2022 年度　卒　業　論　文

リアルタイム 3DCG における
半透明物体の描画に関する研究

指導教員：渡辺 大地 教授

メディア学部 ゲームサイエンスプロジェクト
学籍番号　M0119038
牛島 イアン

2023 年 2 月

2022 年度　　卒　業　論　文　概　要

論文題目

リアルタイム 3DCG における
半透明物体の描画に関する研究

メディア学部 **学籍番号 : M0119038**	**氏名**	牛島 イアン	**指導教員**	渡辺 大地 教授

キーワード	リアルタイム 3DCG、半透明物体、レンダリング、 ラスタライズ、OIT、アルファブレンド

ゲームなどのリアルタイム CG において、半透明物体が重なり合った際の高品質かつ高速な描画は長年の課題である。例えば窓や髪、エフェクトといった半透明部分を含む物体が多数重なり合うように描画する際、奥から順に描画しなかった場合には、深度テストにより奥の物体が前の物体を通して見えないことがある。深度テストを無効化しても、奥にあるはずの物体が前にあるかのように見えてしまう。しかし、リアルタイムでポリゴンの描画の順序を並び替える処理は CPU 負荷が非常に高い。しかも、半透明のポリゴン同士が交差している場合など、一部のケースでは並び替えが不可能なため、望ましい結果にならないことがある。このような問題を改善するのに望ましいのが OIT (Order-Independent Transparency: 順序に依存しない半透明描画) である。

本研究では、OIT の既存手法の品質を検証し、その最新手法の一つである「Moment Transparency」において、アルファ値が小さいかのように、後ろの物体が不自然に見えてしまう現象が発生していたことが分かった。これを改善するため、本手法では後ろの物体の面が最も前にある物体の面に近いほど、後ろの面のアルファ値を小さくする補正をかけて本来想定される色に近づけた。これによって半透明物体同士の距離が近い複雑なシーン、例えば「ガラスの床の上に魔法陣といった半透明のエフェクト」や、「カップの中の氷」などをより高品質に描画することが可能となった。

目 次

図 目 次

第 1 章

はじめに

1.1 研究背景と目的

ゲームなどのリアルタイム CG において、半透明物体が重なり合った際の高品質かつ高速な描画は長年の課題である。例えば窓や髪、エフェクトといった半透明部分を含む物体が多数重なり合うように描画する際、奥から順に描画しなかった場合には、深度テストにより奥の物体が前の物体を通して見えないことがある。深度テストを無効化しても、奥にあるはずの物体が前にあるかのように見えてしまう。しかし、リアルタイムでポリゴンの描画の順序を並び替える処理は CPU 負荷が非常に高い。しかも、半透明のポリゴン同士が交差している場合など、一部のケースでは並び替えが不可能なため、望ましい結果にならないことがある [1]。このような問題を改善するのに望ましいのが OIT (Order-Independent Transparency: 順序に依存しない半透明描画) である。

ゲームなどのリアルタイム 3DCG において、3D モデルの変形と投影を行うと、画面上の XY 座標のほかに、奥行きを表現する「深度」が Z 値として得られる。深度は原則として OpenGL においては -1 と 1 の間 [2]、 Direct3D と Vulkan においては 0 と 1 の間に収まるようになっている [3][4]。ほとんどの場合、奥（カメラより遠く）になるほど数値が大きく、前（カメラの近く）になるほど数値が小さくなる。これを用いて、ポリゴンをラスタライズして出力されるピクセルの深度は「深度バッファ」に格納され、その後出力されるピクセルは同じ位置のピクセルの既存ピクセル深度より小さい場合にのみ、色と深度が格納（上書き）される。この処理を深度テストという。これによって 3DCG において奥にあるポリゴンが前にあるように見えてしまう現象が無くなる。

しかしながら、この説明は物体が全て不透明であると仮定した場合にのみ有効である。半透明の場合、ピクセルを上書きまたは無視するだけでは済まず、色を混合する必要があるため、正しい結果を得るには奥から順に処理する必要がある。

まず、OIT を使用しない場合の半透明物体の描画について説明する。半透明物体を描画する際、一般的には、出力ピクセルの色とアルファ値は式 1.1 によって算出される [5]。式 1.1 では、C_{src} を入力ピクセル、C_{dst} を既存ピクセル（上書きするピクセル）、C_{out} を出力ピクセル、A_{src} を入力ピクセルのアルファ値とし、C は RGBA の 4 つの成分を持っているとする。また、C_{src} の RGB 成分はアルファ値が予め乗算されているとする。

$$C_{out} = C_{src} + (1 - A_{src})C_{dst} \tag{1.1}$$

しかしながら、式 1.1 では、入力ピクセルが前にある物体であり、既存ピクセルが奥にある物体であることが前提である。そのため、前と奥の物体が入れ替わると結果が異なることがないよう、奥から順にポリゴンを描画する必要がある。式 1.1 を用いて奥から順に描画する処理を OVER という [6]。

なお、OVER を実現するために描画の順序が正しくなるように並び替えようとしても、並び変える処理はポリゴンの数が増えるほど CPU にとって急激に高負荷になる。特にモデル品質が大きく向上し、ポリゴン数が大きく増加した昨今にそのような処理を行うのは現実的ではない。そのため、多くのゲームエンジンにおいてはその処理を簡略化し、ポリゴン単位ではなく、物体（エフェクトや 3D モデルなど）ごとのおおよその位置がカメラから遠い順に描画することが多い。多くのシーンにおいてはそれで足りるが、一部のシーンにおいては、ポリゴン単位で並び替えした際に特定条件下で発生する「奥にあるはずの物体が前にあるかのように見えてしまう」現象がさらに悪化してしまう。そのような問題を解決するのが OIT である。

本研究では、OIT を実現する既存手法の品質を改善する方法を提案することを目的とした。本稿では特に「Moment Transparency[7]」において、半透明物体同士の距離が近い際に奥の物体が

不自然に映りこむ現象の改善方法を提案する。本手法によって半透明物体同士の距離が近い複雑なシーン、例えば「ガラスの床の上に魔法陣といった半透明のエフェクト」や、「カップの中の氷」などをより高品質に描画することが可能となった。

1.2 論文の構成

本論文は全 5 章で構成する。第 2 章で OIT を実現する既存手法を説明し、第 3 章では既存手法のうち Moment Transparency に加えた改善を説明する。その後、第 4 章で改善手法の結果を評価し、第 5 章でまとめとする。

第 2 章

既存手法について

本章では、既存の OIT 手法について説明する。

2.1 既存手法の概要

Bavoil ら [8] は、シーンにある半透明物体を繰り返し描画し、前と奥の深度が格納されている深度バッファとポリゴンの深度を毎回比較することで、アルファブレンディングにおいて物体の前後関係を正しく処理する Dual Depth Peeling を提案している。複雑な形状の半透明物体を正しく描画できるが、物体を繰り返し描画する必要があり、場合によっては処理速度が非常に遅くなるというデメリットがある。

今給黎 [9] は、マルチサンプルフレームバッファに書き込むサブサンプル数をフラグメント毎にランダムに変化させ、アルファトゥカバレッジによって色を算出する「確率的な出力サブサンプル数制御に基づく描画順序非依存な半透明描画」を提案している。比較的品質の高い結果が得られるが、ランダム性を起因とするノイズが発生するというデメリットがある。Temporal Anti-Aliasing (TAA)[10] と Bilateral フィルタ [11] を導入することで軽減できるが、完全には除去できないとしている。

