GPU による点群ベース陰関数曲面の直接的レンダリング*

金井 崇　　大竹 豊　川田 弘明　加瀬 究
東京大学大学院総合文化研究科　理化学研究所 VCAD モデリングチーム

概要

陰関数曲面は，将来的に有望な形状表現の一つであり，より少ない数のプリミティブでオブジェクトの詳細かつ滑らかな幾何学的形状を表現することができる．本論文では，点群ベース陰関数曲面の一つである SLIM（*Sparse Low-degree IMplicit*）曲面を対象とした，プログラマブル GPU による高速なレンダリング手法を提案している．本手法はレイキャスティング法にもとづく直接的な描画手法を採用しており，光線と陰関数曲面の交点算出やブレンディングのための点の選択にかかわる処理を，プログラマブルシェーダの一つであるフラグメントプログラムの中で行っている．大容量のオブジェクトに対しては，SLIM 曲面の階層的なデータ構造を利用することで，LOD レンダリングや視錐カリングを効率的に行うことができる．GPU の特徴である処理の並列性を最大限に利用することで，CPU による処理よりはるかに高速にレンダリングできることを実証する．

1　はじめに

陰関数表現（*Implicit Representation*）は，古くより知られる曲面表現である．blobby model [10]，メタボール [19]，soft objects [18]，RBF（*Radial-Basis Functions*）[14, 4] などはすべてこの種の形状表現である．これらの点群ベース陰関数曲面（*Point-Based Implicit Surface*）と呼ばれる表現手法では，CG の分野において良く利用されているポリゴン（メッシュ）表現よりも少ない数のプリミティブによって，オブジェクトの詳細かつ滑らかな幾何学的形状を表現できる．また，複数の陰関数曲面のブレンドや集合演算により，厳密なソリッドモデルを定義することができる．

その中で，近年 MPU（*Multi-level Partition of Unity*）陰関数曲面 [13] や SLIM（*Sparse Low-degree IMplicit*）曲面 [12] が台頭し，将来的にも有望な形状表現の一つとなっている．これらの曲面表現では，各点に対し異なる大きさを持つサポート球が用意され，その中に，プリミティブとして一つ以上の低次の陰関数多項式（*Implicit Polynomial Function*）が定義される．曲面上の点の位置や導関数ベクトルは，隣接する複数の陰関数値の重み付和により計算される．MPU 陰関数曲面や SLIM 曲面は，それ自体が木構造による階層構造を持つため，異なる解像度の陰関数曲面を迅速に取り出すことが可能である．

一方で，これらの陰関数表現を表示する手段として，大まかに分けると次の二つの方法が考えられる．一つは，マーチングキューブ法 [9] 等の等値面生成手法を利用してポリゴンを作成し，そのポリゴンを表示する方法である．この方法は，しかしながら，ポリゴンおよび等値面を生成するための空間場を中間的なデータとして作成する必要があり，余分な空間量を消費する．また，陰関数曲面を変形するような場合，そのつどポリゴンの方もアップデートする必要があるため，特にアニメーションの際には不利である．

もう一つは，レイトレーシング法やレイキャスティング法等を用いて直接陰関数曲面を表示する方法である．このタイプの表示手法として，点群モデルに対するレイトレーシング [2, 16, 1, 15] や，ボリュームレンダリングの高速化 [6, 17, 8, 7] が行われている．陰関数曲面においても，西田ら [11] や Groot ら [5]，大竹ら [12] による直接的な表示手法が提案されている．これらの手法は，画素毎に視点から光線を伸ばし，オブジェクトとの交点を計算することで，正確な位置や法線にもとづく色計算を行うことができるため，表示の質は極めて高いと言える．しかし，多くの計算量を伴うため，現状では，CPU による計算ではリアルタイム表示が難しいのも事実である．

本研究では，近年進展著しいプログラマブル GPU (*Programmable Graphics Processing Unit*) を用いて，等値面ポリゴンや等値面を生成するためのボリューム空間を定義することなく，点群ベース陰関数曲面を直接レンダリングする手法を提案する．現在の GPU は，その高い並列性と処理の柔軟性から，科学技術計算プロセッサとしても注目を浴びている．本手法では，陰関数曲面の点座標や法線を，プログラマブルシェーダによってピクセル単位で計算することで，レイキャスティング法による高速なレンダリングを実現している．[12] では，SLIM 曲面に対し，ソフトウェアを利用した平行投影による直接的なレンダリング手法を提案しているが，本手法は透視投影を実現しており，かつ，GPU を利用することでより高速なレンダリングを可能としている．

