

平成 17 年度卒業論文

3 D ポリゴン半透明表示法 B S P の
改良アルゴリズム

指導教員
佐々政孝教授

東京工業大学情報科学科

田中 雄次郎

概要

2次元多角形のポリゴンによって表現された3次元モデルを高速に画面に表示する技術はコンピュータグラフィックスにおいて基幹をなすものであり、長年にわたって研究されてきたテーマの一つである。

中でも、複雑な構造を持つ機械モデルや建築モデルの内部を透視したり、着目する部品を強調するために他の部分を透過性にしたたりする場合に、それら透過部分のポリゴンを半透明にして表示する技術が必要となる。また、雲や霧、水、ガラス製品、光の効果などコンピュータグラフィックスで扱われる多くのオブジェクトの表示に半透明ポリゴンが用いられることが多い。

これら半透明なポリゴンを多数含むシーンにおいて視点をリアルタイムに移動させながら画面へのモデル表示を行うには、毎フレームごとに視点から見たポリゴンどうしの前後関係を高速に求める必要がある。

従来、前処理としてポリゴンを2分木状に構造化しておくことで、正確な前後関係を求める方法が広く利用されてきた。しかし、視点位置が変わるたびに、木のすべてのノードを辿る必要があるこれらの従来手法においては、面数に比例して辿る時間が増大するため、大規模なモデルではリアルタイムな処理が困難となる。

そこで、様々な高速化手法も試みられてきたが、一般に木を辿る時間と木のサイズとはトレードオフの関係にあり、メモリコストを現実的な値に抑えながら高速化を実現することがこれまでの課題であった。

本研究では、2分木を辿る手法を改良し、処理時間を低減する新しい手法を提案する。

また、今日一般に用いられるポリゴンモデルに適用したとき、従来手法よりも高速で実現可能であることを実験により示す。

目次

1	序論	1-1
1.1	はじめに	1-3
1.2	本研究の目的	1-4
1.3	本研究の構成	1-4
2	関連研究および解決すべき課題	2-1
2.1	関連研究	2-2
2.1.1	可視面決定手法	2-2
2.1.2	zバッファ法	2-2
2.1.3	優先順位アルゴリズム	2-3
2.2	半透明ポリゴンの扱いについて	2-8
2.2.1	アルファブレンディング	2-8
2.2.2	ハードウェアを用いた混合処理	2-9
3	改良アルゴリズム	3-1
3.1	アルゴリズムの概要	3-2
3.2	アルゴリズムの流れ	3-2
3.3	初期リストの構築	3-2
3.4	ノードへの参照を持つ配列の構築	3-4
3.5	リストの入れ替え	3-5
4	実験および考察	4-1
4.1	評価実験のためのモデルおよび実験環境	4-2
4.2	結果	4-4
4.2.1	メモリコスト	4-4
4.2.2	実行時間	4-4
5	まとめと展望	5-1
5.1	まとめ	5-2
5.2	展望	5-2

目次

2.1	ポリゴン間の優先順位が定まらない例	2-4
2.2	BSP 木の構築	2-6
2.3	BSP 木ノードのデータ構造	2-6
2.4	BSP 木構築のための擬似プログラム	2-7
2.5	BSP 木トラバースのための擬似プログラムコード	2-8
2.6	アルファブレンディング	2-9
2.7	Depth Peeling 法	2-10
3.1	改良アルゴリズム	3-3
3.2	リストのデータ構造	3-3
3.3	BSP 木と構築されるリストの関係	3-4
3.4	リストの入れ替え	3-5
3.5	ノードへの参照を持つデータ構造	3-5
3.6	リストと SwitchingPolygon の配列との関係	3-6
3.7	next・prev の付け替え	3-6
4.1	PUMP (4,904 polygons).	4-2
4.2	CAR (10,093 polygons).	4-2
4.3	HORSE (96,966 polygons).	4-3
4.4	PUMP の結果	4-4
4.5	CAR の結果	4-5
4.6	HORSE の結果	4-5
4.7	PUMP のグラフ	4-6
4.8	CAR のグラフ	4-7
4.9	HORSE のグラフ	4-8

表目次

4.1	実験環境	4-3
4.2	入力ポリゴン数, 出力ポリゴン数, メモリコスト	4-4

第 1 章

序論

第 1 章

序論

本章では，本研究の解決すべき課題，特に半透明ポリゴンを多数含むようなポリゴンモデルのリアルタイムレンダリングに関わる問題について説明し，本研究の目的について述べる．また，本論文の構成について述べる．

1.1 はじめに

平面ポリゴンによって表現される物体をモデルという。モデルを画面に描画するには、視点から見た各ポリゴンの見え方をピクセル毎に決定する必要がある。ポリゴンの一部が他のポリゴンによって隠されている場合は、その隠された部分は不可視であり、半透明なポリゴンによって覆われている場合は、重なって見える部分を半透明なポリゴンの色と混ぜ合わせながら画面に描かなければならない。可視面決定と呼ばれるこの処理は、コンピュータグラフィックスにおいては不可欠の処理であり、これまでに多くの手法が提案されてきた。

代表的なものにZバッファ法 [3]、優先順位アルゴリズムがあるが、それぞれに描画速度やメモリコスト、描画結果の精度などにおいて一長一短があり、ターゲットとなるモデルや用途によって使い分けられている。

特にリアルタイムに描画を行う場合は、グラフィックスハードウェアのサポートを受けたZバッファ法を利用することが一般的になっている。しかし、各ピクセル毎に最も視点に近い点のみを求めるこの方法では、視点から奥にあるものから順番に色を混ぜ合わせながら重ね書きする必要がある半透明ポリゴンの描画には、そのままでは適用できない。

今日では、雲や霧、水、窓やガラス製品、光の効果などコンピュータグラフィックスで扱われる多くのオブジェクトの表示に半透明ポリゴンが用いられることが多い。また、複雑な構造を持つポリゴンモデルの内部構造を透視したり、着目する部分を強調するためにそれ以外の部分を透過性にする場合に、それら透過部分のポリゴンを半透明にする必要がある。

ハードウェアを用いて、描画順序に依存せずに半透明ポリゴンを扱う手法もいくつか提案されている [1][2] が、数十から数百のポリゴンの重なりが発生するようなシーンに対しても適度なメモリコストで高速に描画を行うことは難しく、また、特別なアーキテクチャを用意しなければならない場合が多いため、未だ汎用的な手法とはなっていない。

そこで、モデルの描画に先立ち、すべての半透明ポリゴンを視点から遠い順に並び替えておく優先順位アルゴリズムが、比較的高速に処理を行うことが可能なことから一般に広く用いられている。また、このアルゴリズムはアンチエイリアシングの扱いや、高精細なベクター画像の出力にも適している。また、モデルの描画に単独で用いられる場合のみならず、Zバッファ法と組み合わせて表示の高速化を行ったり、完全に視界から見えないモデルを特定し、描画処理から外す用途に用いられたり、また、表示目的以外にもモデルどうしの衝突判定に用いられたりとその利用範囲は広い。

しかし、優先順位アルゴリズムはその処理のすべてをハードウェア上で行うことが出来ず、CPUによって如何に高速にポリゴンの並べ替えを行うかが課題となっている。

優先順位アルゴリズムの中でも、従来、前処理としてポリゴンが他のポリゴンと関係を持つことで、効率よく前後関係を求める方法として BSP 法が広く利用されてきた [7]。しかし、視点位置が変わるたびに、BSP 木のすべてのノードを辿る必要があるこの方法では、面数に比例して木を辿る時間が増大するため、大規模なモデルではリアルタイムな処理が困難となる。

そこで、BSP を辿ることを高速化させるさまざまな改良が試みられてきたが、一般に探索時間と木のサイズとはトレードオフの関係にあり、メモリコストを現実的な値に抑えながら高速化を実現することが課題であった。

1.2 本研究の目的

本研究では、オブジェクトの表示がリアルタイムでしかも視点の移動が微細であるときに有効な表示アルゴリズムを提案する。BSP木をたどる手法として一般には再帰関数が用いられている。本手法ではポリゴンを優先順位の高いものから並べ、リスト構造として保持し、順番に描画する。視点が移動するたびに、ポリゴンの優先順位が変化するために、リストの順序も変更させなければならないが、視点の移動が微細である場合には、表裏状態が変化するポリゴンが少ないため、優先順位の変化も僅かとなり、リストの入れ替えはほとんど行われず、実行時間に対する負担も少なくなり、有効な手法と考えられる。