McGuire ら [12] は、色とアルファ値それぞれの加重平均から色を算出する WBOIT (Weighted Blended Order-Independent Transparency) を提案している。遠くと近くの半透明物体の間の距離が長くなおかつその物体同士が重なっている時は、並び変えによる描画と比較して不正確になるというデメリットがあるため、半透明物体同士の距離が一貫して近いシーンにしか向いていない。WBOIT では、半透明物体のアルファ値と深度から透過率を変化させる「重み」を算出し、

重みを色に乗算した後、その結果を 1 つ目の出力画像に加算すると同時に、アルファ値も 2 つ目の出力画像に加算する。その後、1 つ目の画像の R、G、B をそれぞれ個別にその画像の A で割ったものが出力の RGB となり、2 つ目の画像のアルファ値を 1 から引いたものが出力のアルファ値となり、最終的な RGBA の色が不透明物体の画像とブレンド処理によって合成される。

また、ハードウェアレベルでの OIT の例として、ドリームキャスト [13] に半透明の三角形を並び替える機能が備わっていたことが挙げられる [14]。

なお、Unity や Unreal Engine といった大手ゲームエンジンで OIT は現時点で標準機能として備わっていない。Unity においては単純にオブジェクトが奥から順に描画される [15]。Unreal Engine においてはオブジェクトごとに優先度を 0 前後の値として設定し、値が小さいほど先に描画するが、優先度が同じオブジェクトは奥のオブジェクトが先に描画されるように並び替えされる [16]。

2.2 Moment Transparency について

Sharpe[7] は、ピクセル毎に格納されている 4 つのモーメントと入力深度を比較し、色を算出する関数で McGuire らの WBOIT 手法 [12] の透過率関数を置き換える Moment Transparency を提案している。手法をそのまま実装すると WBOIT と比較して処理が遅くなる。しかし、モーメント及び視覚的深度 (optical depth) のバッファの解像度を低くすると、僅かなアーティファクトと引き換えに WBOIT に近い処理速度とメモリ使用量を実現できるとしている。なお、4 つのモーメントを使用する手法は、Peters ら [17] がシャドウマッピング手法で用いたものを由来としている。

Moment Transparency ではまず第 1 段階で、半透明物体のアルファ値のみから 1〜4 乗 (次) モーメントを RGBA として加算すると同時に、アルファ値から算出した視覚的深度を別の出力画像に加算する。その次の第 2 段階で、Peters ら [17] のシャドウマッピング手法において影の強度を算出する関数と視覚的深度を用いて重みを算出し、以降は WBOIT と同様の処理を行う。図 2.1

は Moment Transparency の処理の流れを図示したものである。この図の第 2 段階が WBOIT に追加された段階である。

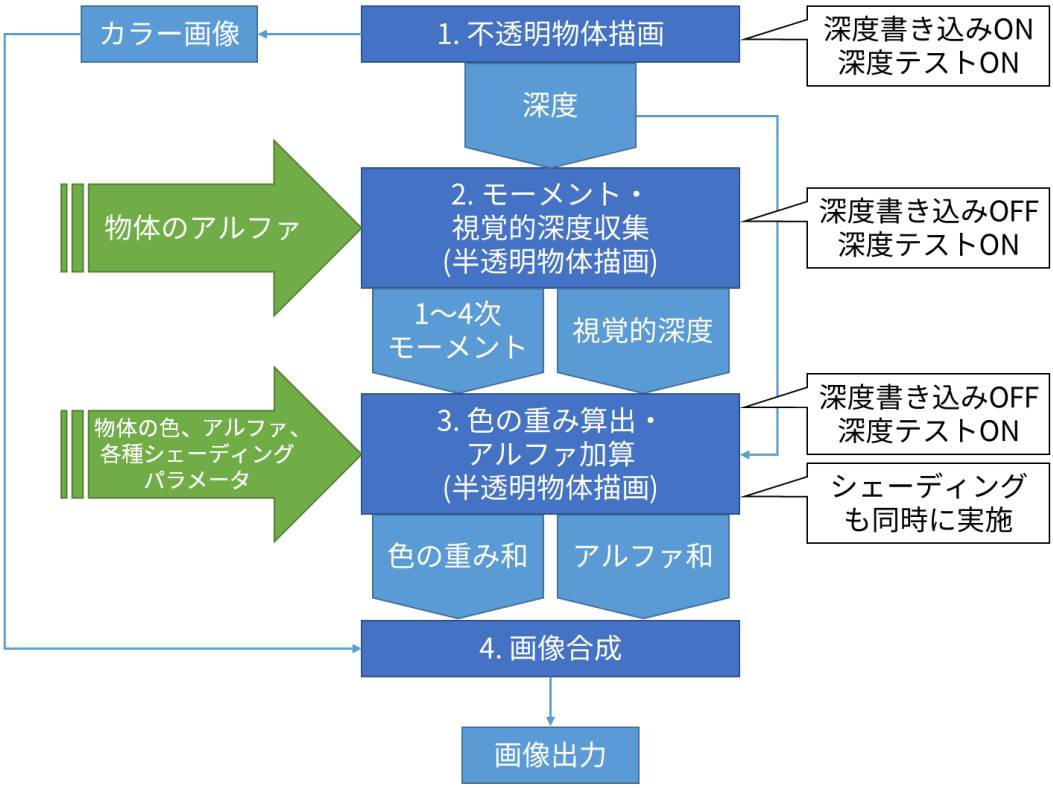

図 2.1　Moment Transparency の処理の流れ

2.3　既存手法の問題点

Moment Transparency[7] では概ね高い品質の半透明描画が得られるが、カメラを向いている 2 つの面が近接している際には、後ろの面が不自然に映り込む (想定以上に見えてしまう) 現象が発生する。図 2.2 は 2 つの物体が重なっている部分に不自然な描画が見られる状態を示す。図 2.3 は不自然箇所の拡大図である。

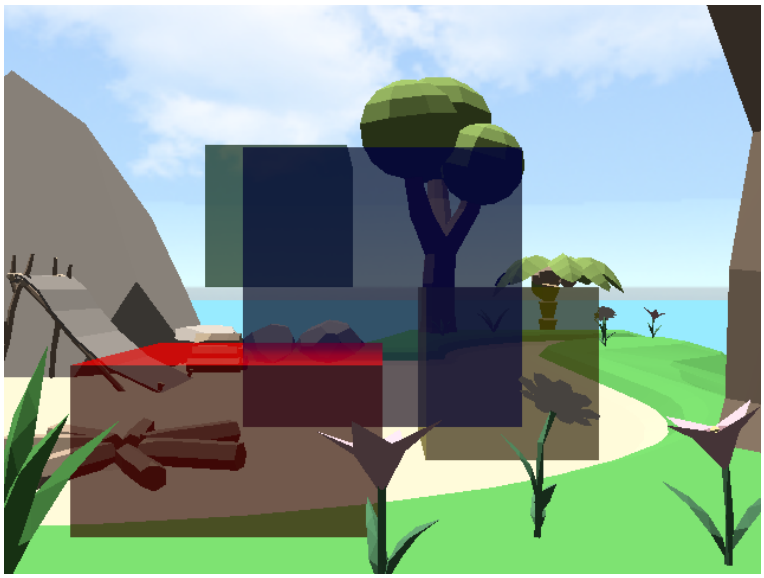

図 2.2　Moment Transparency をそのまま実装した場合の出力

図 2.3　図 2.2 の不自然に見える部分を拡大した画像

第 3 章

改善手法

Moment Transparency の問題点は、物体が近接している際に補正を行うことで改善できると考えられる。まずアルファ値からモーメントを算出する段階で、全ての半透明物体の深度の最小値を 3 つ目の出力画像に収集する。その次の重みを算出する段階で深度の最小値 z_{min} よりピクセルの深度 z が大きい場合、その差に適切な補正係数 m (ここでは 100 とする) を掛け、その結果を 0 と 1 の間に制限したものを Moment Transparency における通常の重み w_{in} に乗算することで、最終的な重み w_{out} を算出する。つまり、面と面の交差点に近づくほど奥の面のアルファ値が小さくなり、前の面が本来想定される色に近くなる。式 3.1 は w_{out} の算出式である。

