2007－CG－127 (3)
2007／7／9

GPU を用いたメタボールの高速レンダリング

金森 由博† 西田 友是†

メタボールは広く使われている陰関数曲面のひとつであり、曲面は粒子が構成する密度場の等値面として表現される。近年、粒子ベースシミュレーションの結果を多数のメタボールを用いて可視化することがよく行われているが、レンダリングのコストが非常に高いことが問題である。そこで本稿では、GPU を用いたメタボールの高速表示法を提案する。等値面は、ポリゴン化するのではなく、ピクセルごとに陰関数を評価して計算される。この計算において、視線上で等値面に寄与する複数のメタボールを抽出する必要があるが、提案法ではデプステストを反復的に行うことでこれを実現する。既存のレイトレーシング手法で用いられる高速化のためのデータ構造は不要であり、動く多数のメタボールを高速に表示することが可能である。

Fast Rendering of Metaballs on GPUs

YOSHIHIRO KANAMORI† and TOMOYUKI NISHITA†

Metaballs are implicit surfaces widely used to model curved objects, represented by the isosurface of a density field defined by a set of particles. Recently, the results of particle-based simulations have been often visualized using a large number of metaballs, however, such visualizations require high rendering costs. In this paper we propose a fast technique for rendering metaballs on GPUs. Instead of using polygonization, the isosurface is directly evaluated in a per-pixel manner. For such evaluation, all metaballs contributing to the isosurface need to be extracted along each viewing ray. We handle this by iteratively performing depth-tests. Our method doesn't require acceleration data structures often used in existing ray tracing techniques, and can display a large number of moving metaballs quickly.

1. はじめに

陰関数曲面は滑らかな曲面を表現できるため、コンピュータグラフィクスの分野で広く使われてきた。それらのうち、代表的なもののひとつがメタボール[9]と呼ばれる、粒子ベースの陰関数曲面である。最近ではメタボールは、粒子ベースシミュレーションの結果を可視化する際に用いられることが多い。こういったシミュレーションでは数千～数万の粒子が用いられるため、多数のメタボールを高速かつ高品質に可視化する手法が望まれる。

各メタボールは密度分布を持ち、それらが構成する密度場の等値面によって滑らかな曲面が表現される。その主な可視化方法は、ポリゴン化[7),14)]やレイキャスティング[10),16)]である。

等値面をポリゴン化する場合、生成される曲面の品質は空間分割の解像度に依存し、計算コストとトレードオフの関係にある。低解像度であれば高速に曲面を生成できるが、曲面にアーティファクトが発生する。さらに、細かい粒子からなる曲面 (流体のしぶきなど) を発見できない場合がある。逆に高解像度であれば細かい形状まで表現できる反面、計算コストが高く、インタラクティブなアプリケーションには向かない。

一方レイキャスティングを用いる場合、計算量は主

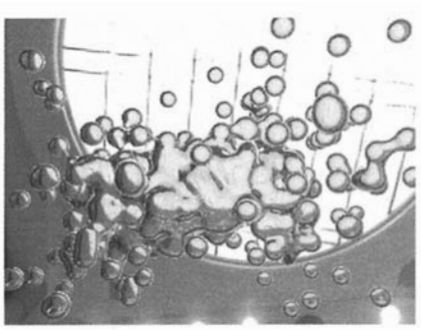


図 1 1,000 個のメタボールが動くアニメーションのスクリーンショット. 描画速度は 480 × 360 で 15.1fps.

