グラフィックスハードウェアによるボロノイ図の描画とその周辺

Fast Drawing of Voronoi Diagrams Using Graphics Hardware and Related Topics

山本修身（名城大学・理工学部）

Osami YAMAMOTO (Meijo University)

キーワード： ボロノイ図，グラフィックスハードウェア
Keywords: Voronoi diagrams, graphics hardware

1. はじめに

グラフィックスハードウェアは３次元図形を高速にコンピュータのディスプレイ上に描画するための専用ハードウェアである．近年では高級なワークステーションだけでなく，一般用途の多くのパーソナルコンピュータに，このようなグラフィックスハードウェアが搭載されている．グラフィックスハードウェアの性能向上は近年めざましいものがあり，現在までのところ 12ヶ月で速度が 2 倍になっている．これは中央処理装置（CPU）の性能向上の速度よりも速い．また，近年では本来の 3 次元図形を描くための計算能力に加えて，ある種のプログラムをグラフィックスハードウェア内部に与えることによって，特殊な陰影付けなど，より複雑な計算を行わせる機能を備えている [2, 8]．この機能は，グラフィックスハードウェア内部のプログラミングをある程度可能にするものであり，グラフィックスハードウェアを単に普通の画像の描画に利用するだけではなく，より柔軟に色々な計算に利用する可能性を与えている．

ボロノイ図は計算幾何学における基本的な手法の 1 つであり多くの応用範囲を持つ．問題としている空間に幾つかの母点と呼ばれる点が与えられたとき,「どの母点に一番近いか」という基準によって，その空間を分割したものがボロノイ図であり，オペレーションズリサーチをはじめ種々の分野において幾何的あるいは地理的な問題を解くための強力な道具となっている．各種のボロノイ図の計算アルゴリズムについては計算幾何学の発展と共に多くの研究が行われてきている [3, 6, 7, 9]．

本稿ではグラフィックスハードウェアによる各種ボロノイ図の描画について概観し，さらにボロノイ図の双対図形であるドローネ三角形分割や 3 次元凸包の計算に言及する．また，最後の部分ではボロノイ図の描画効率について考察する．

2. グラフィックスハードウェアによるボロノイ図の原理

グラフィックスハードウェア (GPU) は 3 次元空間中の図形を画面に表示するためのハードウェアである．グラフィックスハードウェアを用いてボロノイ図を描画するためには，与えられた母点に対応する 3 次元図形を画面上に描画すれば良い．たとえば，ユークリッド平面上の n 個の母点 p_i $(i = 1, \ldots, n)$ のボロノイ図は方程式

$$z = \|x - p_i\|_2 \ (i = 1, \ldots, n) \qquad (1)$$

で表現される n 枚の円錐面をそれぞれ母点固有の色で画面上に描画し，z 軸方向から眺めれば良い (図 1 参照) [4]．3 次元グラフィックスライブラリ OpenGL [5, 10] を用いて，この方法によって得られたランダムな 100 点のボロノイ図を図 2 に示す．このようにして得られるボロノイ図と通常の方法によって得られるボロノイ図は本質的に異なるものであり，ここで求めているものはあくまでボロノイ図のイメージである．さらに，このイメージは連続的なものではなく，画面上のピクセル位置における色情報によって表現されたボロノイ図の概形に他ならない．一方，通常の計算によって得られるボロノイ図は，それぞれの母点に対応する領域の形を頂点の座標として表現したものである．また，これは「画面の大きさ」や「ピクセルの細かさ」などの制約を受けない．

