保守的シャドウマップを用いたFrustum Traced Shadowsの高速化

-Accelerating Frustum Traced Shadows Using Conservative Shadow Maps-

溝口 智博　　徳吉 雄介

株式会社スクウェア・エニックス

E-mail: mizotomo@square-enix.com, tokuyosh@square-enix.com

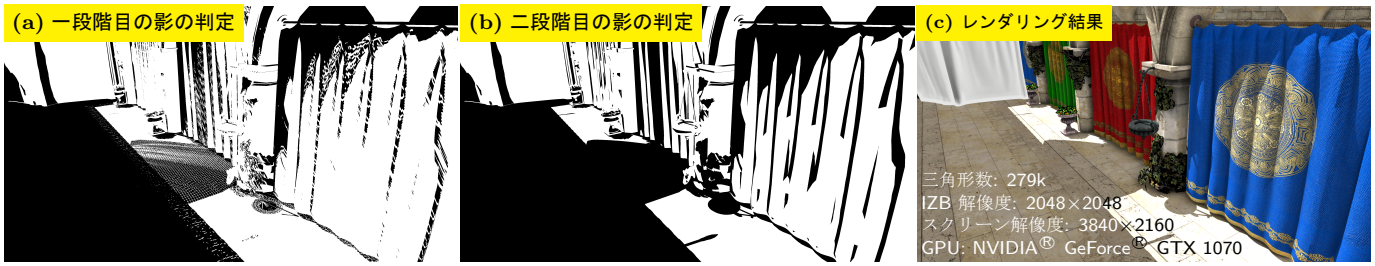


図 1: 本論文のハードシャドウの描画パイプラインで出力した visibility mask buffer (a，b) と最終レンダリング結果 (c)．保守的シャドウマップを用いて事前に影を粗く判定することで (a)，厳密な影の判定を行う必要のあるシェーディング点を削減する．これによってハードシャドウの計算時間を 7.4 ms から 3.7 ms へ減らすことができた．

概 要

本論文は Frustum Traced Shadows 法によるハードシャドウの計算を高速化する描画パイプラインを提案する．この Frustum Traced Shadows 法は近年ゲームのようなリアルタイムアプリケーションで用いられているが，シャドウマップ法と比べると計算コストが高いという問題がある．そこで本論文ではこの Frustum Traced Shadows 法のパイプラインに保守的シャドウマップを用いた影の判定処理を組み込み，二段階の影の判定を行うことで高速化する．更に本論文は既存手法より精度の高い保守的シャドウマップの実装についても述べる．4K のスクリーン解像度における実験の結果，高速化の度合いはシーンによってばらつきがあるものの平均すると約 2.4 倍影の描画速度を向上できることが確かめられた．

1　はじめに

Wyman et al. [7] が提案した Frustum Traced Shadows 法はレイトレーシングと同品質のハードシャドウをリアルタイムで描画する技術であり，近年では NVIDIA® Corporation が提供する NVIDIA® GameWorks で利用することができる [1]．この手法は Ubisoft Entertainment S.A. の Tom Clancy's The Division™ 等のゲームで使用されている [5] が，通常のシャドウマップ法 [6] と比べると処理時間が大きいという問題がある．そこで本論文はこの Frustum Traced Shadows 法を高速化する影の描画パイプラインを提案する．

Frustum Traced Shadows 法では Irregular Z-Buffer (IZB) [7] に光源から見たシェーディング点を Per-Pixel Linked Lists [8] を用いて保存し，各シェーディング点に対してフラスタムトレーシングによる厳密な影の判定を行う．しかしながらこのフラスタムトレーシングによる影の判定はシェーディング点とシーンの三角形数が多いと計算コストが高くなってしまうという問題がある．そこで本論文では Frustum Traced Shadows 法のパイプラインに保守的シャドウマップ [3] を組み込むことで二段階の影の判定を行う．保守的シャドウマップは必ず影となる保守的な深度を保存したシャドウマップである．通常のシャドウマップが影か否かの判定に誤差を持つのに対し，保守的シャドウマップは必ず影であることが保証された領域を部分的に検出することができる．この保守的シャドウマップは元々静的なシーンにおいてシャドウレイの数を削減するために開発されたものであるが，本論文では動的なシーンにおいて IZB のテクセルに格納されたシェーディング点をカリングするために用いる．つまりこの保守的シャドウマップによってカリングされなかったシェーディング点にのみフラスタムトレーシングによる影の判定を行うことで計算コストを削減する．Wyman et al. の Frustum Traced Shadows 法では IZB に格納されるシェーディング点を用いて三角形をカリングするので，本論文が提案するシェーディング点のカリングによって更に三角形もより多くカリングすることができるようになる．また本論文ではシェーディング点のカリング効率をより向上させるため，精度の高い保守的シャドウマップの実装についても述べる．

