修 士 論 文

並列演算に適したバウンディングボリューム階層によるレイトレーシングの高速化

北陸先端科学技術大学院大学
知識科学研究科知識システム基礎学専攻

木村 秀敬

2007 年 3 月

修 士 論 文

並列演算に適したバウンディングボリューム階層によるレイトレーシングの高速化

指導教官　　宮田 一乗 教授

北陸先端科学技術大学院大学
知識科学研究科知識システム基礎学専攻

550018 木村 秀敬

審査委員　宮田 一乗 教授（主査）
審査委員　杉山 公造 教授
審査委員　吉田 武稔 教授
審査委員　由井薗 隆也 准教授

提出年月: 2007 年 2 月

Abstract

Photorealistic rendering of computer graphics is an important issue for long years. Today, we can see great results of researches in movies, video games, and etc. Rasterization is an effective algorithm for rendering in interactive frame rates. However, it is an approximate algorithm, so it has a lack of photorealism. On the other hand, rendering techniques based on ray tracing are slower than rasterization, however, the quality of rendering is highly improved.

It is said that ray tracing was relatively slower than other rendering techniques. However, recent progress of computer makes it possible to render computer graphics by means of ray tracing in interactive frame rates. A lot of techniques to accelerate ray tracing are existing. Among them, spatial and hierarchical scene subdivision to decrease computation cost of ray-object intersection is most effective. The improvements of hardware are also contributing to accelerate ray tracing. The use of single instruction multiple data (SIMD) that enables parallel computation of multiple data is necessary for maximizing potential of recent CPUs.

This thesis describes an acceleration technique of ray tracing by means of recent CPU architectures. Bounding volume hierarchies (BVH) that are used to accelerate ray tracing are improved to make them applicable to SIMD. We construct BVH as quad-tree that is easy to apply SIMD while others mostly used binary-tree. Our techniques achieved shallower tree construction and lesser use of memory than binary-tree. Furthermore, traversal with parallel computation enabled fast ray tracing.

概要

3 次元モデルの写実的なレンダリングは，コンピュータグラフィックスの分野において重要な課題である．数十年に渡って積み重ねられてきた研究の成果は，目を見張るほど美しい映画やゲームの映像に見て取れる．現在，高速なレンダリングを目的としたシステムでは，ラスタライゼーションという手法が用いられている．しかし，ラスタライゼーションは近似的な手法であるため，物理的な正確さに欠ける．一方でオフラインレンダリングは，ラスタライゼーションより速度面では劣るが，レンダリングの品質は大幅に改善されている．オフラインレンダリングのうち多くの手法は，本論文のテーマとして取り上げるレイトレーシングを基礎としている．

レイトレーシングは処理に時間のかかる手法といわれてきたが，近年のコンピュータの発展に伴い，インタラクティブな速度でのレンダリングが可能になってきている．レイトレーシングを高速化するには様々なアプローチが存在する．その中でも，レイが物体に当たる位置を計算する際の計算量を少なくすることを目的とした空間データ構造の改善が重要な課題として取り上げられている．また一方では CPU の発展も進んでいる．複数データに対する並列演算を可能とする Single Instruction Multiple Data（SIMD）という技術は，近年の CPU を最大限に活用する上で必要な技術である．

本論文では，近年の CPU の機能を生かしたレイトレーシングの高速化について述べる．空間データ構造には Bounding Volume Hierarchies（BVH）を用い，これを CPU から利用しやすい形に改良する手法を述べる．従来までは BVH は 2 分木を用いて構築されることが多かったが，本論文では SIMD の機能を持つ CPU による効率的な探索を実現するために，4 分木による BVH を構築した．提案する手法によって構築された BVH は，階層の深さが浅くなり，消費するメモリも少なくなった．そして SIMD による並列演算機能を活用することにより，高速なレイトレーシングが可能となった．

謝辞: 2年間の研究を振り返って

2 年間の JAIST での研究生活を経て，レイトレーシングについての修士論文を執筆するというゴールに至ったことは，私にとって大変感慨深いものである．ここに至るまで，多くの方々とのインタラクションがあった．ここでは私の研究生活を振り返らせていただくとともに，私を支えていただいた方々に感謝の意を表したい．

2004 年の 12 月，突然の依頼にも関わらず，宮田一乘教授が私の研究室訪問を快く受け入れてくださった．まだ CG の初心者であった私の相談を親身に受けていただいたことで，私は JAIST への受験を決心した．インタラクション 2005 に伺ったときには当時の研究室の先輩方にも会い，春からの新生活が待ち遠しくなった．

そして入学式の翌日，私は仮配属期間にも関わらず，個性あふれる宮田研究室での研究生活を始めた．入学式翌日に最初に会った枡野君をはじめ，留学生だとは気づかなかった杜君と出会い，3 人体制かと思っていたところにやってきた武井君，益田君，垣内君，そしてさくら祭での出会いが忘れられない藤井君の合計 7 人が集まった．

宮田研 7 人のメンバーに，伊豫田君，伊藤君を加え，バーチャルリアリティコンテスト IVRC に向けたプロジェクトが走り出した．このプロジェクト，特に球魂は単なるコンテストへの出展だけにとどまらず，その先の国内，海外での出展へとつながり，私にとって大きな収穫となった．さらに，新たな人間関係を築くこともできた．球魂の製作や CEATEC での展示に関しては日立金属株式会社様に感謝したい．またフランスでの短期留学，Laval Virtual での展示に関しては ENSAM P&I 研究所のしらいあきひこ先生に感謝したい．

私の研究題目が決まるまでは紆余曲折があった．そもそも，中間発表での題目は今のそれとはかけ離れたものであった．中間発表の審査員の方々や，新しく研究室に加わった後輩を含む研究室の仲間からの助言があり，現在の題目へと変更した．もしあの時題目を変更していなかったら，私は修士論文を書き上げられた自信がない．助言を下さった方々には感謝したい．また個人的にレイトレーシングに関する助言を頂いた藤田将洋氏にも感謝したい．

最後に．宮田先生，題目を変えたり色々と振り回してしまい申し訳ありませんでした．寄り道は多かったですが，これでようやくゴールできそうです．2 年間の暖かい御指導ありがとうございました．そして研究室の友人たち，色々と楽しかったです．機会があったらまたどこかで会いましょう．

目 次

図 目 次

表 目 次

第1章　はじめに

本章では，コンピュータグラフィックスを用いて写実的な映像表現を目指した従来の研究成果について概説し，本論文が解決する課題について述べ，本論文の構成を紹介する

1.1　背景

3次元モデルの写実的な映像表現（レンダリング）は，コンピュータグラフィックス（以下，CG）の分野において重要な課題である．数十年に渡って積み重ねられてきた研究の成果は，目を見張るほど美しい映画やゲームの映像に見て取れる．このような写実的なレンダリングが求められている背景には，様々な分野の産業からの需要がある．

エンターテインメントの用途では，映画やゲームにおける応用が盛んである．現在の映画と半世紀前の映画とを比較すると，映像の迫力という点では大きな差がある．現実には存在しないようなクリーチャーが映画に登場する場合，従来は特殊メイクなどで実現していたが，近年ではコンピュータグラフィックスを合成して解決する場合が多い．ゲームでは，まだある程度の隔たりがあるとはいえ，写実的な人間や車などのレンダリングが可能となっている．

製品を実際に作る前に，その完成後の姿を見ることができるのは，コンピュータグラフィックスの発展による成果である．建築物は，建造した後でデザインが気に入らなかったからといって建て直すことは不可能である．デザイン段階でコンピュータを使ってモデリングし，写実的なレンダリングを行うことによって，事前にデザインの良し悪しが検証できる．建築物だけでなく，プロトタイプを作るためにコストがかかる自動車のような工業製品でも同様のことが行われている．

現在，高速なレンダリングを目的としたシステムでは，ラスタライゼーションという手法が用いられている．しかし，ラスタライゼーションは近似的な手法であるため，物理的な正確さに欠ける．OpenGL や DirectX といった API（Application Programming Interface）はラスタライゼーションによってリアルタイムでのレンダリングを実現しているが，それらによる写実的なレンダリングは困難である．そのため，それらのシステムはゲームなどのインタラクティブなシステムで使われることはあっても，映画等のように写実性を要求される分野で使われることはない．

一方でオフラインレンダリングと呼ばれる手法は，ラスタライゼーションより速度面では劣るが，レンダリングの品質は大幅に改善されている．高精度なレンダリングで有名な

Maya® のウェブサイトにある Fake or Photo[1]では，表示される画像が写真であるか CG であるかを選択するクイズを楽しめる．ここで表示される画像が写真かどうかを見極めるのは大変難しく，オフラインレンダリングが持つ写実性の高さを示している．オフラインレンダリングの基礎的な技術は，本論文のテーマとして取り上げるレイトレーシングである．レイトレーシングは処理に時間のかかる手法といわれてきたが，近年のコンピュータの発展に伴い，インタラクティブな速度でのレイトレーシングによるレンダリングが可能になってきた [Wal04]．専門家の間では，レイトレーシングがラスタライゼーションに取って代わるのはいつかという議論 [AKSS02] が行われるほどである．

レイトレーシングを高速化するには様々なアプローチが存在する．その中でも，レイが物体に当たる位置を計算する際の計算量を少なくすることを目的とした空間データ構造の改善が，重要な課題として取り上げられている．2006 年にはインタラクティブなフレームレートでのレイトレーシングを目的としたシンポジウムである *The 2006 IEEE Symposium on Interactive Ray Tracing*[2]が世界で初めて開催され，その中でも空間データ構造は大きな注目を集めた．また一方ではハードウェアの発展も進んでおり，それを生かすようなレイトレーシングのアルゴリズムが提案されている．新しく登場してくるコンピュータのアーキテクチャに適応した空間データ構造を用いることで，レイトレーシングはさらに高速化されるであろう．

1.2 本論文が解決する課題

本論文では，レイトレーシングを高速化するために，空間データ構造の 1 つである Bounding Volume Hierarchies （BVH）に 4 分木を適用する．一般に BVH の構築には 2 分木が用いられることが多いが，これを 4 分木とすることによって木構造の構築に使用されるメモリ量を削減し，近年のプロセッサに採用されている SIMD 命令で高速化することを目的とする．

1.3 本論文の構成

本論文の構成は以下のとおりである．2 章では，3 次元形状をレンダリングするための一手法としてレイトレーシングを紹介し，高品質な画像をレンダリングするに当たって時間がかかる原因について述べる．そしてそれを高速化するために用いられる空間データ構造やハードウェアの機能について述べる．3 章では，BVH を 4 分木にするための手法について述べる．まず一般的な 2 分木による BVH の構築手法について述べたあと，それを 4 分木，またさらに分岐数の多い木に拡張するための手法について提案する．そして提案した手法によって構築された BVH が従来の 2 分木 BVH とどのように異なるかについて

[1] http://www.autodesk.com/eng/etc/fake_or_foto/index.html
[2] http://www.sci.utah.edu/RT06/index.html

考察する．4 章では，近年の CPU が持つ機能である SIMD について解説した後，4 分木として構築した BVH に対して SIMD 命令を効果的に用いてトラバースする手法を提案し，SIMD 命令を用いなかった場合との実行速度の比較を行う．そして 5 章では，本論文で提案した手法とその結果についてまとめ，今後の課題を論じる．

第2章　レイトレーシングとその高速化

本章では，3 次元形状をレンダリングするための一手法としてレイトレーシングを紹介し，高品質な画像をレンダリングするに当たって CG 映像のレンダリング処理に時間がかかる原因について述べる．そしてそれを高速化するために用いられる空間データ構造やハードウェアの機能について述べる．

2.1　3 次元物体のレンダリング

2.1.1　モデリングとレンダリング

3 次元 CG 映像を生成するためには，その空間を**モデリング**する必要がある．図 2.1 にモデリングされた 3 次元空間の一例を示す[1]．

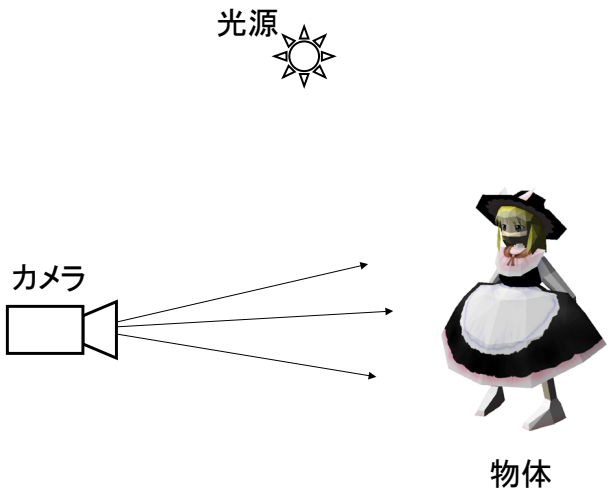

図 2.1: モデリングされた 3 次元空間

モデリングとは，個々の 3 次元物体の形状，材質，色柄，などのデータを作り，それを 3 次元空間に配置して，視点（位置，画角など）や光源（種類と位置など）の設定などを行う一連の作業である [芸術 03]．図 2.1 では，左側に**カメラ**，右側に**物体**，頭上に**光源**が

[1] ここで使用している 3 次元物体は http://www.siobi.info/gallery/th_paper_marisa.htm から引用した．

配置されている．物体は光源によって照らされ，その様子はカメラで撮影される．コンピュータ内でこの過程をシミュレーション，または近似して2次元画像を生成することが，いわゆる**レンダリング**（*rendering*）と呼ばれる処理である．

物体は三角形，直方体，球，曲面などの**プリミティブ**（*primitive*，基本形状）によって構成される．三角形はこれらのプリミティブの中でも最も単純であり，レンダリングなどの処理にかかる時間が短くてすむ．また，他のプリミティブを近似して表現することができる．しかし，実際に三角形を直接編集して物体をモデリングするのは困難であるため，Autodesk® 3ds Max® や Autodesk® Maya® などのモデリングソフトでは，より高度なプリミティブを扱って物体をモデリングすることができる．レンダリングのために球や曲面のように関数近似されたプリミティブを三角形で近似する場合には，その分割数を増やすことでより精度の高い近似が可能となる．本論文ではプリミティブとして三角形のみを取り扱うこととする．

モデリングした3次元空間をレンダリングする手法には，**レイトレーシング**（*ray tracing*，**光線追跡法**とも呼ばれる）や，それを応用した手法，また**ラスタライゼーション**（*rasterization*）がある．Slusallek はこれらを大別して以下のように表現した [SS05][2]．

> **Ray Tracing** Given a ray and set of primitives, efficiently compute the subset of primitives hit by the ray.
>
> **Rasterization** Given a set of rays and a primitive, efficiently compute the subset of rays hitting the primitive.