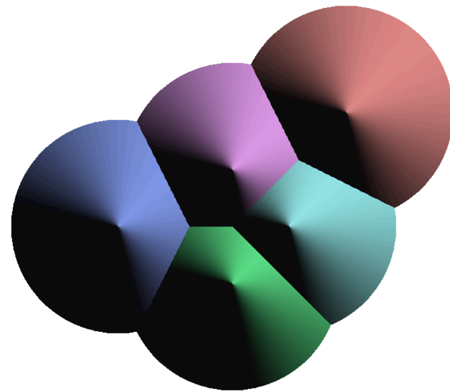

図 1: 頂点が同一平面上に置かれるように並べた円錐面を下から見た様子．それぞれの円錐面を固有の色で着色するとユークリッド平面上のボロノイ図となる．

前者の方法のメリットとして，種々のボロノイ図に対してほぼ同じ原理で描画することができ，また，描画のため

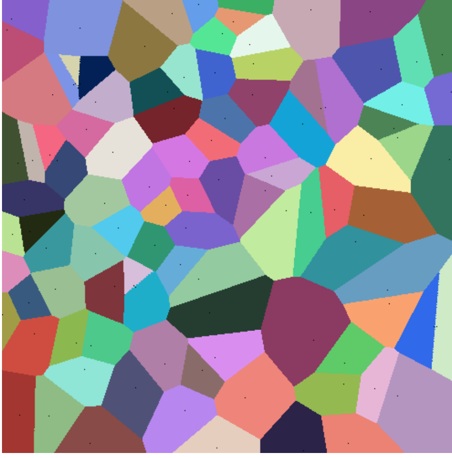

図 2: ユークリッド距離に関するランダムな 100 点のボロノイ図.

図 3: マンハッタン距離に関するランダムな 100 点のボロノイ図.

のプログラムも通常の方法とは比べ物にならないくらい単純であるということが挙げられる．ユークリッド平面上での通常のボロノイ図の描画については，多くの高速アルゴリズムやプログラムが知られているが，そうではないボロノイ図をこれほど手軽に描くことは難しい．

前述のユークリッド平面上のボロノイ図は，実はより簡単に描くことができる．まず，前述の円錐面の代わりにつぎの放物面を描くことを考える：

$$z = \|x - p_i\|_2^2. \ (i = 1, \ldots, n) \tag{2}$$

一般にグラフィックスハードウェアでは，3 次元空間中の複雑な曲面を描くことは難しいが，この場合には，3 次元空間のすべての点に変換 $\varphi : (x, y, z) \in R^3 \mapsto (x, y, z - x^2 - y^2) \in R^3$ を適用することによって，すべての放物面が平面

$$z = \|p_i\|_2^2 - 2(p_i, x). \ (i = 1, \ldots, n) \tag{3}$$

に変換される．また，この変換は x 座標と y 座標を変えず，z 座標の大小関係を変えない．したがって，この変換によって得られる画面上のイメージは円錐面によって得られたものと変わらない．方程式 (3) は平面を表すので，結果として n 個の母点に対応する平面を描画することによりボロノイ図が得られる．この変換によって描画速度を数倍高速化することができる [11]．しかし，それぞれのピクセルの深さを表現するバッファ（z バッファ）の精度が低いと，描画されるイメージが数値誤差により非常に不安定になることが知られており，ある程度高精度な z バッファを用いる必要がある [11]．

同じ平面上のボロノイ図であっても用いる距離を変えた場合，描画すべき要素の形状が変化する．たとえば，x 座標の差の絶対値と y 座標の差の絶対値の和を距離として定義するマンハッタン距離 $d(p_1, p_2) = |p_{1,x} - p_{2,x}| + |p_{1,y} - p_{2,y}|$ によるボロノイ図を描くのであれば，断面が正方形になる同じ形状の四角錐を描画すれば良い．ただし，$p_1 = (p_{1,x}, p_{1,y})$，$p_2 = (p_{2,x}, p_{2,y})$ とする． すなわち，ある点 p からの距離が z であるような点 $C(p, z) = \{q \in R^2 \mid d(p, q) = z\}$ が深さ z に配置されるような図形を描画すれば良い．このような図形は四角錐面となる．この図形を要素として用いれば，マンハッタン距離に関するボロノイ図を描くことができる．図 3 にランダムな 100 点に関するボロノイ図を示す．

グラフィックスハードウェアを用いれば，それぞれの母点に重みのついたボロノイ図についても簡単に描画することができる．このようなボロノイ図の描画では，それぞれの母点に対応した図形の形状を変化させればよい．ある母点 x を中心として，そこから任意の点 y までの距離を $d(x, y) = a\|x - y\| + b$ のように測った場合，前述のユークリッド距離に関するボロノイ図については，円錐面の開き具合と深さ方向の位置をそれぞれ a, b に対応させて変えれば良い．また，このようなボロノイ図と多少異なった構造を持つものにラゲールボロノイ図がある．ラゲールボロノイ図は距離を $d(x, y) = \|x - y\|^2 - r^2$ と定義したものである．r はそれぞれの母点に固有の量である．このようなボロノイ図の領域の境界は直線であることが知られている．ラゲールボロノイ図を作るためには，前述の放物面を基に平面によって描かれるボロノイ図のアルゴリズムを変形し，n 枚の平面

