは物体数に比例して増加する. これは典型的な空間と時間のトレードオフである. もし, 10 億の物体を扱うなら, グリッドの一辺の解像度は 1,000 となり, 10^9 個のボクセルを保持することになる. オクトリーなどの非一様空間分割を利用すれば, この問題を軽減することはできるが, メモリ内のみでの空間分割では, この問題を解決することはできない. たとえば, 10 億個の小さな球が, 格子状に一定間隔で分布している状況を扱うにはどうすればよいだろうか.

一様空間分割は, その単純さと効率のよさにおいて魅力的である. そこで, ここでは問題を一様空間分割に対して直接的に解決することを考え, 非一様な状況は, 引き続き複数のグリッドによって扱う. なお, 本研究では深く検討していないが, 非一様な状況に対する他の有効な手法としては集積物体[*1][Kirk88] の利用も考えられるだろう. つまり, 空間全体に対しては一様空間分割を利用し, 空間的に集中している物体群を集積物体として扱うのである. この場合においても, ここで述べる改善は組み合わせて利用することができる.

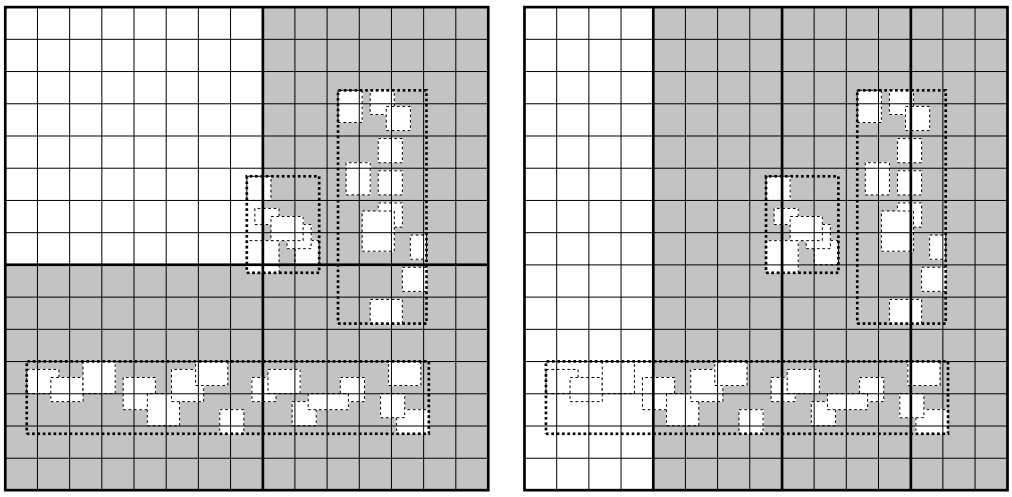
次節では, 前処理と交点計算それぞれについて, どのように解像度の制限を取り除くかを述べる. なお, 交点計算については, トラバースと実際の交点計算の 2 つの段階に分けて論じる.

4.3.1 前処理

前処理において, 基本手法ではグリッド全体の空間を一度に処理していたが, これを部分空間ごとの処理に変更する. たとえば, $1,000 \times 1,000 \times 1,000$ のグリッドは, $500 \times 500 \times 500$ の 8 つの部分空間に分けて処理するのである. それぞれの結果を併合することで, 最終的にグリッド全体に対する結果が得られる.

この手続きはシーンデータに対する複数のパスを新たに要求するが, それぞれのパスのコストは, バウンディングボックス階層を利用することで削減できる. もし, ある部分空間が小さくなれば, 多くのバウンディングボックスがその部分空間と重なりを持たなくなる. そこで, もし上位の階層に位置するバウンディングボックスが重なりを持たなければ, その子孫にあたるすべてのバウンディングボックスの処理を省くことができる. なお, 注意しなければならないのは, バウンディングボックスは, グリッド全体のバウンディングボックスの一辺に相当する長さの辺を持っているかもしれないという点である. このよう

[*1] Aggregate Object. 複数の物体を高速化手法によるデータ構造に格納し, 全体を単独の物体として扱う手法.



(a) 立方体状の分割 (太い実線) は, 非立方体状のバウンディングボックス (太い点線) のカリングの可能性を高める. (b) たとえば層状の分割を行なうと, 下にあるバウンディングボックスに対してカリングを行なうことができない.

図 4.2 前処理における部分空間への分割

なバウンディングボックスのカリングの可能性を高めるため, 部分空間への分割は立方体状になるように行なわれる (**図 4.2**).

4.3.2 トラバース

基本手法では各光線を読み込むと同時に, ボクセルをトラバースしていた. これは実装を容易にするものの, 高解像度グリッドではスラッシングを引き起こすことになる. これを解決するため, まず**アルゴリズム 4.5**のように交点計算の処理を変更し, 個々の光線に対するトラバースの回数を減らす.

重要なのは, 次のパスにおける最初の候補ボクセルが 'あらかじめ' 求められている点である. 各光線が次のパスまで生き残るかどうかを改めてトラバースすることなく判定できるため, 最後の **for** ループではトラバースをせずに済んでいる. この変更はやや複雑にみえるかもしれないが, いわゆる '番兵' の簡単な応用である (次のパスにおける最初の候補ボクセルが番兵にあたる).

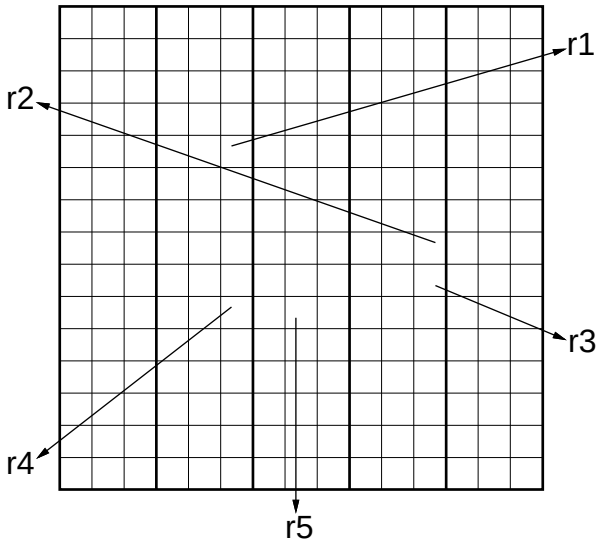
次に実際のトラバースの処理において, スキャンライン法と同様の処理により, ディスク上に格納されたボクセルに対するランダムアクセスを回避する. つまり, グリッド全体をメモリに格納できるブロックに層状に区切り, 各ブロックを順に (ちょうどスキャンライン法でスキャンライン単位の処理をするのと同様に) 処理する. **アルゴリズム 4.5**中の "··· トラバースを行なう" という部分の実際の処理は**アルゴリズム 4.6**のようになる

アルゴリズム 4.5 交点計算 (改善手法/トラバース回数の削減)

```
{ 初期化 }
while ディスク上に未処理の光線が存在する do
  for all ディスク上の光線 do
    while 光線でメモリが満たされていない do
      光線をディスクから読み込む
      3DDDA を初期化
    od
    メモリ中の光線に対し, 最初の候補ボクセルが見つかるまでトラバースを行なう
    for all メモリ中の光線 do
      if 最初の候補ボクセルがある then
        そのボクセル座標と光線の ID をディスクへ書き出す
      fi
    od
  od
od
{ メインループ }
while ディスク上またはメモリ中に未処理の光線が存在する do
  for all ディスク上の光線 do
    while 光線でメモリが満たされていない do
      最初の候補ボクセルがある光線をディスクから読み込む
      光線をディスクから読み込む
      3DDDA を初期化
    od
    メモリ中の光線の本数から登録数を決定
    メモリ中の光線に対し, 次のパスにおける最初の候補ボクセルが見つかるまでトラバース
    を行なう
  od
  for all ディスク上の物体 do
    ...
  od
  for all メモリ中の光線 do
    if 交点情報が完全 then
      交点情報をディスクに書き出す
    fi
  od
od
```

(**図 4.3**).