$$w_{out} = w_{in} \min(\max(m(z - z_{min}), 0), 1) \tag{3.1}$$

なお、式 3.1 では算出値が 0 と 1 の間の範囲から外れると不具合が発生するため、min と max によって制限している。

図 3.1 は提案手法の処理の流れを図示したものである。既存手法である Moment Transparency に追加した処理は「最小深度」の部分となる。

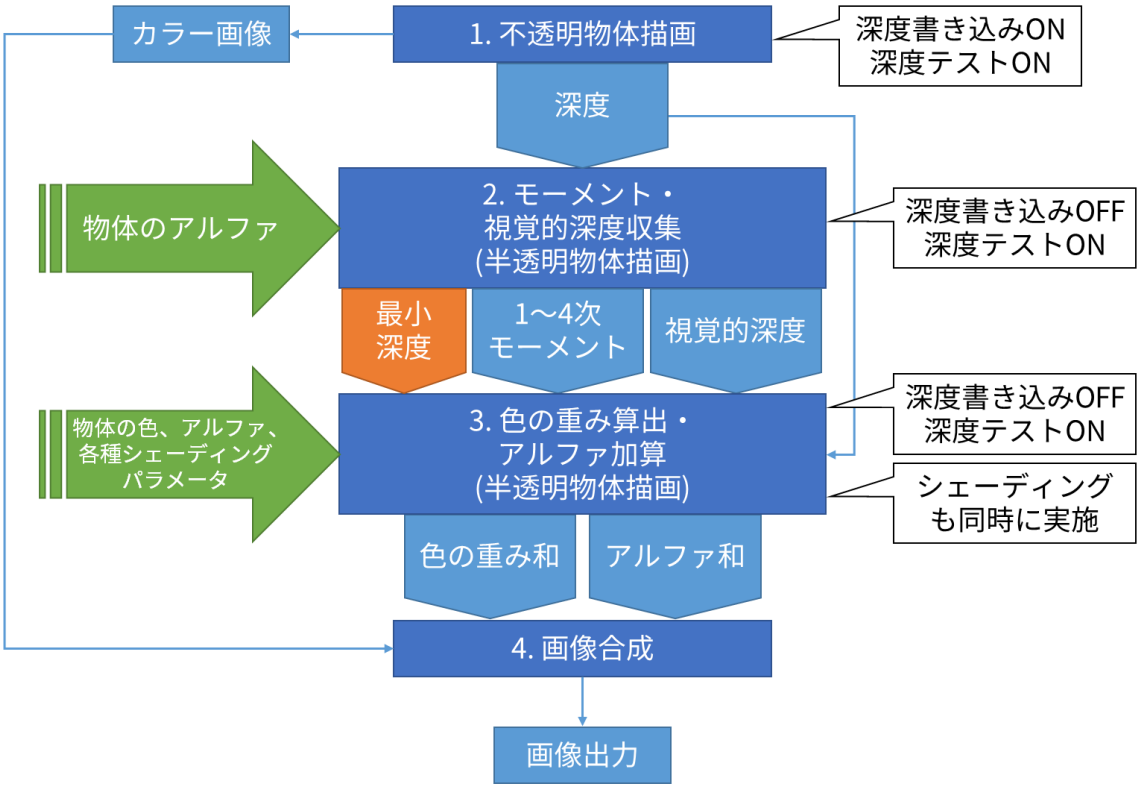

図 3.1　改善手法の処理の流れ

第 4 章

評価と分析

図 4.1 は Moment Transparency と本手法の出力結果の比較画像を示し、図 4.2 は比較箇所の拡大画像を示す。改善によって、半透明物体が近接している際は物体同士が接近する部分がはっきりと見えるようになった。