Wyman et al. の実装ではフラスタムトレーシングを行う中で影と判定されたシェーディング点を逐次 IZB から除

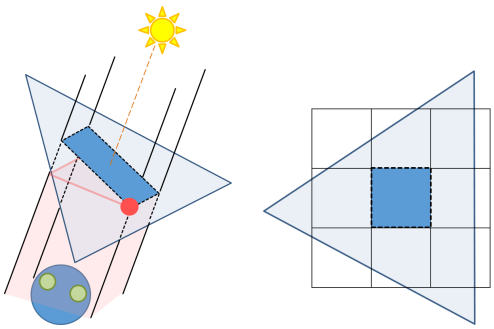


図 2: 必ず影となるシェーディング点（左）と三角形で完全に覆われたテクセル（右）．三角形で完全に遮蔽された領域（ピンクの領域）に含まれるシェーディング点（緑の点）は必ず影となる．この領域は保守的な深度（赤い点の深度）を用いて表現される．

プログラム 1: 保守的シャドウマップのピクセルシェーダー（HLSL）．

```
void main(float4 p : SV_Position, float2 s : BARYCENTRIC){
  float2 dx = ddx(s) * 0.5;
  float2 dy = ddy(s) * 0.5;
  float2 a = dx + dy;
  float2 b = dx - dy;
  if(s.x<max(abs(a.x), abs(b.x)) || s.y<max(abs(a.y), abs(b.y))
    || s.x+s.y>1.0-max(abs(a.x+a.y), abs(b.x+b.y))){
    discard;
  }
}
```

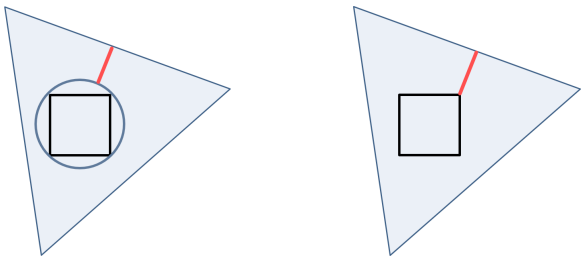
去していたが，本パイプラインはフラスタムトレーシングを行う前にシェーディング点をカリングするので効率が良い．このカリングで用いる保守的シャドウマップは単一の三角形で完全に覆われたテクセルに保守的な深度を保存することによって生成されるため，本パイプラインは保守的シャドウマップのテクセルよりも大きな三角形が存在するシーンに対して特に効果的である．

以上をまとめると本論文の貢献は以下の通りである．

- Frustum Traced Shadows 法のパイプラインに保守的シャドウマップを組み込むことで影の描画を高速化する．
- 保守的シャドウマップの精度を向上させることで IZB に格納されるシェーディング点をより多くカリングする．

2 保守的シャドウマップ

本パイプラインの流れを説明する前にまず保守的シャドウマップについて述べる．保守的シャドウマップには必ず影となる保守的な深度が格納されなければならない．そのため，この深度を出力するにはまずテクセルが単一の三角形で完全に覆われているか否かを判定する必要がある（図 2）．Hertel et al. [3] は保守的シャドウマップの画像空間に投影した三角形の各辺からテクセルの外接円までの距離を計算することによってこの判定を行った（HLSL コードを付録に掲載する）．しかしこの手法は三角形で完全に覆われたテクセルを必要以上に保守的に検出してしまうという問題がある（図 3a）．そこで本論文では保守的シャドウマップの精度を向上させるために，より厳密な判定を既存手法とほぼ同等の計算時間で行う実装を提案する．