2　GPU による SLIM 曲面の直接的レンダリング

本節では，SLIM 曲面 [12] を対象とした，GPU による陰関数曲面の直接的レンダリングについて説明する．SLIM 曲面は，サポート球を含むノードの木構造により定義される．ここではまず，木構造の葉ノードのみをレ

* Direct Rendering of Point-Based Implicit Surfaces on GPUs, by Takashi KANAI (The University of Tokyo, Graduate School of Arts and Sciences), Yutaka OHTAKE, Hiroaki KAWATA, and Kiwamu KASE (RIKEN, VCAD Modeling Team)

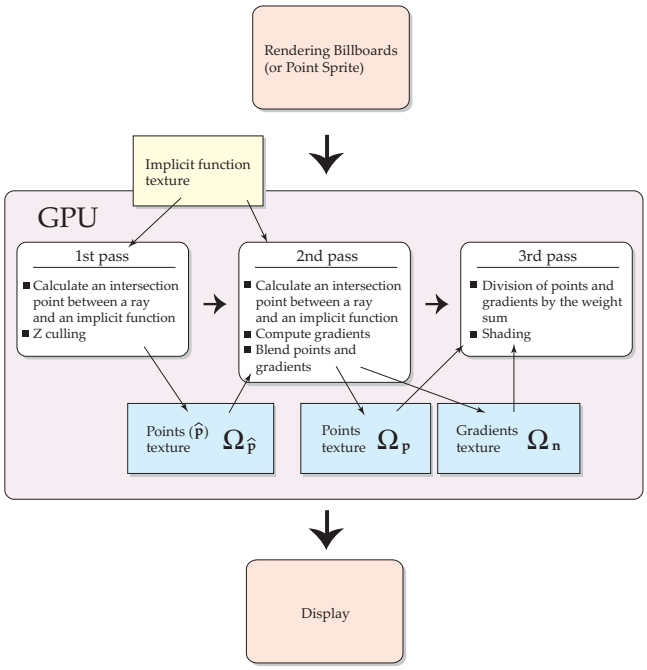

図 1　GPU による描画アルゴリズムの概要.

ンダリングすることを考える．葉ノードの集合における，各ノードのサポート球の半径および球の中心座標をそれぞれ r_i, $\mathbf{c}_i$ $(i = 1, \ldots, N, N$ は葉ノードの数$)$ とする．ここで説明の便宜上，各ノードはすべて 10 個の係数より構成される 2 次の陰関数多項式 $f_i(\mathbf{x})$ $(\mathbf{x} = (x, y, z))$:

$$f_i(\mathbf{x}) = a_i^0 x^2 + a_i^1 y^2 + a_i^2 z^2 + a_i^3 xy + a_i^4 yz + a_i^5 zx + a_i^6 x + a_i^7 y + a_i^8 z + a_i^9 = 0, \quad (1)$$

のみを持つものとする．ただし，他の次数でも同様に計算することができる．また，これらの陰関数多項式は，サポート球の中心を原点とする相対座標系で定義されているものとする．

図 1 に，本手法のレンダリング手順について図示したものを示す．本手法で提案するレンダリングアルゴリズムは，GPU によるスプラットのレンダリング手法（たとえば [3]）をその基本としている．その核となる処理は，3 パスにより構成される，以下のフラグメントプログラムでの処理である．

1. 視点から各フラグメントに対して光線をのばし，陰関数 $f(\mathbf{x}) = 0$ との交点 $\mathbf{p}$ を計算する．視点に一番近い交点を $\hat{\mathbf{p}}$ とし，テクスチャに格納する．
2. 1. と同様の処理により，$\hat{\mathbf{p}}$ のテクスチャを利用し，交点 $\mathbf{p}$ と勾配 $\nabla f(\mathbf{p})$ を，サポートの中心点 $\mathbf{c}$ との距離に応じた重み付ブレンドにより計算する．それぞれ別のテクスチャに格納する．
3. 2. で作成した $\mathbf{p}$ と $\nabla f(\mathbf{p})$ のテクスチャを利用し，曲面上の位置と法線ベクトルをそれぞれ計算する．シェーディング処理を行い，画面上に色を出力する．