† 東京大学大学院情報理工学系研究科コンピュータ科学専攻
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo

にスクリーンの解像度に依存し、細かい粒子であっても高品質で滑らかな曲面を表現できる。しかし、レイキャスティングの計算コストは一般に非常に高い。等値面を表示するために、視線上で等値面に寄与するメタボールを抽出したのち、視線と等値面の交差判定のための方程式を解く必要がある。

本稿では、GPU 上で高速にメタボールのレイキャスティングを行う手法を提案する。提案法では視線上で等値面に寄与するメタボールを抽出するために、depth peeling[3] を用いる。この処理では多数の球を描画する必要があるが、等値面に寄与しない球を除去することで描画コストを減らす。視線と等値面の交差判定のための方程式のソルバとして Bézier Clipping[11] を用いることで、高次の多項式の密度関数で定義されたメタボールでも高速に解を求めることができる。提案法では、多数のメタボールが動くアニメーションを高速に表示できる (図 1)。提案法は粒子ベースの流体シミュレーション結果のプレビューに有効であると考える。

2. 関連研究

この節ではまず、メタボールを定義する密度関数についての関連研究を簡単に紹介する。次に、メタボールをレンダリングする手続きの概要を述べる。そして、近年の GPU による曲面の表示法について関連研究を述べる。

2.1 メタボールの密度関数

メタボール i の密度関数 f_i は、メタボール i の中心点からの距離 r の関数であり、r が増加するに従って単調に減少するようなものが用いられる。メタボール i の有効半径 (f_i のサポート)R_i に対し、$r \geq R_i$ のとき $f_i(r) = 0$ となる。N 個のメタボールによって構成される等値面上の点は次の式を満たす:

$$f(x,y,z) = \sum_{i=0}^{N-1} q_i f_i - T = 0 \qquad (1)$$

ここで T は閾値 (我々は $T = 0.5$ を用いる)、$\{q_i\}$ は密度の係数である。等値面の法線ベクトルは $-\nabla f(x,y,z)$ から得られる。

コンピュータグラフィクスの分野で最初に提案された粒子ベースの陰関数曲面は、Blinn[1] の *blob* である。Blinn は密度関数としてガウス関数を用いた。しかしガウス関数は無限のサポートを持つため、等値面の計算にはすべての blob を考慮する必要があり、計算コストが高い。そこで、有限のサポートを持つ密度関数として、区分的 2 次多項式[9]、4 次多項式[8]、6 次多項式[15] を用いたものが提案された。これらは次数が高いものほど滑らかな曲面が表現できる一方、交差判定のための方程式の求解のコストが増大する。西田らは上記の 4 次式、6 次式の密度関数で定義されたメタボールについて、Bézier Clipping を用いて高速に交差判定を行った[10]。提案法においても上記の 4 次式、6 次式の密度関数を対象とする。

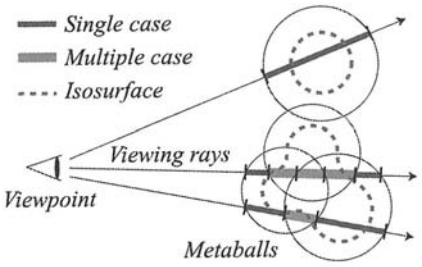


図 3 視線と等値面の交差判定.

2.2 視線と等値面の交差判定

提案法が基礎とする、西田ら[10] の視線と等値面との交差判定処理について説明する。メタボールの密度関数が有限のサポートを持つものとする。有限のサポートを持つ密度関数を用いる場合、視線と各メタボールの交点の間の区間ごとに解くべき方程式が変わるため、区間ごとに計算を行う。まず、視線と各メタボールの交点を計算し、視点に近い順にソートする。そして交点と交点の間の各区間において、視線と交差するメタボールの個数によって異なる処理を行う (図 3):

Single Case (視線と交差するメタボールが 1 つ):
等値面は球面になるため、等値面を持つ球と視線の交差判定を行う。この球の半径は前計算できるので、実行時は高速に処理できる。

Multiple Case (視線と交差するメタボールが複数):
視線と交差するメタボールを抽出し、交差判定のための Bézier 形式 (付録 A.1 参照) の方程式を立式し、Bézier Clipping により求解する。

西田らは上記の処理をスキャンライン法によってピクセルごとに逐次的に処理した。この処理は並列化が可能なので、GPU による高速化が期待できる。GPU を用いるために解決すべき課題は次の 2 点である:

課題 1: 等値面に寄与するメタボールの抽出

課題 2: 交差判定のための方程式の求解

上記の課題 2 に関しては、Pabst らが提案した GPU 上で Bézier Clipping を行う手法[12] が利用できる。

本稿では課題 1 を解決する。すなわち、視線上の各区間において等値面に寄与するメタボールのリストを、GPU 上で保持・更新するアルゴリズムを提案す

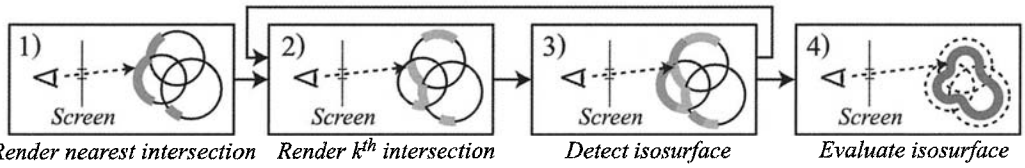


図 2 提案法によるレンダリングの概要．1) メタボールを描画し，視線とメタボールの交点のうち，視点に最も近い交点を求める．2) 視線とメタボールの k 番目 ($1 \leq k < N_{peel}$，N_{peel} はユーザ指定の数) の交点を求める．3) $k-1$ 番目および k 番目の交点の間で，Bézier Clipping を用い，ピクセルごとに視線が等値面と交差するか検査する．2 および 3 を $N_{peel}-1$ 回ループする．等値面が検出されたピクセルについては，ループ内の計算を省略する．4) ピクセルごとに等値面を評価し，シェーディングする．

る。この貢献により、GPU の並列計算能力を活用して等値面のレイキャスティングが可能となる。

2.3 GPU を用いた曲面の表示

曲面の高品質なレンダリングのために、ポリゴン化せず、GPU 上で関数を直接評価する手法が提案されている。NURBS[12]、細分割曲面[17]、*SLIM* 曲面[5] などを表示するものがある。

Loop ら[6] は GPU を用いて Bézier 四面体で定義される曲面のレンダリングを行った。この中で、4 次式で定義されるメタボールのレンダリングも行っているが、多数のメタボールを扱う方法は提示されていない。また、扱える密度関数の次数は 4 次までである。提案法では多数のメタボールを高速に表示することができ、次数は 4 次以上でも可能である。

岩崎ら[4] は、粒子ベースの流体シミュレーションの結果をメタボールを用いて可視化した。彼らは空間をグリッドに分割し、各格子点での密度関数を評価する際、GPU を用いて高速化を行った。彼らの方法ではポリゴン化と同様、小さい粒子からなる曲面を発見できない場合がある。提案法ではピクセルごとにレイキャスティングを行うことにより、そのような曲面も表示できる。

3. アルゴリズム

3.1 概　　要

提案法は視点に最も近い等値面のみを表示する。本稿において、視点から視線とメタボールの交点までの距離を t と表記し、これを視線のパラメータとする。

CPU で計算する場合、各視線上で等値面を検出するために、視線とメタボールのすべての交点を求めてソートする必要がある (2.2 節参照)。しかし GPU で計算する場合、各ピクセルですべての交点を記憶するのは限られたメモリのため難しい。一方、ピクセルによっては等値面が視点に近い数区間内で検出されるため、視点に近い順に交点が得られれば、すべての交点を記憶する必要はない。そこで提案法では、各ピクセルにおいて等値面検出に必要な情報だけを保持する。視線と交差するメタボールの ID および交点を、depth peeling[3] を用いて視点に近い順に取得し、その情報を更新する。そして等値面が検出されたら、そのピクセルの更新を終了する。この処理の概要を図 2 に示す。

以下、等値面検出のための情報をどのように保持・更新するかを述べ、視線と交差するメタボールの ID および交点をどのように取得するかを述べる。そして depth peeling の描画コストを下げる方法を述べる。