$$z = \|p_i\|_2^2 - 2(p_i, x) - r_i^2 \ (i = 1, \ldots, n) \tag{4}$$

を描画すれば良い [11]．ランダムな 100 点に関するラゲールボロノイ図を図 4 に示す．

つぎに，線分のボロノイ図について考えてみる．線分のボロノイ図は点と点との距離を線分と点との距離に置き換えたものである．線分は 2 つの平面上の 2 つの点 p_1 と p_2 を用いて $b = \{\lambda_1 p_1 + \lambda_2 p_2 \in R^2 \mid \lambda_1 + \lambda_2 = 1, \ 0 \le \lambda_1, \lambda_2 \le 1\}$ と定義できる．このとき，この線分と任意の点 p との距離はつぎのように表現される：

$$d(b, p) = \begin{cases} \|p_1 - p\| & t \le 0, \\ \varphi(p_1, p_2, p) & 0 < t < 1, \\ \|p_2 - p\| & 1 \le t. \end{cases} \tag{5}$$

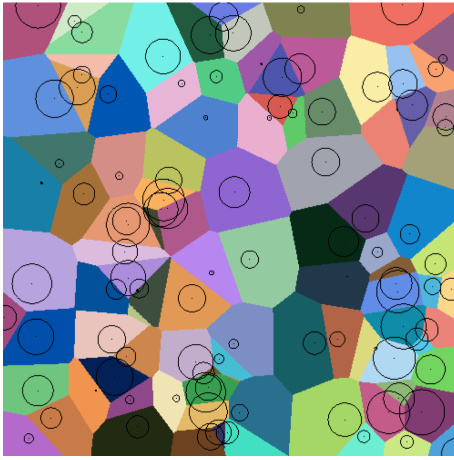

図 4: 平面上のランダムな 100 点のラゲールボロノイ図.

ただし，$\varphi(p_1,p_2,p) = (\|p_1-p\|^2\|p_1-p_2\|^2-((p_1-p_2)\cdot(p_1-p))^2)^{1/2}/\|p_1-p_2\|$ であり，$t=(p_1-p)\cdot(p_1-p_2)/\|p_1-p_2\|^2$ とする．この式から， 平面上の $t \leq 0$ の部分と $1 \leq t$ の部分には，通常のボロノイ図を描くための円錐面要素の一部を用いることができる．これに対して，$0 < t < 1$ の部分は平面の一部である．これらの図形をそれぞれの線分に対応した色で描画すれば，線分のボロノイ図を描くことができる．

また，平面上のボロノイ図以外に球面上のボロノイ図を考えることもできる．この場合，球面上の 2 点間の距離は 2 点間の測地線の距離ということになる．球面の半径を 1 として，球面上の 2 点の中心からの位置ベクトルを p_1 および p_2 とおくと，2 点の距離はこの 2 つのベクトルの成す角に等しいことから，

$$d(p_1,p_2) = \cos^{-1}(p_1,p_2) \tag{6}$$

と表すことができる．ただし，(p_1,p_2) は p_1 と p_2 の内積を表す．しかし，ここでボロノイ図を描くために必要なのは大小の比較である．すなわち，与えられた各母点のうちどの母点に一番近いかを見つけ出すことができれば良い．したがって，より単純に，

$$d(p_1,p_2) = 1-(p_1,p_2) \tag{7}$$

と定義すれば十分である．この定義に基づき，球面上のある点を $p_0=(p_{0,x},p_{0,y},p_{0,z})$ とおけば，通常のボロノイ図を描くときに用いる円錐面に対応する曲面 P_{p_0} は，

$$S_{p_0} = (0,0,1) - S_2'A \tag{8}$$

と表すことができる [11]．ただし，S_2 は原点を中心とする単位球面のうち z 座標が非正の部分であり，A はつぎのような 3×3 の行列である：

$$A = \begin{pmatrix} -1 & 0 & p_{0,x} \\ 0 & -1 & p_{0,y} \\ 0 & 0 & p_{0,z} \end{pmatrix}. \tag{9}$$

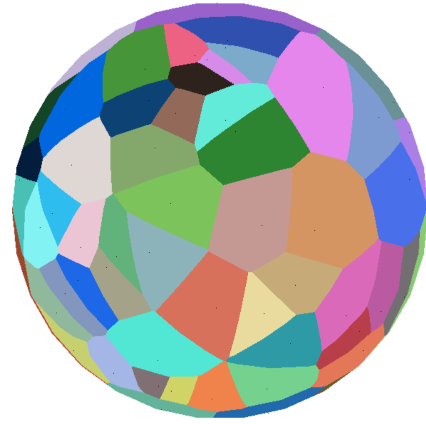

図 5: 球面にランダムに分布した 80 母点による球面ボロノイ図.