以上の方法により, ディスク上に格納されたボクセルに対し, 最大で 2 回の参照しか必要とせず, また参照を逐次的に行なうことができる. なお, 必要なメモリをさらに削減するため, 複数の軸に対する分割を行なうことも考えられるが, これにはディスクに対するより多くの参照が必要である.



r1 から r5 の光線があるとする．各光線は, その現在のボクセル座標に対応するブロック (太線で示されている) に登録される．次に, これらのブロックが左右両方の順に処理される．r1 と r3 は左から右のパスで, r2 と r4 は右から左のパスで処理される．r5 はどちらのパスで処理してもよい．

図 4.3 層状の分割を利用したトラバース

アルゴリズム 4.6 交点計算 (改善手法/ブロック単位のトラバース)

```
光線を X 軸上の各ブロックに登録する
for all ブロック (左から右) do
  ブロックのボクセルをディスクから読み込む
  for all ブロック中の光線 do
    ボクセルのトラバースを行ない, 候補ボクセルの座標を記録
    if ブロックの外へ出た then
      次のブロックに光線を登録
    fi
  od
od
for all ブロック (右から左) do
  上の for ループと同じ処理を行なう
od
```

4.3.3 交点計算

トラバースの後, 光線を格納するためのボクセルをどのように保持すべきだろうか．簡単な方法としては, 前処理と同様のものが考えられる．つまり, 部分空間ごとにボクセルの保持とすべての物体に対する処理を繰り返し, 最後にすべての結果を併合する．これによって正しい結果を得ることはできるが, 交点計算では光線や交点情報を保持するための

領域が必要であり, 部分空間をより小さくとらなければならない. これは結果として物体に対するパスを大きく増加させることになる.

実際には光線を保持するボクセルは全体のごく一部であり, ほとんどのボクセルは空である. そこで, 部分空間への分割による処理ではなく, 2 階層になったグリッドの利用を考える. たとえば, $1,000 \times 1,000 \times 1,000$ のグリッドを, $100 \times 100 \times 100$ の上位グリッドと, それらに格納される $10 \times 10 \times 10$ の下位グリッドとして表現する. この方法は Jevans らによる高速化手法 [Jeva89] に似たものであるが, 物体ではなく, 光線をボクセルに登録する点が異なる.

このような 2 階層グリッドにより, 必要なメモリ量を大きく削減できるが, それだけではすべての場合に十分であるとはいえない. たとえば, すべての上位ボクセルに対してなんらかの光線が含まれている場合を考えよう. 光線が含まれている上位ボクセルに対し, 単純に下位のグリッドの割り当てを行なえば, 以前と同様に大量のメモリが必要になってしまう. そこで, 次のような方法によって問題を解決する. この方法がうまく動作するのは, 上位ブロックのうち, 大量の光線を持つものはごく一部に限られるためである.

(1) 一定数の下位グリッドをメモリに割り当てる.
(2) 各上位グリッドごとに格納される光線を数える.
(3) 含まれている光線の本数が最も多い上位グリッドから順に下位グリッドを割り当てる. ここで, 下位グリッドの個数には制限があるため, すべての上位グリッドが下位グリッドを割り当てられるとは限らない.
(4) 光線を登録する. もし, 下位グリッドがある場合は, そのボクセルに登録する. そうでなければ, 光線はそのボクセル座標と共に, 上位グリッドに直接登録する.

上位ボクセルに直接登録された光線については, 各バウンディングボックスを読み込んで処理を行なう際, 光線が実際にバウンディングボックスに対するボクセルに属しているかどうかを調べなければならない. したがって, 上位ボクセルへの登録はオーバーヘッドを生むが, 上位ボクセルあたりの光線数が少ない場合には必ずしも不利とはいえない. なぜなら, 下位ボクセルの参照も, 下位グリッドにローカルな座標への変換など, 別のオーバーヘッドを持っているからである.

以上により, 必要なメモリ量を減らし, 最悪の場合でも必要なメモリ量を一定以下に限定でき, かつ速度の極端な低下を招くことのない方法が得られる. 上位, 下位グリッドの最適な解像度を求めるのは一般に難しいが, 本研究では上位ボクセルあたりの光線数を削減することを狙い, 次のような方法を採用することで良好な結果を得ている.

(1) 上位, 下位グリッドそれぞれの軸あたりの解像度 R_u, R_l が, ユーザによってこれらの最大値で初期化される.

(2) グリッドの解像度を Klimaszewski の方法 [Klim97] によって決め, 軸あたりの解像度の最大値を R とする.
(3) もし $R \leq R_u$ なら, 単階層グリッドを用いる.
(4) もし $R > R_u$ なら, まず $R_l = \min(R_l, \lceil R/R_u \rceil)$, つぎに $R_u = \min(R_u, \lceil R/R_l \rceil)$ として軸あたりの解像度を再定義し, 2 階層グリッドを用いる.

4.4 実験結果

本節では, これまでに述べたアルゴリズムの実装による実験結果を示す. 実装は個々の処理に応じて C++で記述したサブプログラムからなり, sray という名前のシェルスクリプトで統合している. 実装は **3.5 節**で用いたものを土台に新たに書き起こしたものであり, 条件を揃えるため, 以下で基本手法と呼んでいるものは古い実装ではなく, この新しい実装の改善部分を抑制したバージョンを作成して利用している. 実行環境は Linux を稼動している Pentium II 450MHz, メモリ 256MB の PC である. 生成画像の解像度など, 各種実験条件は SPD の指定に沿っている. 使用した SPD のバージョンは 3.13 で, シーン mount のコンパイルには `JENKINS_HASH` を指定した. なお, 改善についてのパラメータの設定は次の通りである.

- バウンディングボックス階層の構築 (**4.2.2 節**):
  $s = 3$, $n = 1,000$, $l = 4$ (最大で 5 階層). balls, gears, trees については $t = 0.5$ として 2 つのトップレベルボックスを, 他のものについては $t = 1.0$ として 1 つのトップレベルボックスを生成させた. 生成された階層はシーンおよびサイズファクタに依存しており, たとえば balls では 3, rings では 4 から 5 となっている.
- 2 階層グリッド (**4.3.3 節**):
  $R_u = 256$, $R_l = 4$.