3.2 等値面に寄与するメタボールの保持・更新

等値面検出のための情報として、提案法ではメタボールの交点での視線のパラメータ t およびメタボールの ID リストを保持する。この ID リストは、視線のある区間 (2.2 節参照) において等値面に寄与するメタボールの ID のリストである。メタボールは球形なので視線と 2 回交差し、1 回目に交差するときリストに追加し、2 回目に交差するときにリストから削除すればよい。これを踏まえ、ID がひとつ与えられたら、もしリストにその ID がまだなければリストに追加し、すでにあればリストから削除する。

この ID リストは各ピクセルで保持・更新する必要がある。これを効率的に行うために、GPU の N_{buf} 個のバッファへの同時描き込み機能を活用する (N_{buf} の最大値は GPU に依存し、shader model 4.0 に対応した GPU では 8 である)。スクリーンと同サイズの浮動小数点 RGBA フォーマットのテクスチャを N_{buf} 枚用意する。これらをひとまとめにして、RGBA の各チャンネルに 1 つずつ浮動小数点値を記憶し、各ピクセルでサイズが $4N_{buf}$ の配列として扱う。この配列には、視線のパラメータ t、ID の個数、そして $4N_{buf}-2$ 個の ID を記憶する。我々の実験では $N_{buf}=4$、すなわち記憶する ID は最大で 14 個とした。等値面が検出された場合、その印としてパラメータ t の符号を負にする。

```
// ListTex[2] : t および ID リストを保存するテクスチャ
// TmpTex : t およびひとつの ID を保存するテクスチャ
ListTex[0], ListTex[1] および TmpTex を 0 で初期化;
すべてのメタボールの前面を描画 (図 2 の 1);
t および ID を ListTex[0] に保存;
for (k = 1;  k < N_peel;  k++)
  すべてのメタボールの前面と背面を描画 (図 2 の 2);
  if (ListTex[0] の t より t が小さい)
    フラグメントを破棄;
  endif
  t および ID を TmpTex に保存;
  ListTex[0] と TmpTex を入力として
  等値面を検出 (図 2 の 3);
  ListTex[0] と TmpTex を入力として
  更新された ID リストを ListTex[1] に保存;
  ListTex[0] と ListTex[1] を入れ替える;
endfor
ListTex[0] を入力として等値面を評価 (図 2 の 4);
```

図 4 提案法によるレンダリングの擬似コード.

3.3 メタボールの ID および交点の取得

提案法では depth peeling[3] を応用し、視線と交差するメタボールの ID および交点を、視点に近い順に取得する。前節で述べた、視線のパラメータ t およびメタボールの ID リストを保存するテクスチャを入出力用に 2 組用意し、これらのテクスチャにメタボールの ID および交点での視線のパラメータ t を書き込んでいく。図 2 に対応する、提案法によるレンダリングの擬似コードを図 4 に示す。

3.4 Depth peeling の最適化

提案法では depth peeling において、メタボールの有効半径を持つ球の表・裏の両面を何度も描画する必要がある。多数の球を描画する場合、この処理が全体のボトルネックとなる。そこでこの処理を高速化するため、以下の 2 つを検討した:

- 高速な球の描画方法
- 描画する球の個数の削減

まず、描画方法の 1 つの案として、球をポリゴンとして描画する方法が考えられる。しかし、視線と球の交点を正確に求めるために細かい分割が必要であり、1 つの球を描画するために多数の頂点を座標変換しなければならなくなる。特に、球のスクリーンでのサイズが数ピクセル程度の場合に描画効率が悪い。そこで提案法では、球の中心点と半径のみから球を描画する。まず、頂点シェーダに中心点と半径を与え、透視投影された球の描画範囲を、スクリーン上で矩形領域として特定する (付録 A.2 参照)。そして矩形領域内の各フラグメントについて、フラグメントシェーダで視線と球との交差判定を行い、視線が球と交差しなければフラグメントを破棄する。同様の手法は透視投影された円盤の描画[2] や 2 次曲面の描画[13] に用いられている。この手続きにより、特に視点から球が遠い場合、ポリゴンを用いる場合よりも非常に高速に球を描画できる。なお、球の前面と背面の両方を描画するためには 2 回に分けて描画を行うことになる。

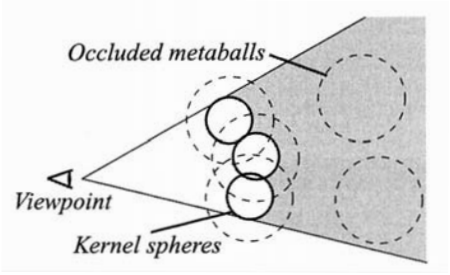


図 5 カーネル球を用いたメタボールのカリング.