以下の副節で，これら 3 パスの処理を含めた各手順についての詳細を説明する．

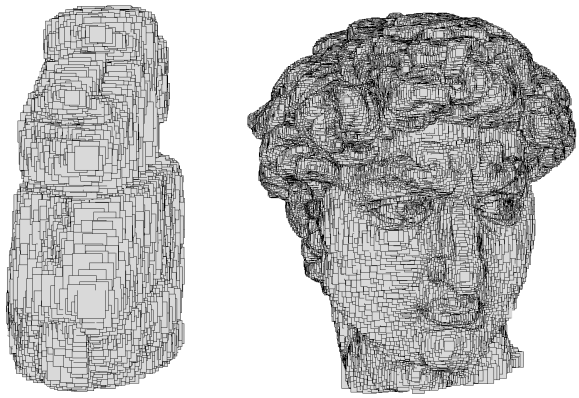

図 2　ビルボードによるサポート領域の描画.

2.1　ビルボードによるサポート領域の描画

1 パス目と 2 パス目では，視点からの光線と陰関数との交点を計算する．これらの計算はフラグメントプログラムの中で行われ，そのために必要なフラグメントは，陰関数を囲むようなサポート球を描画し，ラスタ化することで得られる．しかしながら，サポート球の描画は計算の負荷が非常に高い．通常，球の描画には球を近似したポリゴンを用いるが，ポリゴンを細かく分割すればするほど近似精度が高くなる一方で，描画すべきポリゴンの数は飛躍的に増加するためである．

その代わりに，ここでは球の直径を一辺とする四辺形を描画する．これより，四辺形のポリゴンを一つだけ描画すれば良いので，球のポリゴンと比較して計算負荷は大幅に少なくて済む．ただし，任意の視点において，球が投影されるスクリーン上の領域と同じ大きさの領域を，四辺形の描画においても必ず確保する必要がある．このため，四辺形は視点の変化によらず常に正面を向いている必要がある．

ここでは二通りの方法について検討する．一つはビルボードを用いた方法であり，もう一つは GPU の一機能であるポイント・スプライト（`GL_ARB_point_sprite`）を利用した方法である．ビルボードの場合，四辺形が正面を向くように四隅の頂点座標を CPU により計算し，この 4 点の頂点座標を GPU に受け渡す．一方で，ポイント・スプライトの場合は，四辺形の中心座標（この場合はサポート球の中心座標 $\mathbf{c}_i$）1 点のみを GPU に受け渡す．スプライトのスクリーン座標における大きさは，頂点プログラムの中で算出する．したがって，GPU へのデータの受け渡しを考えると，ポイント・スプライトの方が効率が良い．

ただし，ポイント・スプライトの方では，スプライトのスクリーン座標の最大値がハードウェア依存で固定されている[*1]．そのため，特にオブジェクトを拡大表示したとき，スプライトがある一定の大きさ以上にならず，その結果穴が開いてしまう，という問題が起こる．このことから，本手法では主にビルボードを用いることにしている．図 2 に，ビルボードによるオブジェクトの表示

[*1] 環境変数 `GL_POINT_SIZE_MAX_ARB` で調べることができる．ちなみに nVIDIA GeForce 7900 GTX では 63.375 である．

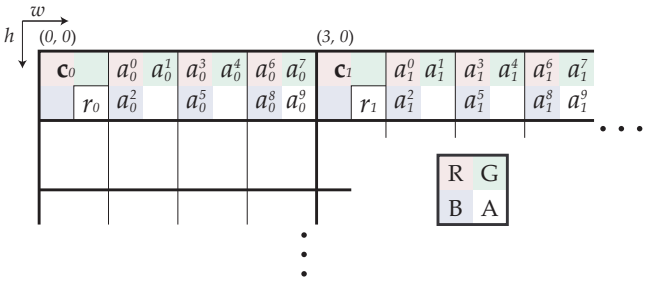

図 3　陰関数曲面パラメータの二次元浮動小数点テクスチャへの格納.