1.3 本研究の構成

本論文は、全5章から成る。2章では、可視面決定手法、特に優先順位アルゴリズムに関する先行研究の紹介を行い、また、本研究が目的の一つとしている半透明ポリゴンの描画の原理の解説と課題を示す。提案する3章では新しい手法を述べる。4章で実験結果および考察について述べ、5章でまとめと今後の展望について述べる。

第 2 章

関連研究および解決すべき課題

第 2 章

関連研究および解決すべき課題

本章では、優先順位アルゴリズムを中心に可視面決定に関わる先行研究について述べ、また半透明ポリゴンの描画についてその原理およびハードウェアを用いた表示手法の現状と問題点について概説する。

2.1 関連研究

3次元ポリゴンモデルを画面に表示するには可視面、不可視面の判定を行い不可視面を消去する、いわゆる陰面消去処理を行うことが不可欠である。また、特にモデルが大規模になる場合は、視点が存在する領域に対して完全に不可視となる面、あるいは不可視となる面の集合をあらかじめ特定し、陰面消去処理から除外しておくことで表示を高速化することも重要になってくる。また、重なり合った半透明ポリゴンを表示するには、その透過性を考慮した面の見えかたを決定する必要がある。視点に対して、このような可視領域を求める問題を可視面決定問題という。

2.1.1 可視面決定手法

可視面決定問題は、3次元グラフィックスにおける重要なテーマの一つであり、これまでに多くの可視面決定アルゴリズムが提案されてきた。主なものに z バッファ法、レイトレーシング法、スキャンライン法、優先順位アルゴリズムといった方法があるが、それぞれに一長一短があり、用途や適用するモデルによって使い分けられている。以下、それぞれの手法について、その概要と特性を説明する。

2.1.2 z バッファ法

z バッファ法（デプスバッファアルゴリズムとも呼ばれる）はグラフィックスハードウェアを利用することで大変な高速化が図れる為に近年は特にリアルタイム処理が必要な場面には最も広く利用されるアルゴリズムとなっている [3]。

アルゴリズムの要件は、色の値を記憶するフレームバッファだけでなく、 z （奥行き値）の値をピクセルごとに記憶するための同じエントリ数を持った z バッファも使えるようにしてあることである。 z バッファは0に初期化されているが、これは後方クリッピング面の z 値を表している。また、フレームバッファも背景色で初期化される。

なお、 z バッファに記憶できる最大値は、前方クリッピング面の z 値を表している。ポリゴンは任意の順でスキャン変換され、フレームバッファに入れられる。スキャン変換のプロセス中、ポリゴン上の点 (x,y) が、既にバッファ上に色とデプス値 (*depth*, 深さ) が保存されている点よりも、観測者から見て近ければ、新しい点の色とデプスでフレームバッファおよび z バッファを置き換える。

このアルゴリズムでは、ポリゴンをあらかじめ視点から近い順にソートしておく、オブジェクト同士を比較する必要もなく有効である。全体のプロセスとしては、固定された x と y に関して、色とデプス値を調べ、それぞれ最大のデプス値とその時の色を見つける作業のみとなる。

z バッファとフレームバッファにはその時点での最大の z に関するデータがピクセルごとに記録してある。よって、オブジェクトを前から後ろに向かっておおよそソートし、最も近いオブジェクトを最初に表示しておけば、各バッファへのアクセス回数が低減され、計算の効率をさらに向上させることができる。その意味においては、後に述べるような全てのポリゴンを視点から近い順に並べ替える優先順位アルゴリズムをこの z バッファ法に組み入れることで z バッファ法そのものの高速化を図ることも出来る。

また、 z バッファアルゴリズムは、オブジェクトがポリゴンでなくてもよい。色と z 値が投影点ごとに決まれば、どのようなオブジェクトのレンダリングにも使えることが利点の一つでもある。しかも、ポリゴン同士が交差するような場合に対処するための処理を明示的に行う必要がない。また、可視面の計算にかかる時間は、オブジェクト中のポリゴン数には依存しない傾向にある。この理由は、ビューボリューム中のポリゴン数が増えるとともに、各ポリゴンによってカバーされるピクセル数が減っていくことが多いためである。

ただし、 z バッファアルゴリズムはピクセル単位での可視面判定を行うため、エイリアシングの影響を受けやすい。Aバッファ法 [2][5] では、エイリアシングの問題に取り組むため、重み付けなしのエリアサンプリングに離散近似を行っている。

また、各ピクセル毎に最も視点に近い点のみを保持するこの方法では、視点から奥にあるものから順番に色を混ぜ合わせながら重ね書きする必要がある半透明ポリゴンの描画にはそのままでは適用できない。また、各ピクセル単位で色が決定されるため拡大表示印刷に耐えうるような高精細画像の出力が困難、といった問題もあり、ハードウェアのサポートを受けずに高精細なレンダリングを行う場合には逆に遅いアルゴリズムとなる。

2.1.3 優先順位アルゴリズム

優先順位アルゴリズム (*priority algorithms*) とは、視点に対するポリゴン間の前後判定を行い、モデルを構成するすべてのポリゴンの優先順位を決めるというもので、オブジェクトをこの順位に従ってフレームバッファへ描画することで最終的に正しい画像が生成されるようになる。

例えば、どのポリゴンも z 軸に対して (z 軸に投影して) 互いに重ならないのであれば、 z に関して降順にポリゴンをソートし、その順でレンダリングすれば、視点に対して奥にあるポリゴンを手前にあるポリゴンが上書きしていくことになり、結果として不可視面が消去されることになる。これは1つのレイ (視線) が同時に2つのポリゴンを貫通するとき、視点に近いポリゴン上のピクセルが遠くにあるポリゴンのピクセルを遮蔽するためである。

また、ポリゴン同士が z 軸に対して重なっているような場合でも、正しいレンダリング順を

決めることができる。だが、ポリゴン同士が互いに循環的に重なる場合や、互いに貫通する場合には、正しいレンダリング順は存在しない(図 2.1)。このようなケースでは、ポリゴンを複数に分割して線形に並ぶようにしなければならない。優先順位アルゴリズムでは、オブジェクトスペースすなわち 3 次元空間上においてポリゴン間の奥行きと比較や分割を行う。ただし、スキャン変換だけは、すでに描かれたポリゴンのピクセル値を参照し、上書き、あるいは混合処理を行うためグラフィックス装置の能力に依存する。

そして、ソートされたポリゴンのリストは、ポリゴン単位で作成されるため、どのような解像度にも即座に対応できる。すなわち、ピクセル単位で奥行き情報を求めている Z バッファ法と異なり、拡大表示によって表示精度が落ちることはなく、ベクター画像の出力や自然なアンチエイリアシング処理が可能となる。

ポリゴン単位の奥行き情報を求める優先順位アルゴリズムは、他の可視面決定手法と組み合わせることで表示の高速化に用いられたり、また衝突判定や視点位置の探索など、表示以外の分野でも種々に用いられる汎用的な手法と言える [8]。

しかし、ポリゴン間の奥行き比較をソフトウェアによって行わなければならないこの手法では、扱うポリゴン数に応じて処理が遅くなってしまう、という問題がある。そこで、優先順位アルゴリズムでは、ソート順の決め方、分割ポリゴンの選び方、前処理によるポリゴンデータの構造化等、様々な手段を使い分けることで高速化が図られている。

以下に、これら優先順位アルゴリズムとして分類される手法について、その代表的なものを紹介する。

デプスソート法

最も単純な優先順位アルゴリズムとして各オブジェクト(ポリゴン)の重心と視点とのユークリッド距離によってオブジェクトを降順、あるいは昇順にソートする、というデプスソート法がある。このような方法は処理が単純であるため、高速なレタリングが必要とされる場合に適していると言える。しかし、この方法では重心の位置とポリゴンの重なり具合によって前後関係が正しく表されない視点位置が存在することがあり、また図 2.1 のような場合にはどのように優先順位を定めても隠面消去に失敗してしまう。