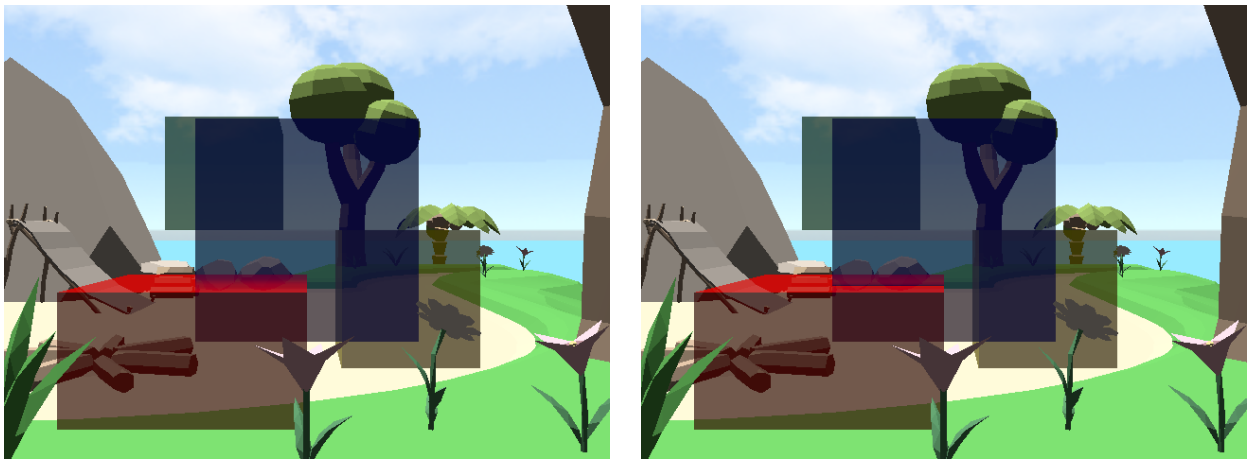

図 4.1　Moment Transparency と本手法の出力画像比較

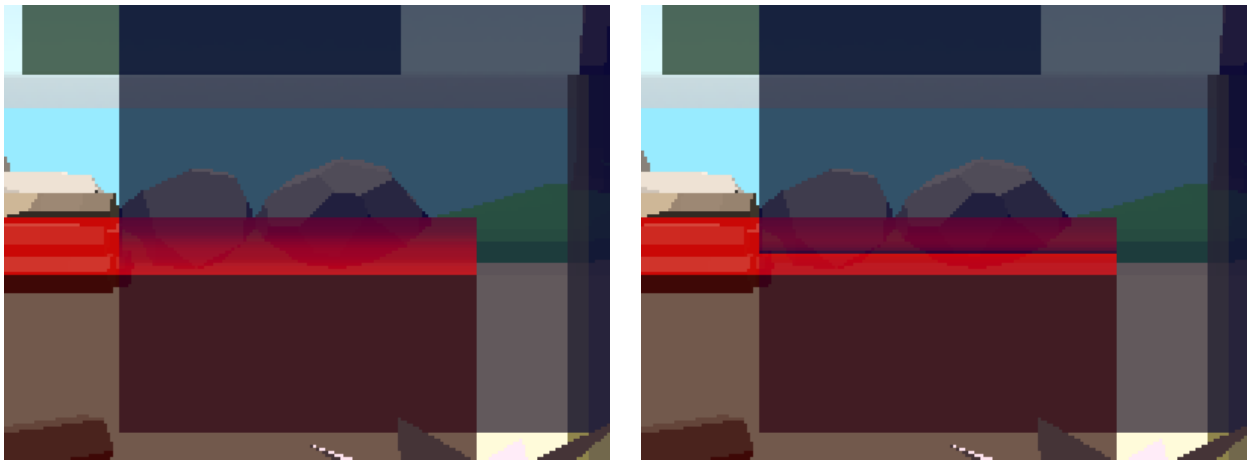

図 4.2　Moment Transparency と本手法の出力画像拡大図比較

図 4.3 と図 4.4 は別角度からの比較画像を示す。

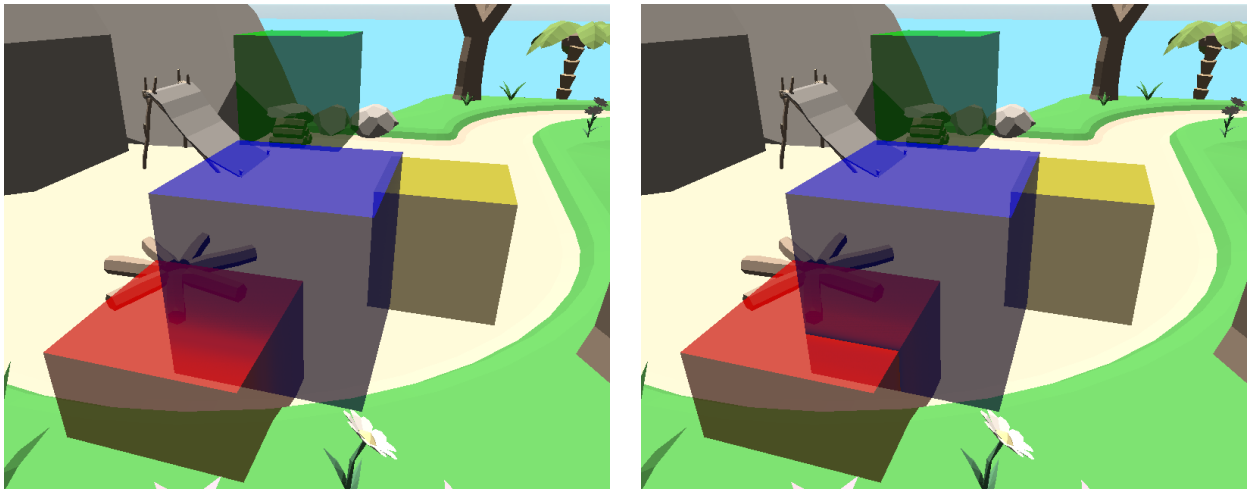

図 4.3　Moment Transparency と本手法の出力画像比較 (別角度)

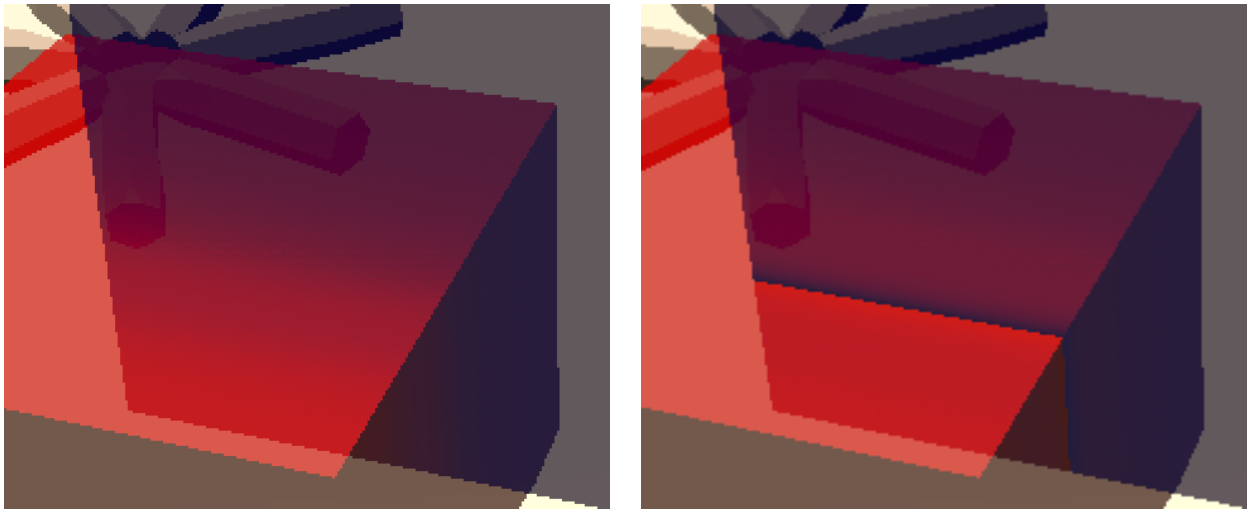

図 4.4　Moment Transparency と本手法の出力画像拡大図比較 (別角度)

図 4.5 と図 4.6 は別形状に対する出力比較画像を示す。より複雑なモデルを使用した場合にも、ある程度改善されている。

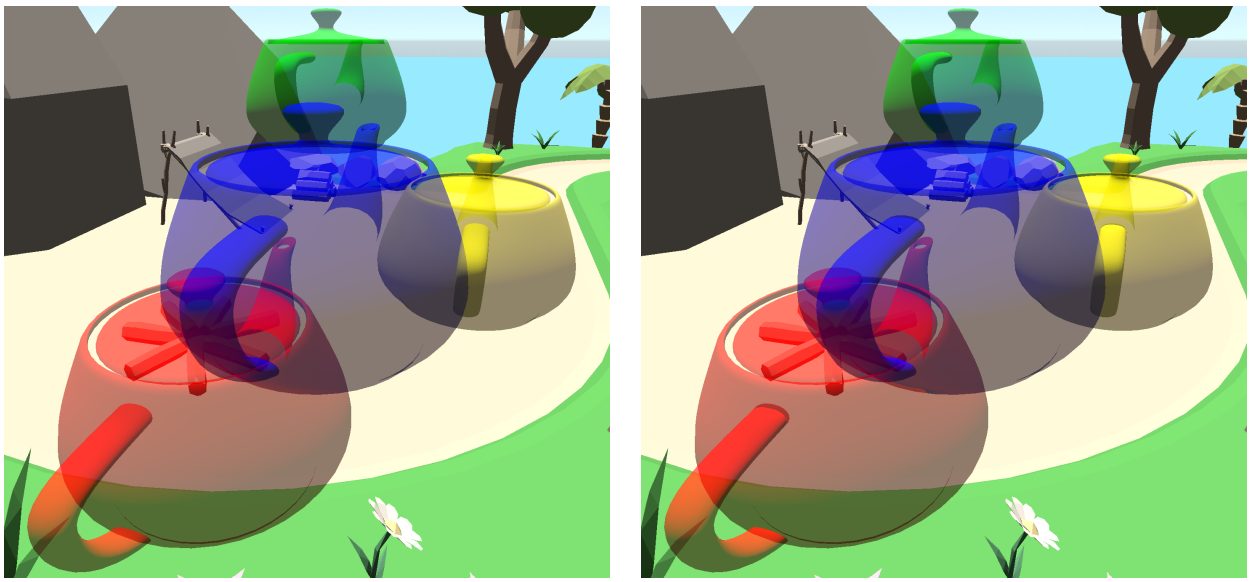

図 4.5　Moment Transparency と本手法の出力画像比較 (ティーポットモデル使用)