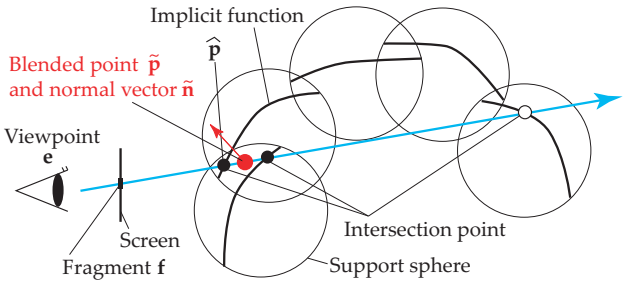

図 4　陰関数群と光線の交点算出.

例を示す.

2.2　陰関数テクスチャの作成

1 パス目と 2 パス目において，フラグメントプログラム内で陰関数曲面のパラメータを参照するために，ここではあらかじめ，これらのパラメータを二次元の浮動小数点テクスチャとして格納しておく．各ノードにつき格納すべきパラメータは，サポート球の中心座標 $\mathbf{c}_i$，半径 r_i，陰関数多項式の係数 $(a_i^0, a_i^1, \ldots, a_i^9)$ の計 14 個である．

図 3 に，本手法におけるテクスチャへの格納方法を示す．陰関数曲面に関する計算に必要な 14 個のパラメータは，一つのピクセルに RGBA 4 つの値を持つ，二次元の浮動小数点テクスチャに格納する．一つのノードのパラメータを格納するのに 4 つのピクセルを必要とする．GPU でこれらのパラメータにアクセスするために，各ノード i は，対応する 4 つのピクセルの先頭のピクセル座標 (w_i, h_i) を保持しておく．ビルボードレンダリングの際，四隅の頂点を指定するときに，このピクセル座標をテクスチャ座標として同時に指定する．この座標はラスタ化後のフラグメントに引き継がれ，フラグメントプログラムの中で，対応する陰関数曲面のパラメータを取得する際に利用される．テクスチャの大きさは，その最大容量がハードウェアによって決まっている[*2]ため，プリミティブの数がこれを超える場合，複数のテクスチャに格納する必要がある．

2.3　陰関数曲面の位置と法線ベクトルの計算

交点の算出.　図 4 に，陰関数群と光線との交点算出の模式図を示す．視点から各フラグメントへ光線を伸ばし，この光線とぶつかる陰関数曲面との交点を算出する．この計算は，サポートの中心点を原点とする局所座標系で行われる．このため，まず視点とフラグメントの座標を，交点計算の前に局所座標系へ変換する．一つのフラグメントにおいて，そのスクリーン座標（NDC, *Normalized Device Coordinate*）を $\mathbf{f}^s$ とすると，あるノード i の局所座標系への変換式は以下の通りである.

$$\mathbf{f} = (\mathbf{PM})^{-1}\,\mathbf{f}^s - \mathbf{c}_i, \tag{2}$$

ここで $\mathbf{P}, \mathbf{M}$ はそれぞれ投影行列，モデルビュー行列であり，$\mathbf{c}_i$ はノードのサポート球の中心点座標である．同様に局所座標系へ変換した視点の座標 $\mathbf{e}$ を用いて光線 $\mathbf{x} = \mathbf{f} + t\,(\mathbf{f} - \mathbf{e})$ が定義される．この光線とノード i の陰関数多項式 $f_i(\mathbf{x})$ との交点,

$$f_i\,(\mathbf{f} + t(\mathbf{f} - \mathbf{e})) = 0, \tag{3}$$

は，パラメータ t に関する二次方程式となり，解析的に解くことができる．交点は，サポート球の中に含まれる点のみを考慮する．すなわち，交点とサポート球の中心座標との距離とサポート球の半径 r_i を比較し，距離が半径よりも大きい場合は考慮しない．場合によっては交点が二つ求まる可能性があるが，その場合は，視点との距離を計算し，より小さい（視点により近い）点を選択する.