最初の結果は, 比較的小規模なデータ (50,000,000 個/6GB まで) についてのものである. **表 4.1**は **4.2 節**で述べた改善——幾何形状とバウンディングボックスの分離 (S), 幾何形状に対する遅延処理 (G), バウンディングボックス階層による遅延処理 (B)——がどのように全体の計算時間に影響するかを示したものである. 各グリッドの解像度は **4.3.3 節**で述べた方法によって決められたが, これらのシーンでは単一階層グリッドが利用可能なため, 2 階層グリッドは用いられていない.

この結果から, まず S+G が 1.88 から 2.30 倍の速度の改善になっていることがわかる. 予測されるように, この改善はデータサイズに比例して大きくなっている. これに対し, B は効果があるときとそうではないときで大きな差があり, B にはバウンディングボックス

表 4.1 物体数 50,000,000 までのシーンに対する計算時間

	物体数	サイズ (MB)	生成時間	基本手法	S+G	S+G+B	
balls6	597,872	28	0:00:33	0:05:23	1.11	1.13	(1.02)
balls7	5,380,841	251	0:04:45	0:14:18	1.48	1.57	(1.06)
balls8	48,427,562	2,263	0:44:50	1:17:16	1.88	2.00	(1.07)
gears18	851,473	107	0:03:09	0:06:12	1.28	1.34	(1.05)
gears49	17,176,755	2,167	1:03:28	0:37:36	1.97	2.40	(1.21)
gears61	33,139,227	4,181	2:02:32	1:03:58	1.88	2.57	(1.37)
gears69	47,962,315	6,051	2:57:17	1:27:21	1.96	2.90	(1.48)
mount9	524,292	49	0:01:08	0:04:53	1.27	1.34	(1.06)
mount10	2,097,156	194	0:04:32	0:10:41	1.64	1.86	(1.13)
mount11	8,388,612	776	0:18:06	0:28:45	2.12	2.67	(1.26)
mount12	33,554,436	3,104	1:12:11	1:23:15	2.14	3.06	(1.43)
rings36	972,361	86	0:01:29	0:08:46	1.27	1.51	(1.19)
rings94	16,877,701	1,499	0:26:21	1:01:11	1.85	3.07	(1.65)
rings118	33,279,541	2,955	0:52:10	1:42:10	1.83	3.62	(1.98)
rings135	49,755,601	4,418	1:18:12	2:31:03	1.70	4.48	(2.63)
teapot124	998,448	126	0:03:45	0:06:24	1.58	1.70	(1.08)
teapot516	17,302,512	2,189	1:05:13	0:52:49	2.22	2.58	(1.17)
teapot719	33,596,713	4,250	2:06:42	1:25:21	1.99	2.57	(1.29)
teapot877	49,986,369	6,323	3:08:32	1:53:10	2.00	2.58	(1.29)
tetra9	262,144	24	0:00:33	0:01:03	1.22	1.24	(1.02)
tetra10	1,048,576	97	0:02:13	0:01:52	1.52	1.60	(1.05)
tetra11	4,194,304	388	0:08:53	0:05:23	2.02	2.19	(1.09)
tetra12	16,777,216	1,552	0:35:32	0:17:37	2.30	2.53	(1.10)
tree17	524,287	47	0:00:42	0:03:53	1.05	1.04	(0.99)
tree19	2,097,151	186	0:02:55	0:05:17	1.14	1.14	(0.99)
tree21	8,388,607	744	0:11:42	0:09:44	1.31	1.31	(1.00)
tree23	33,554,431	2,880	0:46:30	0:25:12	1.38	1.39	(1.00)

1 列目のラベルはシーン名とサイズファクタを表す. ‘生成時間’ は SPD のコマンドの実行とバイナリ形式への変換の計算時間, ‘基本手法’ は何も改善を加えない場合の計算時間 (時間:分:秒) を表す. 次の 2 列は ‘基本手法’ に対する計算時間の改善率を示す. つまり, 基本時間を改善率で割ったものが, 実際の計算時間になる. ‘S’ は ‘幾何形状とバウンディングボックスの分離’ を, ‘G’ は ‘幾何形状に対する遅延処理’ を, ‘B’ は ‘バウンディングボックス階層による遅延処理’ を表す. ‘S+G+B’ の前処理は, バウンディングボックス階層の構築時間も含む. 括弧内の値は, ‘S+G+B’ の ‘S+G’ に対する改善率を示す.

表 4.2 2 階層グリッドのオーバーヘッド

	64/2	32/4
balls8	0.89	0.90
gears69	0.90	0.86
mount12	0.81	0.86
rings135	0.93	0.83
teapot877	0.89	0.92
tetra12	0.84	0.77
tree23	0.93	0.91

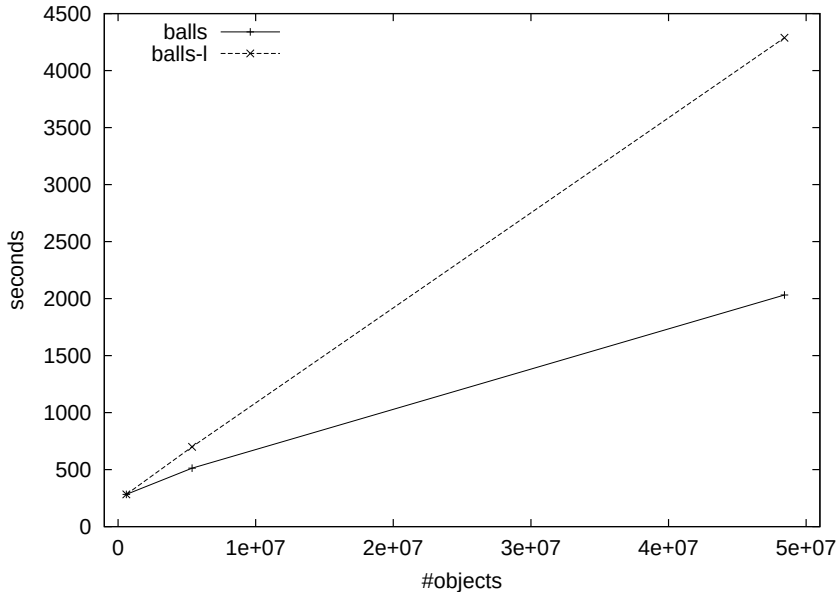
表 4.1のデータに対し, 人為的にグリッドの解像度に制約を与えた結果. 数値は単階層グリッドに対する改善率を示す. 単階層グリッドの一辺の最大解像度は 128 である. これに対し, '64/2' および '32/4' は 2 階層グリッドを表し, それぞれ上位/下位グリッドの最大解像度が 64/2 および 32/4 であることを示している.

階層の構築というオーバーヘッドもある. このため, S+G+B の結果の変動は大きく, 基本手法に対する改善率は 1.39 から 4.48 となっている. しかし, S+G に対する最悪の改善率が 0.99 にとどまっていることからもわかるように, B は効果がない場合でも, それほど大きなオーバーヘッドにはならない. これは, 最下層より上位にあるバウンディングボックスの総数が, 物体数に比べて遥かに少ないからである.