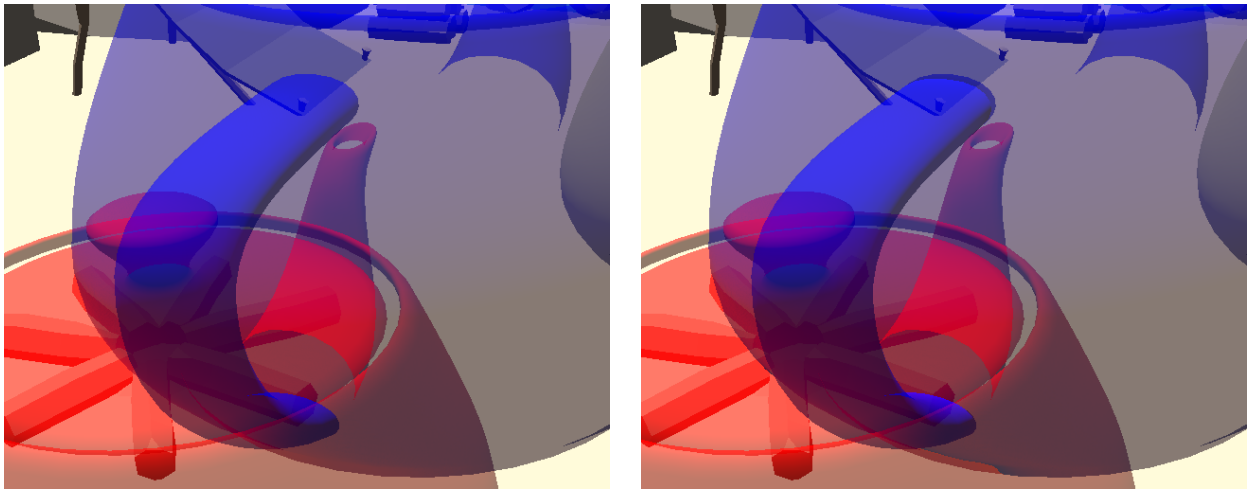

図 4.6　Moment Transparency と本手法の出力画像拡大図比較 (ティーポットモデル使用)

なお、図 4.1 から図 4.6 までの出力には、異なるスペックの PC を用いて実行速度の検証を行った。表 4.1 にデスクトップ PC 環境を、表 4.2 にノート PC 環境をそれぞれ示す。出力画像の内容はどちらも同一であった。ただし、出力画像の解像度はディスプレイ解像度とは関係なく 1280x720 とし、垂直同期はオフにしてフレームレートは無制限とした。また、拡大していない図は 1280x720 の出力画像から切り取ったものである。開発環境に関しては、プログラミング言語

は Rust、グラフィックス API は Vulkan を使用し、どちらの PC においても同じソースコードを使用した。

表 4.1　デスクトップ PC の仕様

CPU	AMD Ryzen 9 7900X
RAM	DDR5-6000 16GBx2
GPU	AMD Radeon RX 5700 XT
GPU VRAM	GDDR6 8GB バス幅 256 ビット
GPU ドライバー	Mesa 22.3.2 RADV
OS	Arch Linux (Linux カーネルバージョン 6.1.4.arch1-1)

表 4.2　ノート PC の仕様

CPU	AMD Ryzen 7 5800H
RAM	DDR4-3200 16GBx2
GPU	NVIDIA GeForce RTX 3060 Laptop
GPU VRAM	GDDR6 6GB バス幅 192 ビット
GPU ドライバー	GeForce Game Ready Driver 527.56
OS	Windows 10 Pro 64-bit 22H2

どちらの PC においても、フレーム時間が 2ms 以下 (500fps 以上) と非常に短くなっており、その誤差も非常に大きい。現時点では描画している半透明物体の数が少なく、ライティング処理も簡易的なものになっており、CPU 負荷が GPU 負荷を大きく上回っていると考えられる。そのため、今回の GPU 負荷の計測は断念した。今後はより複雑な半透明物体を大量に描画した際の GPU 負荷の調査が課題だと考えられる。

本手法の最も大きな問題点として、メモリ使用量の増加が挙げられる。本手法で追加された深度最小値を格納する画像のフォーマットに 32 ビット浮動小数点を採用しているため、フル HD (1920x1080) 画像の場合は約 7.91 MiB、4K (3840x2160) 画像の場合は約 31.6 MiB 増加することになる。本手法のベースとなっている Moment Transparency では RGBA の全てに 32 ビット浮動小数点が用いられるフォーマットを使用するなど、元々メモリ使用量が高いため、メモリ速

度が低い内蔵 GPU などでは処理速度がさらに低下すると考えられる。

なお、16 ビットのフォーマットを使用したところ、精度の低さが原因と思われる大きな不具合が発生し、OIT 手法が全く機能していないに等しかったため、今回は不採用に至った。精度が問題とならないよう、今後改善の余地があると考えられる。

また、今回は深度と深度最小値の差に掛ける補正係数 m を 100 とすると良好な結果が得られたが、カメラの near 値や far 値、物体の配置などによっては調整が必要になると考えられる。本研究では near は 0.25、far は 5000 に設定した。参考までに、m を調整した場合の出力を図 4.7 から 4.22 までで示す。$m = 1$ や $m = 5$ といった極端に小さい値に関しては赤以外のキューブが青のキューブを通して暗く見えてしまう現象が発生したが、それ以外に関しては切り取った部分以外に変化は見られなかった。