1 パス目の処理: $\hat{\mathbf{p}}$ の算出.　通常は一つのフラグメントに対し複数のビルボードが重複して投影されるため，複数の交点が求まることとなる．1 パス目では，このうちより視点に近い点（図 4 の左の黒塗の点）を選択し，この点を $\hat{\mathbf{p}}$ とする．これは，Z バッファを利用した Z カリングにより行うことができる．$\hat{\mathbf{p}}$ を求める目的としては，2 パス目で $\hat{\mathbf{p}}$ の遠くにある交点（例えば図 4 の白抜きの点）を排除するためである．得られた $\hat{\mathbf{p}}$ は，次のパスでアクセスするために，一旦浮動小数点テクスチャ $\Omega_{\hat{\mathbf{p}}}$ に格納する．なおここでは，FBO (*Frame-Buffer Objects*) による `Render_To_Texture` を利用することで，GPU - CPU 間の転送にかかるボトルネックを解消している.

Z 値の補正.　フラグメントはビルボードから計算されるため，現在の Z 値はビルボードを投影変換した結果得られる値である．通常，陰関数曲面との交点との Z 値とビルボードからのそれは異なるため，正しくカリングできない，という問題を引き起こす．このような問題が起こり得る状況を図 5 に示す．ビルボードの Z 値により，結果的に遠くにある陰関数曲面の交点が選ばれてしまう.

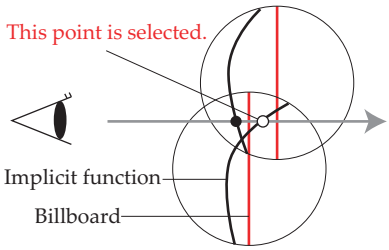

図 5　Z カリングに関する問題点．白抜きの点が誤って選ばれる.

[*2] 環境変数 `GL_MAX_RECTANGLE_TEXTURE_SIZE_EXT` で調べることができる．ちなみに nVIDIA GeForce 7900 GTX では 4,096 であるため，(4,096×4,096) / 4 = 4,194,304 の数のプリミティブを格納できる.

この問題に対処するため，ここでは求まった交点を再び投影変換して得られる Z 値を利用して，Z 値の補正を行う．交点 $\mathbf{p}$ を投影変換することによる新たな Z 値 Z' は，以下の式で求められる:

$$Z' = \frac{\mathbf{p}^s.z - Z_n}{Z_f - Z_n}, \quad \mathbf{p}^s = \mathbf{PMp}, \tag{4}$$

ここで Z_n, Z_f は，それぞれ前方，後方クリップ面における Z 値であり，$\mathbf{p}^s.z$ は $\mathbf{p}$ の投影変換後のスクリーン座標系における位置 $\mathbf{p}^s$ の z 座標である．式 (4) で計算した Z' を，フラグメントプログラムの中でデプス値と置き換える．

位置と法線ベクトルの計算． 2 パス目と 3 パス目で，各フラグメントにおける陰関数曲面の位置 $\tilde{\mathbf{p}}$ と法線ベクトル $\tilde{\mathbf{n}}$（図 4 の矢印付黒塗の点）を計算する．これらは，光線上の $\hat{\mathbf{p}}$ の近傍点 $\mathbf{p}_j$ ($j = 1 \ldots M$, M は近傍点の数) に対する PU (*Partition of Unity*) [12] により計算される:

$$\tilde{\mathbf{p}} = \frac{\sum_j^M \omega_j \mathbf{p}_j}{\sum_j^M \omega_j}, \tag{5}$$

$$\tilde{\mathbf{n}} = \frac{\mathbf{n}}{|\mathbf{n}|}, \quad \mathbf{n} = \frac{\sum_j^M \omega_j \nabla f(\mathbf{p}_j)}{\sum_j^M \omega_j}, \tag{6}$$

ここで $\nabla f(\mathbf{p}_j)$ は，点 $\mathbf{p}_j$ における陰関数曲面 f の勾配 (*gradient*) である．ω_j は重みであり，ここではガウス関数を用いている．ここでの問題は，フラグメントプログラムでは式 (5), (6) のうち割り算を行うことができないことである．このため，ここでは 2 パスでの処理に分割する．2 パス目では重み付和を計算し，その結果を利用して，3 パス目で割り算のみを行うことで $\tilde{\mathbf{p}}, \tilde{\mathbf{n}}$ を計算する．