ここで文中に出てくる *ray*（**レイ**）とは，図 2.2 と図 2.3 に示すように，カメラから画素を通り抜けて物体に向けて発せられる線である．レイトレーシングとは，レイが交差するプリミティブを求める処理である．ラスタライゼーションとは，与えられたプリミティブに交差するレイを求める処理である．3次元空間を投影変換した2次元画像は，その内の全ての画素の色を計算することによってレンダリングできる．

それぞれの画素は**シェーディング**（*shading*，陰影付け）を行って色の計算が行われる．シェーディングでは，光源の種類，物体面の性質（反射率など），および光源，面，視点の相互関係を用いた計算が行われる [芸術 03]．物体によって異なる質感を表現するために，様々な種類のシェーディングモデルが用いられている．モデルには，マットな質感を表現するための Lambert モデルや，それを改良した Oren-Nayar モデル，金属の質感を表現するための Blinn モデルなどがある [PH04]．

画素の色を求める処理の概要だけを比較すると，レイトレーシングとラスタライゼーションは似た処理のように思える．しかしながら，実際にはアルゴリズムもその性質も非常に異なる．特に「レンダリングの品質」と「計算時間」という特徴に対して比較すると一長一短であるという二律背反した関係にあるため，場面に応じてどちらを用いるかが選択される．

[2] これはコンピュータグラフィックス関連の国際会議である SIGGRAPH 2005 での Course（特定分野に関する講義）で用いられた資料からの引用である．SIGGRAPH では近年レイトレーシングに関する Course が開かれている．

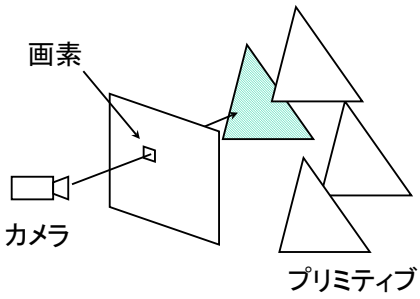

図 2.2: レイトレーシング

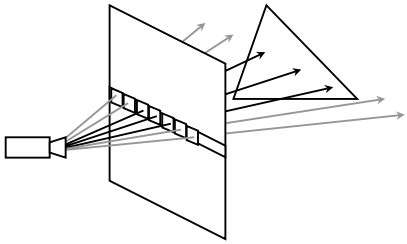

図 2.3: ラスタライゼーション

2.1.2 レイトレーシングによるレンダリング

レイトレーシングとは，一般にはレンダリング手法という意味で用いられることが多いが，これについて Wald は彼の博士論文 [Wal04] で以下のように説明している．

> ... the term “ray tracing” actually covers a wide range of different meanings, ranging from the basic *concept* of efficiently finding an intersection between a ray and a set of primitives, over the classical recursive ray tracing *rendering algorithm* (and its variants), down to more general *ray tracing based algorithms* that use ray tracing in one or another form.

つまり，レイトレーシングという単語の持つ意味の範囲は広く，その内容を限定しないと誤解を招くことがある．この論文では彼の定義に従い，**コンセプト**，**レンダリングアルゴリズム**，**レイトレーシングを基本としたアルゴリズム**というように 3 つの意味で使い分けるが，特に指示がないときはコンセプトとしてのレイトレーシングとしてこの単語を用いる．

レイトレーシングを用いてレンダリングを行うために用いられるアルゴリズムには，フォトンマッピング（photon mapping）[Jen96] やメトロポリス光輸送（metropolis light transport）[VG97] などがある．これらは 3 次元空間内の光の相互反射を考慮した大域照明という考えを取り入れており，写実的なレンダリングが可能である．工業製品のプロトタイピングや，映画，CM などに用いられる画像や映像をレンダリングするためには，これらのアルゴリズムが用いられることが多い．一般に，大域照明を用いたレンダリングには，長い計算時間を要する．

2.1.3 ラスタライゼーションによるレンダリング

ラスタライゼーションは非常に高速なレンダリングが可能である．特に 21 世紀に入ってからは GPU による非常に高速なレンダリングが可能となっている．また，GPU はラス

タライゼーションを高速化させるだけでなく，その過程をプログラマブルシェーダによってカスタマイズすることを可能にした [宮田 05]．従来はラスタライゼーションによって得られる画像の品質は低かったが，このプログラマブルシェーダによって品質が改善されつつある [Pha05]．また，GPU を汎用の計算機ととらえ，レイトレーシングやそれによるレンダリングを GPU 上で行う研究も行われている [PBMH02]．

2.2 レイトレーシング

本論文では，写実的なレンダリングを得意とするレイトレーシングを取り上げる．**レイトレーシング**とは，3 次元モデルを写実的にレンダリングするために有効なアルゴリズムで，1968 年に Appel が陰面消去を解決する手法として提案した．3 次元モデルをレンダリングする際には，視点からモデルを見たときに表側にある部分のみをレンダリングしなければならない．この課題を解決するための手法が**陰面消去**である．実際には陰面消去にはいくつかのアルゴリズムが存在するが，それらの 1 つとしてレイトレーシングが提案された．

レイとは，太さを持たない 3 次元空間上の線分，または半直線である．レイの原点を $\boldsymbol{O}$，方向を $\boldsymbol{D}$ とすると，レイ $\boldsymbol{R}$ は式 2.1 で表される．

$$\boldsymbol{R}(t) = \boldsymbol{O} + t\boldsymbol{D} \tag{2.1}$$

一般に，レイはカメラから画素を通り抜けるように生成され，シーン内でカメラに最も近いプリミティブとの交差点を探索する．レイがプリミティブに衝突したときの t の値は t_{hit} と表記する．交差点を探索する手法で最も単純なものは，レイと全てのプリミティブとの交差判定を行い，その中で最も原点からの距離が短い点を探索する手法である．レイの長さは t の範囲を t_{max} 未満に制限することで決めることができる．また，レイが物体の表面から発せられる場合には，計算誤差によってその表面自体を交点として誤認してしまう場合がある．t の範囲を ϵ より大きい値に制限することで，このような事態を防ぐことができる．

三角形とレイとの交差判定を行うためには，ユークリッド座標系以外での座標系における計算が有効である．そのための座標系としては，**バリセントリック座標系（***barycentric coordinates***）**が代表的なものとして知られている [MT97]．三角形の頂点をそれぞれ $\boldsymbol{V_0}$, $\boldsymbol{V_1}$, $\boldsymbol{V_2}$ とすると，バリセントリック座標系上での三角形上の点 $T(u, v)$ は式 2.2 で表すことができる．

$$\boldsymbol{T}(u, v) = (1 - u - v)\boldsymbol{V_0} + u\boldsymbol{V_1} + v\boldsymbol{V_2} \tag{2.2}$$

このとき，u と v は，$u \geq 0, v \geq 0, u + v \leq 1$ という条件を満たす必要がある．レイとの交点を求めることは，$\boldsymbol{R}(t) = \boldsymbol{T}(u, v)$，つまり式 2.3 を解くことと等しい．

$$\boldsymbol{O} + t\boldsymbol{D} = (1 - u - v)\boldsymbol{V_0} + u\boldsymbol{V_1} + v\boldsymbol{V_2} \tag{2.3}$$

この式を変形させることによって，式 2.4 を得ることができる．

$$[-\boldsymbol{D}, \boldsymbol{V_1} - \boldsymbol{V_0}, \boldsymbol{V_2} - \boldsymbol{V_0}] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = [\boldsymbol{O} - \boldsymbol{V_0}] \tag{2.4}$$

線型方程式である式 2.4 の解を求め，t, u, v が条件を満たせば，レイと三角形が交差していると言える．

このようなアルゴリズムにしたがって見つかった点に対して適切なシェーディングをすることでレンダリングが可能となる．しかし，Appel のアルゴリズムは，元々陰面消去を目的としていたため，単純なシェーディングしかすることができなかった．

2.3 二次レイの発生

Whitted は 1980 年に反射，屈折を考慮したレイトレーシングによるレンダリングアルゴリズム [Whi80] を確立した．以下に Whitted のアルゴリズムについて詳述する．

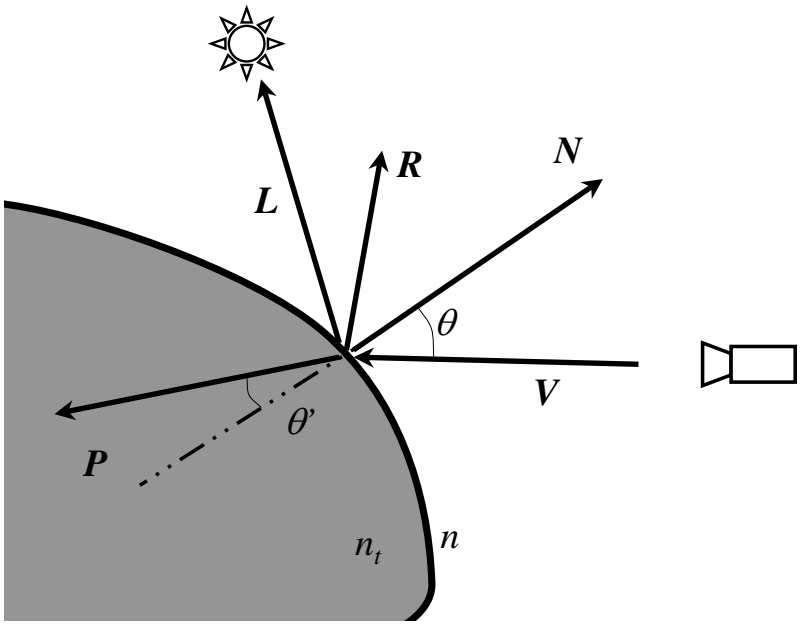

図 2.4: 反射，屈折の関係

図 2.4 に示すとおり，法線 $\boldsymbol{N}$ を持つプリミティブ上の点から視点に届く光の輝度 I には，反射方向からの輝度 S と屈折方向からの輝度 T が影響している．これらの輝度は，視点から発せられたレイの方向ベクトル $\boldsymbol{V}$，そしてそれが反射，屈折する方向に向かうベクトル $\boldsymbol{R}$，$\boldsymbol{P}$ に沿って伝播する光を表現している．これらを用いて I を表現すると式 2.5

となる．

$$I = I_a + k_d \sum_{j=1}^{j=ls} (\boldsymbol{N} \cdot \boldsymbol{L_j}) + k_s S + k_t T \tag{2.5}$$

右辺の第 1 項は，環境光 I_a による輝度で，3 次元空間内を一様に明るくする．第 2 項は，光源からの照明による輝度で，Lambert の法則による拡散反射を計算している．$\boldsymbol{L_j}$ は光源方向へ向かうベクトルで，光源は ls 個あるとする．k_d は拡散反射による輝度に対する係数である．第 3 項，第 4 項は反射，屈折による輝度で，k_s, k_t はそれぞれ S, T に対する係数である．$\boldsymbol{R}$ は入射角と反射角が等しいという法則から式 2.6 によって導くことができる [SM03]．

$$\boldsymbol{R} = \boldsymbol{V} - 2\boldsymbol{N}(\boldsymbol{V} \cdot \boldsymbol{N}) \tag{2.6}$$

また，θ を $\boldsymbol{R}$ と $\boldsymbol{N}$ のなす角（反射角），θ' を $\boldsymbol{P}$ と $-\boldsymbol{N}$ のなす角（屈折角）とすると，$\boldsymbol{P}$ は式 2.7 で求まる．

$$\boldsymbol{P} = \frac{n(\boldsymbol{V} + \boldsymbol{N}\cos\theta)}{n_t} - \boldsymbol{N}\cos\theta' \tag{2.7}$$

入射光側の屈折率 n と屈折光側の屈折率 n_t を与えると，θ と θ' の関係はスネルの法則 $n\sin\theta = n_t \sin\theta'$ から計算できる．

$$\cos^2\theta' = 1 - \sin^2\theta' = 1 - \frac{n^2(1-\cos^2\theta)}{n_t^2} \tag{2.8}$$

以上の関係から $\boldsymbol{P}$ をまとめると，式 2.9 のように式中から θ と θ' を取り除くことができ，三角関数の計算を省くことができる．

$$\boldsymbol{P} = \frac{n(\boldsymbol{V} - \boldsymbol{N}(\boldsymbol{V} \cdot \boldsymbol{N}))}{n_t} - \boldsymbol{N}\sqrt{1 - \frac{n^2(1-(\boldsymbol{V} \cdot \boldsymbol{N})^2)}{n_t^2}} \tag{2.9}$$

このとき，式 2.9 の根号の中が負数になると，全反射が起こる．全反射が起こると屈折光が発生せずに，反射光のみが I に寄与する．

k_s と k_t はフレネルの法則から導くことができるが，計算を簡単にするために式 2.10，式 2.11 に示す Schlick の近似式を使うことができる．

$$k_s = R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5 \tag{2.10}$$

$$k_t = 1 - k_s \tag{2.11}$$

Whitted のアルゴリズムによって反射や屈折の表現が可能となり，レイトレーシングによるレンダリングは写実的なものに近づいた．しかし，反射や屈折によってレイが増えたため，物体との交差判定の回数も増加することとなり，結果的にレンダリングにかかる時間が長くなるという問題が発生した．ここで新しく発生したレイは**二次レイ**（*secondary rays*）と呼ばれる．特に物体がガラスなどの材質で構成されている場合は，反射，屈折による二次レイの数が指数的に増加してしまう．後に Cook らは，分散レイトレーシング [CPC84] でモーションブラーや被写界深度などの表現を可能としたが，ここでも二次レイ

を増やすことが必要となった．レイトレーシングを用いた写実的なレンダリングについては [PH04] や [SM03] などの書籍が詳しい．

レイを増やすことによって得られる画像の品質は良くなるが，レンダリングにかかる時間が長くなってしまう．そのため，レイトレーシングを高速化するために様々な手法が考案された．

2.4 空間データ構造の改善による高速化

レイトレーシングを高速にするためのアプローチは複数ある．Wald の博士論文 [Wal04] の第 3 章 “A Brief Survey of Ray Tracing Acceleration Methods” では以下のような点が挙げられている．

- Computing less Samples in the Image Plane
  - Pixel-Selected Ray Tracing and Adaptive Sampling
  - Vertex Tracing
  - The Render Cache
  - Edge-and-Point-Image (EPI)
- Reducing the Number of Secondary Rays
  - Shadow Caching
  - Local Illumination Environments
  - Adaptive Shadow Testing
  - Single Sample Soft Shadows
  - Pruning the Ray Tree
  - Reflection Mapping and Shadow Maps
  - Sample Reuse for Animation
  - First Hit Optimization
- Accelerating Ray-Scene Intersection
  - Faster Intersection Tests
  - Spatial and Hierarchical Scene Subdivision