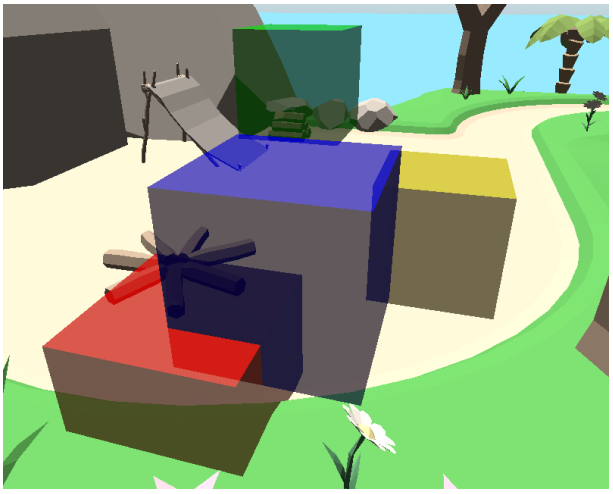

図 4.7　$m = 1$

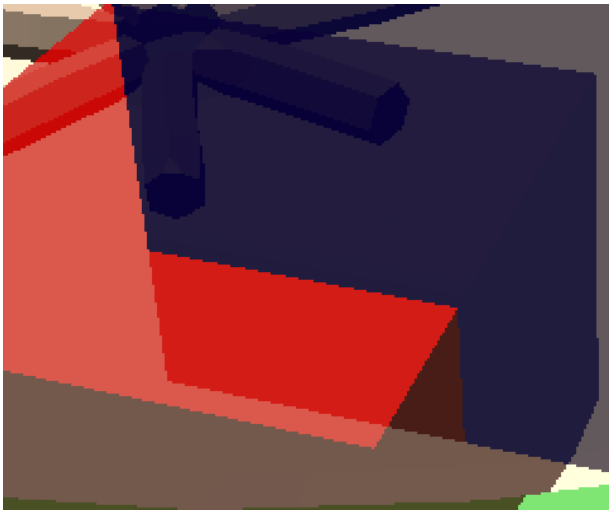

図 4.8　$m = 1$ (拡大)

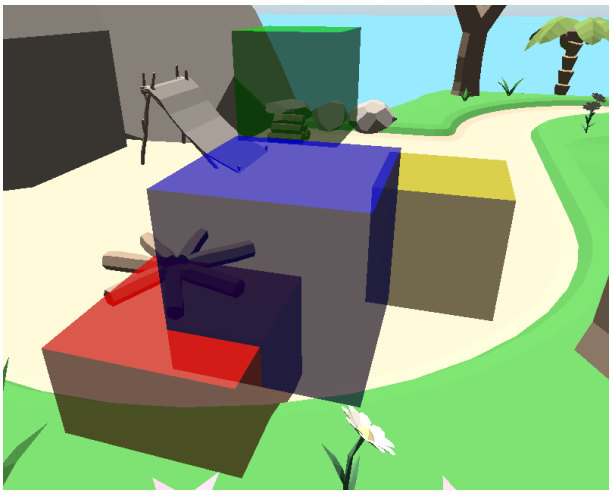

図 4.9　$m = 5$

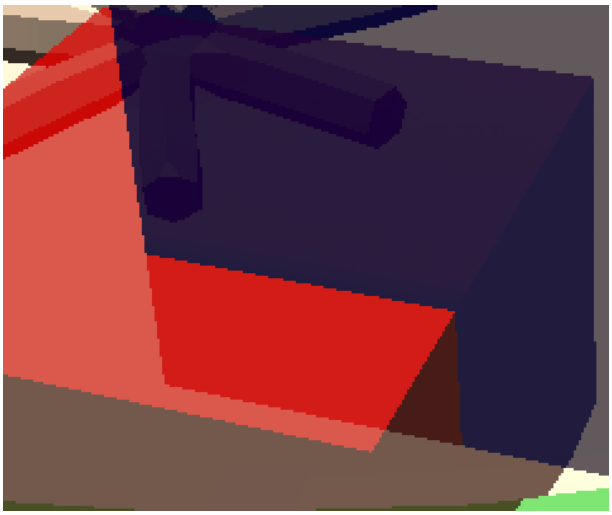

図 4.10　$m = 5$ (拡大)

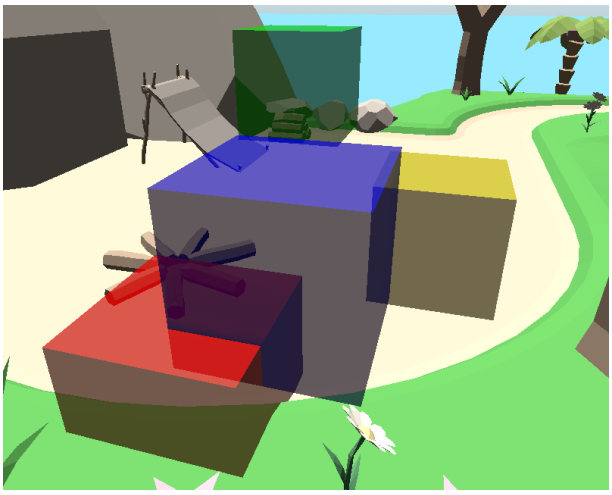

図 4.11　$m = 25$

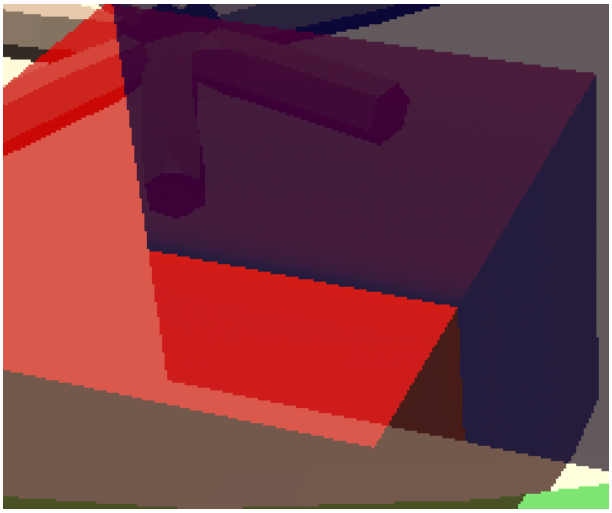

図 4.12　$m = 25$ (拡大)

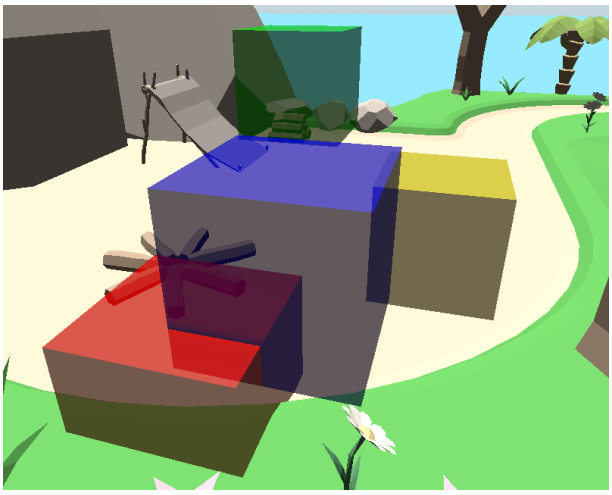

図 4.13　$m = 50$

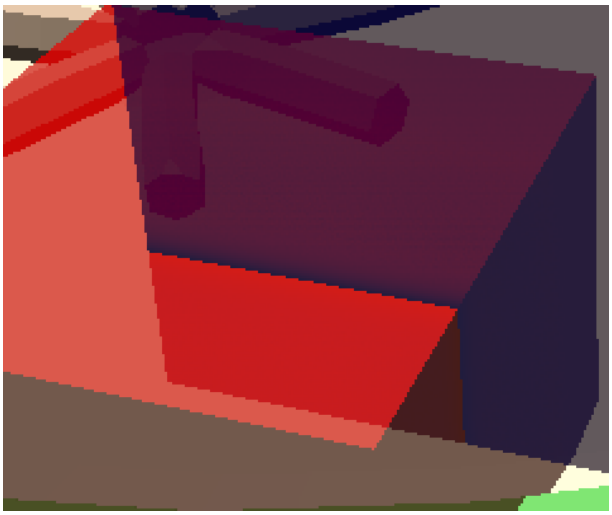

図 4.14　$m = 50$ (拡大)