2 パス目の処理: 交点のブレンディング． 2 パス目のフラグメントプログラムにおいて，まず $\Omega_{\hat{\mathbf{p}}}$ から $\hat{\mathbf{p}}$ を取得する．対応するピクセル座標は，フラグメントのスクリーン座標から計算できる．

次に，1 パス目と同様に交点 $\mathbf{p}$ を求め，$\hat{\mathbf{p}}$ との距離とサポート球の半径 r_i とを比較することで，$\hat{\mathbf{p}}$ の近傍点（例えば図 4 の右の黒塗の点）以外の点を排除する．これらの近傍点を用いて，重み付線形和 $(\sum_j^M \omega_j \mathbf{p}_j, \sum_j^M \omega_j)$, $(\sum_j^M \omega_j \nabla f(\mathbf{p}_j), \sum_j^M \omega_j)$ を求め，それぞれを浮動小数点テクスチャ $\Omega_{\mathbf{p}}, \Omega_{\mathbf{n}}$ に格納する．重み付線形和の計算は，GPU の機能の一つであるブレンディング処理で行うことができる．これは，Z バッファによるデプステストを無効にすると同時にブレンディング (`GL_BLEND`) を有効にし，さらにブレンド関数として「入力のアルファ値による重み付和」となるように設定する[*3]．この状態で，フラグメントプログラムのピクセル出力として 4 つの値 $(\mathbf{p}_j, \omega_j)$, $(\nabla f(\mathbf{p}_j), \omega_j)$ をそれぞれ指定すればよい．ここでは，GPU の一機能である Multiple Draw Buffer を利用することで，交点と勾配を別々のテクスチャに格納することができる．

[*3] `glBlendFunc(GL_SRC_ALPHA, GL_ONE)` と設定する．

図 6 david モデルの LOD レンダリングによる SLIM 曲面のノード数の制御．左: 葉ノードによる描画（932,720 ノード）．右: LOD ノードによる描画（126,466 ノード）．

3 パス目の処理: $\tilde{\mathbf{p}}, \tilde{\mathbf{n}}$ の算出とシェーディング． 3 パス目では，まずフレームバッファの解像度（＝テクスチャの解像度）と同じ四辺形ポリゴンを一つ描画する．すると，ラスタ化により得られるフラグメントの位置は，そのフラグメントのピクセル座標となる．このピクセル座標を用いて，2 パス目で生成した浮動小数点テクスチャ $\Omega_{\mathbf{p}}, \Omega_{\mathbf{n}}$ よりピクセル値 $(\sum_j^M \omega_j \mathbf{p}_j, \sum_j^M \omega_j)$, $(\sum_j^M \omega_j \nabla f(\mathbf{p}_j), \sum_j^M \omega_j)$ を取得する．4 番目の成分で 1 〜 3 番目の成分をそれぞれ除算することにより，位置 $\tilde{\mathbf{p}}$ と法線ベクトル $\tilde{\mathbf{n}}$ を求める．最後に，これらの値を用いてシェーディングを行い，画面に色を出力する．

2.4 LOD レンダリングと視錐カリング

SLIM 曲面は，個々の陰関数をノードとする木構造により構成される．この構造をもとに，サポート球を利用することで，LOD レンダリングや視錐カリングのためのラフなチェックができる．LOD レンダリングでは，根（*root*）ノードから順に子ノードを辿っていき，あるノードにおいて，サポート球の半径のスクリーン座標での長さが，ある閾値（ここでは数ピクセル）以下の場合，そのノードの親ノードをレンダリングする．視錐カリングにおいても同様のアルゴリズムが利用できる．上記の判定のかわりに，球が視錐の中に入っているかどうかをチェックすればよい．

図 6 には，david モデルの 葉ノードのレンダリングと LOD レンダリングによる描画結果の比較を示す．LOD レンダリングによって，視覚的にほとんど差のない描画結果を，より少ないノード数によって得られることが確認できる．

図 7　描画時間の計測に用いた例題モデルの本手法による描画例．上段左より: dino, armadillo, lucy. 下段左より: feline, xyzrgb dragon. 枠内は lucy の顔部分の拡大図.