項目だけ列挙しても，これだけ多岐に渡って最適化がなされるべき点があることが分かる．そして Wald はこれらの中から “Spatial and Hierarchical Scene Subdivision” が最重要であると位置づけた．本論文では，これを**空間データ構造**と呼ぶ．

空間データ構造が用いられる目的は，レイとプリミティブの交差判定の数を減らすことである．3 次元物体が N 個のプリミティブで構成されているとすると，総当りで交差判定を行った場合，計算量は $O(N)$ となる．物体を高い精度でモデリングしようとすると，用いられる三角形の数は数十万から数千万という数になり，これらに対して一つ一つ交差判定を行うと，その計算時間は非常に長くなる．空間データ構造を用いると，必要な交差判定の数は激減する．例えば本論文で取り上げる BVH のように木構造を持つ空間データ構造は，計算量の増加を大体 $O(\log N)$ に抑えることができるため，総当りで交差判定を行うよりも計算時間が短い．

空間データ構造の構築には数多くのアルゴリズムが提案されている．Havran はこれらを用いたレイとプリミティブの交差判定を “ray shooting algorithms” と呼び，kd-tree，BSP tree，グリッド，BVH，octree などの 12 個のアルゴリズムを比較して，博士論文の第 3 章にまとめた [Hav01]．結果は kd-tree が平均的に高いパフォーマンスを示したと述べられている．しかし，全ての状況に対応した完全優位なアルゴリズムがないことも同時に述べられている．このような空間データ構造に関する研究は，現在も注目を浴びる研究分野である [RSH05][WIK+06]．近年では，変形するシーン[3]に対して BVH が有効であることが示された [WBS07]．

2.5 ハードウェアによる高速化

レイトレーシングの高速化にはアルゴリズムの改善が大きく寄与してきたが，近年では実装面での改善も研究されている．並列計算の機能を用いたり，レイトレーシングのためのハードウェアを製作するなどといった取り組みが，こういった研究の成果である．

1990 年代後半から，パーソナルコンピュータに搭載されている CPU に，並列計算機能が搭載され始めた．Intel® の CPU は MMX，SSE，SSE2，SSE3 といった命令群 [Inta][JZ] を，AMD® の CPU は 3D Now!等の命令群 [AMD] を，並列計算用の命令として持っている．CG や音楽などを取り扱うソフトウェアには，複数のデータに対して同時に同じ計算をさせる処理が多くある．こういったソフトウェアは，CPU による並列計算を活用することによって実行速度を大幅に改善している．このような並列計算を行うための命令は，*SIMD*（*Single Instruction Multiple Data*）**命令**と呼ばれる．SIMD 命令を使うことによって，複数のデータに対して同じ計算処理を一度に行うことができる．例えば SSE を使えば，32 ビット浮動小数点の四則演算が 4 つ同時にできる．近年のレイトレーシングの高速化を目指した研究では，このような SIMD 命令が用いられることが多い [GM03]．

[3]**変形する**というのは，時間変化に対してプリミティブ数が増減しないことを表す．一般に言う動的シーンでは，プリミティブ数が変化することがあるので，ここではそれと区別して変形するシーンという表現を使っている．

GPU（*Graphics Processing Unit*）は3DCGのレンダリングに特化した構造をしており，頂点やピクセルなどを並列に演算するハードウェアを持っている．そのため，CPUよりも並列計算が得意である．例えば，2006年にNVIDIA® から発売されたGeForce8800[NVIa]は，スカラー演算を行うStreaming Processorを128個搭載おり，それぞれは1.35GHzで動作する．GPUを用いたプログラミングは，シェーダ言語によって行うことができる．またCUDA[NVIb]を用いるとC言語を用いたGPUプログラミングが可能である．Purcellらや Carrらは GPUを用いたレイトレーシングを実装した [PBMH02][CHH02].

さらに近年では，**マルチコア** *CPU* が登場し始めている．コアというのは，従来のCPUに相当するもので，演算装置単体を意味する場合もあれば，それに付随するキャッシュも含めた意味になることもある．これらのCPUを構成するコア数が2つならデュアルコア（dual core），4つならクアッドコア（quad core）などと呼ばれており，それらを総称してマルチコア（multi-core）という単語が使われる．マルチコアCPUも並列計算の高速化に寄与する要素として考えることができる．

また，Sony，IBM，東芝の三社は共同で *Cell*[Intc][Son] を開発し，2005年にその技術仕様について公開した．IntelやAMDのように，全てのコアが等しい性能を持つマルチコアCPUとは異なり，Cellは PPE，SPEと呼ばれる2種類のコアを合計で9つ搭載している．このうち SPEは Cellに8つ搭載されており，SIMDによる並列演算を得意とするため，プログラムの書き方によっては高いパフォーマンスを発揮することができる．Benthinらは Cellを使ってインタラクティブなフレームレートのレイトレーシングを実装した [BWSF06].

第3章　分岐数を2から拡張したBVHの構築

本章では，一般に2分木で表現されることが多いBVH（Bounding Volume Hierarchies）の分岐数を増やす手法について述べ，構築されたBVHの構造について考察する．まず，BVHの概念や利点について述べ，その構築手法を説明する．次に2分木で構築されてきたBVHを4分木にした場合にどのような変化が起こるかを考察する．最後にオブジェクトメディアン，SAHという2つの手法を用いてBVHの分岐数を2から拡張する手法について述べ，実験結果について考察する．

3.1　Bounding Volume Hierarchies

本論文では，空間データ構造の一種であるBVH[Cla76][RW80]を取り上げる．BVHとは，その名が示すとおりバウンディングボリュームの階層構造であり，レイと3次元物体との交差判定にかかる計算量を減らすために用いられる[1]．まずは，バウンディングボリュームについて説明する．

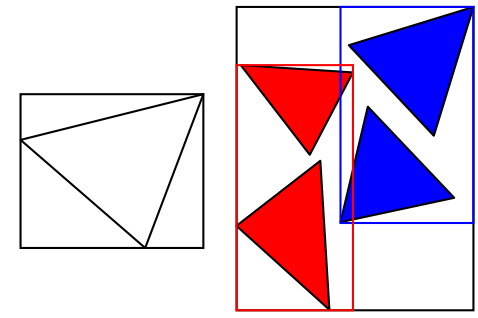

図 3.1: 三角形を取り囲むバウンディングボリュームと，その階層構造

バウンディングボリューム（Bounding Volume，以下**ボリューム**と略す）とは，図3.1（a）のようにプリミティブを取り囲む立体である．これは2次元空間での図に見えるが，視線方向の奥行きもあると考え，三角形や長方形は3次元空間に存在するものとする．ボ

[1]Whittedのレイトレーシング[Whi80]では，すでにBVHの有用性が論じられている．

リュームには，レイとの交差判定にかかる計算時間が短いものが適している．そのため，球 [Whi80] や直方体 [RW80] などの形が候補として考えられるが，本論文ではそれらの中でも広く用いられている *Axis Aligned Bounding Box*（*AABB*，軸に並行な直方体）をボリュームとして用いる．ボリュームは，可能な限り小さいサイズでプリミティブを取り囲まなければならない．そのため，AABB の面は三角形の頂点と接する．

ボリュームに取り囲まれる三角形は 1 つだけではなく複数とすることもできる．図 3.1（b）に，赤い三角形 2 つ，青い三角形 2 つをそれぞれ取り囲むボリューム，そしてそれら全体を囲む黒いボリュームを示す．この場合にも，ボリュームはそれが含む三角形全てを最小のサイズで取り囲まなければならない．ここで，赤，青，黒のボリュームの間に階層構造が成り立っていることに着目する．赤と青のボリュームは，黒のボリュームに含まれる形になっているので，これらは黒が親で，赤と青が子となる階層構造を形成しているといえる．

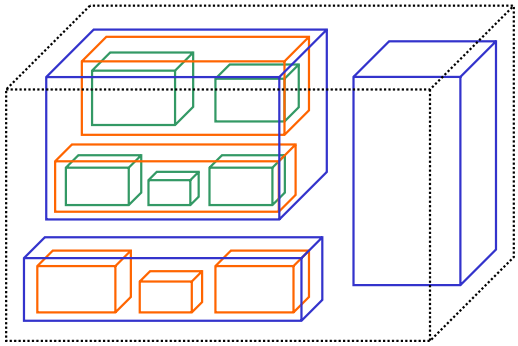

図 3.2: ボリュームの階層構造

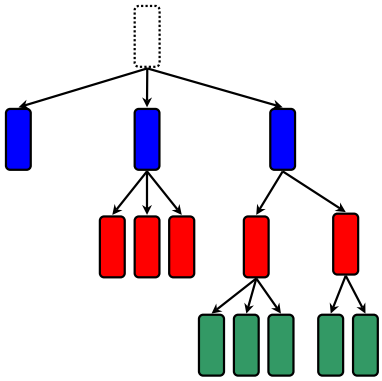

図 3.3: 木構造で表現した階層構造

さらにボリューム同士の階層関係の規模が大きくなると，図 3.2 のようになる．簡略のために三角形は表示していない．黒の破線で示したボリュームは他の全てのボリュームを取り囲み，青，赤，緑とボリュームが小さくなるにしたがって階層が深くなっている．これを分かりやすく木構造で示したのが図 3.3 である．

木構造中の四角を**ノード**と呼び，あるノードの下にあるノードを**子ノード**と呼ぶ．それぞれのノードの色はボリュームの色に対応している．ここで，木構造の末端にあるノードを**葉ノード**，そうでないノードを**内部ノード**，最上位に位置するノードを**ルートノード**と呼ぶ．ルートノードは内部ノードの一種として扱う．

三角形の形状，質感などの情報を保持する必要があるのは，葉ノードだけである．内部ノードが含む三角形を知りたい場合には，葉ノードに至るまで子ノードをたどって行けばよい．よって，内部ノードは直接三角形の情報を持つ必要はない．

レイトレーシングは，この階層構造によって高速化される．任意のレイと，BVH で表される 3 次元物体との交差判定を行うとき，まずレイとルートノードのボリュームとの交差判定が行われる．ルートノードのボリュームは 3 次元物体を完全に取り囲んでいるた

め，レイがボリュームと交差していなければ，レイが 3 次元物体に交差することはない．交差している場合は，レイが 3 次元物体と交差する可能性があるといえる．この場合，交差判定の対象が子ノードへと移る．子ノードでもレイとボリュームとの交差判定を行い，交差するならさらに下の子ノードへと進んでいく．そして交差判定の対象が葉ノードに到達すると，ようやくレイと三角形との交差判定が行われる．このように階層構造をたどってレイと交差する三角形を求める一連の過程を**探索**，または**トラバース**と呼ぶ．

ここで，BVH がレイトレーシングに与えるの利点を 2 つ見出すことができる．1 つは，交差判定を行う数を減らすことができるということである．総当りでレイと N 個の三角形との交差判定を行う場合，探索に要する計算量は $O(N)$ となる．しかし，BVH を用いることによって，全ての三角形と交差判定を行う必要はなくなる．BVH を用いた場合の計算量は，階層の深さによって変化するが，大体 $O(\log(N))$ となる．もう 1 つの利点は，AABB で物体を表すのでレイとの交差判定に要する時間が短いということである．

空間データ構造の中には，他にも kd-tree，octree，グリッドといったものが存在する．これらの手法は空間を分割しているため，物体を分割する BVH とは異なる．以降，kd-tree のような特性を持つ空間データ構造の構築方法を空間分割法と総称する．空間分割法は，図 3.1 のように三角形が配置されていた場合，三角形が配置されている空間を分割する．これは，例えば $Z = 0$ で定義されるような X-Y 平面で空間を分割するという意味である．分割された 2 つの空間を三角形がまたいでいるような場合には，その三角形は両方の空間に存在するものとして扱われる．つまり，空間分割法において，三角形は複数の葉ノードに存在することがある．これに対して BVH では，物体を三角形単位で分割する．例えば，A, B, C の 3 つの三角形があった場合，それらを [A, B] と [C] や，[A, C] と [B] といったようなグループに分ける．このため三角形が複数のノードに存在することはない．図 3.1 で青い三角形が赤いボリューム内に，また赤い三角形が青いボリューム内に食い込んでいるが，これはあくまで見た目上の問題である．

3.2 BVH の構築

BVH を構築する手法には，物体を分割してプリミティブにしていくトップダウンな手法 [KK86][Smi98] と，プリミティブを集めて物体を構築していくボトムアップな手法 [GS87] がある．本論文では [WBS07] でその有用性が指摘されているトップダウンな手法を取り上げ，それらの中でもプリミティブ数が同じになるように分けていく手法と，発見的手法によって分割点を探す手法を用いる．

例として図 3.4 に示すような 3 次元物体の BVH を構築する過程について説明する．一般に，木構造の分岐数は木全体にわたって一定であることが多く，ほとんどの論文で論じられているのは分岐数が 2，つまり木が 2 分木である場合のみである．よって，ここでは 2 分木で BVH を構築する場合について説明する．2 分木で BVH を構築するときは，子ノードを作る段階で，常にその数が 2 つになるようにしなければならない．

ルートノードには，BVH を構築する対象となる 3 次元物体の全ての三角形が含まれる．

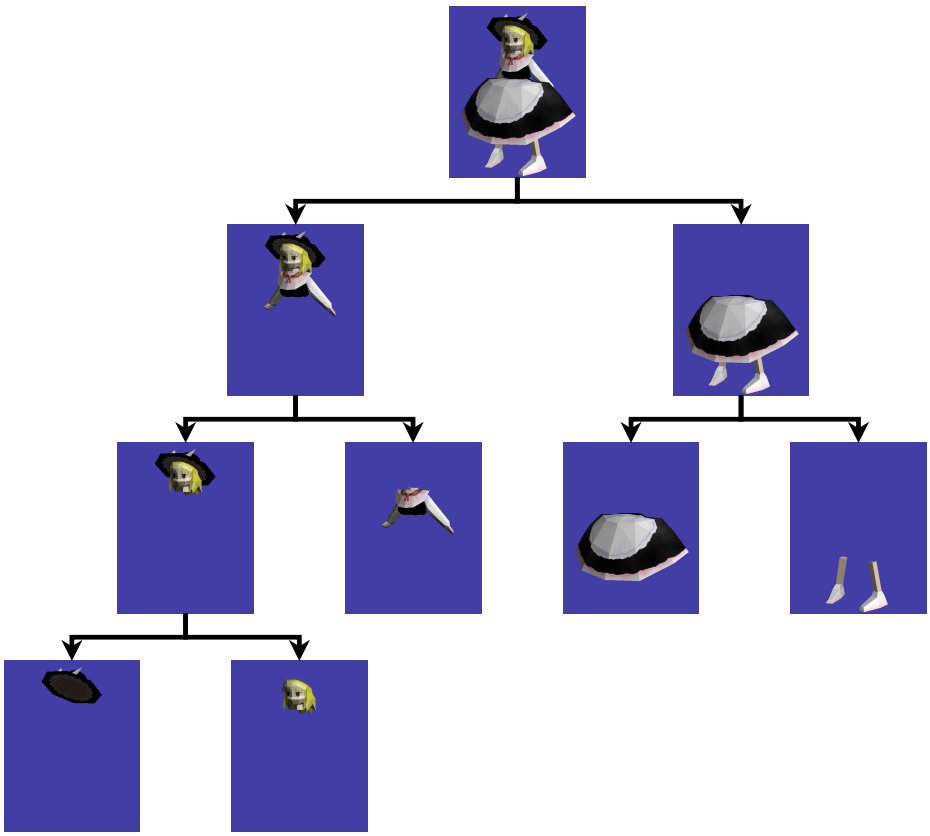

図 3.4: 2 分木による BVH

子ノードを構築するためには，これらの三角形を 2 つのグループに分ける必要がある．ここでは説明上分かりやすいように上半身と下半身という 2 つの部位に分け，子ノードとする．このとき，分かれた部分ごとに AABB を計算し，ノードの情報として保持する．そしてこれらのノードをさらに 2 つの部位に分け，階層構造を構築していく．すると最終的に図 3.4 のような木構造が完成する．ここでは説明上，元の物体を 5 つの部位に分けて最小単位としているが，本論文で提案するアルゴリズムでは，この分割作業を三角形 1 つになるまで繰り返す．一般的に，全てのノードは，ノード自身が示す 3 次元物体のプリミティブを取り囲む AABB についての情報を持つ．