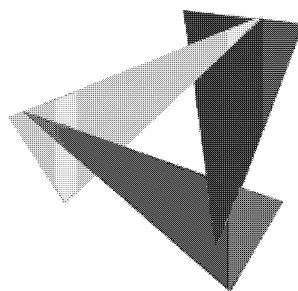


図 2.1: ポリゴン間の優先順位が定まらない例

このような場合にも正しく処理するためには、ポリゴンを分割するなどの対策が必要となり、アルゴリズムはより複雑になる。

BSP 法

デプスソート法は、各ポリゴン間の相対位置がダイナミックに変化する場合にも用いることが出来る優先順位アルゴリズムの1つであったが、一般にポリゴンモデルはそのほとんどが静的である（形状が変化しない）ことが多く、主に視点位置のみを変化させながらその視点に対する可視面決定を行うことが多い。このように、シーン中のポリゴンモデルのほとんどが静的であることを仮定した場合には BSP (Binary Space Partitioning) 法や Octree 法、k-d tree 法といった高速かつ安定に前後判定を行う方法が存在し、特に BSP 法は今日でもリアルタイムウォークスルーや 3D ゲームソフトに多く用いられる優れた方法と言える [4][6]。

モデル空間に対して、主に視点位置と視線方向のみが変化することを仮定すれば、各ポリゴンどうしの位置関係が不変であることを利用し、ポリゴンを前もって構造化しておくことで、描画時において高速にポリゴン間の奥行き比較を行うことが可能となる。

特に 2 分木状に構造化する BSP 法はそのシンプルな構造と再帰的なアルゴリズムを適用可能なことから最もよく利用されている手法である。この方法はモデルから 1 つずつポリゴンを選択しながらそのポリゴンを含む平面によって他のポリゴンを階層的に順次 2 分割していき、すべてのポリゴンを 2 分木 (BSP tree) として構造化する、というものである (図 2.2, 説明は後述)。ポリゴン間の前後関係はこの 2 分木の全てのノードにおいて、子である 2 つのポリゴン集合のうち、どちらが視点に近いかをノードポリゴンの表裏によって判定していくことによって求めることが出来る。

以下に、この BSP 木の構築方法について詳しく述べる。

BSP 木の各ノードは全てのポリゴンのなかから適当に選択されたポリゴンであり、どれを選び出してもアルゴリズムは正しく動作する。これら分割ポリゴンはシーンを 2 つの半空間に分割していく。これにより、一方の半空間には分割ポリゴンの法線を基準にして表側にあるすべてのポリゴンが格納され、もう一方の半空間には分割ポリゴンの裏側にあるすべてのポリゴンが格納される。分割ポリゴンによって定義される平面と別のポリゴンとが干渉すれば、この平面によってポリゴンを切断し、それぞれ平面の表側と裏側の半空間に割り当てる。

以上の操作を全てのポリゴンに対して再帰的に繰り返すことで、BSP 木が構築される。すなわち、ルートポリゴンの表側と裏側にある半空間からそれぞれ 1 つずつポリゴンを選択し、それぞれ表側と裏側の子のノードとしていく。これを子ポリゴンが 1 個だけになるまで繰り返す。

BSP 木の構築のための疑似プログラムコードを図 2.4 に示す。なお、図 2.3 は BSP 木のノードのデータ構造を表したものである。

図 2.2 は 2 次元空間において 5 つのポリゴンを用いて BSP 木を構築する様子を段階的に表したものである。面の集合の中で適当なものを 2 分木のルートの面として決め、残りの面をルートの面の front 側と back 側に分ける。もし、ルートの面の front 側にも back 側にも存在する面がある時は、ルートの面を通る平面で面を切り、front 側と back 側に分ける。面の集合が front 側と back 側に分けた後、それぞれで新たにルートとなる面を決めて作業を再帰的に行っていく。ルートとなる面がなくなれば作業は終了する。

以上が前処理における BSP 木の構築方法であったが、次に、BSP 木を用いてポリゴンモデルを画面に描画する方法について解説する。

BSP 木をルート (ルートポリゴン) から正しい順序でたどれば、どのような視点からでも奥行きの順に並んだポリゴンリストを得ることができる。

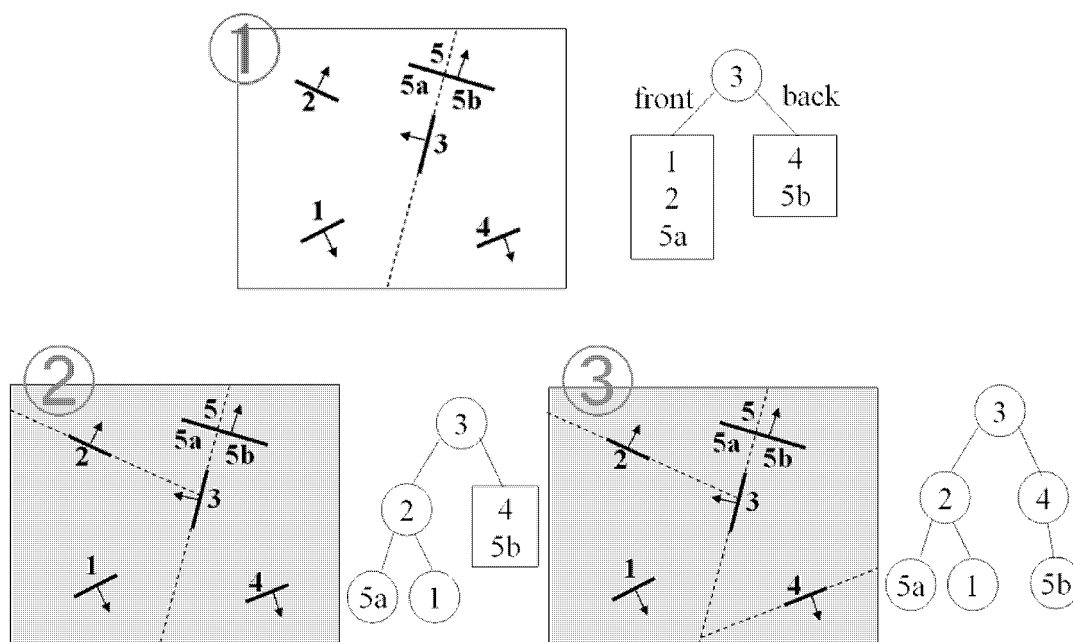


図 2.2: BSP 木の構築

```

typedef struct{
    polygon root;
    BSP_tree *frontChild, *backChild;
}BSP_tree;

```

図 2.3: BSP 木ノードのデータ構造

視点があるノードポリゴンの表側の半空間にある場合、アルゴリズムではまず、ノードポリゴンの裏側にある半空間の子ポリゴンをすべて描画してから、ノードポリゴンに戻り、次にノードポリゴンの表側にある半空間の子ポリゴンをすべて描画する。逆に視点がノードポリゴンの裏側の半空間にあれば、ノードポリゴンの表側にある半空間の子ポリゴンをすべてたどってから、ノードポリゴンに戻り、最後にノードポリゴンの裏側にある半空間の子ポリゴンをすべてたどる。各ノードの子にあたるノードにおいても、このアルゴリズムを再帰的に適用する。

BSP 木をたどって表示する疑似プログラムコードを図 2.5 に示す。

以上のアルゴリズムからも分かるように、BSP 木を高速にたどるためには木のルートから各リーフまでの深度をなるべく浅く均一にし、バランスのとれた木の形状にする必要がある。そのため、どのポリゴンを選択して各サブツリーのルートにするかで、アルゴリズムのパフォーマンスは著しく変わってくる。また、あるノードポリゴン（サブツリーのルート）を選択する際に、子にあたるポリゴンの分割回数が最小になるようにしたい。しかし、このような面を自動的に決定することは極めて難しい問題である [7]。

この BSP 法では分割面によるポリゴンの切断が多数発生し（理論上は入力ポリゴン数 N に対して最悪の場合出力ポリゴン数が $O(N^3)$ のオーダーで増加する）、それに伴い BSP 木のサ

```

BSP_tree *BSP_makeTree( polygon *polyList)
{
    polygon root;
    polygon *backList, *frontList;
    polygon p, backPart, frontPart;    /* 各ポリゴンが凸であると仮定 */

    if(polyList == NULL)
        return NULL;
    else{
        root = BSP_selectAndRemovePoly(&polyList);
        backList = NULL;
        frontList = NULL;
        for(each remaining polygon p in polyList){
            if(polygon p in front of root)
                BSP_addToList(p, &frontList);
            else if(polygon p in back of root)
                BSP_addToList(p, &backList);
            else{                                /* ポリゴン p は分割が必要 */
                BSP_splitPoly(p, root, &frontPart, &backPart);
                BSP_addToList(frontPart, &frontList);
                BSP_addToList(backPart, &backList);
            }
        }
        return BSP_combineTree(BSP_makeTree(frontList),
                                root, BSP_makeTree(backList));
    }
} /*BSP_makeTree*/