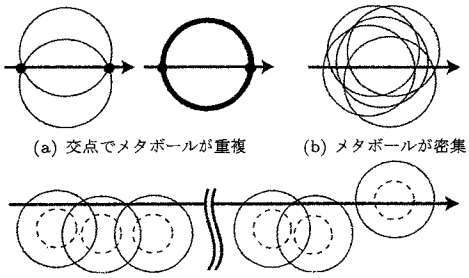
次に、描画するメタボールの個数の削減について述べる。2.2 節の Single Case で述べた、等値面をなす球を「**カーネル球**」と呼ぶことにする。視点から見て、カーネル球に隠れるメタボールは等値面に寄与しない、ということに注目し、カリングを行う (図 5)。この処理には GPU の遮蔽問い合わせ機能 (*occlusion query*) を用いる。Occlusion query を用いる場合は物体を描画する必要があるため、より描画コストの低いビルボードを用いる。まず、視点から見てカーネル球の後側にビルボードを配置して描画し、視線のパラメータ t を保存する。続いて occlusion query を用いてメタボールの手前に配置したビルボードがスクリーンに描き込まれるかを調べる。このときスクリーンに描画されないメタボールをカリングすれば、depth peeling で描画するメタボールを削減できる。このカリング処理はメタボールが密集している場合に特に有効であり、描画すべきメタボールの個数を 10〜20%程度まで減らすことができる。

3.5 特殊なケース

ここまで述べたアルゴリズムで検討されていない状況 (図 6) について補足する。

視線との交点において複数のメタボールが重なる場合 (図 6a)、視線のパラメータ t を比較しただけでは 1 つのメタボールの ID しか得られない。この場合は、ID がリストに保存されていないことを確認し、すでにリストに保存されている場合はその ID を含むフラグメントを破棄する。

メタボールが密集している場合 (図 6b)、ID リストの個数の上限 $K = 4N_{buf} - 2$ に達する可能性がある。すると交差判定ができず、穴が生じてしまう。このような場合は、その時点の t で交点が発見されたことに



(a) 交点でメタボールが重複　(b) メタボールが密集

(c) 視線が何度もメタボールを通過

図 6 特殊なケース.

表 1 ループ回数 N_{peel} による描画時間の違い. 単位はミリ秒.

次数	個数	N_{peel}			
		14	20	30	40
4	1,000	31.6	40.0	52.9	64.9
	2,000	37.9	48.3	64.5	80.6
	4,000	60.6	77.5	103.1	128.2
6	1,000	51.5	63.7	79.4	94.3
	2,000	57.5	71.9	90.9	109.9
	4,000	90.1	114.9	147.1	175.4

する。実験では $K = 14$ で視覚的不具合は見られなかった。

視線がメタボールを何度も通過したのち、等値面と交差する場合 (図 6c) がある。このような場合は、ID リストの個数の上限 K を超えない限り、depth peeling のループの回数 N_{peel} を十分大きくすれば等値面を発見できる。

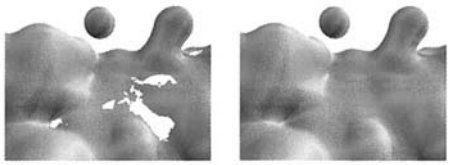
4. 実験結果

実装には C++、OpenGL、Cg を用いた。実験は Intel Core 2 Quad 2.66 GHz、NVIDIA GeForce 8800 Ultra 搭載の PC で行った。以下の実験では特に断りがない限り、解像度は 480×360 である。