子ノードを作るときに三角形を 2 つのグループにどのように分けるかは，効率的な BVH を作るために重要である．できる限りのグループの組み合わせを考え，それらの中から効率的なものを発見するのが好ましいが，可能な組み合わせの数は非常に多く，全てについて試すのは困難である．そこで，ある軸に垂直な面を分割面として三角形を 2 つのグループに分けるという簡略的な手法が広く用いられている．例えば，あるノードに含まれる三角形の中心の X 座標がある値よりも大きいか小さいかで，三角形を 2 つのグループに分けることができる．ここでの三角形の中心座標とは，重心座標ではなく，三角形を取り囲む AABB の中心座標のことである．

以上のように，組み合わせの探索に制限を加えることで，軸，分割面という 2 つのパラメータを変更するだけの探索に探索空間を狭めることができる．また，事前に三角形を軸に沿ってソートしておけば，ノードに格納する三角形をある番号からある番号までといった形で一括して指定することができる．さらに，あるノードにおいて子ノードを探索するときに，探索順序が座標軸に沿ってソートされていると，より高速なトラバースが可能となる [MW04]．より具体的に言えば，AABB を X（または Y，Z）軸に平行な平面で分割した場合，ボリュームの X（または Y，Z）座標値が小さい子ノードから探索したほうが高速となる．

ここまでの処理内容をまとめ，与えられたノードに対する子ノードを作成する擬似コードで表す．

```
if（ノードが含む三角形が 1 つ）
{
    葉ノードを作る
}
else
{
    三角形をソートする
    三角形をグループにまとめる
    グループから子ノードを作る
}
```

構築される BVH の効率を左右するのは，ノードの分割に使用するための軸，分割面の探索，つまり擬似コード中の「三角形をグループにまとめる」にあたる処理である．これらを探索するアルゴリズムが優れているほど，より速いレイトレーシングが可能な BVH を構築することができる．

任意の分岐数を持つ BVH を構築するためのノードは，以下のような構造体で記述することができる．

```
struct Node
{
    Aabb aabb;
    int index;
    Node** children;
};
```

aabb はノードが含む三角形を取り囲む AABB である．index は，ノードが内部ノードの場合は子ノードを作るときに切断した軸の情報で，葉ノードの場合は含まれるポリゴンの番号である．children は，内部ノードの場合は子ノードを指すポインタの配列で，葉ノードの場合は NULL である．

3.3　2分木と4分木の理論的比較

本論文では，木構造の分岐数を4とした木構造（4分木）でBVHを構築することを提案する[2]．実装するに当たって，まずは2分木を4分木としたときにどのような変化が生じるかについて机上で考察する．

葉ノードには1つのプリミティブのみが含まれるように分割する．2分木の場合と異なり，4分木では内部ノード数が一定数になることが保障されない．図3.5と図3.6にプリミティブが16個ある場合に考えられる2分木と4分木を示す．

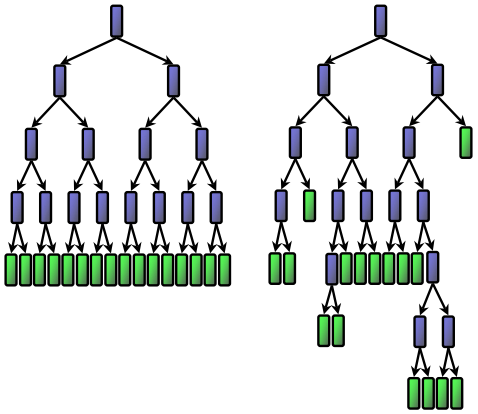

図 3.5: 2分木

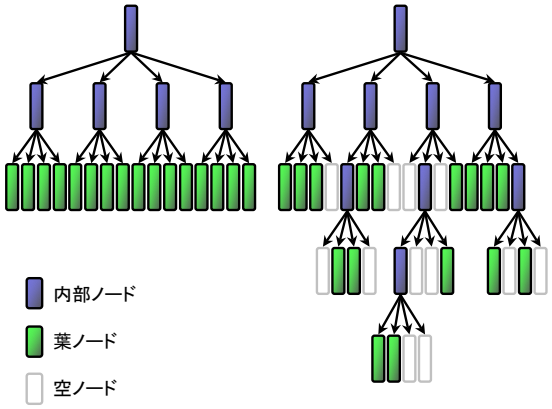

図 3.6: 4分木

それぞれの図の左側は，木をバランス木とした場合である．葉ノードに至るまでの階層の深さは，2分木で4，4分木で2となっている．このときの階層の深さの数え方は，ルートノードでの深さを0として，ノードをたどるたびに深さが1増えるとしている．また，内部ノード数はそれぞれ15個，5個である．バランス木を構築すれば，それぞれの分岐数において階層の深さと内部ノード数は最小となる．

これらのバランス木に対して，右側に示したのはその構造が乱れた木である．構造が乱れると，4分木では**空ノード**が発生することがある．空ノードとは，図3.6において色の薄い四角で示されたノードで，葉ノードの一種である．4分木において，それぞれの内部ノードは必ず4つのノードを持たなければならない．しかし，場合によってはそこに入るべきノードが存在しないことがある．そのような場合に空ノードが発生してしまう．

構造が乱れた場合の変化で両方の木に共通するのは，階層の深さが深くなっていることである．階層の深さは，そのまま探索効率に影響する．木をたどるときにレイとバウンディングボリュームとの交差判定を計算する数が少ないほど，速いレイトレーシングが可能となる．4分木のみで起こっている変化は，内部ノード数が増えていることである．2分木では構造が乱れて階層がいくら深くなっても，内部ノード数が変わることがない．しかし4分木では，空ノードの発生という問題があるために内部ノードが増加する．図3.6

[2]4分木という単語は2次元空間の分割にも用いられ，一般に4分木というとその意味を指すことが多い．ここでいう4分木は，それとは異なるものを指していう単語である．

ではバランス木で5個だった内部ノードが，構造を乱すことで9個に増えている．この数はさらに増えることもある．また，総合的に言えば，内部ノード，空ノードの増加をまとめて，ノード数が増えているとも言える．ノード数が増えれば，BVH の構築に必要なメモリ量も増えてしまうことになる．

空ノードが発生してしまう問題に対して，本論文ではノードを重複させる手法を提案する．例えば，分岐数が4であるにも関わらず，ノードに含まれる三角形が3つしかないような場合には，子ノードのうち2つが同じ三角形を参照するようにする．より一般的に言えば，あるノードに含まれる三角形の数が分岐数に満たない場合，子ノードのうちのいくつかが同じ三角形を参照するようにする．本論文で提案する SIMD 命令によるトラバースでは，子ノードには常に有効なノードが存在していなければならない．そのため，子ノードを NULL ポインタとするようなことができないため，このような処理が行われる[3]．

以上までで論じたように，BVH の木構造を2分木から4分木にすることによって，階層の深さが浅くなる代わりに，ノード数が増加するという変化がある．レイトレーシングを BVH で高速化するためには，2分木が用いられることが多かった．理由は主に探索効率によるものである．レイがあるノードに達すると，子ノードの AABB 全てと交差判定しなければならない．このときに分岐数が多いほど，交差判定をする回数が増え，計算時間がかかる．本論文では，後述する並列演算によりこの問題を解決する．

3.4 オブジェクトメディアンでの分割

3.4.1 オブジェクトメディアンでの分割による BVH の構築

あるノードの子ノードが持つ三角形の数が等しくなるような分割手法 [KK86][Smi98] を**オブジェクトメディアン（*object median*）での分割**と呼ぶ．オブジェクトメディアンでの分割が用いられる理由は，実装が簡単で，BVH の構築に要する時間が短いためである．本論文では，まず典型的なこの手法を用いて，BVH の分岐数を変化させた場合の振る舞いを考察する．

ソート済みの T 個の三角形に，番号が0番から $T-1$ 番までついていたとすると，これらの三角形を2つのグループ（0番と1番）に分けるアルゴリズムは以下の擬似コードのように非常に単純なものとなる．

```
0 番目から T/2-1 番目までの三角形をグループ 0 番に入れる
T/2 番目から T-1 番目までの三角形をグループ 1 番に入れる
```

ソートするときの軸は，ノードの階層が深くなるごとに，X 軸，Y 軸，Z 軸と順に変更する．これによって AABB のサイズが均等に小さくなる．

オブジェクトメディアンで分割することによって，2分木はバランス木となるため，木構造の階層が浅くなる．また，分岐数が多くなっても，それに近い形の木が構築される．

[3] このことについては4章で詳述する．

そのため，各葉ノードへ至るまでの交差判定計算の数が一定している．全ての三角形が同じ確率でレイと交差するのであればこの手法を用いて構築されたBVHによるレイトレーシングは非常に高速である．しかし，実際にはレイと交差する可能性が高い大きな三角形や，その逆に小さな三角形が存在するため，期待するほどの速度は出ない．

本論文では，BVHの構築をするとき，分岐数が2より多い場合に起こる変化について実験する．オブジェクトメディアンでの分割によってBVHを構築する場合には，分岐数を2から増やす手法は単純なものである．分岐数を4とすると，元の擬似コードをそのまま拡張した以下のような手法が考えられる．

0番目からT/4-1番目までの三角形をグループ0番に入れる
T/4番目から2*T/4-1番目までの三角形をグループ1番に入れる
2*T/4番目から3*T/4-1番目までの三角形をグループ2番に入れる
3*T/4番目からT-1番目までの三角形をグループ3番に入れる

さらに分岐数をBで表し一般化すると，以下のように書くことができる．

```
for i = 0 to B-1
{
    i*T/B番目から(i+1)*T/4-1番目までの三角形をグループiに入れる
}
```

この手法により，図3.7のようにX軸に沿ってソートされた8つの三角形があった場合，子ノードに格納される三角形は2つずつとなる．

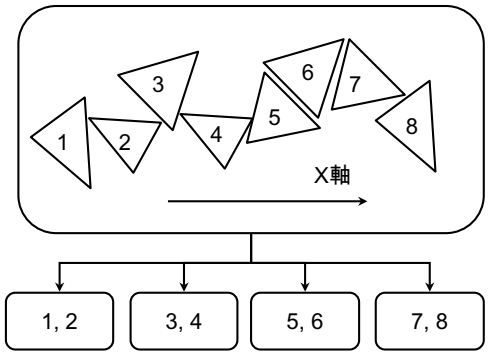

図 3.7: オブジェクトメディアンによる分割（分岐数4の場合）

しかし，三角形の数が4つに満たない場合，例えば2つしかないような場合には，グループ0番と1番に1つ目の三角形，グループ2番と3番に2つ目の三角形が入ってしまう．こうなると，本来発生すべき空ノードが発生しなくなってしまう．そこで，三角形が重複せずにグループに入るように次に示すようなアルゴリズムを改良する．

```
for i = 0 to B-1
{
    for j = i*T/B to (i+1)*T/B-1
    {
        j 番目の三角形をグループ i に入れる
    }
}
```

`for j = A to B` の `A` と `B` の小数点以下が切り捨てられ，`for` 内部のループは `A` が `B` 以上の場合にしか実行されないものとすると，三角形の重複はなくなり，空ノードが発生する．

3.4.2 実験と考察

オブジェクトメディアンでノードを分割した BVH において，分岐数を 2 から増やした場合に，ノード数，また階層の深さがどのように変化するかを実験した．実験に用いた 3 次元物体の名称と三角形数を表 3.1 に示す．これらの物体のうち，balls4，gears4，tree10 は，レイトレーシングの性能評価に用いられることが多い *SPD*（Standard Procedural Database）[Hai87] によって生成した．それぞれの物体の BVH の木構造の分岐数は 2-16 の範囲で 2 つおきに実験を行った．

表 3.1: 実験に使用する物体

物体名	図番号	三角形数
cloister	図 3.8	81,410
balls4 (SPD)	図 3.9	797,150
gears4 (SPD)	図 3.10	36,610
tree10 (SPD)	図 3.11	270,206

葉ノードの平均深さ

まず，葉ノードの平均深さについて考察する．各物体に対する葉ノードの平均深さの変化は，図 3.12 のようになった．分岐数が増えると，葉ノードの平均深さは減少する．分岐数が 2 から 4 に増えたとき，深さは約半分になっている事が分かる．これは先に述べた理論的考察でも考えられたことである．それ以上分岐数を増やした場合の平均深さの減り方は穏やかになる．分岐数が 4 から 8 へと 2 倍になっても，平均深さは半分とまでは行かない．これは空ノードの発生に原因があると考えられる．また，分岐数が増えてくると，平均深さが整数になる現象が見られた．例えば，分岐数が 10 のときに，cloister と gears4 の平均深さはちょうど 5 である．平均化したときの小数点以下の値が出ないということは，