```

図 2.4: BSP 木構築のための擬似プログラム

```

void BSP_displayTree(BSP_tree *tree)
{
    if(tree != NULL){
        if(viewer is in front tree -> root){
            /* 後ろの子, ルート, 前の子をトラヴァース */
            BSP_displayTree(tree -> backChild);
            displayPolygon(tree -> root);
            BSP_displayTree(tree -> frontChild);
        }else{
            /* 前の子, ルート, 後ろの子をトラヴァース */
            BSP_displayTree(tree -> frontChild);
            displayPolygon(tree -> root);
            BSP_displayTree(tree -> backChild);
        }
    }
}
} /*BSP_displayTree*/

```

図 2.5: BSP 木トラヴァースのための擬似プログラムコード

イズが出力ポリゴン数に比例して巨大化してしまう、という問題がある。木の各ノードではそのノード面の表裏判定が行われるが、木の巨大化に伴いこの表裏判定回数が増大し、結果として処理時間が増大することになる。2分木構造を用いてポリゴンの優先順位を決定するときには、この表裏判定の回数が処理時間に大きく影響する。BSP法の場合、ポリゴン数 N に対して表裏判定回数が $O(N)$ であり、木を保持するのに必要なメモリコストもまた $O(N)$ である。

2.2 半透明ポリゴンの扱いについて

ここでは、本研究で提案する優先順位アルゴリズムが対象の一つとする半透明ポリゴンの扱いについて述べる。

2.2.1 アルファブレンディング

半透明のポリゴンを画面に描画する場合、すでにフレームバッファに描かれている色と新しいポリゴンの色とを適当な比率で混ぜ合わせる必要がある。このような処理をブレンディング（混合処理）と呼び、各ピクセル毎に次のように色が決定されることになる。

描こうとしている色を $(R_s, G_s, B_s, \alpha_s)$ 、すでに描かれている色を $(R_d, G_d, B_d, \alpha_d)$ とするとブレンディングされた新しい色は図 2.6 のようにして決定される。ただし、 α はアルファ値と呼ばれピクセルの透過性を表す。すなわち、 $\alpha = 0$ ならばそのピクセルは完全な透明となり、 $\alpha = 1.0$ ならば不透明となる。

以下簡単のため、すべてのポリゴンのアルファ値を 0.5、すなわち描こうとするピクセルの色と既に描かれているピクセルの色とを同率で合成する場合を考える。たとえば背景 C の上に半透明の物体 A, B が共に重なりあって見える場合には、視点から奥にあるものから C,

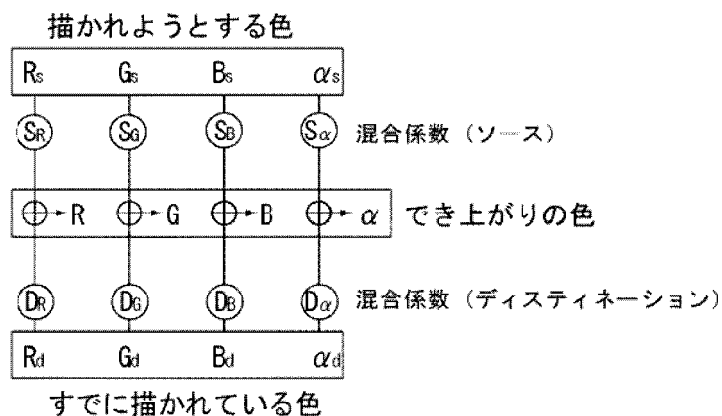


図 2.6: アルファブレンディング

A, B の順になっているとすれば, まず C と A とが合成され, そこにさらに B が合成されるため, 結果としてピクセル値は

$$\frac{1}{2}\left(\frac{1}{2}C + \frac{1}{2}A\right) + \frac{1}{2}B = \frac{1}{4}A + \frac{1}{2}B + \frac{1}{4}C$$

となる.

一方奥から C, B, A の順に重なり合っているとすれば

$$\frac{1}{2}\left(\frac{1}{2}C + \frac{1}{2}B\right) + \frac{1}{2}A = \frac{1}{2}A + \frac{1}{4}B + \frac{1}{4}C$$

となり, 結果の色はブレンディングの順番に依存することが分かる.

よって, 厳密に半透明物体間の重なりを扱いたければ, あらかじめ全ての半透明物体の前後関係を調べ, 奥にあるものから順に描くことが必要になる.

2.2.2 ハードウェアを用いた混合処理

ソフトウェアによるポリゴンのソートを行わず, ハードウェアによって半透明ポリゴンを扱うための手法もいくつか提案されている. 以下, ハードウェアによる半透明ポリゴンの描画について現在提案されている手法のいくつかを紹介し, その問題点を挙げる.

Depth Peeling 法

Depth Peeling 法は多層のカラーバッファとデプスバッファからなる仮想フレームバッファを作成し, 最も奥の仮想フレームバッファから順番に描画をすることによって実現される手法である [1].

図 2.7 に Depth Peeling 法の概略を示す. 図では, オブジェクトを 3 つの仮想フレームバッファに分けている.

仮想フレームの作成方法は

- フレームバッファに格納された色をテクスチャとして保存する.
- フレームバッファに格納されたデプス値をテクスチャとして保存する.

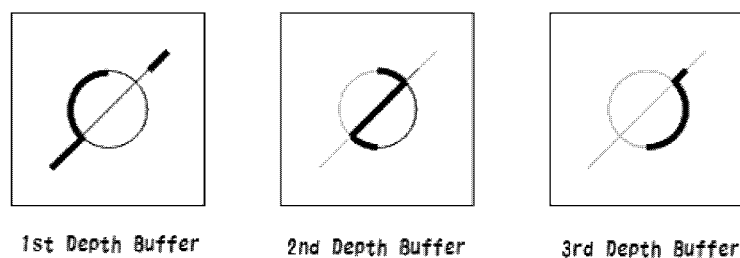


図 2.7: Depth Peeling 法

- テクスチャに保存されたデプス値とフラグメントのデプス値を保存する。

のようになる。

これらの機能を用いて n 回のマルチパスレンダリング (n は仮想フレームバッファの数) を行うことで、 n 枚のカラーバッファおよびデプスバッファをテクスチャとしてグラフィックスハードウェアのメモリに格納することができる。仮想フレームバッファの作成が終わると、それぞれのカラーバッファをアルファブレンディングすることにより、半透明表示を行うことができる。仮想フレームバッファは視点に依存するので視点が変更される度に作成する必要がある。

この手法では、ハードウェアを用いてフラグメント単位のソートを行うものの、フラグメントの重なり分だけのマルチパスが必要であり、数十枚以上の半透明ポリゴンが重なるような場合には処理が遅くなり、またメモリコストも増大するという欠点がある。

R-Buffer 法

R-Buffer 法はフラグメント単位で **Linked List** を持ち、リストのノードとして色とアルファ値、デプス値を、という手法である [2]。視点に対して奥にあるフラグメントからリストを作っていく、新しく挿入されたポリゴンに対しては、そのポリゴンのすべてのフラグメントのデプス値がリストの持つデプス値と比較され、新しくリストの中に挿入される。半透明ポリゴンを描画するには、フラグメント単位でリストから順番に色とアルファ値を取り出し、ブレンディングの計算を行っていく。しかし、視点が変化する度に全てのリストを更新しなければならず、処理が複雑となる。また、すべてのフラグメントをリスト上に保持しなければならず、数十枚以上の半透明ポリゴンが重なるような場合には膨大なメモリが必要である。

第 3 章

改良アルゴリズム

第3章

改良アルゴリズム

本章では、提案する BSP 法の改良アルゴリズムを述べる。

3.1 アルゴリズムの概要

与えられたポリゴンの集合から単一のリスト構造を構成し、視点位置に応じてリストの一部を並び替えることで視点に対するポリゴンの前後関係を効率的に求めるアルゴリズムを提案する。BSP 法がフレームごとに木の全てのノードを再帰的に辿る必要があるのに対し、提案手法では表裏判定に必要なポリゴンのみによる判定を行い、更新の必要があるリストの一部のみを逐次的に入れ替えていくことで、1 フレームあたりのソーティング処理を大きく削減する。