Depth peeling と交差判定のループ回数 N_{peel} を、メタボールの数を変えながら変化させた場合の描画時間の比較を表 1 に示す。N_{peel} が小さ過ぎる場合、等値面が発見できず穴が生じる場合がある (図 7)。しかしながら経験的には、$N_{peel} = 30$ 程度でほとんどの場合に穴が生じないことがわかった。

提案法の精度を調べるために、図 1 の例に関して CPU で計算したリファレンスと GPU で計算した結果を比較した。各ピクセルにおける視線とメタボールの交点について、リファレンスではすべてを計算したのに対し、GPU では 30 個で打ち切った。リファレンスに対する、等値面との交点における視線のパラメータの RMS 誤差は 1.1%であった。

CPU で計算した場合と GPU で計算した場合とで、



(a) $N_{peel} = 14$　(b) $N_{peel} = 30$

図 7 N_{peel} が小さい場合に生じる穴.

表 2 CPU と GPU の描画時間の比較. 単位はミリ秒.

	個数	解像度		
		256^2	512^2	$1,024^2$
CPU	1,000	400.0	1888.0	8439.6
	2,000	628.5	2744.9	12597.6
	4,000	267.3	1270.5	6015.7
GPU	1,000	36.0	117.6	400.0
	2,000	39.2	126.6	454.5
	4,000	36.2	109.9	357.1

描画時間の比較した結果を表 2 に示す。CPU では depth peeling を行う代わりに、各ピクセルに書き込まれるパラメータ t およびメタボールの ID を配列に記憶し、後から t でソートした。密度関数の次数は 6 とし、GPU でのループの反復回数 N_{peel} は 30 とした。CPU 実装においてマルチスレッド化や SIMD 命令の利用は行っていない。CPU に比べ、GPU で計算した場合は解像度が高いほど高速化の割合が大きく、512^2 以上ではおよそ 11〜27 倍高速である。

5. まとめと今後の課題

我々は GPU 上で高速にメタボールのレイキャスティングを行った。Depth peeling を用いて視線上で等値面に寄与するメタボールを抽出し、そのリストを管理するアルゴリズムを提案した。視線と等値面の交差判定には Bézier Clipping を用いることで、高次多項式の密度関数で定義されたメタボールについても、高速に等値面を求めることができた。

今後の課題として、GPU の機能を活用してさらなる高速化を図りたい。また、複雑な屈折や透明物体を考慮するために、視線に一番近い等値面だけでなく、それ以降の等値面についても効率的に計算する手法を開発したい。

参考文献

1) J. F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1(3):235–256, 1982.
2) M. Botsch, M. Spernat, and L. Kobbelt. Phong splatting. In *Eurographics Symposium on Point-Based Graphics 2004*, pp. 25–32, 2004.
3) C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 2002.