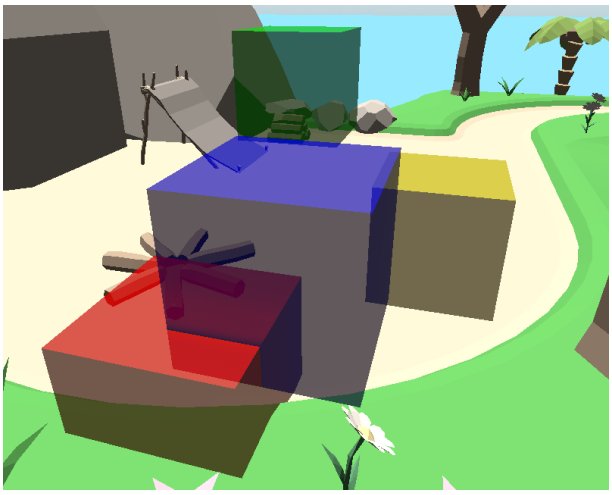

図 4.15　$m = 75$

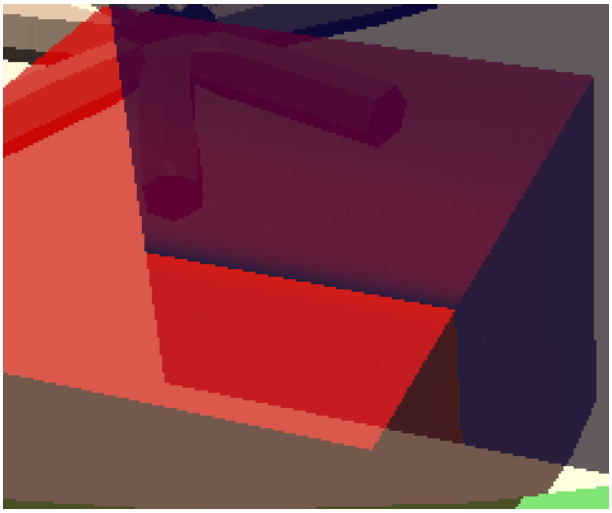

図 4.16　$m = 75$ (拡大)

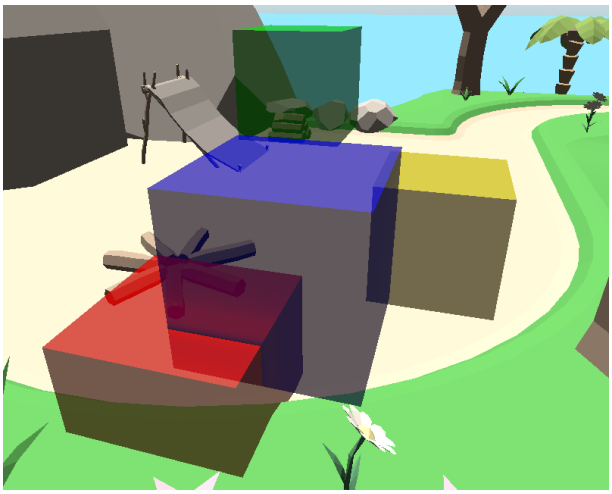

図 4.17　$m = 100$ (既定値)

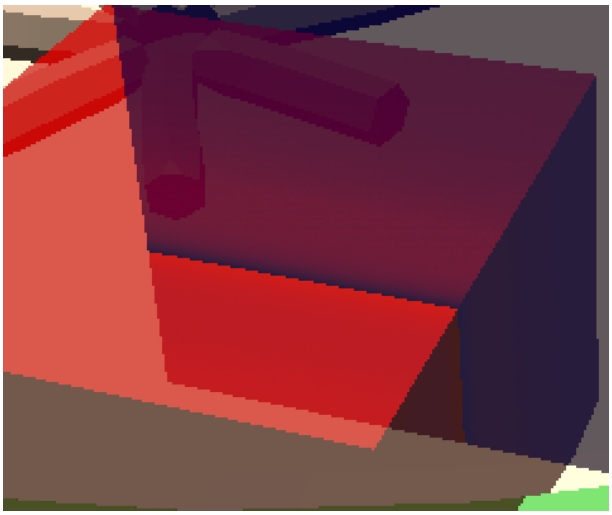

図 4.18　$m = 100$ (既定値) (拡大)

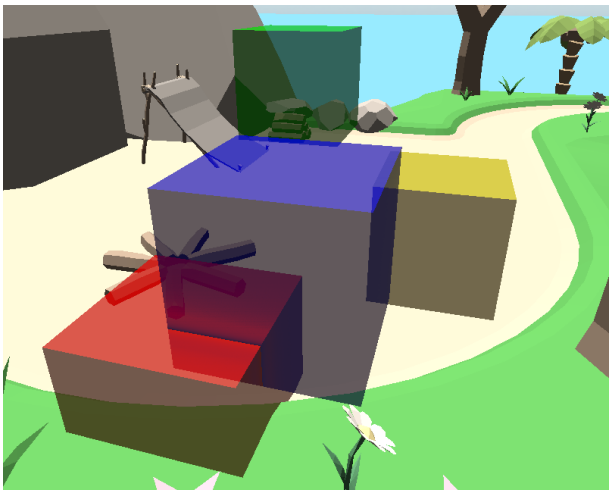

図 4.19　$m = 150$

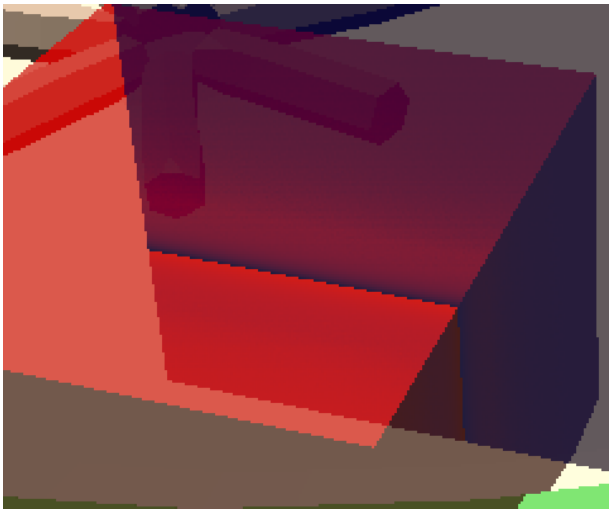

図 4.20　$m = 150$ (拡大)