このアルゴリズムはポリゴン全体の集合から単一のリスト構造である遮蔽関係リストの初期状態をまず構成し、その後、変化する視点位置に応じてリストの一部を並べ替えていくことで高速にポリゴン間の前後関係を求める、というものである。この章では BSP 木のノードとは木のリーフではない要素と定義する。

3.2 アルゴリズムの流れ

改良アルゴリズムは大きく3つの操作からなる。アルゴリズムの流れを図 3.1 に示す。

- 初期リストの構築
- ノードの配列の構築
- 視点の変化に伴うリストの入れ替え

3.3 初期リストの構築

初期の視点におけるリストを構築する。データ構造は図 3.2 のようになる。

本手法では BSP 木を用いないが、BSP 木における表示の順番と本方法で構築されるリストの関係を図 3.3 に示す。図 3.3 の BSP 木に書かれている番号は、ある視点においてノードの表示される順番である。next は次の要素への参照、prev は前に表示した要素への参照、LL, LR, RL, RR は表裏の情報に変化したときに入れ替えるべきリストの先頭と末尾への参照を表す。

```

makeList(); /* 初期リストの作成 */
makeNodeArray(); /* 子を持つノードへの参照を持つ配列の作成 */
while(全視点移動が終了するまで){
  for(i = 0 ~ ノードの数){
    if(現在の視点の face の表裏情報      switchingPolyon[i].state
                                           (直前の視点における表裏情報)){
      changeList(); /* リストの入れ替え */
      changeParent(); /* 親にあたる木の情報の更新 */
      switchingPolyon[i].state を更新
    }
  }
  リストを順番に描画
}

```

図 3.1: 改良アルゴリズム

```

class VisibilityList{
  face polygon;          // ポリゴンデータ
  VisibilityList* next; // リストの次の要素
  VisibilityList* prev; // リストの前の要素
  VisibilityList* LL, LR, RL, RR;
};

```

図 3.2: リストのデータ構造

ノードの表裏情報が変化するとき、ノードの前と後にあるリストの集合を入れ替える作業が必要となる。図 3.3の一部を取り出したものを図 3.4に示す。もし要素4の表裏情報が変化した場合、要素1から要素3までと要素5から要素7までを入れ替える。このときノードの前のリスト(要素1～要素3)の先頭である要素1をLL, 末尾である要素3をLRが指し、同様にノードの後に描画されるリスト(要素5～要素7)の先頭である要素5をRL, 末尾である要素7をRRが指している。LL, LR, RL, RRはリストの交換をするとき、要素のnext, prevの付け替えを行う際に使用する。

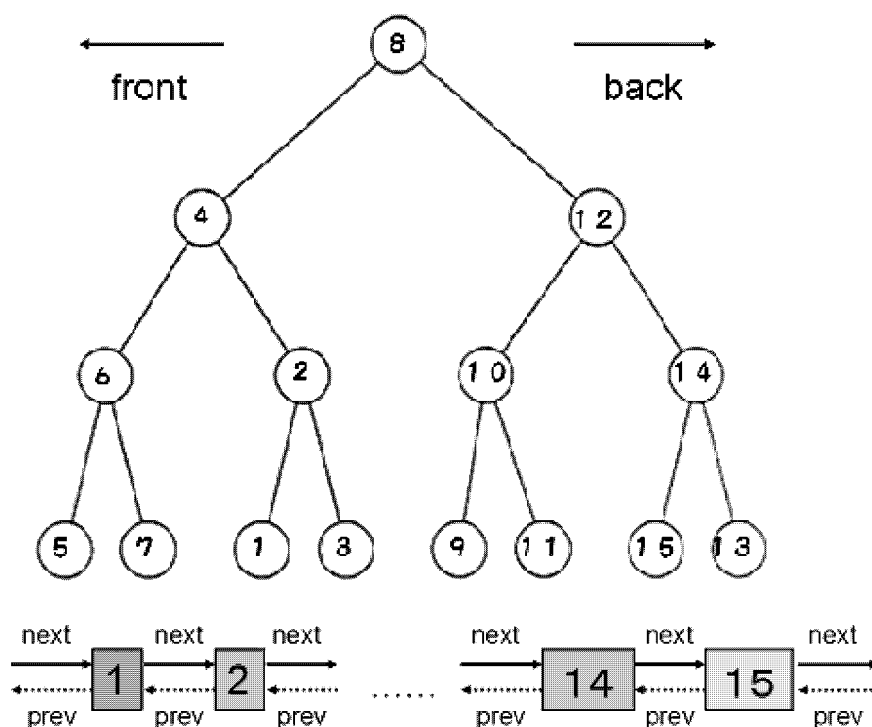


図 3.3: BSP木と構築されるリストの関係

3.4 ノードへの参照を持つ配列の構築

今回の手法ではノードへの参照 (`VisibilityList *self`) を要素として持つ `SwitchingPolygon` というデータ構造を作り、その配列を用意する。図 3.6のように配列の要素である `self` はリストの要素の中でノードになっている要素への参照を持つ。`SwitchingPolygon` の第 1 要素の `self` はリストの要素 2 を指し、第 2 要素の `self` はリストの要素 4 を指し、第 3 要素の `self` はリストの要素 6 を指す。`SwitchingPolygon` の配列の要素数はノードの数と一致する。

視点が連続的に遷移することを仮定すれば、視点の移動に伴う各ノードの表裏の変化も、逐次的なものとなる。よって、全てのノードの表裏の状態を一つ前のフレームと比較し、表裏が反転した場合にリストの入れ替え操作を行う。これにより、BSPを辿る操作が省略され、ソーティングを高速化できる。各 `SwitchingPolygon` は、その表裏の状態を内部変数の“state”に保持し、表裏の変化はこの `state` の値を参照することにより検出される。

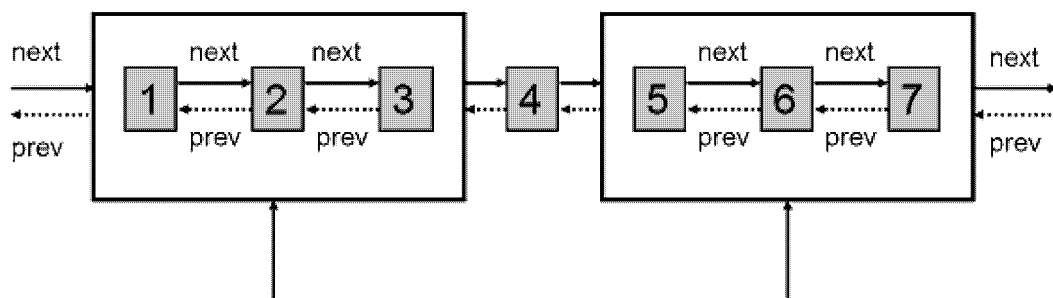


図 3.4: リストの入れ替え

```

class SwitchingPolygon{
    VisibilityList* self; //リストへの参照
    bool state; //視点に対する表裏情報
};

```

図 3.5: ノードへの参照を持つデータ構造

3.5 リストの入れ替え

`SwitchingPolygon` の配列の中で表裏状態が `state` と異なる要素が出てきた場合は、リストを入れ替えるために下図のような操作をする。

$$\begin{aligned}
 [LLのprevのnext] &\iff [RLのprevのnext] \\
 [LRのnextのprev] &\iff [RRのnextのprev] \\
 [LLのprev] &\iff [RLのprev] \\
 [LRのnext] &\iff [RRのnext]
 \end{aligned}$$

(3.1)

この操作を行うと表裏状態が変化したノードに関するリストの入れ替えが行われる。例えば、要素 4 の表裏状態が変化したとすると、BSP 木での左右の部分木に当たる 1, 2, 3 と 5, 6, 7 の表示順を変えなければならない。そこで図 3.4 のように要素 4 の前後のリストを入れ替えるために、図 3.7 のようにリストの `next` と `prev` を付け替える。この作業を表裏情報が変化した全てのノードにおいて実行すると、できあがったリストは正しい順序にソートされたことになる。図 3.1 で示される通り、視点が変わる毎に表裏状態の変化した全てのノードに関するリストを入れ替えると変化した視点における正しいポリゴンのソートが行われる。