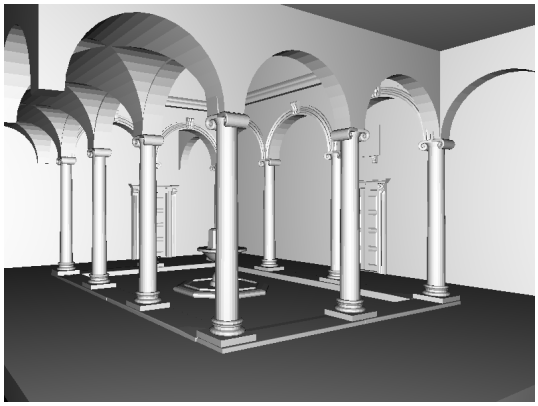

図 3.8: cloister

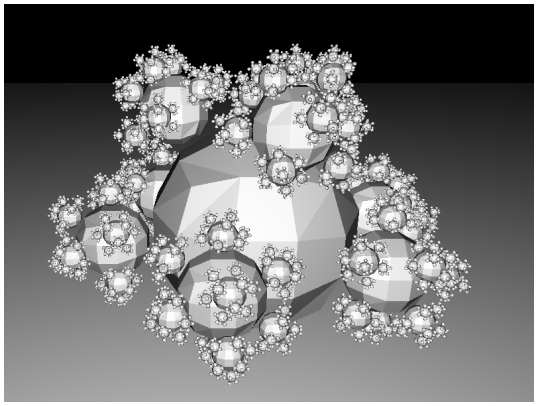

図 3.9: balls4

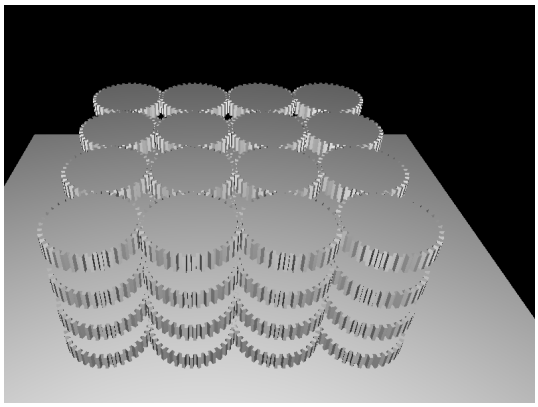

図 3.10: gears4

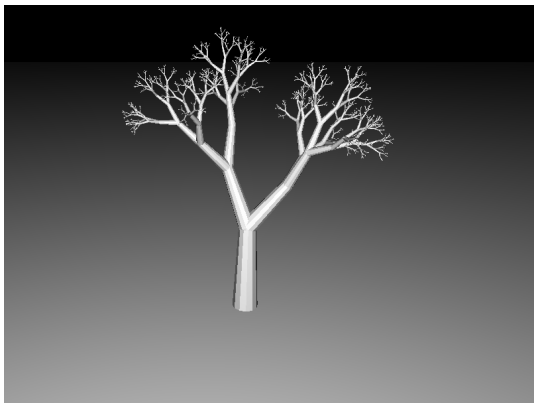

図 3.11: tree10

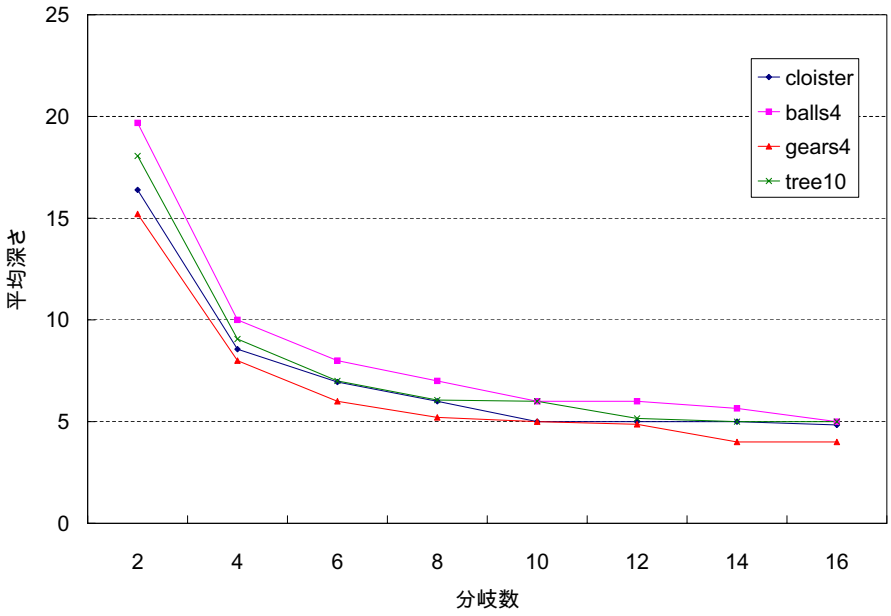

図 3.12: オブジェクトメディアンで分割した場合の葉ノードの平均深さ

それぞれの葉ノードの深さが全て同じである可能性が高いといえる．つまり，このようなケースではバランス木が構築されている可能性が高い．

ノード数

次に，ノード数の変化について考察する．物体 cloister におけるノード数の変化を図 3.13 に示す．分岐数が 2 であるときには，空ノードがないため，内部ノード数と葉ノード数を足した数が全ノード数となる．内部ノード数と葉ノード数はグラフ上では同じように見えるが，実際には葉ノード数が 1 だけ多い．分岐数を増やした場合の内部ノード数は，分岐数が 2 のときよりも減ることはないが，単調減少しているわけでもない．同様に，空ノード数も単調増加しているわけではない．これが極端に現れているのが分岐数が 10 のときである．ここでは内部ノードと空ノードがともに減っている．逆に分岐数が 14 の時には空ノードが非常に増える．

分岐数が 10 のときに空ノードが少ない原因について考察する．分岐数を B，葉ノードの深さを D とすると，空ノードも含めた葉ノードの数 N は $N = B^D$ となる．分岐数が 8, 10, 12 のときの葉ノード数を計算し，表 3.2 にまとめた．ここで，単純化のために，全ての葉ノードの深さは一定であると想定している．

本論文では，葉ノードに入る三角形は 1 つとしている．そのため，物体を構成する三角形を BVH で表現するには，葉ノードの数が三角形数よりも多くなければならない．物体 cloister は約 8 万個の三角形から構成されているため，表 3.2 で太字で示した葉ノード数であれば，三角形を全て格納することができる．それぞれの分岐数に対して 1 つずつ太字で

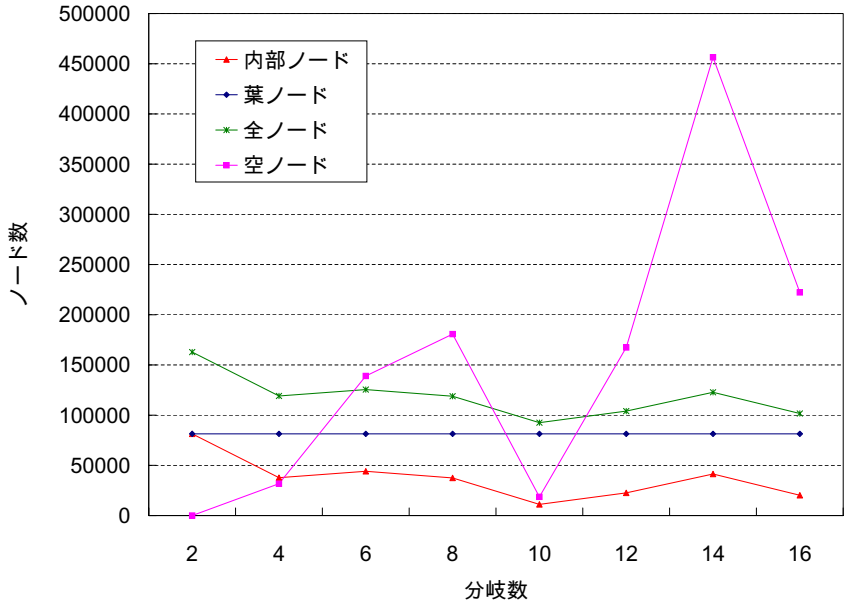

図 3.13: オブジェクトメディアンで分割した場合のノード数

表 3.2: 葉ノードの深さと分岐数による葉ノード数の変化

分岐数	全ての葉ノードの深さ		
	4	5	6
8	4,096	32,768	**262,144**
10	10,000	**100,000**	1,000,000
12	20,736	**248,832**	29,585,984

示した部分があるが，これらの中で最も数が小さいのが分岐数 10，葉ノード深さ 5 のときの 10 万である．他の場合には 20 万以上という数になっている．葉ノード数が少ないということは，空ノードの発生も少ないということを意味している．よって，分岐数が 10 のときは空ノード数が少なく抑えられている．

以上のことから，必ずしも分岐数 10 において空ノード数が少なくなるわけではないということが言える．物体 cloister の持つ三角形数は，分岐数 10，葉ノード深さ 5 において偶然にも空ノードが少なくなるような数であった．例えば三角形数が 24 万であれば，分岐数 12，葉ノード深さ 5 のときに空ノードが少なくなるであろう．

使用メモリ

最後に，BVH の構築に必要なメモリ量について考察する．BVH の構築に必要となるメモリ量は分岐数，内部ノード数，葉ノード数に依存しており，構造体 Node の定義から導くことができる．AABB は各座標の最大値と最小値の組み合わせ，つまり 6 つの 32 ビッ

トfloat値で表現できるので，そのサイズは24バイトとなる．子ノードはその木の分岐数によって異なる．これらの情報から，それぞれの分岐数で使用されているメモリ量が計算できる．

$$M_{in}(D) = 28 + 4 * D \tag{3.1}$$

$$M_{ln} = 32 \tag{3.2}$$

$$M_{all}(D, N_i, N_l) = N_i M_{in}(D) + N_l M_{ln} \tag{3.3}$$

M_{in} は内部ノードの使用メモリ，M_{ln} は葉ノードの使用メモリ，M_{all} はBVH全体の使用メモリである．また D は分割数，N_i は内部ノード数，N_l は葉ノード数である．式3.1を元に，分岐数を2から16の間で変化させたときの内部ノードの使用メモリ量を計算し，表3.3にまとめた．

表 3.3: 分岐数の変化による内部ノードの使用メモリ

分岐数	2	4	6	8	10	12	14	16
使用メモリ [Byte]	36	44	52	60	68	76	84	92

例えば分岐数を2から16に変化させた場合，内部ノードが使用するメモリは $92/36 = 2.56$ 倍となる．生成されるBVHの内部ノード数が2.56分の1よりも少なければ，分岐数を2として構築したBVHよりも使用メモリが少ないということになる．

図3.14に，各物体から構築したBVHが使用するメモリ量を示す．メモリ量は，実験から得られたノード数を式3.3に適用して計算した．

分岐数が2のときと比べて，他の分岐数ではほとんど使用メモリ量が少なくなっている．分岐数が多い部分では，1つ1つのノードが消費するメモリ量が多いため，空ノードが増えることによるメモリ消費が多くなる．全ての物体に対して特定の分岐数で使用メモリ量が最低になるということはないと思われる．これは先ほどノード数の変化でも考察したとおり，物体の持つ三角形の数に依存するからである．

3.5 コスト関数を用いた分割

3.5.1 分岐数を2より大きくしたSAHによる分割

本論文におけるコスト関数とは，子ノードを生成したときに，その探索にかかる計算量を見積もる関数のことである，これを最適化することで効率的な探索が可能なBVHが構築できる．レイトレーシングには *Surface Area Heuristic*（SAH）というコスト関数がよく用いられる．kd-treeの構築にはSAHが用いられることが多かった．BVHの構築については，Goldsmithらによるボトムアップな構築のためのコスト関数が提唱されていた [GS87]．しかし，この手法によってレイトレーシングはあまり高速化されなかった．Havranによ

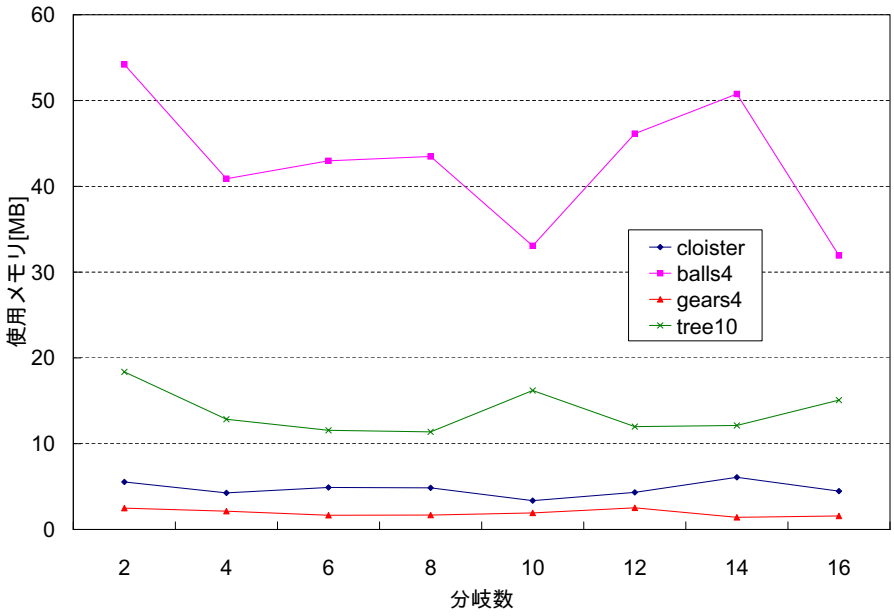

図 3.14: オブジェクトメディアンで分割した場合の使用メモリ

る空間データ構造の比較 [Hav01] では BVH の構築にこの手法が用いられたため，BVH が他の空間データ構造から非常に劣っているとされた．しかし，2006 年に Wald が 2 分木による BVH の構築に式 3.4 で示される SAH を適用し，変形するシーンに対する高速なレイトレーシングに成功した [WBS07].

$$T = 2T_{AABB} + \frac{A(S_1)}{A(S)}N(S_1)T_{tri} + \frac{A(S_2)}{A(S)}N(S_2)T_{tri} \tag{3.4}$$

ここで S は分割対象のノードに含まれる三角形の集合，S_1，S_2 は分割されたノードに含まれる三角形の集合である．関数 $A(S)$，$N(S)$ は，それぞれ S を取り囲む AABB の表面積，S に含まれる三角形の数を表す．T_{AABB}，T_{tri} は，それぞれレイと AABB，レイと三角形の交差判定にかかる計算時間である．

S を S_1 と S_2 に分ける組み合わせの数を減らすために，オブジェクトメディアンと同様にある軸に垂直な面での分割に限定する．軸については X，Y，Z の 3 軸について全て探索する．まとめると，SAH を用いた擬似コードは以下のようになる．

```
for each 軸
{
    軸に沿って三角形をソートする
    for each 分割面
    {
        分割面によって三角形を S1 と S2 のグループに分ける
        SAH を用いてコストを計算する
```

```
            コストが最小ならそのときの軸と分割面を記憶する
        }
    }
```