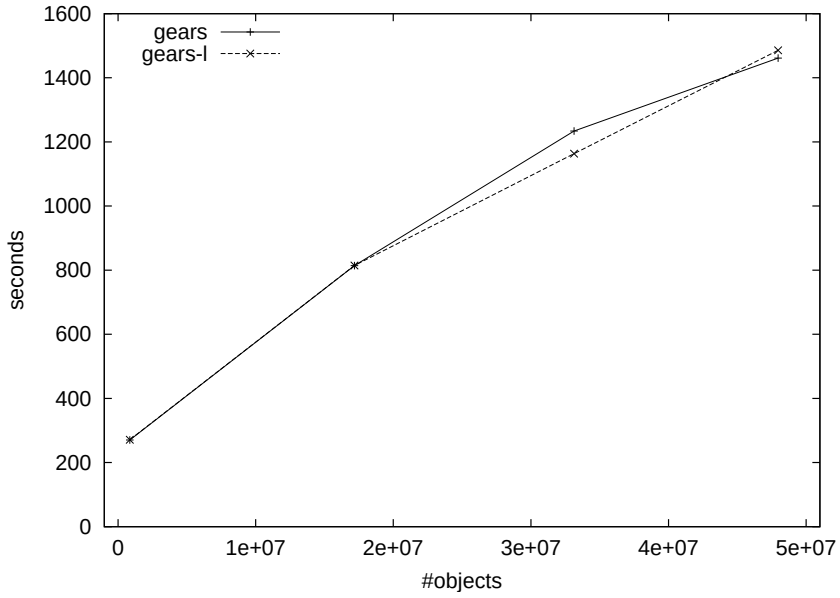
図 4.4から**図 4.7**, および**表 4.2**は, 高解像度グリッドの有無による差を示している (高解像度グリッド以外の改善はすべて適用した). **図 4.4**から**図 4.7**を見ると, 解像度が制限された単階層グリッドでは計算時間が途中から線形に増加しているのに対し, 高解像度グリッドでは緩やかな増加にとどまっていることがわかる. gears のように高解像度グリッドが単階層グリッドより遅い結果を生じている場合もあるが, 計算時間の増加傾向は同様であり, いずれ高解像度グリッドの方が速くなるであろうことが予測できる.

このように絶対時間において高解像度グリッドの方が遅い場合があるのは, **表 4.2**に示されているように 2 階層グリッドには単階層グリッドにはない階層を扱うオーバーヘッドが存在するからである. この表から, 2 階層グリッドは単階層グリッドの場合の 1.1 から 1.3 倍の時間がかかっていることがわかる. もちろん, このオーバーヘッドは大規模なシーンに対して高解像度グリッドを実現できることで報われる.

最後に, 大規模なデータに対する結果を示す. **表 4.3**および**表 4.4**は**図 4.9**の画像を生成した際の結果を示している. また, **図 4.8**は, これらの表の内容を棒グラフで示したものである. ここで, rings368 と rings368m の違いは視野の情報のみであり, rings368m では全物体のうち約 1,000,000 個だけが視野に含まれている. また, S+G および S+G+B の効果を明確にするため, これらの画像を生成する際には, 基本手法についても高解像度グリッドの改善を適用している.



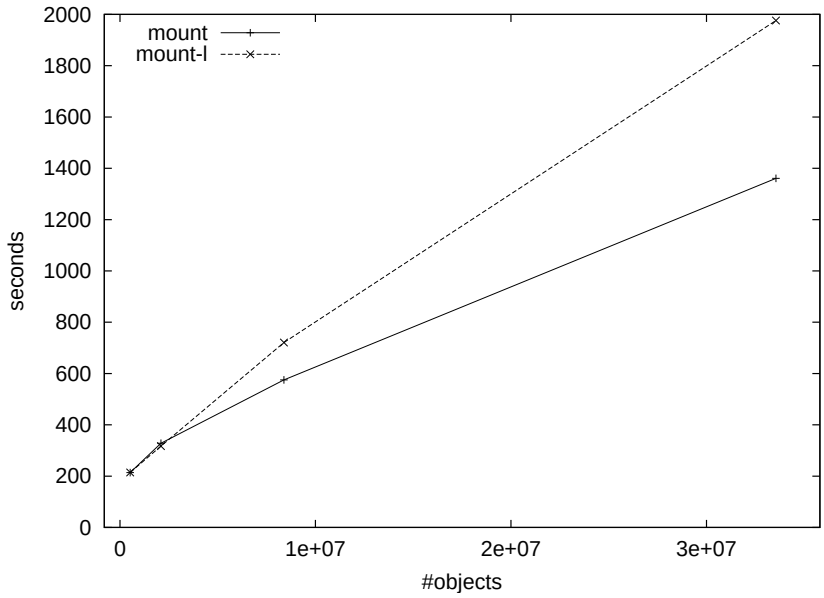
balls



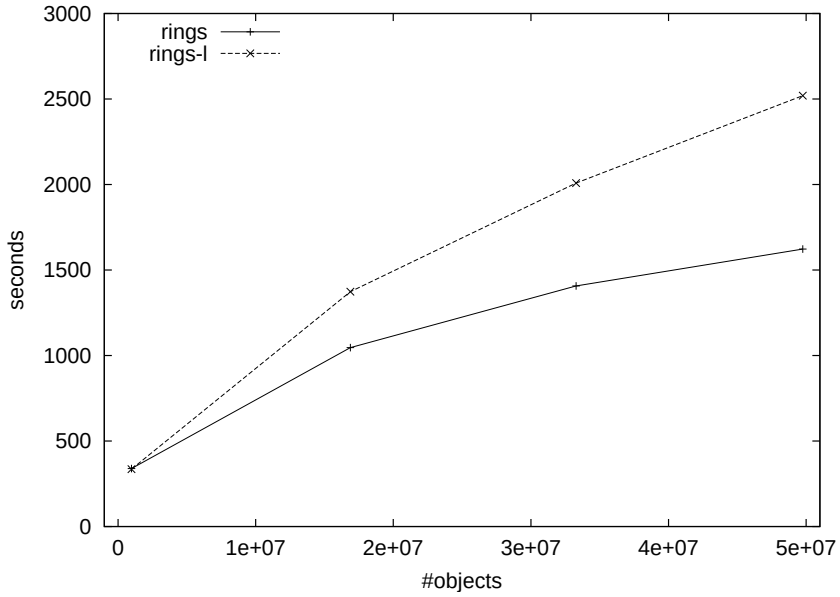
gears

サフィックスがないものが高解像度グリッド, '-l' がついているものが一辺あたりの最大解像度を 128 に制限した単階層グリッドである.