(a) Hertel et al.　　(b) 本手法

図 3: 三角形で完全に覆われたテクセルの検出．Hertel et al. [3] の実装はテクセルの外接円から三角形の各辺までの距離を保守的シャドウマップに投影した空間で計算するのに対し (a)，本論文ではテクセルの 4 隅から三角形の各辺までの距離の最小値を重心座標系で計算することで Hertel et al. よりも厳密な判定を行う (b)．

2.1 三角形で完全に覆われたテクセル

プログラム 1 に本論文の保守的シャドウマップのピクセルシェーダーを示す．三角形で完全に覆われたテクセルを厳密に判定するために，テクセルの四隅の重心座標系 (u, v, w) の各軸の最小値（即ち三角形の各辺からテクセルまでの符号付距離）を用いる（図 3b）．無限遠光源を仮定するとこれらの値は以下の式で与えられる．

$$\begin{aligned} u(x,y) - \max(|a_u|, |b_u|), \\ v(x,y) - \max(|a_v|, |b_v|), \\ w(x,y) - \max(|a_w|, |b_w|). \end{aligned} \tag{1}$$

ここで x, y は保守的シャドウマップのテクセル中心の位置，そして $[a_u, a_v, a_w], [b_u, b_v, b_w]$ は重心座標系におけるテクセルの対角線ベクトルの長さを半分にしたものであり，$\mathbf{s}(x,y) = [u(x,y), v(x,y), w(x,y)]$ とすると以下の式で与えられる．

$$[a_u, a_v, a_w] = \frac{\partial \mathbf{s}(x,y)}{2\partial x} + \frac{\partial \mathbf{s}(x,y)}{2\partial y},$$
$$[b_u, b_v, b_w] = \frac{\partial \mathbf{s}(x,y)}{2\partial x} - \frac{\partial \mathbf{s}(x,y)}{2\partial y}.$$

式 (1) がひとつでも 0 未満のときテクセルは三角形からはみ出る．従って，テクセルが三角形で完全に覆われている

か否かは以下の条件式で与えられる．

$$u(x,y) < \max(|a_u|, |b_u|),$$
$$v(x,y) < \max(|a_v|, |b_v|),$$
$$w(x,y) < \max(|a_w|, |b_w|).$$

ここで $w(x,y) = 1 - u(x,y) - v(x,y)$，$|a_w| = |a_u + a_v|$，$|b_w| = |b_u + b_v|$ であるので，w 軸に関する判定は $u(x,y), v(x,y)$ を用いて記述することもできる．現在の我々の実装ではこの $u(x,y), v(x,y)$ をジオメトリシェーダーを用いて生成しているが，HLSL のシェーダーモデル 6.1 から使用可能となる予定の SV_Barycentrics [2] を用いることでこれらの値をジオメトリシェーダーで生成せずともピクセルシェーダーで直接使用することができるようになる．従ってこの実装は将来高速化できる可能性を持っている．

2.2 保守的な深度

保守的シャドウマップにはテクセルを三角形に投影した四角形上の深度よりも遠い値を保存する必要がある（図 2）．無限遠光源を仮定するとこの四角形は平行四辺形であるため，対角線ベクトルより，保存すべき保守的な深度の下限値 $z_{\min}$ を求めることができる．

$$z_{\min} = z(x,y) + \frac{\max\left(\left|\frac{\partial z(x,y)}{\partial x} + \frac{\partial z(x,y)}{\partial y}\right|, \left|\frac{\partial z(x,y)}{\partial x} - \frac{\partial z(x,y)}{\partial y}\right|\right)}{2}.$$

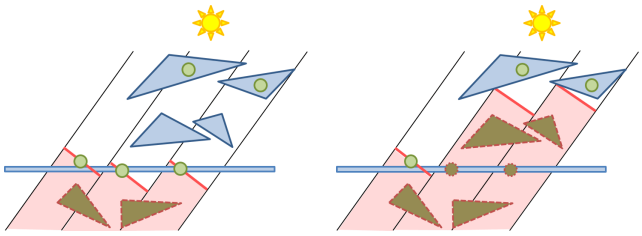
ここで $z(x,y)$ はテクセル中心における三角形の深度である．この $z_{\min}$ はピクセルシェーダーで計算することもできるが，ピクセルシェーダーで計算した深度を出力すると GPU ラスタライザーの深度カリングを活用できないという問題がある．そのため深度に関しては本論文でも Hertel et al. [3] と同様に GPU ラスタライザーがサポートする Slope Scaled Depth Bias を使用することで $z_{\min}$ を近似する．Slope Scaled Depth Bias を用いると，GPU ラスタライザーが出力する値 z_{out} は以下の式で与えられる．

$$z_{\text{out}} = z(x,y) + d \ \max\left(\left|\frac{\partial z(x,y)}{\partial x}\right|, \left|\frac{\partial z(x,y)}{\partial y}\right|\right).$$