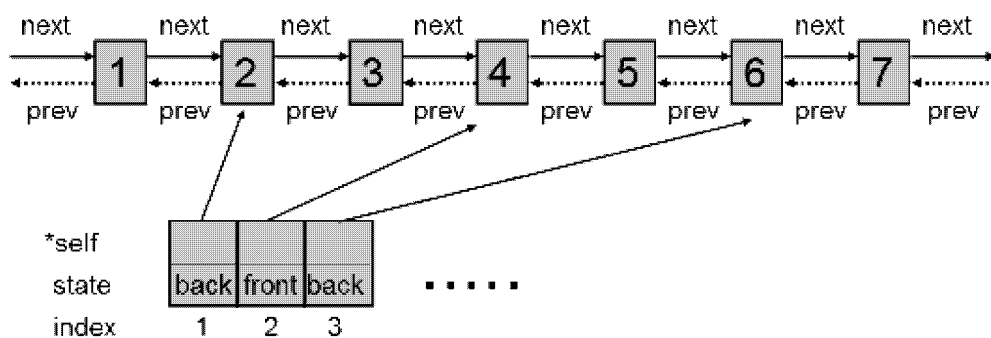


図 3.6: リストと SwitchingPolygon の配列との関係

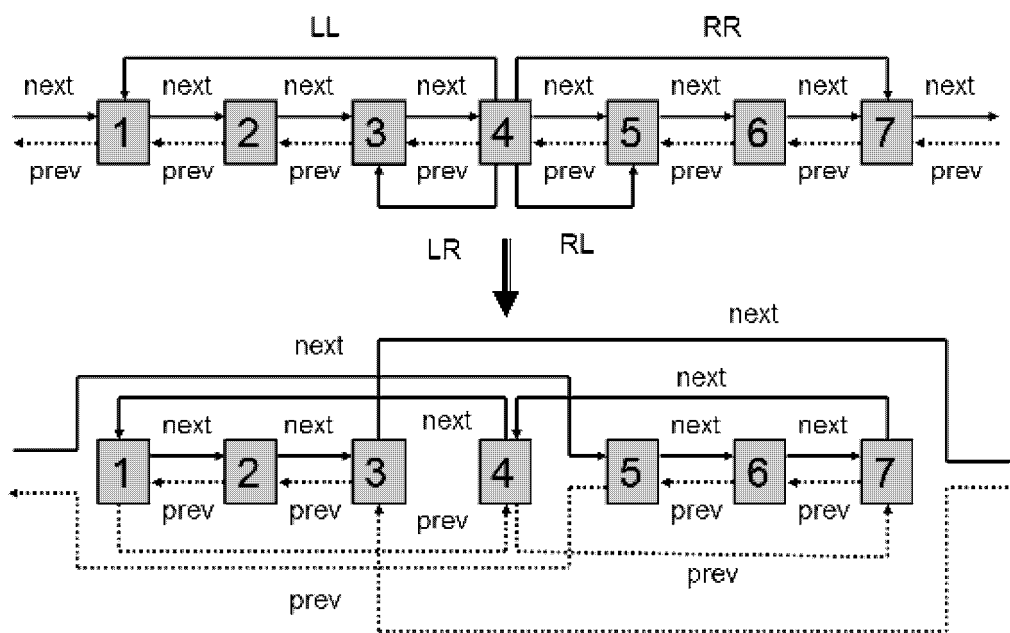


図 3.7: next・prev の付け替え

第4章

実験および考察

第4章

実験および考察

4.1 評価実験のためのモデルおよび実験環境

本研究で提案する手法を，図 4.1 ~ 図 4.3 に示す 3 つのモデルに対して適用し，今日最も広く用いられている優先順位アルゴリズムである再帰関数を用いた方法と速度およびメモリコスト面からの比較を行った．

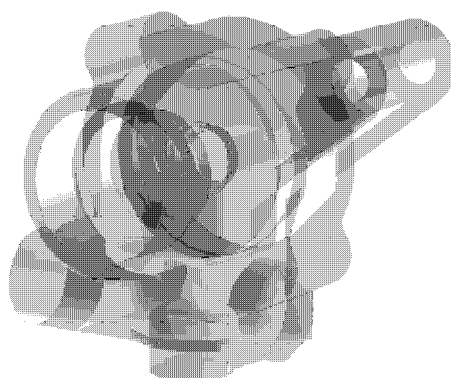


図 4.1: PUMP (4,904 polygons).

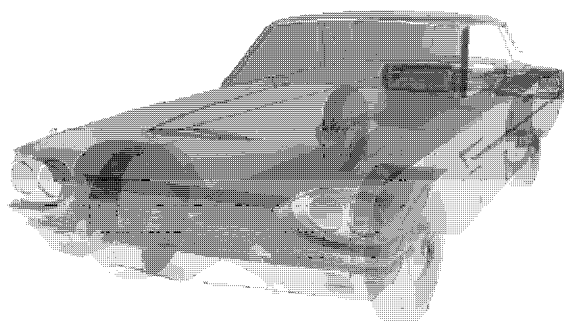


図 4.2: CAR (10,093 polygons).



図 4.3: HORSE (96,966 polygons).

図 4.1は機械部品のモデルであり，4904面から成る．中央に1つの穴が開いており，トーラスと同位相である．図 4.2の車モデルは車体および車体とは切り離された4つのタイヤから成り，全ポリゴン数は10093面である．車体，タイヤはそれぞれ球位相でありモデルに対して外側を向いている面を表としている．図 4.3の馬のモデルは96,966面，全体が球位相である．

なお，本研究で提案する改良アルゴリズムやBSP法は共にポリゴンをデータ構造化するための前処理に数秒から数分の時間を要する．しかし，この前処理は各モデルに対し初めに1回施されるのみであるため，インタラクティブフェーズにおいてフレームレートに直接影響を与えるソーティング処理よりも重要性は低い．ソーティングに要した時間は，一定の幅で角度を連続的に変化させ200フレームを描画し，全ての作業にかかった時間と描画に関する作業を除いた時間を計算した．角度の変化は 1° 、 2° 、 5° 、 10° 、 20° 、 50° 、 100° で調べた．また，テストの全行程と直接描画に関する作業の2種類を計測し，描画以外での作業にどれほどの効果がでるか見た．メモリコストの比較においては，本手法およびBSP法において前処理で構築される構造を保持するのに必要なメモリコストを計算により求めている．

なお，改良アルゴリズムおよびBSP法は共に同じ計算機環境において実装し，実験を行った．実装，実験環境を以下に示す．

表 4.1: 実験環境

ハードウェア	
CPU	Intel Pentium4 2.8 GHz
Main Memory	512MB
GPU	ATI Mobility Radeon 9700
OS	Windows XP Professional
実装言語	C++
Graphics API	OpenGL

4.2 結果

4.2.1 メモリコスト

提案手法とおよび比較に用いる BSP 法をこれら 3 つのモデルに適用した結果のメモリコストを、表 4.2 に示す。入力ポリゴン数とは最初に与えられるオブジェクトのポリゴン数であり、出力ポリゴン数とは BSP 木を作ったときの要素数である。BSP を作成するときには、面を切り合うので木の要素数は初めに与えられたポリゴン数より多くなる。本手法は BSP 法よりコストが増えているが、リストのほかに SwitchingPolygon の配列を用意したことが要因として考えられる。

表 4.2: 入力ポリゴン数，出力ポリゴン数，メモリコスト

	PUMP	CAR	HORSE
入力ポリゴン数	4,904	10,093	96,966
出力ポリゴン数	1,3915	36,944	633,913
BSP のメモリコスト (MB)	1.3915	3.6944	63.391
本手法のメモリコスト (MB)	1.8171	4.8701	86.289

4.2.2 実行時間

実行時間の結果を表 4.5 ~ 4.6 および図 4.8 ~ 4.9 に示す。結果を確認すると、変化する角度が小さいときは本手法は BSP 木を再帰関数で辿るとほぼ同程度かわずかな高速化が見られる。しかし、角度の変化が大きくなると、視点の移動による表裏情報が変わるノードが多くなり、同時にリストの入れ替えの回数が増え、その結果、再帰関数より多くの時間を要することになる。

CAR	角度の変化	1°	2°	5°	10°	20°	50°	100°
全行程	再帰関数	3.12594sec	3.12203sec	3.12593sec	3.1284sec	3.12587sec	3.10504sec	3.08466sec
	改良アルゴリズム	3.15034sec	3.17418sec	3.2161sec	3.29565sec	3.4119sec	3.6557sec	3.82069sec
描画以外	再帰関数	0.913501sec	0.910159sec	0.909244sec	0.905875sec	0.901619sec	0.882599sec	0.845461sec
	改良アルゴリズム	0.813782sec	0.831531sec	0.887224sec	0.965639sec	1.08016sec	1.31268sec	1.48007sec
	表裏の変化した割合	0.43%	0.78%	1.87%	3.71%	7.15%	17.25%	28.00%