図 4.4 高解像度グリッドの有無による計算時間の変化



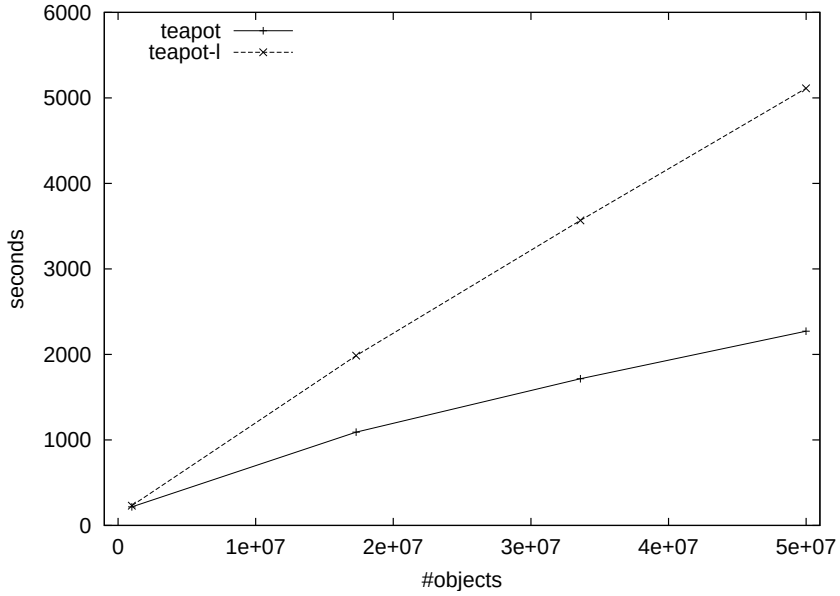
mount



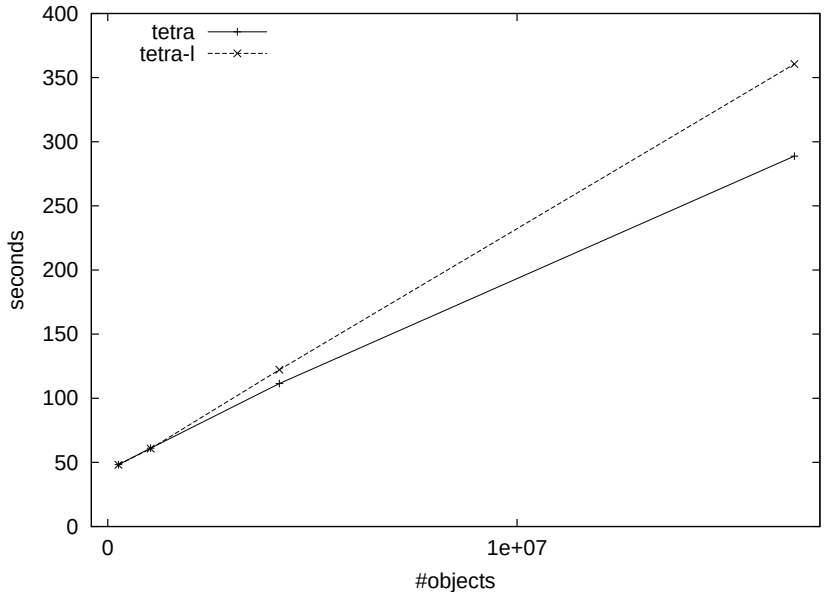
rings

グラフの形式については**図 4.4**を参照.

図 4.5 高解像度グリッドの有無による計算時間の変化 (続き 1)



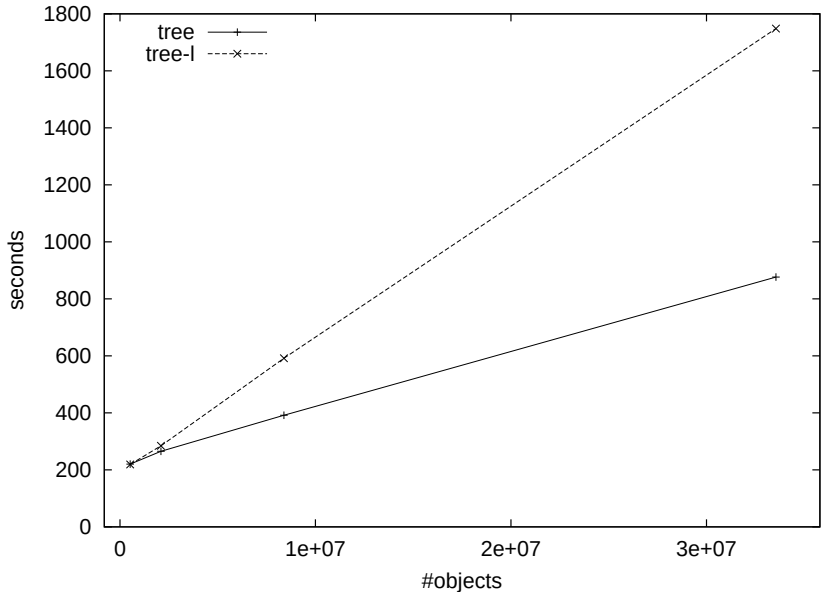
teapot



tetra

グラフの形式については**図 4.4**を参照.

図 4.6 高解像度グリッドの有無による計算時間の変化 (続き 2)



tree

グラフの形式については**図 4.4**を参照.

図 4.7 高解像度グリッドの有無による計算時間の変化 (続き 3)

表 4.3 大規模データに対する計算時間 (全体)

	物体数	サイズ (MB)	生成時間	基本手法	S+G	S+G+B	
mount14	536,870,916	49,664	19:16:15	19:26:15	2.04	5.33	(2.62)
rings368	1,000,787,041	88,857	26:37:42	50:54:40	1.84	12.43	(6.76)
rings368m	1,000,787,041	88,857	26:37:42	30:40:26	1.65	13.65	(8.26)

表の形式については**表 4.1**を参照.

表 4.4 大規模データに対する計算時間 (前処理/トレーシング)

	前処理				トレーシング			
	基本手法	S+G	S+G+B		基本手法	S+G	S+G+B	
mount14	6:31:37	2.10	8.87	(4.23)	12:54:37	2.01	4.44	(2.21)
rings368	11:03:30	1.82	5.61	(3.09)	39:51:10	1.84	18.76	(10.17)
rings368m	11:03:30	1.82	5.61	(3.09)	19:36:56	1.57	70.78	(45.01)

表の形式やデータの内容については, **表 4.1**, **表 4.3**を参照.