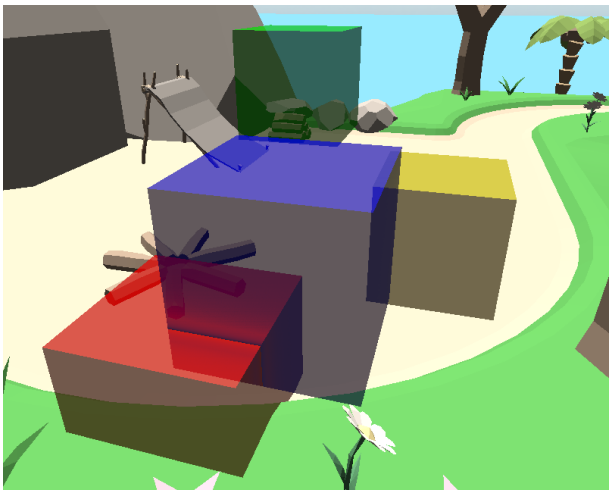

図 4.21　$m = 200$

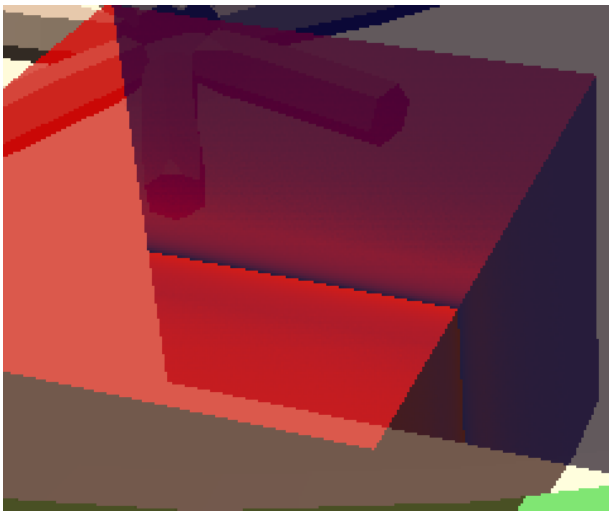

図 4.22　$m = 200$ (拡大)

第 5 章

まとめ

OIT の既存手法として、 McGuire ら [12] の WBOIT では色とアルファ値からの加重平均で色を算出しているのに対して、Sharpe[7] の Moment Transparency では WBOIT を改造してアルファ値からのモーメントで色を算出している。 Moment Transparency では半透明物体同士が近接している際、前の物体のアルファ値が小さいかのように、後ろの物体が不自然に見えてしまう現象が発生していた。これを改善するため、本手法では後ろの物体の面が最も前にある物体の面に近いほど、後ろの面のアルファ値を小さくする補正をかけて本来想定される色に近づけた。これにより、ゲームなどのリアルタイム CG において、半透明物体を用いた複雑なシーン、特に半透明物体同士が近いシーンをより高品質に描画することが可能となった。

今後の展望として、本手法のメモリ使用量の削減のほかに、半透明物体をより多く重ねた際の品質に関する調査が必要である。しかし、本手法のメモリ使用量を削減したとしても、メモリ使用量に対して品質が大きく改善されているといえないこともある。そこで、本手法のようにフレームバッファやシェーダーを複数用いるのではなく、ドリームキャストの半透明描画機能 [14] のように、GPU 上での三角形の並び替えによる半透明物体の描画も調査対象である。テッセレーションシェーダーといったポリゴンを分割する処理との組み合わせにより交差や突起における不正確な描画を改善することも調査対象と考えられる。

なお、GPU 上での並び替えを実現する具体的な方法としては、マージソート [18] やクイックソート [19] など、並列化可能で GPU 上で実行可能なソートアルゴリズムをコンピュートシェーダーで実装することが考えられる。ただしその場合、マテリアルといったパラメータも三角形ごとに紐づける必要があるため、レンダリングパイプラインを従来の CPU 主体ではなく GPU 主体

とし、GPU 側で処理を完結 [20] させると、CPU でのバインドなどによるボトルネックを避けることで効率良く実装できると考えられる。

謝辞

本研究に取り組むにあたり多くの指導をしていただいた渡辺先生と阿部先生に感謝を申し上げます。また、時折大変ご迷惑をおかけして申し訳ありませんでした。そして、展示等で多くの方々から意見をいただいたことにも感謝しております。特に展示の際に CG を専門分野とする柿本先生からいただいたハードウェア処理に関する意見は、研究を進めるにあたっての大きなヒントと励みになりました。

参考文献

[1] M. E. Newell, R. G. Newell, and T. L. Sancha. A solution to the hidden surface problem. In *Proceedings of the ACM Annual Conference - Volume 1*, ACM '72, p. 443 – 450, New York, NY, USA, 1972. Association for Computing Machinery.

[2] gldepthrange - opengl 4 reference pages. `https://registry.khronos.org/OpenGL-Refpages/gl4/html/glDepthRange.xhtml`, 2014. 参照: 2023.2.13.

[3] Depth buffers (direct3d 9). `https://learn.microsoft.com/en-us/windows/win32/direct3d9/depth-buffers`, July 2021. 参照: 2023.2.13.

[4] Vulkan ® 1.3.240 - a specification. `https://registry.khronos.org/vulkan/specs/1.3/html/chap24.html#vertexpostproc-clipping`, January 2023. Chapter/Section 24.2. 参照: 2023.2.13.

[5] Thomas Porter and Tom Duff. Compositing digital images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, p. 253 – 259, New York, NY, USA, 1984. Association for Computing Machinery.

[6] Alex Dunn and Louis Bavoil. Transparency (or translucency) rendering. `https://developer.nvidia.com/content/transparency-or-translucency-rendering`, October 2014. 参照: 2023.1.18.

[7] Brian Sharpe. Moment transparency. In *Proceedings of the Conference on High-Performance Graphics*, HPG '18, New York, NY, USA, 2018. Association for Computing Machinery.

[8] Louis Bavoil and Kevin Myers. Order independent transparency with dual depth peeling. Technical report, NVIDIA Corporation, February 2008.

[9] 今給黎隆. 確率的な出力サブサンプル数制御に基づく描画順序非依存な半透明描画. Technical Report 6, 株式会社セガゲームス, aug 2015.

[10] Jonathan Korein and Norman Badler. Temporal anti-aliasing in computer generated animation. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '83, p. 377 – 388, New York, NY, USA, 1983. Association for Computing Machinery.

[11] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pp. 839–846, 1998.

[12] Morgan McGuire and Louis Bavoil. Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques (JCGT)*, Vol. 2, No. 2, pp. 122–141, December 2013.

[13] ドリームキャスト — セガハード大百科. `https://sega.jp/history/hard/dreamcast/`. 参照: 2023.2.13.

[14] Sebastian Wloch. Optimizing dreamcast microsoft direct3d performance. `https://docs.microsoft.com/en-us/previous-versions/ms834190(v=msdn.10)`, March 1999. 参照: 2023.1.16.

[15] Unity 2021.3 - scripting api: Transparencysortmode. `https://docs.unity3d.com/ScriptReference/TransparencySortMode.html`, January 2023. 参照: 2023.1.19.

[16] Using transparency in materials - unreal engine 5.1 documentation. `https://docs.unrealengine.com/5.1/en-US/using-transparency-in-unreal-engine-materials/`. 参照: 2023.1.19.

[17] Christoph Peters and Reinhard Klein. Moment shadow mapping. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, i3D '15, p. 7 – 14, New York,

NY, USA, 2015. Association for Computing Machinery.

[18] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, Vol. 68, No. 10, pp. 1381–1388, 2008. General-Purpose Processing using Graphics Processing Units.

[19] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *ACM J. Exp. Algorithmics*, Vol. 14, , jan 2010.

[20] Victor Blanco. Gpu driven rendering overview. `https://vkguide.dev/docs/gpudriven/gpu_driven_engines/`, 2021. 参照: 2023.2.13.