3　結果と議論

図 7 に，描画時間の計測に利用した例題モデルの本手法によるレンダリング結果を示す．ポリゴン化によるレンダリングでは，解像度が低い場合にはジャギーが発生することがある．一方，本手法では，フラグメント単位で位置や法線ベクトルを計算し利用するため，高品質なレンダリングを実現できることが確認できる．なお，GPU 上での制約により，交点のブレンディングは 16 ビットでしか行うことができないが，レンダリング結果を見る限りでは，特にそれによる視覚的な不具合は見てとれないことも確認できる．

表 1 には，本論文で掲載されたモデルに対する描画時間を計測した統計結果を示す．このうち，CPU による描画時間は， Ohtake らによって実装されたソフトウェア [12] の実行時間を参考として掲載している．ただし [12] では平行投影によるレンダリングを提案しており，そのため本手法よりも計算が単純化されていることを追記しておく．

表より，GPU を利用した本手法は，CPU によるレンダリングよりもおおよそ 3 〜 7 倍の描画速度が得られていることが確認できる．特に，ノード数の少ないモデルに対する描画速度がより効果的である．これは，GPU による処理の並列化の効果が如実に現れているものと考えられる．本手法は，レイキャスティング法をベースとした手法であることから，塗りつぶされる画面上のフラグメントの数が大きいほど，それだけ描画時間がかかる．そのことを実証する例として，moai と dino のレンダリング結果を比較してみると，ノード数は moai の方が少ないにもかかわらず，dino の描画の方が 2 倍ほど高速である．これは，最終的に塗りつぶされた画面上のフラグメントの数が，moai の方がはるかに大きいことを意味している．

大容量のモデル，特に表の下二つのモデルに関しては，本手法による高速化の効果はそれほど現れていない．これは，モデルを描画する最初の段階の，CPU から GPU へデータを転送する処理がボトルネックになっていると推測できる．このような例では，転送すべきデータ（ノード）の数自体を減らすことが効果的である．実際，LOD レンダリングによって，処理速度が 2 〜 3 倍ほど向上していることが確認できる．

4　結論と展望

本論文では，点群ベース陰関数曲面の一つである SLIM 曲面を対象とした，プログラマブル GPU による高速なレンダリング手法について提案した．本手法では，フラグメント毎に曲面の位置や法線を計算することから，高品質なレンダリングが実現できることを実証した．CPU によるレンダリング時間との比較では，おおよそ 3 〜 7 倍の速度向上が見込め，特に大容量モデルに対しては，LOD レンダリングによりさらに 2 〜 3 倍の速度向上が見込めることを合わせて実証した．

今後の展望としては二つの方向性が考えられる．一つは，折り目や角などの鋭角特徴（*sharp feature*）を持つ SLIM 曲面に対応することである．SLIM 曲面では，鋭角特徴は特別なケースとして処理されることから，描画に関してもそれに合わせる必要がある．もう一つは，他の点群ベース陰関数曲面（メタボールや blobby model など）に対する直接的なレンダリング手法を確立することである．ただし，これらの曲面のレンダリングに関しては，本手法の枠組みで十分に対応可能であると考えている．

model	#total nodes	#leaf nodes	CPU (fps)	GPU (fps)	GPU (LOD) (fps)	#LOD nodes
moai	6,948	5,421	2.91	21.53	-	-
dino	7,783	6,009	7.09	44.92	-	-
feline	20,891	16,175	4.00	23.79	-	-
armadillo	31,712	24,662	4.27	22.97	-	-
xyzrgb dragon	242,018	185,050	2.79	12.86	-	-
lucy	915,151	691,978	1.12	3.50	6.45	130,696
david	1,221,100	932,720	0.93	2.63	7.08	126,466

表 1　描画時間の計測の統計結果．左より，モデル名，総ノード数，葉ノード数，CPU による描画時間，GPU による葉ノードの描画時間，GPU による LOD ノードの描画時間，LOD ノード数を示す（描画時間の単位はすべて fps）．計測時の描画ウインドウの大きさはすべて 512×512 としている．描画時間は CPU: Pentium D 840 (3.2GHz), GPU: nVIDIA GeForce 7900 GTX の環境で計測.