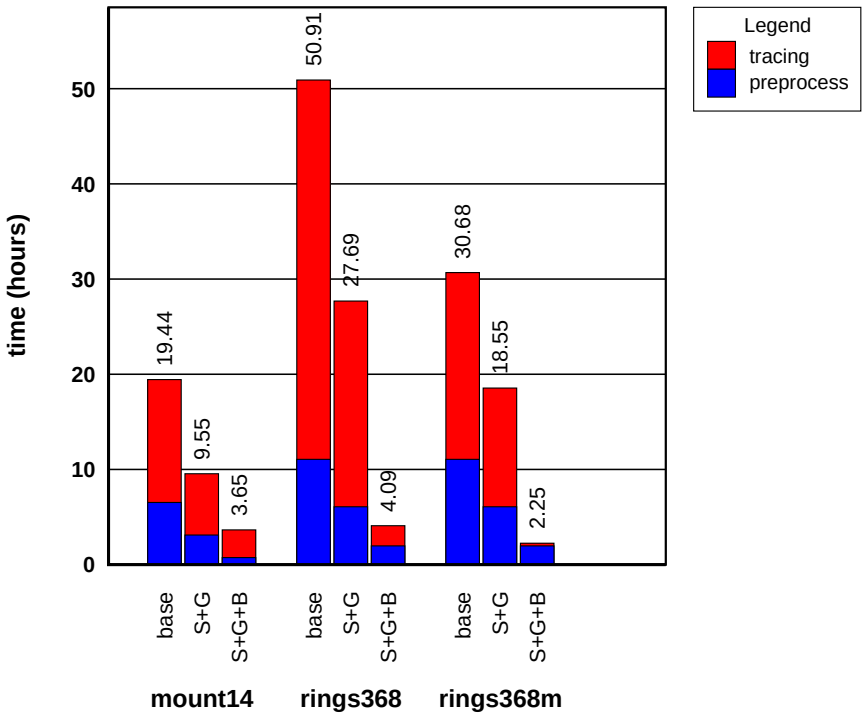


表 4.1, **表 4.3**を棒グラフで示したものである．各シーンごとに左が ‘基本手法’, 中央が ‘S+G’, 右が ‘S+G+B’ を示し, 青が前処理, 赤がトレーシングの時間を示している．

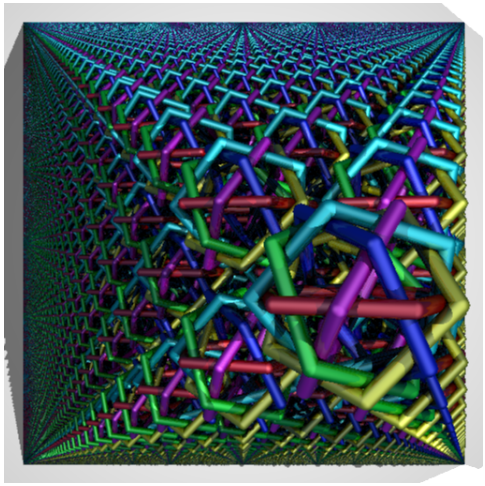
図 4.8 大規模データに対する計算時間

これらの結果から, 大規模データでは改善手法によって計算時間が大きく短縮されることがわかる．改善率は全体で 5 から 14 に及び, 特に rings368m の結果は象徴的である．rings368m では全物体のうち, 0.1%程度だけが視野に入っており, 大部分の物体についての処理は B によって行なわれずに済む．このため, 基本手法で 19.6 時間, S+G でも 12.5 時間かかっているトレーシングの時間が, S+G+B では 17 分で済んでいる．また, 先に述べたように高解像度グリッドはいずれの場合も適用しているが, もしそうしなければ, いずれの改善率もさらに 10 倍以上になるだろう．

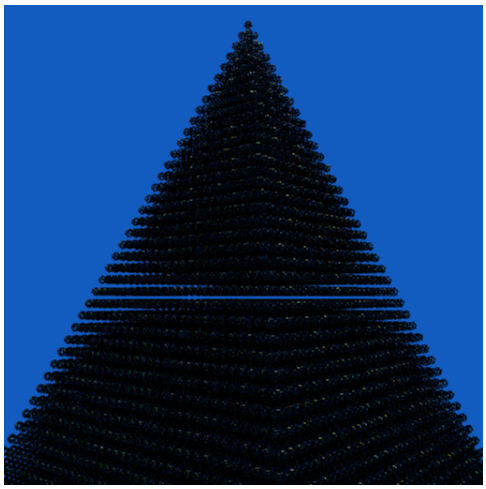
なお, B による処理は, 視野に入る物体のみを選び出す単純なカリングとは異なる．rings では反射や屈折を生じる面があり, 視野の外に出て行く光線も追跡されるため, 単純なカリングでこの状況を扱うことはできない．

この他, 生成時間を見てみると, 極めて長い時間を必要としている．これは SPD の指示に厳格に従った結果であるが, このことから大規模データを効率よくレンダリングするため, モデリングシステムとレンダリングシステムがより緊密に協調することが重要であるといえる．

mount14



rings368



rings368m

rings368 および rings368m では**図 3.9**とは異なり，反射した光線についても計算している．

図 4.9 大規模データに対する生成画像

第 5 章

結 論

本論文では, 大規模データに対するレイトレーシングを扱った. 以下では, 本論文における成果と今後の研究について述べる.

5.1 成 果

本論文では, 大規模データを処理するために必要な考え方とこれまでの研究について整理し, これらに基づいて新しい手法を構築した.

最初に示した基本手法では, 幅優先レイトレーシングの本質は光線とシーンデータの役割を入れ替えることにあること, したがって, 主流の高速化手法を組み合わせられることを示し, 具体的には一様空間分割と組み合わせたアルゴリズムを考案した. そして, このアルゴリズムの詳細として, メモリの消費量や交点計算の効率を考えた光線の部分的な登録処理を示した. 次に, 空間分割の解像度の増加に伴うシーンデータ参照回数の増加を避けるため, 登録数の動的な変更とボクセル境界での登録を導入した. また, 無駄な交点計算を削減するため, 過去のボクセル位置を利用した判定と, メモリを光線によって可能な限り満たす処理を導入した. シェーディングについては, 必要な処理を交点計算から完全に分離し, 物体順に光線および交点情報をソートする処理を示した. これらから, 全過程においてディスク上のデータへの逐次的なアクセスを維持するアルゴリズムを得た.

次に示した改善手法では, 基本手法の問題点を改めて検討し, 主要な問題点として無駄なデータ処理と空間分割の解像度の制限を挙げた. そして, これらを改善するため, まず, 無駄なデータ処理に対し, 基本的な遅延評価とバウンディングボックス階層によるカリングを導入した. 次に, 高解像度グリッドを実現するため, 前処理およびトラバースでは 2 次記憶化したグリッドを導入し, 実際の交点計算の段階では 2 階層グリッドを導入した. この高解像度グリッドの実現には, 無駄なデータ処理に関する改良, 特にバウンディングボックス階層によるカリングを用いた効率のよい処理が含まれる. これらから, 基本手法

の頑健さを保ったまま大きく高速化したアルゴリズムを得た.

性能評価は, それぞれの手法についての完全な実装を行い, 公開された実験用シーンデータを用いて行なった. 基本手法については, 通常のレイトレーシングの実装との比較により, 手法固有のオーバーヘッドを調べた. また, 通常のレイトレーシングと比べ, 任意のデータに対しても常に安定して動作することを示した. これらの比較により, 次の改善手法を含むすべての結果を, 既存の高速化手法や将来の研究と比較することが可能である. 改善手法については, まず基本手法との比較的小規模なデータを用いた比較により, それぞれの改良を検証した. また, 異常ともいえる大規模なデータについても比較を行った. 以上により, 本研究は, 他の手法では扱うことが困難ないしは不可能と考えられるようなデータにおいても頑健かつ高速な処理が可能であることを示した.