ここで d は Slope Scaled Depth Bias である．$d = 1$ に設定することで GPU ラスタライザーは $z_{\text{out}} >= z_{\min}$ を満たす保守的な深度を自動的に出力する．

3 影の描画パイプライン

本論文では Frustum Traced Shadows 法のパイプラインに 2 章で述べた保守的シャドウマップを生成する処理と，その保守的シャドウマップを用いた粗い影の判定処理を新たに追加することで IZB に格納されるシェーディング点をカリングする．このカリングによってシェーディング点だけでなく，フラスタムトレーシングにおける三角形フラグメントのカリングの効率も向上させることができる．



(a) 既存手法　(b) 本手法

図 4: 既存手法と本手法のパイプラインにおける三角形フラグメントのカリングの比較．図 (a) の場合，最遠のシェーディング点よりも遠い位置に存在する三角形フラグメントをカリングするが図 (b) であれば必ず影となる領域（ピンク色の領域）に存在する三角形フラグメントもカリングすることができる．

3.1 粗い影の判定

本パイプラインにおける粗い影の判定は IZB の生成パス内で行う．また本論文ではシェーディング点に対する IZB の投影処理と保守的シャドウマップの投影処理を統一するため，両者を同じ解像度に設定する．この投影空間においてシェーディング点が影か否かを保守的シャドウマップの深度を用いて判定し，影となるシェーディング点を IZB に格納する前にカリングする．もしシェーディング点が影と判定された場合，このシェーディング点に対応する visibility mask buffer のピクセルに影を出力し（図 1a），そうでなければ IZB のリストにこのシェーディング点を追加する．この粗い影の判定は IZB の生成処理にオーバーヘッドを発生させるが，シェーディング点が削減される分リストの構築コストが軽減されるため，実際には生成時間は短縮される．

3.2 フラスタムトレーシングによる影の判定

フラスタムトレーシングによる影の判定ではシェーディング点がシーンの三角形によって遮蔽されるか否かを正確に判定する．このフラスタムトレーシングは光源から見た三角形をラスタライズし，ピクセルシェーダーで IZB に格納されたシェーディング点を順次参照することで行われる．単純な実装だとこの影の判定は光源から見えるシーンの三角形フラグメント全てに対して実行されてしまうため，Wyman et al. [7] は GPU ラスタライザーで深度バッファを用いて三角形フラグメントをカリングし，高速化を行っている．この深度バッファは IZB に格納されるシェーディング点のリストの中で光源から最も遠い点の深度を保存することによって作成される．本パイプラインでは IZB に格納されるシェーディング点が削減されるため，既存手法よりもより光源に近い深度が深度バッファに保存される．従って本論文が提案するシェーディング点のカリングを用

表 1: ハードシャドウの計算時間（ms）.

Sponza (三角形数: 279k) SanMiguel (三角形数: 5.3M) PowerPlant (三角形数: 12.8M)