本論文は Wald が提案したこの手法を分岐数が 2 より多い BVH に対して適用する手法を提案する．例として分岐数が 4 の場合を考える．2 分木のときと同様に，分割対象のノードに含まれる三角形の集合を S とし，分割後の三角形の集合を S_1，S_2，S_3，S_4 とする．可能な組み合わせを減らすため，Wald の手法と同様にノードの分割をある軸に垂直な面に限定する．しかし 2 分木のときは分割面を $3(N(S)-1)$ 個の候補の中から探索すればよかったのに対し，4 分木のときの候補の数は爆発的に増える[4]．そのため，軸に垂直な面での分割に加え，さらなる制約条件をつけなければ計算時間が長くなってしまう．

そこで，組み合わせ数を抑えるために，SAH による分割を再帰的に用いる方法を提案する．分岐数を 4 にする場合は，最初に従来までの SAH による分割を用いて S を $S_{1,2}$ と $S_{3,4}$ の 2 つのグループに分ける．そして，分割されたそれぞれのグループに対して再度 SAH による分割を行い，$S_{1,2}$ を S_1 と S_2 に，$S_{3,4}$ を S_3 と S_4 に分ける．このとき，ソートに用いる軸は最初の分割で用いた軸を用いるため，本来の SAH による分割とは若干異なるアルゴリズムを用いる．このように 2 段階の分割を行うことによって，S を 4 つのグループに分けることができる．この手法は図 3.15 に示すとおり，従来の SAH を用いた 2 分木の構築と同じ計算をし，違う木を構築するものである．よって BVH の構築に要する計算時間は 2 分木のときとほぼ変わらない．

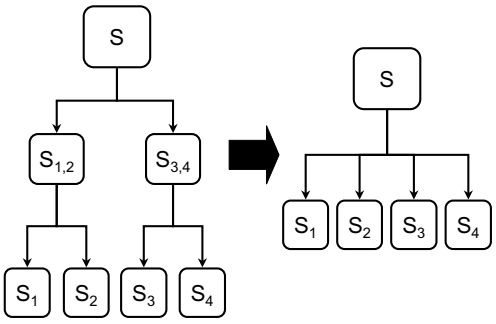

図 3.15: 4 分木の段階的構築

生成された 4 つの集合に対し，さらに SAH による分割を行うことで，分岐数を 8，16

[4]具体的な数は解析的に求められる．N 個の三角形に対して番号を 0 番から $N-1$ 番までつける．また，ソートする軸は簡単のため固定する．三角形の集合を分けるとき，0 番から i 番までと $i+1$ 番から $N-1$ 番まで分けるような切断面を切断面 i と呼ぶことにする．1 番目の切断面 P_1 が取りうる範囲は $[0, N-4]$．同様に 2 番目の P_2 は P_1 を用いて表すと $[P_1+1, N-3]$，3 番目の P_3 は $[P_2+1, N-2]$ となる．これから組み合わせの数を計算すると $\Sigma_{i=0}^{N-4}\Sigma_{j=i+1}^{N-3}\Sigma_{k=j+1}^{N-2}1$ となる．この式の展開は難解だが，大まかに言えば三角形の数 N に対して $O(N^3)$ で増えていく．さらに，これを拡張し分岐数を M とした場合，組み合わせは $O(N^{M-1})$ で増加する．

と増やしていくことができる．このように，この手法では三角形の集合を繰り返し 2 つに分割していくため，生成される木の分岐数は 2 の累乗に制限される．

3.5.2 実験と考察

SAH を用いて分割した BVH において，分岐数を 2 から増やした場合に，ノード数，また階層の深さがどのように変化するかを実験した．実験は，オブジェクトメディアンのときと同様に 4 種類の物体を対象にして行った．BVH の木構造の分岐数は 2，4，8，16 の 4 種類で行った．

葉ノードの平均深さ

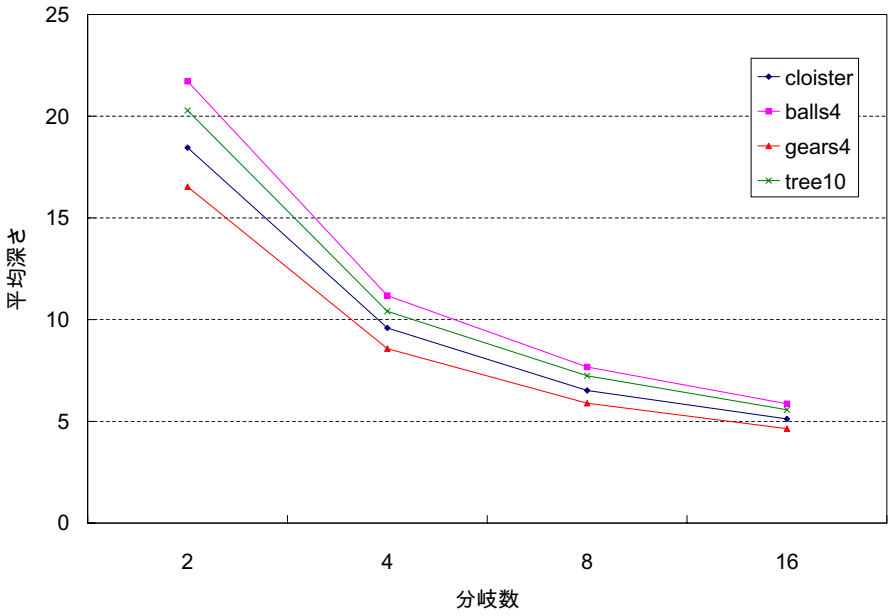

図 3.16: SAH による分割をした場合の葉ノードの平均深さ

まず，葉ノードの平均深さについて考察する．各物体に対する葉ノードの平均深さの変化は図 3.16 のようになった．分岐数が 2 の時には，オブジェクトメディアンの結果と比較して平均深さが 1 割程度深くなっている．分岐数を 2 から増やしていったときの平均深さの変化は，オブジェクトメディアンのときと変化はない．一般に階層構造の探索は深さが浅いほど計算量が少ないものとされる．しかし，SAH は探索コストを見積もった上で階層構造を構築しているため，階層が深くなることだけで速度が悪化するとは言い切れない．

このことを検証するため，追加的な実験を行った．実験はオブジェクトメディアンと SAH の 2 つの手法で構築された BVH を用いてレンダリングを行うものである．実験の結

果得られたレンダリング時間を図 3.17 に示す．それぞれのグラフは，OM で示したものがオブジェクトメディアンでの分割，SAH で示したものが SAH を用いた分割を用いた場合の結果である．この結果から，階層が深いにも関わらず，明らかに SAH のほうが効率的な BVH を構築できていることが分かる．

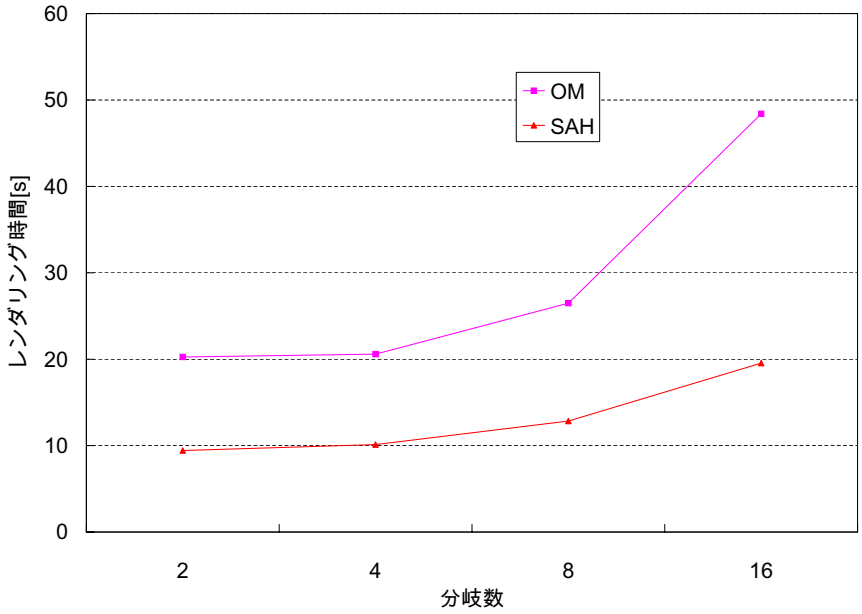

図 3.17: BVH の構築手法によるレンダリング時間の変化

ノード数

次に，ノード数の変化について考察する．図 3.18 に全体のノード数と空ノード数の変化を示した．

オブジェクトメディアンでは，分岐数が 8 のとき極端にノード数が増えるといったような不安定な変化を示していた．それに対して SAH を用いた場合にはノード数が滑らかに変化していることがわかる．SAH を用いて分割した場合は，様々な分岐数に渡ってノード数を平均的に少なくする効果が出ているのではないかと考えられる．

使用メモリ

最後に，BVH の構築に必要なメモリ量について考察する．オブジェクトメディアンのときと同様に BVH の構築に必要なメモリ量を求め，図 3.19 にまとめた．

オブジェクトメディアンでは分岐数を変化させたときの使用メモリ量の変化が，物体によってまちまちであった．例えば，分岐数が 10 のとき，cloister の使用するメモリは他の分岐数に比べて低くなったが，tree10 の使用するメモリは逆に多くなっていた．しか

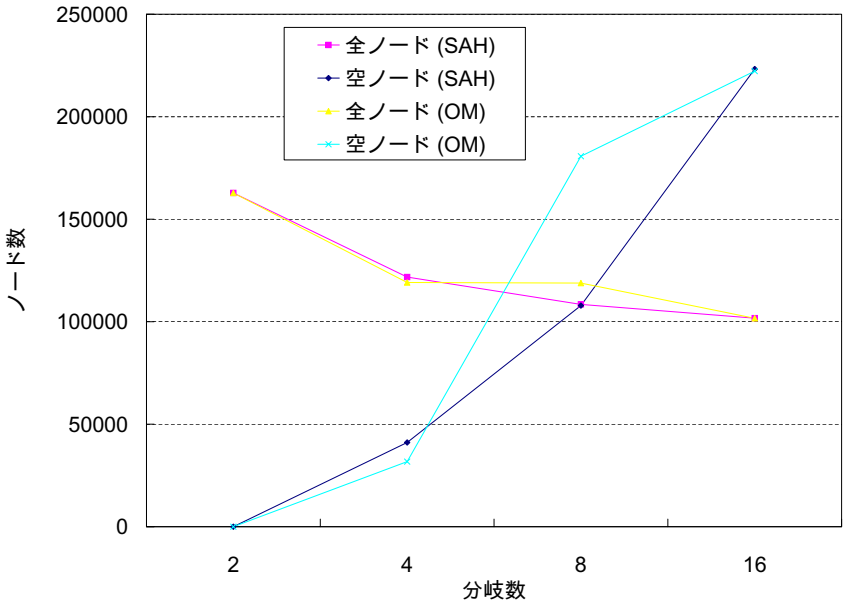

図 3.18: SAH による分割をした場合のノード数

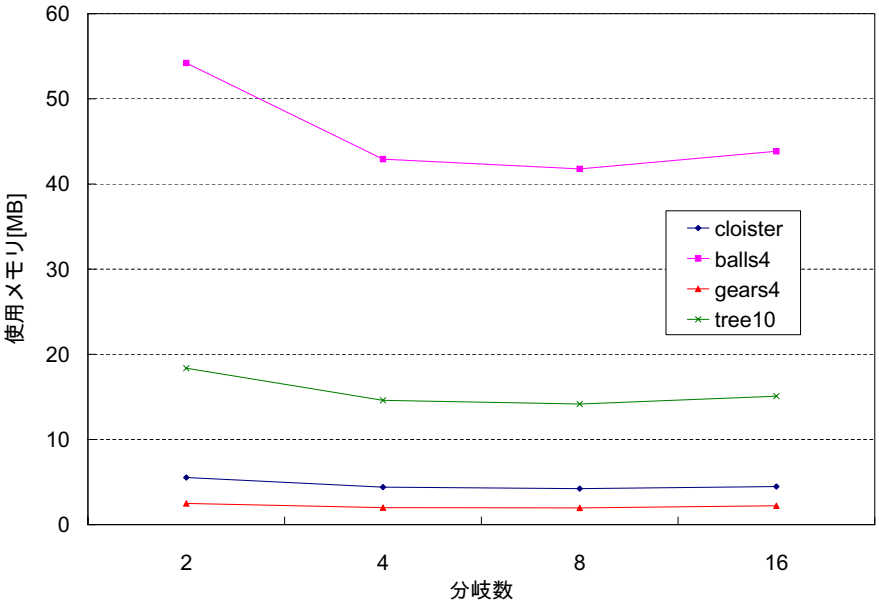

図 3.19: SAH による分割をした場合の使用メモリ

しSAHを用いた実験結果では，物体を問わずに使用メモリ量の変化は一定である．分岐数が4と8のときに使用メモリが少なくなり，16になるとやや増える．理由については，ノード数の変化で述べたとおりSAHが平均的にノード数を少なくした結果であると考えられる．

3.6 まとめ

従来まで2分木で構築されることが多かったBVHに対して，分岐数を増やす手法を提案し，生成されたBVHの構造を検証した．

まず，BVHによってレイトレーシングが高速化される理由について述べ，その構築手法について述べた．そして木構造の分岐数を増やすことでどのような変化が起こるかについて考察した．分岐数が2の場合には，どのような木構造を作ってもノード数は一定である．しかし分岐数を増やすことによってノード数が増加する場合があり，そのようなときには空ノードが発生する．

次に，BVHの構築段階でノードを分割する手法として，オブジェクトメディアンでの分割と，SAHを用いた分割の2種類をとりあげ，それぞれの手法について分岐数を2から増やす手法について提案した．オブジェクトメディアンでの分割については，ノードの持つ三角形を分岐数で等分して子ノードの持つ三角形とした．SAHを用いた分割については，三角形を一度2つのグループに分割した後，それらを繰り返し分割することで子ノードの持つ三角形とした．オブジェクトメディアンでの分割では分岐数が任意に指定できるのに対し，SAHを用いた分割では分岐数は2の累乗に制限される．

最後に，提案した手法を用いてBVHを構築し，BVHのノード数，空ノード数，葉ノードの平均深さ，使用メモリをまとめた．分岐数を2から増やすことで使用メモリが減る場合が多かった．SAHを用いて分割したときには，分岐数を変えた場合の使用メモリがどの物体に対しても同じような変化を示した．また，葉ノードの平均深さはオブジェクトメディアンで分割したときのほうが浅いが，レンダリング速度はSAHを用いた分割のほうが速い．これらの結果から，SAHを用いてBVHの分岐数を増やせば，使用メモリが少なく，高速なレイトレーシングが可能であると考えられる．

第4章　SIMD命令による4分木BVHのトラバース

本章では，4分木BVHのトラバーサルにSIMD命令を用いて高速化する手法について述べる．まずプログラムの実行速度を加速する手法のSIMD命令と，それに適したデータ構造について説明する．そして4分木BVHをトラバースするときに効率的にSIMD命令を用いる手法を提案する．最後に提案手法を用いたレイトレーシングによるレンダリングを実装して実行時間を計測し，考察する．

4.1　SIMDによる並列演算

4.1.1　SIMD化されるCPU

SIMD（*Single Instruction Multiple Data*）とは，近年のCPUアーキテクチャの進化において，マルチコアとともに注目されるキーワードである．SIMDを用いることで，複数のデータに対する同時並行計算が可能となる [Inta].