図 4.4: PUMP の結果

PUMP	角度の変化	1°	2°	5°	10°	20°	50°	100°
全行程	再帰回数	1.2787sec	1.28863sec	1.28857sec	1.28871sec	1.28887sec	1.28905sec	1.28898sec
	改良アルゴリズム	1.2792sec	1.28864sec	1.28863sec	1.28881sec	1.2889sec	1.31463sec	1.37373sec
描画以外	再帰回数	0.280341sec	0.277787sec	0.275346sec	0.2704sec	0.261186sec	0.23266sec	0.208809sec
	改良アルゴリズム	0.273831sec	0.276637sec	0.286271sec	0.303292sec	0.329103sec	0.402622sec	0.462105sec
	表裏の変化した割合	0.45%	0.83%	1.99%	4.00%	7.30%	19.00%	29.00%

図 4.5: CAR の結果

HORSE	角度の変化	1°	2°	5°	10°	20°	50°	100°
全行程	再帰回数	58.2528sec	58.4011sec	58.2607sec	58.2572sec	58.2916sec	58.5692sec	58.2771sec
	改良アルゴリズム	56.994sec	57.3578sec	58.4635sec	59.42sec	61.305sec	66.4276sec	70.2688sec
描画以外	再帰回数	19.4084sec	19.4328sec	19.4516sec	19.4211sec	19.4454sec	19.4479sec	19.4045sec
	改良アルゴリズム	17.1715sec	17.5612sec	18.5632sec	19.6611sec	21.455sec	26.5325sec	30.5196sec
	表裏の変化した割合	0.47%	1.01%	2.37%	4.69%	9.09%	21.62%	36.15%

図 4.6: HORSE の結果

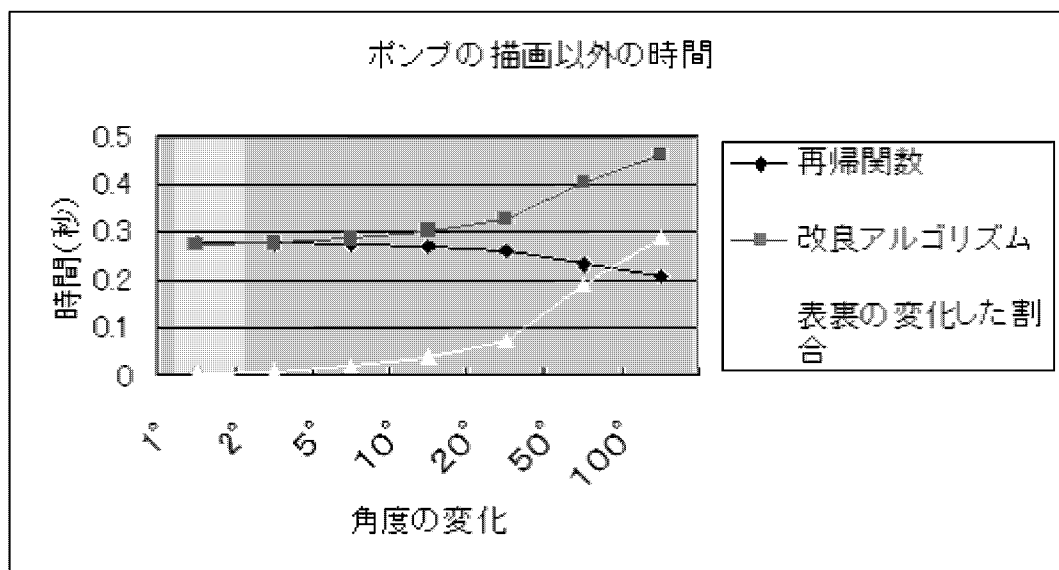
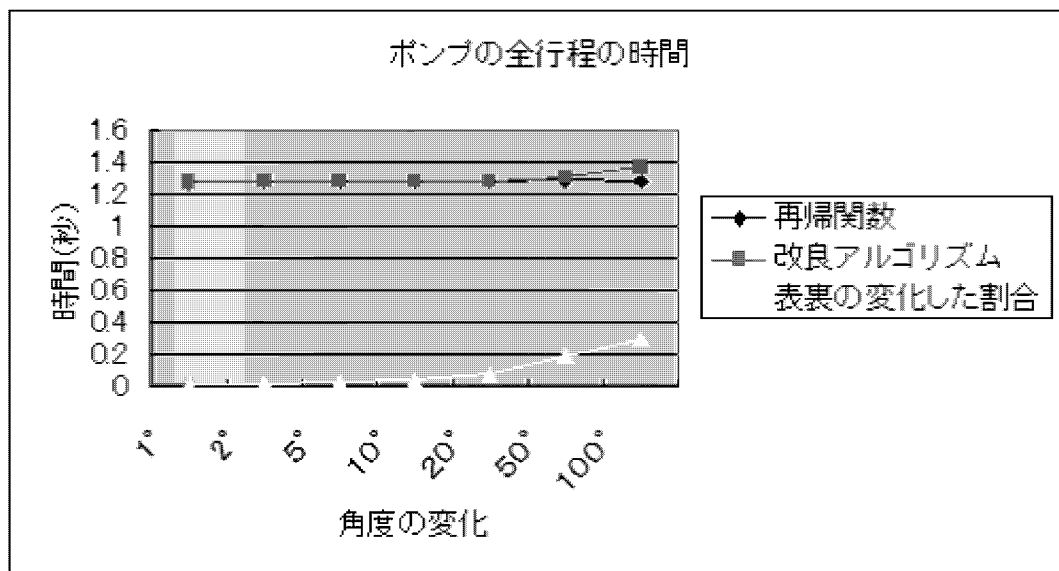