2K 解像度	既存手法	**本手法**	既存手法	**本手法**	既存手法	**本手法**	既存手法	**本手法**	既存手法	**本手法**	既存手法	**本手法**
光源のビューフラスタム構築	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
保守的シャドウマップ生成	–	**0.17**	–	**0.20**	–	**1.89**	–	**1.90**	–	**4.34**	–	**4.30**
IZB 構築	0.74	**0.43**	0.38	**0.28**	0.63	**0.32**	0.70	**0.42**	0.30	**0.26**	0.30	**0.27**
フラスタムトレーシング	1.11	**0.54**	3.51	**0.29**	10.67	**8.18**	21.15	**8.74**	21.68	**9.96**	12.62	**10.24**
合計	1.88	**1.17**	3.92	**0.80**	11.33	**10.42**	21.88	**11.09**	22.01	**14.59**	12.95	**14.84**
4K 解像度	既存手法	**本手法**	既存手法	**本手法**	既存手法	**本手法**	既存手法	**本手法**	既存手法	**本手法**	既存手法	**本手法**
光源のビューフラスタム構築	0.10	0.10	0.10	0.10	0.13	0.12	0.11	0.11	0.12	0.12	0.10	0.10
保守的シャドウマップ生成	–	**0.35**	–	**0.47**	–	**2.10**	–	**2.08**	–	**4.72**	–	**4.59**
IZB 構築	3.03	**1.67**	1.66	**1.16**	2.58	**1.18**	2.82	**1.65**	1.24	**1.07**	1.22	**1.10**
フラスタムトレーシング	3.99	**1.23**	8.63	**0.45**	15.73	**8.83**	35.49	**9.46**	29.74	**10.29**	15.33	**10.79**
合計	7.12	**3.35**	10.39	**2.18**	18.44	**12.23**	38.42	**13.30**	31.10	**16.20**	16.65	**16.58**

表 2: 保守的シャドウマップの実行結果.

	表 1b のシーン		表 1e のシーン	
	生成時間	カリング効率	生成時間	カリング効率
Hertel et al.	0.48 ms	55.5%	4.75 ms	35.8%
本手法	0.47 ms	56.0%	4.72 ms	37.4%

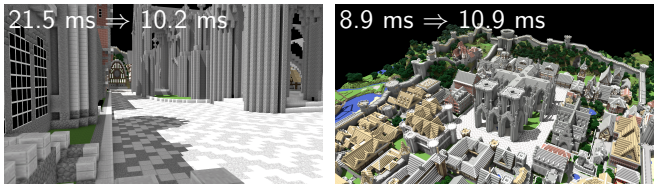
いることによってシェーディング点だけでなく，三角形フラグメントも Wyman et al. より多くカリングすることができるようになる（図 4）．本パイプラインではこの深度バッファの作成も IZB の生成時に行う．深度バッファへの深度の書き込みは GPU ラスタライザーを用いて行う必要があるので，本論文ではスクリーンの各ピクセルをポイントプリミティブとして扱うことで頂点シェーダーを用いた単一のパスで IZB と深度バッファを同時に作成する．

4 実験結果

提案したパイプラインによる実験結果を示す．本実験では NVIDIA® GeForce® GTX 1070 GPU を用いてハードシャドウの計算時間を計測した．実験には 2K のスクリーン解像度（1920×1080）に対して 1024×1024 解像度の IZB を，4K のスクリーン解像度（3840×2160）に対して 2048×2048 解像度の IZB を用いた．これらは本実験環境下で最も良いパフォーマンスが得られた解像度の組み合わせである．本論文では既存手法 [5, 7] と違って IZB の空間分割を行っていないが，光源のビューフラスタムの大きさは既存手法と同様にシェーディング点のバウンディングボックスを用いて決定した．

計算時間. 既存の Frustum Traced Shadows 法と本手法の各パスの計算時間を表 1 に示す．計算時間を比較すると高速化の度合いに違いはあるものの，平均すると 2K 解像度で約 1.9 倍，4K 解像度で約 2.4 倍影の描画速度が向上することが確認された．この高速化の度合いに振れ幅があるのはカリングされるシェーディング点の数が三角形の大きさに依存するためであり，保守的シャドウマップのテクセルより大きな三角形によって遮蔽されるシェーディング点が多いほど効率が良くなるからである．表 1f において，2K 解像度では本手法の合計計算時間が既存手法よりも遅くなっているが，4K 解像度ではカリングによる高速化が保守的シャドウマップを生成するオーバーヘッドを上回っている．3.1 節で述べたように本論文では IZB と同じ解像度（2K スクリーンで 1024×1024，4K スクリーンで 2048×2048）の保守的シャドウマップを用いる．保守的シャドウマップの解像度が高いと，三角形で完全に覆われたテクセルが多くなるためシェーディング点のカリング効率を向上させることができる．従って 4K スクリーンにおける計算時間が既存手法よりも高速化されたと考えられる．