IntelはIA-32アーキテクチャと呼ばれるCPUの設計コンセプトの下，20年以上にわたる技術革新を続けてきた．その歴史の中，MMXをはじめとし，SSE（Streaming SIMD Extentions），SSE2，SSE3といった技術の中で，SIMDというキーワードが登場してきた[1].

MMX Pentium IIやPentium MMXに採用された技術で，図4.1に示すような8つの64ビットレジスタ（MMXレジスタ）を用いた整数の並列演算が可能である．このレジスタに収めるデータとしては，8つのバイト型（8ビット整数），4つのワード型（16ビット整数），2つのダブルワード型（32ビット整数）の3つがある．8つのバイト型のように，複数の値をまとめたデータ型をpacked型（packed type）という．

SSE Pentium IIIに採用された技術で，図4.2に示すような8つの128ビットレジスタ（XMMレジスタ）を用いることで，packed，またはスカラ（scalar，単一のデータと

[1] ここで述べている内容については，IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture[Inta] から引用している．CPUベンダが公開している資料は何千ページという量に及ぶが，一次情報源からの情報は貴重であるため，ダウンロードだけでもしておくことをお勧めする．

いう意味で packed の対義語）の単精度浮動小数点[2]を扱うことができる．また MMX レジスタは 16 個に拡張されている．

SSE2 Pentium 4 や Intel Xeon に採用された技術で，XMM レジスタの用途に packed 倍精度浮動小数点[3]と 128 ビット packed 整数が追加されている．

SSE3 ハイパースレッディング（Hyper-Threading）をサポートした Pentium 4 に採用された技術で，新規命令に加え，浮動小数点から整数へのデータ変換やスレッド同期が可能となっている．

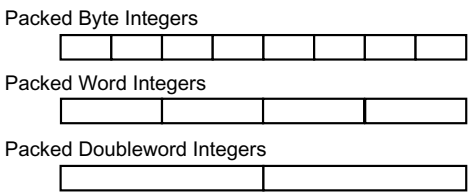

図 4.1: MMX レジスタ

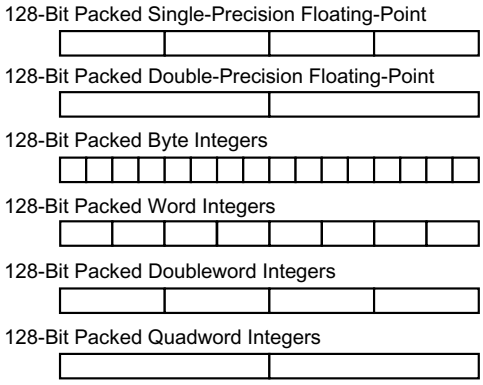

図 4.2: XMM レジスタ

このように，CPU にビット長の長いレジスタが搭載されることによって，複数のデータを並列に処理することができるようになった[4]．このことを表す言葉として single-instruction multiple-data，略して *SIMD* という単語が用いられるようになった．そしてそのような処理をプログラムするために，新たな命令（instructions）が追加されていった（今後このような複数データを対象とした命令を *SIMD* **命令**と呼ぶ）．SIMD 命令には，四則演算だけでなく，平方根を求める演算や，最大値，最小値を求める演算など，プログラムの高速化に有用な命令が多数ある．今後単に SSE と言った場合には，MMX 等で追加された命令のことを指す．

AMD の CPU は，当初 *3DNow!* という独自の SIMD 命令を持っていたが，近年では Intel の SSE と互換の命令を実装するという戦略に移行している [AMD]．また，Motorola の G4，IBM の G5 といった Macintosh 向けの CPU でも *AltiVec*[App] を使用した SIMD プログラミングが可能である．しかし，Apple が Intel の CPU を Macintosh に搭載することにしたため，消費者向けコンピュータにおいて AltiVec を使用する機会は少なくなった．他に AltiVec が使用できる CPU としては，Cell プロセッサ[5]が挙げられる．

[2]いわゆる 32 ビット浮動小数点で，C/C++言語では float 型である．
[3]いわゆる 64 ビット浮動小数点で，C/C++言語では double 型である．
[4]2007 年には新しく SSE4 が登場する予定である．
[5]正確に言えば，Cell プロセッサ内部の PPE で AltiVec を用いることができる．

4.1.2 SIMD 命令

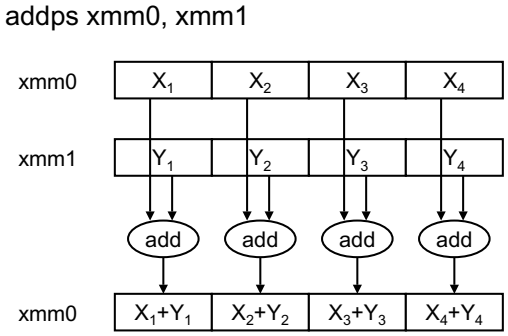

図 4.3: SIMD 命令の例

SSE を用いた演算の例として addps を取り上げる．addps は，図 4.3 のように単精度浮動小数点の加算を 4 つ同時に行う．加算に用いられる引数は 128 ビット長の XMM レジスタに格納される．addps はこれら 4 つの値に対して同時に加算演算を行うことができる．SSE 等の命令セットには他にも多数の命令が用意されており，これらを効率的に用いることで今までより数倍の実行速度を持つソフトウェアを書くことができる．

addps はアセンブラ言語での命令なので，これらを直接用いてプログラムするのは手間がかかるとともに，コンパイラによる最適化が期待できない．このような問題を解消するために，*intrinsics* というものが提供されている [Intb]．intrinsics は形式上は C/C++言語の関数であるが，コンパイル時にインライン展開されることで関数の呼び出しコストがかからず，実行速度はアセンブラ言語を直接書いた場合と変わりない．例えば，addps 命令に対する intrinsic としては_m128_add_ps がある．intrinsics を用いるときには__m128 というデータ型が用いられる．これは 128 ビットのデータであり，XMM レジスタに直接収めることができる．intrinsics や__m128 データ型を用いることで，通常の C/C++言語のプログラムに対する自然な形での SIMD 命令を用いた高速化をすることができる．

4.1.3 3 次元座標を取り扱うプログラミングへの応用

一般に 3 次元座標は $X_1, Y_1, Z_1, X_2, Y_2, Z_2, ...$ という順序でメモリ上に配置されている．このようなデータ構造は Array of Structures（AoS）と呼ばれている．AoS は文字通り構造体の配列であり，例としては次のようなコードが挙げられる．

```
struct Vertex {
    float x, y, z;
} vertices[4];
```

vertices は，構造体を要素として持つ配列変数である．これに対し Structure of Arrays（SoA）は X_1, X_2, ..., Y_1, Y_2, ..., Z_1, Z_2, ... という配置である．

```
struct Vertex {
    float x[4], y[4], z[4];
} vertices;
```

vertices は，配列を要素として持つ構造体である．AoS と SoA のメモリ上での配置を比較すると図 4.4 のようになる．

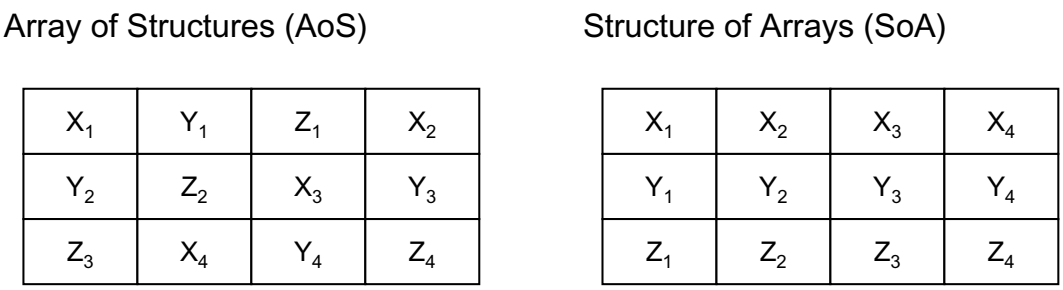

図 4.4: AoS と SoA のデータ構造

ここで，SoA の構造体が要素として持つ変数が 4 つの float 型で表されていることに着目する．このような 4 つの単精度浮動小数点の値は，XMM レジスタで扱うことができる __m128 型を用いて表すことができるため，SoA のデータ構造は以下のようにも書くことができる．

```
struct Vertex {
    __m128 x, y, z;
} vertices;
```

__m128 型にしたデータに対して SIMD 命令を用いることによって，4 つの 3 次元座標に対する並列処理が可能となる．__m128 型を用いて float 型の変数を処理したい場合には，float 型の配列を __m128 型に読み込む _mm_load_ps や，逆に __m128 型から float 型の配列に読み込む _mm_store_ps 等の intrinsics を用いる．

SIMD 命令は従来までのレイトレーシングの高速化に関する研究でもその有効性が指摘されてきた．Geimer らは SIMD 命令を用いたレイトレーシングのプログラムを複数のプラットフォーム上で動かせるフレームワークを提案した [GM03]．Reshetov らや Wald らは，SIMD 命令を効率的に利用してインタラクティブなフレームレートでのレイトレーシングによるレンダリングを実現した [RSH05][Wal04]．これらの研究では，複数のレイをまとめてトレースすることで高速化を実現している．

4.2　4分木BVHのトラバース

4分木では，あるノードの下には4つの子ノードがある．通常は，それらの子ノードが持つAABBに対して個別にレイとの交差判定を行い，交差すればその子ノードから先へと探索を進める．本論文では，SIMD命令を用いることによって，4つの子ノードが持つAABB全てに対して同時にレイとの交差判定を行う手法を提案する．

4.2.1　データ構造の改善

ここまで用いてきたデータ構造では，各ノードは子ノードの持つボリュームを全て囲むボリュームを保持していた．このままのデータ構造で4つの子ノードのボリュームに対する並列演算をしようとした場合，子ノードの持つボリュームデータをいったん全て収集し，それらをSoAとして並べ替える必要が生じる．そのため，ノードが子ノードのボリュームを直接持つようなデータ構造に改良した．ノードを表す構造体は以下のようになる．

```
struct Node
{
    __m128 aabbMin[3], aabbMax[3];
    int index;
    Node* children[4];
};
```

4.2.2　必要な情報の事前計算

トラバースをする過程で，行われるレイとAABBとの交差判定の回数は非常に多い．例えば，物体cloisterを解像度800x600でレンダリングした場合の交差判定の回数は約1,300万回である．そのため，交差判定の計算は必要最低限である必要がある．本論文では，トラバースの前に計算可能な情報は計算しておくという手法をとった．

SIMD命令を用いて1本のレイと4つのAABBとの交差判定をするためには，まず1本のレイをSoAの形で4本に複製する必要がある．そして1本のレイと1つのAABBとの交差判定を，SIMD命令によって4つ同時に行う．レイはトラバースの過程で内容が変化することがない．そのため事前に4本に複製しておくことでトラバース過程での計算量を減らすことができる．また，レイとAABBとの交差判定を計算するときに，レイの方向ベクトルの逆数を使用する．これについても事前に計算を行った．

4.2.3　レイとAABBとの交差判定のSIMD化

レイとAABBとの交差判定にはslabを用いた手法が広く用いられている[PH04]．slabとは2枚の並行する平面にはさまれた区間のことであり，AABBは軸に垂直な3つのslab

を組み合わせたものとして考えることができる．本論文ではその手法を SIMD 命令が有効に活用される形に変更して使用する．以下に使用したレイと AABB との交差判定を行うプログラムを示す.

```
__m128 tNear, tFar, cmp, t0, t1;

t0 = _mm_set1_ps(ray.mint);
t1 = _mm_set1_ps(ray.maxt);

// X 軸
tNear = _mm_mul_ps(_mm_sub_ps(aabbMin[0], ray.o4[0]),
                   ray.invRayDir[0]);
tFar = _mm_mul_ps(_mm_sub_ps(aabbMax[0], ray.o4[0]),
                  ray.invRayDir[0]);
t0 = _mm_max_ps(_mm_min_ps(tNear, tFar), t0);
t1 = _mm_min_ps(_mm_max_ps(tNear, tFar), t1);

// Y 軸
tNear = _mm_mul_ps(_mm_sub_ps(aabbMin[1], ray.o4[1]),
                   ray.invRayDir[1]);
tFar = _mm_mul_ps(_mm_sub_ps(aabbMax[1], ray.o4[1]),
                  ray.invRayDir[1]);
t0 = _mm_max_ps(_mm_min_ps(tNear, tFar), t0);
t1 = _mm_min_ps(_mm_max_ps(tNear, tFar), t1);

// Z 軸
tNear = _mm_mul_ps(_mm_sub_ps(aabbMin[2], ray.o4[2]),
                   ray.invRayDir[2]);
tFar = _mm_mul_ps(_mm_sub_ps(aabbMax[2], ray.o4[2]),
                  ray.invRayDir[2]);
t0 = _mm_max_ps(_mm_min_ps(tNear, tFar), t0);
t1 = _mm_min_ps(_mm_max_ps(tNear, tFar), t1);

cmp = _mm_cmpgt_ps(t0, t1);
```

交差判定内の全ての計算は intrinsics によって行われる．注釈が必要と思われる変数に関しては表 4.1 にまとめた.