本研究の成果を用いることで, レイトレーシングだけが扱える光学的効果を保ちながら任意の複雑さを持つシーンをレンダリングすることができる. このように質と量の双方において, ユーザに妥協を強いることがない. これが本研究の最も重要な価値である. 本研究の成果は広範な領域に適用することができるが, 大規模な建築物に対する照明シミュレーションや映画の特殊効果など, シーンの複雑さと光学的効果の双方に対して高い要求のある分野で特に有効である.

5.2 今後の研究

本論文では, 形状をレンダリング時に必要になった時点で生成する手続き的オブジェクトや, 形状の複製を利用するインスタンシング, ボリュームデータによる近似などは用いていない. これは実験データで定められている実験手続きに厳格に従った結果であるが, これらの手法はデータの総量を抑えながら複雑なシーンを作ることを可能とするので, 実用的なシステムを作る上では導入を検討すべきである. 手続き的オブジェクトなどの導入は, 本論文の手法が Z バッファや REYES と同様の構造をもっていることから容易であるが, スクリーン上の投影面積のような単純な指標がレイトレーシングには存在しないため, 物体の LOD (Level of Detail: 詳細度) を常に高くしなければならない. したがって, レイトレーシングおける LOD を適切に求める手法の確立が重要である. このような手法が確立されれば, ボリュームデータなどによる非可逆の圧縮や近似も, 誤差を適切に抑えながら行なうことができる.

実験手続きに従ったことによるもう 1 つの結果が, 長時間におよぶデータ生成と前処理である. もし, シーンのさまざまな情報を維持するデータベースが存在し, そうした情報をあらかじめ知ることができれば, こうした処理にかかる時間は大幅に短縮できると考えられる. また, シーンのさまざまな情報は, 実際にレイトレーシングを行なう際にも有効

である. このようなレンダリングをサポートするシステムは特定の用途では作られているが, 十分に汎用性のあるものは未だ存在しない. 手続き的オブジェクトなどの技術も包含した, 高機能, 汎用なシーンデータベースの研究は, これからの興味深い課題である.

参考文献

[Aman87] Amanatides, J. and A. Woo: A Fast Voxel Traversal Algorithm for Ray Tracing, in *Eurographics '87*, pp. 3–10, 1987.

[Apod00] Apodaca, A. A. and L. Gritz: *Advanced RenderMan*, Morgan Kaufmann Publishers, 2000.

[Arna87] Arnaldi, B., T. Priol, and K. Bouatouch: A New Space Subdivision Method for Ray Tracing CSG Modelled Scenes, *The Visual Computer*, Vol. 3, No. 2, pp. 98–108, 1987.

[Arvo89] Arvo, J. and D. Kirk: A Survey of Ray Tracing Acceleration Techniques, in Glassner, A. S. ed., *An Introduction to Ray Tracing*, pp. 201–262, Academic Press, 1989.

[Bado95] Badouel, D., K. Bouatouch, and T. Priol: Distributing Data and Control for Ray Tracing in Parallel, *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, pp. 69–77, 1995.

[Catm74] Catmull, E.: A Subdivision Algorithm for Computer Display of Curved Surfaces, Technical Report UTEC-CSc-74-133, University of Utah, 1974.

[Cook84] Cook, R. L., T. Porter, and L. Carpenter: Distributed Ray Tracing, in Christiansen, H. ed., *SIGGRAPH '84 Conference Proceedings*, Annual Conference Series, pp. 137–145, ACM Press, 1984.

[Cook87] Cook, R. L., L. Carpenter, and E. Catmull: The Reyes Image Rendering Architecture, in Stone, M. C. ed., *SIGGRAPH '87 Conference Proceedings*, Annual Conference Series, pp. 95–102, ACM Press, 1987.

[Deus98] Deussen, O., P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz: Realistic Modeling and Rendering of Plant Ecosystems, in Cohen, M. ed., *SIGGRAPH '98 Conference Proceedings*, Annual Conference Series, pp. 275–286, Addison Wesley, 1998.

[Fuji86] Fujimoto, A., T. Tanaka, and K. Iwata: ARTS: Accelerated Ray Tracing System, *IEEE Computer Graphics and Applications*, Vol. 6, No. 4, pp.

16–26, 1986.

[Funk96] Funkhouser, T. A.: Coarse-Grained Parallelism for Hierarchical Radiosity Using Group Iterative Methods, in Rushmeier, H. and T.-M. Rhyne eds., *SIGGRAPH '96 Conference Proceedings*, Annual Conference Series, pp. 343–352, Addison Wesley, 1996.

[Gaud88] Gaudet, S., R. Hobson, P. Chilka, and T. Calvert: Multiprocessor Experiments for High Speed Ray Tracing, *ACM Transactions on Graphics*, Vol. 7, No. 3, 1988.

[Glas84] Glassner, A. S.: Space Subdivision For Fast Ray Tracing, *IEEE Computer Graphics and Applications*, Vol. 4, No. 10, pp. 15–22, 1984.

[Gold87] Goldsmith, J. and J. Salmon: Automatic Creation of Object Hierarchies for Ray Tracing, *IEEE Computer Graphics and Applications*, Vol. 7, No. 5, pp. 14–20, 1987.

[Gree89] Green, S. A. and D. J. Paddon: Exploiting Coherence for Multiprocessor Ray Tracing, *IEEE Computer Graphics and Applications*, Vol. 9, No. 6, pp. 12–26, 1989.

[Gutt84] Guttman, A.: *R*-trees: a dynamic index structure for spatial searching, *SIGMOD Record (ACM Special Interest Group on Management of Data)*, Vol. 14, No. 2, pp. 47–57, 1984.

[Hain87] Haines, E. A.: A Proposal for Standard Graphics Environments, *IEEE Computer Graphics and Applications*, Vol. 7, No. 11, pp. 3–5, 1987, `http://www.acm.org/pubs/tog/resources/SPD/overview.html`.

[Hain88] Haines, E. A. and M. VandeWettering: Sorting Unnecessary on Shadow Rays for Kay/Kajiya?, *Ray Tracing News*, Vol. 1, No. 9, 1988, `http://www.raytracingnews.org/rtnews4b.html#art6`.

[Hain91] Haines, E. A.: Efficiency Improvements for Hierarchy Traversal in Ray Tracing, in Arvo, J. ed., *Graphics Gems II*, pp. 267–274, Academic Press, 1991.

[Hanr86] Hanrahan, P.: Using Caching and Breadth-First Search to Speed Up Ray-Tracing, in *Graphics Interface '86*, pp. 56–61, 1986.

[Hart91] Hart, J. C. and T. A. DeFanti: Efficient Anti-aliased Rendering of 3D Linear Fractals, in Sederberg, T. W. ed., *SIGGRAPH '91 Conference Proceedings*, Annual Conference Series, pp. 91–100, ACM Press, 1991.

[Jens98] Jensen, H. W. and P. H. Christensen: Efficient Simulation of Light Trans-

port in Scenes With Participating Media Using Photon Maps, in Cohen, M. ed., *SIGGRAPH '98 Conference Proceedings*, Annual Conference Series, pp. 311–320, Addison Wesley, 1998.