http://developer.nvidia.com.
4) K. Iwasaki, Y. Dobashi, F. Yoshimoto, and T. Nishita. Real-time rendering of point-based water surfaces. In *Proc. of Computer Graphics International 2006*, pp. 102–114, 2006.
5) T. Kanai, Y. Ohtake, H. Kawata, and K. Kase. GPU-based rendering of sparse low-degree implicit surfaces. In *Proc. of GRAPHITE '06*, pp. 165–171, 2006.
6) C. Loop and J. Blinn. Real-time GPU rendering of piecewise algebraic surfaces. In *Proc. of SIGGRAPH '06*, pp. 664–670, 2006.
7) W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proc. of SIGGRAPH '87*, pp. 163–169, 1987.
8) S.Murakami and H.Ichihara. On a 3D display method by metaball technique. *Journal of papers given by at the Electronics Communication (in Japanese)*, J70-D(8):1607–1615, 1987.
9) H. Nishimura, M. Hirai, T. Kawai, I. Shirakawa, and K. Omura. Object modeling by distribution function and a method of image generation. *Journal of papers given by at the Electronics Communication Conference (in Japanese)*, J68-D(4):718–725, 1985.
10) T. Nishita and E. Nakamae. A method for displaying metaballs by using Bézier Clipping. *Computer Graphics Forum*, 13(3):271–280, 1994.
11) T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. In *Proc. of SIGGRAPH '90*, pp. 337–345, 1990.
12) H.-F. Pabst, J. P. Springer, A. Schollmeyer, R.Lenhardt, C.Lessig, and B.Froehlich. Ray casting of trimmed NURBS surfaces on the GPU. In *IEEE Symposium on Interactive Ray Tracing 2006*, pp. 151–160, 2006.
13) C. Sigg, T. Weyrich, M. Botsch, and M. Gross. GPU-based ray-casting of quadratic surfaces. In *Eurographics Symposium on Point-Based Graphics 2006*, pp. 59–65, 2006.
14) Y. Uralsky. Dx10: Practical metaballs and implicit surfaces. Technical report, NVIDIA Corporation, 2006. http://developer.nvidia.com.
15) B. Wyvill, C. McPheeters, and G. Wyvill. Data structure soft objects. *The Visual Computer*, 2(4):227–234, 1986.
16) B. Wyvill and A. Trotman. Ray-tracing soft objects. In *CG International '90*, pp. 469–476, 1990.
17) Y. Yasui and T. Kanai. Surface quality assessment of subdivision surfaces on programmable graphics hardware. In *SMI '04: Proceedings of the Shape Modeling International 2004 (SMI'04)*, pp. 129–138, 2004.

付　　録

A.1 各次数の密度関数の Bézier 形式について

西田ら[10] の方法において、密度関数を Bézier 形式に変換する必要がある。

視線と 1 つのメタボールの交点の間の区間の、視線のパラメータを $s \in [0,1]$ とする。M 次の密度関数 $f(s)$ を

$$f(s) = \sum_{m=0}^{M} d_m B_m^M(s) \tag{2}$$

と表現したときの、Bernstein 係数列 $\{d_m\}$ が必要である。ここで B は Bernstein 多項式で $B_m^M(s) = \binom{M}{m} s^m (1-s)^{M-m}$ である。提案法で用いる 4 次、6 次の係数列は次の通りである:

(1)　4 次式の係数

$$d_0 = d_1 = d_3 = d_4 = 0, \quad d_2 = \frac{8}{3}a_i^2 \tag{3}$$

(2)　6 次式の係数

$$d_0 = d_1 = d_5 = d_6 = 0 \tag{4}$$

$$d_2 = d_4 = \frac{16}{27}a_i^2, \quad d_3 = \frac{8(8a_i+5)a_i^2}{45} \tag{5}$$

ここで $a_i = \frac{D_i}{R_i}$ である。D_i は $2\sqrt{D_i}$ が区間長に等しくなるような値 (交差判定のための 2 次方程式の判別式) であり、R_i はメタボール i の有効半径である。

視線のある区間において N 個のメタボールが交差する場合、視線と等値面の交差判定のために次の方程式を用いる:

$$f(s') = \sum_{i=0}^{N-1} q_i \left(\sum_{m=0}^{M} d_m'^i B_m^M(s') \right) - T = 0, \tag{6}$$

ここで $s' \in [0,1]$ はその区間のパラメータ、$\{q_i\}$ は密度の係数、$\{d_m'^i\}$ はその区間におけるメタボール i の Bernstein 係数、T は閾値である。式 (6) は Bézier Clipping を用いて高速に解くことができる。

A.2 透視投影された球の描画範囲

視点座標系において、中心点が (x_i, y_i, z_i)、半径が R_i の球 i があるとする。この球 i のスクリーン上での x 方向および y 方向の描画範囲は次式から計算できる:

$$x = \frac{z_0(x_i z_i \pm R_i\sqrt{x_i^2 + z_i^2 - R_i^2})}{z_i^2 - R_i^2} \tag{7}$$

$$y = \frac{z_0(y_i z_i \pm R_i\sqrt{y_i^2 + z_i^2 - R_i^2})}{z_i^2 - R_i^2} \tag{8}$$

ここで z_0 は視点からスクリーンまでの距離である。