保守的シャドウマップの実行結果. 表 2 に 2048×2048 解像度の保守的シャドウマップを生成した際の実行結果を示す．実行時間について本手法は Hertel et al. [3] とほぼ同等であったが，シェーディング点のカリング効率は比較的三角形が大きなシーン（表 1b）で 0.5%，三角形が小さなシーン（表 1e）で 1.6%良い結果を得ることができた．これは保守的シャドウマップの生成において，三角形で完全に覆われたテクセルを既存手法よりも厳密に判定したためである．表 1b と表 1e のシーンでの実行時間を比較すると実行時間に約 4.3 ms 違いがあるが，各シーンの 2K 解像度と 4K 解像度における保守的シャドウマップの生成時間は



(a) 効率的なケース　　(b) 効率的でないケース

図 5: 4K 解像度における Rungholt モデル（三角形数: 6.7M）のレンダリング結果．図 (a) のシーンでは光源のビューフラスタムが小さいので保守的シャドウマップのテクセルよりも大きな三角形が多くなり，シェーディング点が効率的にカリングされる．一方図 (b) のシーンだとビューフラスタムが大きいのでテクセルよりも小さな三角形が多くなり，保守的シャドウマップによるカリングの効果があまり得られない．

平均 0.25 ms 程しか違いが見られない．これは保守的シャドウマップの生成コストがピクセルシェーダーではなく，バーテックスシェーダー，ジオメトリシェーダー，そしてラスタライザーに大きく依存していることを示しており，三角形数が多くなると計算コストが大きくなる．このオーバーヘッドは 2.1 節で述べた SV_Barycentrics を用いてジオメトリシェーダーを除去することで，将来小さくできると考えられる．

失敗例．　図 5 において提案したパイプラインは (a) の場合効率的であるが，(b) の場合従来のパイプラインよりも処理時間が大きくなる．これは (b) のようなビューフラスタムの大きなシーンだとテクセルよりも小さな三角形が多く描画され，保守的シャドウマップのオーバーヘッドがカリングによる高速化を上回ってしまうからである．この問題に対する解決案として，テクセルよりも大きな三角形のみ保守的シャドウマップに描画することが挙げられる．こうした保守的シャドウマップ用の描画オブジェクトの自動的な選択は今後の課題である．

5　結論

5.1　まとめ

本論文では，Frustum Traced Shadows 法によるハードシャドウの計算を高速化する描画パイプラインを提案した．4K のスクリーン解像度でこのパイプラインを用いることでハードシャドウの計算時間が平均約 2.4 倍高速化された．本パイプラインはスクリーンと保守的シャドウマップの両方の解像度が高くなると，より効率的にハードシャドウを描画することができる．そして保守的シャドウマップのテクセルよりも大きな三角形が多いと本手法はより高速化することができる．本パイプラインでは現状，保守的シャドウマップを生成するためにジオメトリシェーダーを使用しているが，将来 HLSL の SV_Barycentrics を代わりに用いることで生成オーバーヘッドを小さくできることが期待される．

5.2　今後の課題

小さな三角形．　保守的シャドウマップのテクセルよりも小さな三角形は保守的な深度に全く影響を及ぼさないため，この三角形を保守的シャドウマップに描画すると計算コストが高くなってしまうという問題がある．この問題の解決方法としてテクセルよりも大きな三角形のみを自動的に選択して保守的シャドウマップに描画することが挙げられるので今後検証していきたい．

IZB の投影空間の分割．　既存手法と違い，本論文では IZB の投影空間を分割する手法 [5, 7] を使用していない．計算コストを更に削減するために今後実装していきたいと考えている．この分割にはシェーディング点の分布を使用するが [5]，保守的シャドウマップによるカリングを用いて無駄なシェーディング点を削減することができるので，より効率の良い分割を行えるのではないかと考えられる．

保守的シャドウマップの高速化．　三角形で完全に覆われたテクセルの検出は提案手法以外にも Conservative Rasterization Tier3 対応 GPU（Intel® HD 530 や AMD Radeon™ RX Vega シリーズ）であれば，HLSL の SV_InnerCoverage [4] を用いることで検出することもできる．この機能は単純であるが，一方で Conservative Rasterization を使用しなくてはならないため，ピクセルシェーダーの実行回数が多くなってしまうという問題がある．著者らの持つ AMD Radeon™ RX Vega 56 GPU ではドライバーがまだ対応していないためか SV_InnerCoverage が正しく動作しなかったが，将来 SV_Barycentrics を用いた本論文の実装とどちらが高速に動作するか検証していきたいと考えている．