[Jeva89] Jevans, D. and B. Wyvill: Adaptive Voxel Subdivision for Ray Tracing, in *Graphics Interface '89*, pp. 164–172, 1989.

[Kaji89] Kajiya, J. T. and T. L. Kay: Rendering Fur with Three Dimensional Textures, in Lane, J. ed., *SIGGRAPH '89 Conference Proceedings*, Annual Conference Series, pp. 271–280, ACM Press, 1989.

[Kato01] Kato, T., H. Nishimura, T. Endo, T. Maruyama, J. Saito, and P. H. Christensen: Parallel Rendering and the Quest for Realism: The 'Kilauea' Massively Parallel Ray Tracer, pp. IV–1–59, 2001.

[Kay86] Kay, T. L. and J. T. Kajiya: Ray Tracing Complex Scenes, in Evans, D. C. and R. J. Athay eds., *SIGGRAPH '86 Conference Proceedings*, Annual Conference Series, pp. 269–278, ACM Press, 1986.

[Kirk88] Kirk, D. and J. Arvo: The Ray Tracing Kernel, in *Proceedings of Ausgraph '88*, pp. 75–82, 1988.

[Klim97] Klimaszewski, K. S. and T. W. Sederberg: Faster Ray Tracing Using Adaptive Grids, *IEEE Computer Graphics and Applications*, Vol. 17, No. 1, pp. 42–51, 1997.

[Kolb89] Kolb, C.: Rayshade: An Extensible System for Creating Ray-Traced Images, 1989, `http://graphics.stanford.edu/~cek/rayshade/`.

[Lamp90] Lamparter, B., H. Müller, and J. Winckler: The Ray-Z-Buffer — An Approach for Ray Tracing Arbitrarily Large Scenes, Technical report, Universitat Freiburg Institut fur Informatik, 1990.

[Law96] Law, A. and R. Yagel: Multi-Frame Thrashless Ray Casting with Advancing Ray-Front, in *Graphics Interface '96*, pp. 70–77, 1996.

[Leut97] Leutenegger, S. T., J. M. Edgington, and M. A. Lopez: STR: A simple and efficient algorithm for R-tree packing, Technical Report TR-97-14, Institute for Computer Applications in Science and Engineering, 1997.

[Max81] Max, N. L.: Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset, Annual Conference Series, pp. 317–324, ACM Press, 1981.

[Möll99] Möller, T. and E. Haines: *Real-Time Rendering*, A K Peters, Ltd., 1999.

[Müll86] Müller, H.: Image Generation by Space Sweep, in *Computer Graphics Fo-*

rum (Eurographics '86 Proceedings), Vol. 5, pp. 189–195, 1986.

[Müll92] Müller, H. and J. Winckler: Distributed Image Synthesis with Breadth-First Ray Tracing and the Ray-Z-Buffer, in Monien, B. and T. Ottmann eds., *Data Structures and Efficient Algorithms. Final Report on the DFG Special Initiative (v. 594 of Lecture Notes in Computer Science)*, pp. 124–147, Springer-Verlag, 1992.

[Naka97] Nakamaru, K. and Y. Ohno: Breadth-First Ray Tracing Utilizing Uniform Spatial Subdivision, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 3, No. 4, pp. 316–328, 1997, ISSN 1077-2626, `http://computer.org/tvcg/tg1997/v0316abs.htm`.

[Naka01] Nakamaru, K. and Y. Ohno: Enhanced Breadth-First Ray Tracing, *Journal of Graphics Tools*, Vol. 6, No. 4, pp. 13–28, 2001, ISSN 1086-7651, `http://www.acm.org/jgt/papers/NakamaruOhno01/`.

[Neyr98] Neyret, F.: Modeling, Animating, and Rendering Complex Scenes Using Volumetric Textures, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 4, No. 1, pp. 55–70, 1998.

[Phar96] Pharr, M. and P. Hanrahan: Geometry Caching for Ray-Tracing Displacement Maps, in Pueyo, X. and P. Schröder eds., *Eurographics Rendering Workshop 1996*, pp. 31–40, 1996.

[Phar97] Pharr, M., C. Kolb, R. Gershbein, and P. Hanrahan: Rendering Complex Scenes with Memory-Coherent Ray Tracing, in Whitted, T. ed., *SIGGRAPH '97 Conference Proceedings*, Annual Conference Series, pp. 101–108, Addison Wesley, 1997.

[Plun85] Plunkett, D. J. and M. J. Bailey: The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed, *IEEE Computer Graphics and Applications*, Vol. 5, No. 8, pp. 52–60, 1985.

[Rein99] Reinhard, E., A. Chalmers, and F. W. Jansen: Hybrid Scheduling for Parallel Rendering using Coherent Ray Tasks, in *1999 IEEE Parallel Visualization and Graphics Symposium (PVGS '99)*, pp. 21–28, ACM SIGGRAPH, 1999.

[Rubi80] Rubin, S. M. and T. Whitted: A 3-Dimensional Representation for Fast Rendering of Complex Scenes, in Thomas, J. J. ed., *SIGGRAPH '80 Conference Proceedings*, Annual Conference Series, pp. 110–116, ACM Press, 1980.

[Snyd87] Snyder, J. M. and A. H. Barr: Ray Tracing Complex Models Containing Surface Tessellations, in Stone, M. C. ed., *SIGGRAPH '87 Conference Proceedings*, Annual Conference Series, pp. 119–128, ACM Press, 1987.

[Tell94] Teller, S., C. Fowler, T. Funkhouser, and P. Hanrahan: Partitioning and Ordering Large Radiosity Computations, in Glassner, A. ed., *SIGGRAPH '94 Conference Proceedings*, Annual Conference Series, pp. 443–450, ACM Press, 1994.

[Veac97] Veach, E. and L. J. Guibas: Metropolis Light Transport, in Whitted, T. ed., *SIGGRAPH '97 Conference Proceedings*, Annual Conference Series, pp. 65–76, Addison Wesley, 1997.

[Voor91] Voorhies, D.: Space-filling Curves and a Measure of Coherence, in Arvo, J. ed., *Graphics Gems II*, pp. 26–30, Academic Press, 1991.

[Wald01] Wald, I., C. Benthin, M. Wagner, and P. Slusallek: Interactive Rendering with Coherent Ray Tracing, in Chalmers, A. and T.-M. Rhyne eds., *Computer Graphics Forum (Eurographics 2001 Proceedings)*, Vol. 20, 2001.

[Ward94] Ward, G. J.: The RADIANCE Lighting Simulation and Rendering System, in Glassner, A. ed., *SIGGRAPH '94 Conference Proceedings*, Annual Conference Series, pp. 459–472, ACM Press, 1994.

[安部 95] 安部, 大野：並列計算機のためのレイトレーシングアルゴリズム, テレビジョン学会誌, Vol. 49, No. 10, pp. 1266–1271, 1995.

[中丸 92] 中丸：レイトレーシングにおける大量のデータの扱いに関する研究, 工学修士論文, 慶応義塾大学 理工学研究科 計算機科学専攻, 1992.