謝辞

dino は米 Cyberware 社，feline はカリフォルニア工科大，armadillo, xyzrgb dragon, lucy, david はスタンフォード大，moai は会津大学によって提供されたモデルデータである．データの利用に関してここに感謝の意を表する．

参考文献

[1] B. Adams, R. Keiser, M. Pauly, L. J. Guibas, M. Gross, and P. Dutré. Efficient raytracing of deforming point-sampled surfaces. *Computer Graphics Forum (Proc. Eurographics 2005)*, 24(3):677–684, 2005.

[2] A. Adamson and M. Alexa. Ray tracing point set surfaces. In *Proc. Shape Modeling International 2003*, pp. 272–282. IEEE CS Press, Los Alamitos CA, 2003.

[3] M. Botsch and L. P. Kobbelt. High-quality point-based rendering on modern GPUs. In *Proc. 11th Pacific Conference on Computer Graphics and Applications*, pp. 335–343, 2003.

[4] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3D objects with radial basis functions. In *Computer Graphics (Proc. SIGGRAPH 2001)*, pp. 67–76. ACM Press, New York, 2001.

[5] E. de Groot and B. Wyvill. Rayskip: Faster ray tracing of implicit surface animations. In *Proc. 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE '05)*, pp. 31–36. ACM Press, New York, 2005.

[6] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. ACM SIGGRAPH / EUROGRAPHICS Workshop on Graphics Hardware*, pp. 9–16. ACM Press, New York, 2001.

[7] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum (Proc. Eurographics 2005)*, 24(3):303–312, 2005.

[8] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proc. IEEE Visualization 2003*, pp. 287–292. IEEE CS Press, Los Alamitos CA, 2003.

[9] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Computer Graphics (Proc. SIGGRAPH '87)*, pp. 163–169. ACM Press, New York, 1987.

[10] S. Muraki. Volumetric shape description of range data using blobby model. In *Computer Graphics (Proc. SIGGRAPH '91)*, pp. 227–235. ACM Press, New York, 1991.

[11] T. Nishita and E. Nakamae. A method for displaying metaballs by using bezier clipping. *Computer Graphics Forum (Proc. Eurographics '94)*, 13(3):271–280, 1994.

[12] Y. Ohtake, A. G. Belyaev, and M. Alexa. Sparse low-degree implicits with applications to high quality rendering, feature extraction, and smoothing. In *Proc. 3rd Eurographics Symposium on Geometry Processing*, pp. 149–158. Eurographics Association, Aire-la-Ville, Switzerland, 2005.

[13] Y. Ohtake, A. G. Belyaev, M. Alexa, G. Turk, and H.-P. Seidel. Multi-level partition of unity implicits. *ACM Transactions on Graphics (Proc. SIGGRAPH 2003)*, 22(3):463–470, 2003.

[14] V. V. Savchenko, A. A. Pasko, O. G. Okunev, and T. L. Kunii. Function representation of solids reconstructed from scattered surface points and contours. *Computer Graphics Forum*, 14(4):181–188, 1995.

[15] E. Tejada, J. P. Gois, L. G. Nonato, A. Castelo, and T. Ertl. Hardware-accelerated extraction and rendering of point set surfaces. In *Proc. Eurographics / IEEE VGTC Symposium on Visualization*. Eurographics Association, Aire-la-Ville, Switzerland, 2006. to appear.

[16] I. Wald and H.-P. Seidel. Interactive ray tracing of point-based models. In *Proc. 2nd Eurographics Symposium on Point-Based Graphics*, pp. 1–8. Eurographics Association, Aire-la-Ville, Switzerland, 2005.

[17] R. Westermann and B. Sevenich. Accelerated volume ray-casting using texture mapping. In *Proc. IEEE Visualization 2001*, pp. 271–278. IEEE CS Press, Los Alamitos CA, 2001.

[18] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.

[19] 西村, 平井, 河合, 河田, 白川, 大村. 分布関数による物体モデリングと画像生成の一手法. 電子通信学会論文誌, J68-D(4):718–725, 1985.