レイと AABB との交差判定の結果は cmp に格納される．cmp を構成する 4 つの 32 ビット値が，それぞれの AABB に対して交差したかどうかを判定する値となっている．レイ

表 4.1: 交差判定に使用する変数

変数名	型	意味
aabbMin	__m128[3]	AABB の各軸の最小値
aabbMax	__m128[3]	AABB の各軸の最大値
ray.o4	__m128[3]	事前に複製したレイの原点
ray.invRayDir	__m128[3]	事前に複製したレイの方向ベクトルの逆数
cmp	__m128	交差判定の結果

と AABB が交差している場合には，この値が 0 となる．交差していない場合には 32 ビット全てが 1 となる．

4.2.4 空ノードをトラバースしないための対策

BVH を構築するときに空ノードが発生する問題については 3 章で述べたとおりである．ここでは空ノードをトラバースしないための対策について述べる．4 分木においては，あるノードの子ノードは 4 つ必要である．しかし，空ノードが発生するようなケースでは子ノードが不足する．例として，A と B の 2 つしか子ノードがない場合を想定する．本論文では，このようなケースでまず [-, -, A, B] というように後方にノードを詰める．そして余った前方の子ノードに，A を繰り返して格納し，最終的に [A, A, A, B] とする．

トラバースをするときには，まず SIMD 演算によるレイと AABB との交差判定を一斉に行う．次に，cmp の情報を元に，1 番目の子ノードの AABB がレイと交差するならトラバースを進める．2 番目の子ノードについては，AABB とレイが交差していて，かつノードが示す先が 1 番目の子ノードと異なる場合のみ，トラバースを進める．3 番目以降についても同様である．これによって空ノードに対する余計なトラバースを防ぐ．以下にトラバースに使用したプログラムを示す．

```
const int allBits = _mm_movemask_ps(cmp);
if (allBits != 0xf)
{
    if ((allBits & 1) == 0)
        Traverse(node->children[0], ray);
    if ((allBits & 2) == 0 && node->children[1] != node->children[0])
        Traverse(node->children[1], ray);
    if ((allBits & 4) == 0 && node->children[2] != node->children[0])
        Traverse(node->children[2], ray);
    if ((allBits & 8) == 0 && node->children[3] != node->children[0])
        Traverse(node->children[3], ray);
}
```

ここでは cmp が表している交差判定の結果を 4 ビットにまとめるために _mm_movemask_ps という intrinsic を使用している．また，本論文ではレイの原点から近い AABB との交差判定を優先的に行うため，Ordered Traversal[WBS07] を実装している．そのため，ここに示した子ノードを 1 番目から順にトラバースするプログラムとは逆に，4 番目からトラバースするプログラムもある．

4.3 実験と考察

SIMD 命令を用いた 4 分木のトラバースを実装し，レイトレーシングによるレンダリングに要する実行時間を計測した．BVH の構築は 3 章で述べた SAH による分割を用いて行い，対象も 3 章で用いた 4 種の物体とした．実験には，Intel Core Duo T2050 1.60GHz, メモリ 1GB を搭載した Windows PC を用い，コンパイラには Microsoft Visual C++ 2005 を用いた．出力画像の解像度は横 800，縦 600 ピクセルである．シェーディングについては，レイの方向ベクトルと三角形の法線の内積を取り，その値で画素の濃さを決める方法を取った．これはカメラ位置に光源があることに近い．レンダリングの計測の結果得られたレンダリング時間を図 4.5 に示す．

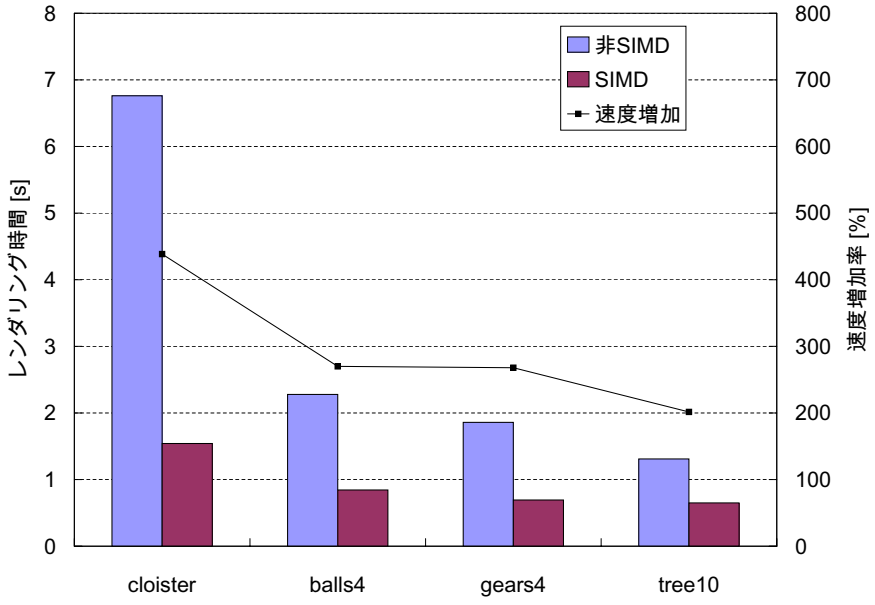

図 4.5: レンダリング時間

レンダリング速度は，SIMD 命令を用いることによって 2.32–4.45 倍に改善された．また，レンダリング時間がかかっている場合のほうがより速度が改善されている．SPD で生成された物体（balls4，gears4，tree10）は，大きな板の上に物体が乗っている形をして

おり，レイの大部分はその板だけに交差する．これとは反対に，cloister はほとんどのレイが物体の複雑な位置に交差する．そのためこのような速度差が出ていると考えられる．

速度の改善が最高でも 4.45 倍という結果は，単純に SIMD 命令を用いた場合の効果が 4 倍と考えると，それ以上の結果であるといえる．これには事前計算や，木構造を 4 分木としたことが影響していると考えられる．例えば，Geimer ら [GM03] によれば，Xeon プロセッサを用いて SIMD 命令を用いたレイトレーシングをした場合，2.4–3.9 倍の速度改善が見られた．実装上の細かい差はあると思われるが，本論文の結果はこれより優れたものであった．

しかし，レイトレーシングを高速化する最先端の研究において，Reshetov ら [RSH05] や Wald ら [WBS07] は 10 倍程度での速度でレイトレーシングを行っている．これにはトラバースにレイのコヒーレンス[6]を用いるかどうかということの影響が大きいと思われる．パケットと呼ばれるレイの束を用いて複数のレイを同時にトラバースする手法はパケットトラバースと呼ばれている．これは，1 本のレイに対して複数の AABB と交差判定をするという本論文で用いている手法とは対極に位置するものである．4 分木の BVH に対してもパケットトラバースが可能となれば，使用メモリ量の削減という利点を生かしたまま，より高速なレイトレーシングが可能となると考えられる．他にも様々な最適化の手法があるが，これらを 4 分木に対しても適用していくことが今後必要であると思われる．

4.4 まとめ

3 章で提案した 4 分木 BVH に対して，SIMD 命令を用いた効率的なトラバースを行う手法を提案した．

まず，近年の CPU が SIMD 化されており，並列計算を用いることで CPU を最大限に利用することが可能となっていることについて述べた．SIMD 命令は，複数のデータに対して同時に同じ演算を行う命令である．このような命令を効率的に用いるプログラムを書くには，データ構造を検討する必要がある．

次に，SIMD 命令を用いて 4 分木 BVH をトラバースする方法について提案した．4 つの AABB との交差判定を行うために，まずノードのデータ構造を適したものへと改善した．また，レイの複製など，トラバース中に変わらない情報については事前に計算するようにした．そして SIMD 命令の intrinsics を用いたプログラムを実装し，レイトレーシングによるレンダリングを行った．結果は SIMD 命令を用いない場合と比較して 2.32–4.45 倍に速度が上昇した．

[6] レイ同士の類似性のこと．主に方向が似ているレイをまとめてトラバースすることが多い．

第5章　まとめと課題

本章では，本論文のまとめとして，提案したBVHの構築とトラバースの手法，ならびにその効果などについて述べる．そして今後改善が求められる課題について述べる．

5.1　本論文のまとめ

本論文は，写実的なCGのレンダリングを可能とするレイトレーシングの高速化について述べたものである．

2章では写実的なCGのレンダリングを可能とするレイトレーシングの基本的なアルゴリズムと，従来の高速化手法について紹介した．高品質なレンダリング結果を得るにはレイを増やす必要があり，計算時間が長くなる．それを改善するために多くの高速化手法が研究されているが，それらの中でも空間データ構造を改善することが効果的である．また，ハードウェアの機能を活用したレイトレーシングの実装することでさらに高速化することができる．

3章では，Bounding Volume Hierarchies（BVH）の木構造の分岐数を増やす手法を提案し，構築されたBVHの比較を行った．BVHは物体を取り囲むボリュームの階層構造であり，レイトレーシングを高速化するために用いられる空間データ構造の1つとして用いられている．この階層構造は一般に2分木として構築される．本論文では，これを4分木，またさらに分岐数が多い木としてBVHを構築する手法を提案した．まず，一般の2分木を4分木にした場合に木構造にどのような変化が生じるかについて理論的な考察を行い，空ノードの発生，使用されるメモリ量の変化などについて述べた．そして実際に2つの階層構造の構築手法を用いた実験を行った．1つ目の構築手法は，物体をオブジェクトメディアンで分割する手法である．この手法は容易に分岐数を増やすことができる．構築されたBVHは階層が浅く，分岐数を変えた場合の使用メモリ量の変化が大きいことが明らかになった．2つ目の構築手法は，SAHというコスト関数を用いて物体を分割する手法である．分岐数の拡張については，分割する場所の探索範囲を狭くするために，2分木BVHを構築する手法を複数回適用するという手法を提案した．構築されたBVHは，階層がやや深くなるが，トラバースの速度はオブジェクトメディアンより速いことが明らかになった．また，分岐数を変えた場合の使用メモリ量の変化が少ないことも明らかになった．

4章では，提案した手法によって構築された4分木に対してSIMD命令を用いてトラバースする手法を提案した．まず，近年のCPUにおいてSIMDによる並列計算の機能が発展し，それらを活用することによってより高速な計算ができることについて紹介した．

次に，SIMD 命令によってレイトレーシングを高速化する手法を提案した．ノードのデータ構造は，あるノードが持つ 4 つの子ノードの AABB を SIMD 命令に適したデータ構造に書き換えて保持させるものとした．そして SIMD 命令の intrinsics を用いた並列演算によってレイトレーシングによるレンダリングを行った．結果は SIMD 命令を用いない場合と比較して最大で 4.45 倍に高速化された．

5.2 今後の課題

3 章での実験から，BVH の木構造の分岐数を増やすことによって使用メモリ量が削減されることは確認できた．しかし，高速化については 4 章で述べたとおり，満足の行く結果は得られなかった．より高速なレイトレーシングを実現するためには，近年の研究成果で用いられている技術を 4 分木にも適用していく必要があるだろう．

また，本論文が 1 本のレイをそれぞれトラバースする手法を用いている背景には，写実的なレンダリングではコヒーレンスが乱れることが多く，パケットトラバースが機能しない可能性があると考えているからである．この問題については推測を元に述べているが，実際にそのようなレンダリングを行った場合にパケットトラバースと速度を比べる必要があると思われる．そのためにも，本手法を用い大域照明を考慮した写実的なレンダリングを実装する必要があるだろう．

本研究に関する研究発表

1. 木村秀敬，宮田一乗．レイトレーシングのための 4 分木バウンディングボリューム階層の構築．映像表現フォーラム：画像電子学会 第 232 回研究会，March，2007.（発表予定）
2. 木村秀敬，宮田一乗．SIMD 命令を用いた木探索によるレイトレーシングの高速化．CG アニメーションカンファレンス，March，2007.（発表予定）

参考文献

[AKSS02] Kurt Akeley, David Kirk, Larry Seiler, and Philipp Slusallek. When will ray-tracing replace rasterization? In *SIGGRAPH '02: ACM SIGGRAPH 2002 Panels*, 2002.

[AMD] AMD. AMD developer central - documentation. http://developer.amd.com/documentation.jsp.

[App] Apple. Velocity engine. http://developer.apple.com/hardwaredrivers/ve/index.html.

[BWSF06] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray tracing on the cell processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The Ray Engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, Vol. 19, No. 10, pp. 547–554, 1976.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pp. 137–145, New York, NY, USA, 1984. ACM Press.

[GM03] Markus Geimer and Stefan Müller. A cross-platform framework for interactive ray tracing. In *Graphiktag im Rahmen der GI Jahrestagung*, Frankfurt am Main, September 2003.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, Vol. 7, No. 5, pp. 14–20, 1987.

[Hai87] Eric Haines. A proposal for standard graphics environments. In *IEEE Computer Graphics ans Applications*, Vol. 7, pp. 3–5, 1987. http://www.acm.org/tog/resources/SPD/.

[Hav01] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, the Faculty of Electrical Engineering, Czech Technical University, 2001.

[Inta] Intel. Intel 64 and IA-32 architectures software developer's manuals. http://www.intel.com/products/processor/manuals/index.htm.

[Intb] Intel. Introduction to intrinsics. http://www.intel.com/cd/ids/developer/asmo-na/eng/59644.htm.

[Intc] International Business Machines. The Cell Project at IBM Research. http://www.research.ibm.com/cell/.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pp. 21–30, London, UK, 1996. Springer-Verlag.

[JZ] Joseph D. Wieber, Jr. and Gary M. Zoppetti. How to use intrinsics. http://shareit.jotxpert.net/WikiHome/Articles/59645.

[KK86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pp. 269–278, New York, NY, USA, 1986. ACM Press.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, Vol. 2, No. 1, pp. 21–28, 1997.

[MW04] Jeffrey Mahovsky and Brian Wyvill. Fast ray-axis aligned bounding box overlap tests with plucker coordinates. *The Journal of Graphics Tools*, Vol. 9, No. 1, pp. 37–48, 2004.

[NVIa] NVIDIA. GeForce 8800. http://www.nvidia.com/page/geforce_8800.html.

[NVIb] NVIDIA. NVIDIA CUDA. http://www.nvidia.com/object/cuda.html.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 703–712, New York, NY, USA, 2002. ACM Press.

[PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[Pha05] Matt Pharr. *GPU Gems 2*. Addison Wesley, 2005.

[RSH05] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *Proceedings of ACM SIGGRAPH '05*, New York, NY, 2005. ACM Press.

[RW80] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pp. 110–116, New York, NY, USA, 1980. ACM Press.

[SM03] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing, Second Edition*. A K Peters, Ltd., Natick, MA, USA, 2003.

[Smi98] Brian Smits. Efficiency issues for ray tracing. *J. Graph. Tools*, Vol. 3, No. 2, pp. 1–14, 1998.

[Son] Sony Computer Entertainment Inc. Cell broadband engine. http://cell.scei.co.jp/.

[SS05] Peter Shirley and Philipp Slusallek. Introduction to real-time ray tracing. In *ACM Siggraph 2005 course 38*. ACM Press, 2005.

[VG97] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, Vol. 26, No. 1, 2007.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, Vol. 23, No. 6, pp. 343–349, 1980.

[WIK+06] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 2006. (Proceedings of ACM SIGGRAPH 2006, to appear).

[宮田 05] 宮田一乘, 高橋誠史, 黒田篤. GPU コンピューティングの動向と将来像. 芸術科学会論文誌, Vol. 4, No. 1, pp. 13–19, March 2005.

[芸術 03] 芸術科学会. CG 基礎用語辞典 誰にでもわかるコンピュータグラフィックス. 芸術科学会 出版部, 2003.