図 4.7: PUMP のグラフ

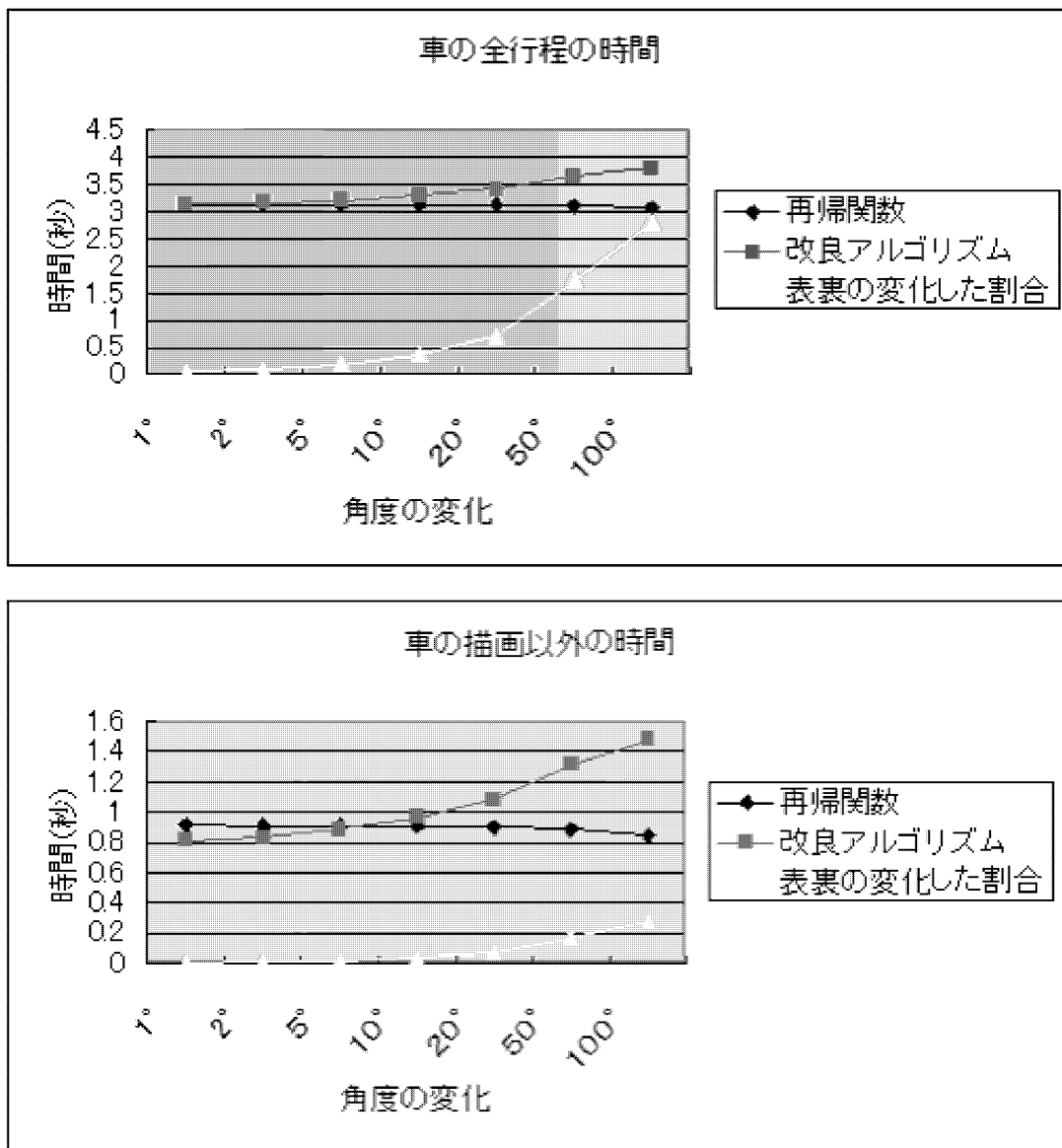


図 4.8: CAR のグラフ

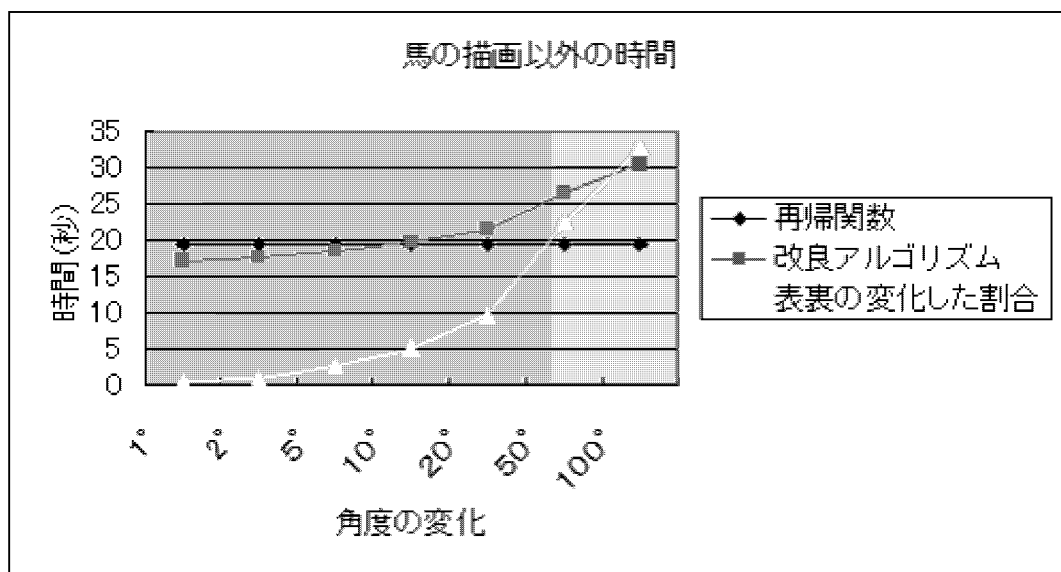
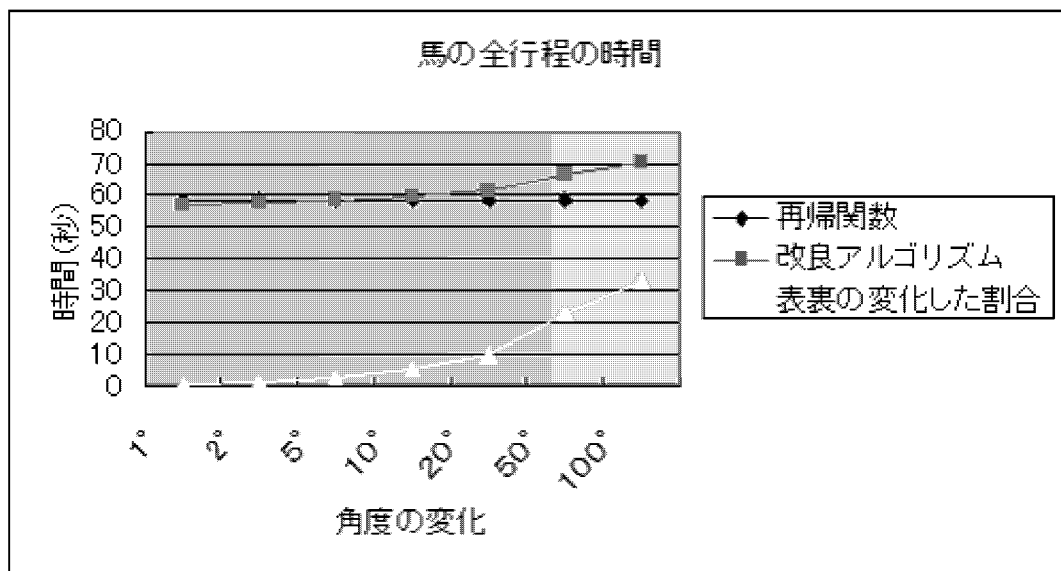


図 4.9: HORSE のグラフ

第5章

まとめと展望

第 5 章

まとめと展望

本章では、本研究のまとめと今後の展望について述べる。

5.1 まとめ

本手法により、視点の移動が小さいときは、既存の **BSP** の再帰関数を用いた方法よりわずかながらの高速化が見られた。視点が移動する幅が大きくなると本手法の効果は失われる。微細な視点の移動をするときは、**BSP** 木のノードを再帰関数によりたどるよりも、変更すべきリストの一部を入れ替える時間が微小であるため、本手法は成功したといえる。

5.2 展望

今日では、雲や霧、水、窓やガラス製品、光の効果などコンピュータグラフィックスで扱われる多くのオブジェクトの表示に半透明ポリゴンが用いられることが多い。半透明ポリゴンを表示する方法にハードウェアを用いて、描画順序に依存せずに半透明ポリゴンを扱う手法がある。しかし、数十から数百のポリゴンの重なりが発生するようなシーンでは適切なメモリコストで実行することができない。また、優先順位アルゴリズムにおいて、デプスソートが提案されているが、オーダーが $O(N \log N)$ であるためにポリゴン数が多い場合は有効ではない。その点 **BSP** 法はメモリコストや計算時間の面で優れた方法である。本手法は **BSP** で再帰関数を用いる方法と比べてわずかな高速化しかできなかったが、この手法は **BSP** だけでなく 2 分木を辿る際、**LeftChild** と **RightChild** のどちらを辿るかがほとんど変化しない場合、リストを構築し、表裏が変化したノードに関わるリストの入れ替えをすることにより再帰関数より高速に実行することができ、**BSP** 以外でも効果を期待できる。

参考文献

- [1] Cass Everitt: 『Interactive Order-Independent TransParency』 NVIDIA OpenGL Applications Engineering , cas@nvidia.com
- [2] Craig M. Wittenbrink. R-buffer: A pointerless A-buffer hardware architecture. In 2001 SIGGRAPH/Eurographics Workshop on Graphics Hardware, pages 73-80, August 2001.
- [3] P. R. Atherton: A Method of Interactive Visualization of CAD Surface Modeles on a Color Video Display. *SIGGRAPH'81* , 279-287,1981.
- [4] J. L. Bentley: Multidimensional Binary Search Trees Used for Associative Searching. *Communication cf the ACM* 18(9), September 1975.
- [5] L. Carpenter: The A-buffer, an Antialiased Hidden Surface Method. *SIGGRAPH'84* , 103-108, 1984.
- [6] L. Doctor, J. Torborg: Display Techniques for Octree-Encoded Objects. *CGA*, 1(3), 29-38, July 1981.
- [7] H. Fuchs, G. D. Abram, and E. D. Grant: Near Real-Time Shaded Display of Rigid Objects. *Computer Graphics*, Vol.17(3), pp.65-72, July 1983.
- [8] B. F. Naylor: A Priori Based Techniques for Determining Visibility Priory for 3-D Scenes. *Ph.D. Dissertation, University cf Texas at Dallas*, May 1981.