謝辞

本論文のポリゴンモデルには Marko Dabrovic 氏と Frank Meinl 氏による Crytek Sponza モデル，Guillermo M Leal LLaguno 氏による San Miguel モデル，ノースカロライナ大学による Power Plant モデル，そして kescha 氏による Rungholt モデルを使用させて頂いた．モデルデータの公開に感謝する．

参考文献

[1] NVIDIA Corporation. NVIDIA Hybrid Frustum Traced Shadows. https://developer.nvidia.com/Hybrid-Frustum-Traced-Shadows.

[2] DirectX Shader Compiler. Shader Model 6.1. https://github.com/Microsoft/DirectXShaderCompiler/wiki/Shader-Model-6.1.

[3] S. Hertel, K. Hormann, and R. Westermann. A Hybrid GPU Rendering Pipeline for Alias-Free Hard Shadows. In *Eurographics 2009 - Areas Papers*, October 2009.

[4] MSDN Library. SV_InnerCoverage. https://msdn.microsoft.com/en-us/library/windows/desktop/dn933281(v=vs.85).aspx.

[5] J. Story and C. Wyman. HFTS: Hybrid Frustum-traced Shadows in "the Division". In *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16, pp. 13:1–13:2, July 2016.

[6] L. Williams. Casting Curved Shadows on Curved Surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, pp. 270–274, August 1978.

[7] C. Wyman, R. Hoetzlein, and A. Lefohn. Frustum-Traced Irregular Z-Buffers: Fast, Sub-Pixel Accurate Hard Shadows. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 22, No. 10, pp. 2249–2261, October 2016.

[8] J. C. Yang, J. Hensley, H. Grün Gruen, and N. Thibieroz. Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum*, Vol. 29, pp. 1297–1304, June 2010.

付録

Hertel et al. [3] の保守的シャドウマップのシェーダーをプログラム 2，3 に示す．この実装ではまず保守的シャドウマップに投影した画像空間で三角形の頂点から対辺までの距離を計算する．この距離はジオメトリシェーダーで計算し，線形補間した値がピクセルシェーダーに渡される．ピクセルシェーダーではこの距離がテクセルの外接円の半径よりも大きいか否かを判定することによって三角形で完全に覆われたテクセルを検出する．

プログラム 2: Hertel et al. の保守的シャドウマップのジオメトリシェーダー（HLSL）．

```
struct Output{
  float4 p : SV_Position;
  float3 d : DISTANCE;
};
[maxvertexcount(3)]
void main(triangle float4 inPos[3] : SV_Position,
          inout TriangleStream< Output > output){
  float2 p0 = inPos[0].xy * g_smResolution;
  float2 p1 = inPos[1].xy * g_smResolution;
  float2 p2 = inPos[2].xy * g_smResolution;
  float2 e0 = p1 - p0;
  float2 e1 = p2 - p1;
  float2 e2 = p0 - p2;
  float2 n0 = normalize(float2(-e0.y, e0.x));
  float2 n1 = normalize(float2(-e1.y, e1.x));
  float2 n2 = normalize(float2(-e2.y, e2.x));
  float d0 = abs(dot(n1, e2));
  float d1 = abs(dot(n2, e0));
  float d2 = abs(dot(n0, e1));
  Output element0 = { inPos[0], float3(d0, 0.0, 0.0) };
  Output element1 = { inPos[1], float3(0.0, d1, 0.0) };
  Output element2 = { inPos[2], float3(0.0, 0.0, d2) };
  output.Append(element0);
  output.Append(element1);
  output.Append(element2);
  output.RestartStrip();
}
```

プログラム 3: Hertel et al. の保守的シャドウマップのピクセルシェーダー（HLSL）．

```
void main(float4 p : SV_Position, float3 d : DISTANCE){
  float DIAG = 1.4142135623730951;
  if (d.x < DIAG || d.y < DIAG || d.z < DIAG)
    discard;
}
```