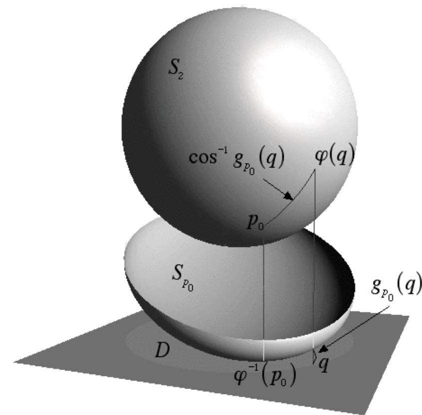

図 6: 球面ボロノイ図を描画するための要素.

この曲面は球面を一次変換した楕円球の一部である (図 6 参照)．この曲面を多角形近似した曲面を描画することにより得られる球面ボロノイ図を図 5 に示す．

本節ではグラフィックスハードウェアによる各種ボロノイ図の描画について述べた．この方法の利点は，色々なボロノイ図をほぼ同じ考え方の下で構成でき，原理の単純さからボロノイ図を手軽に描画できることである．効率については本稿の後の部分で述べるが，用いるハードウェアに依存し，いつでも高速に描画できるとは限らない．特に，これまで計算アルゴリズムが非常に良く研究されてきているユークリッド平面上のボロノイ図などについては，通常の方法で計算する方が効率が良いと考えられる．

3. 3 次元凸包とドローネ三角形分割の計算について

グラフィックスハードウェアは本来画像を得るための道具であって，それ以外の用途に用いるために設計されていない．しかし，描画した画像をメインメモリに転送することは可能であり，この機能を用いることにより，描画した画像から情報を得ることができる．

ドローネ三角形分割は 2 次元のボロノイ図の双対図形

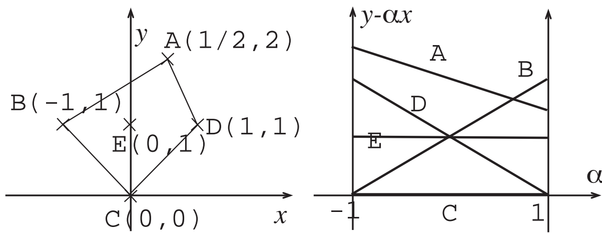

図 7: グラフィックスハードウェアによる 2 次元凸包の計算の原理.

図 8: ある円盤の中にランダムに分布した 10 万点の凸包を計算するためのイメージ.

として定義することができる．すなわち，与えられた母点のボロノイ図中で互いに隣り合う領域を作る 2 つの母点どうしをすべて結んだ平面グラフがドローネ三角形分割である．グラフィックスハードウェアを用いてボロノイ図のイメージを得る場合には，いくつかの問題点が存在する．まず，ボロノイ図のイメージを得る場合には，無限に広い平面全体のイメージを計算することは不可能であるため，ある限られた矩形内のイメージを得ているに過ぎない．従って，この矩形内のイメージの情報からすべての隣接関係を見つけ出すことは原理的に不可能である．また，ボロノイ図のイメージはピクセルによって表現された離散的なものであり，連続的な情報を得ることができない．さらに，ピクセルの個数は膨大であり，この 1 つずつをトレースすると，それだけで計算量が爆発することになる．これらの問題のいくつかは解決可能であり，また，残りのいくつかはある程度の制限の下で解決することができる．

たとえば，隣接関係の問題は，ドローネ三角形分割の問題を 3 次元凸包の問題に置き換えることにより解決する．ここで述べてきたボロノイ図の描画の手法と本質的に同じであるが，2 次元および 3 次元凸包はグラフィックスハードウェアを用いて計算することができる．ここでは 2 次元の凸包についての [11] の計算法の概略について述べる．

平面上の n 点の 2 次元凸包とは，この n 点を含む最小の凸多角形のことである．この多角形を計算するということは，この多角形を構成する頂点を順に列挙することと同値である．ここで，与えられた点を通って傾き α の直線を考える．これらの直線のうちの 2 本は凸包の接線となり，その y 軸との切片は，これらの直線の中でそれぞれ最小と最大になる．傾き α をある範囲で動かして考えれば，切片はそれぞれ α の 1 次式として表現される．たとえば，図 7 の左図では 5 つの点が平面上に置かれていて，凸包を構成する点は A, B, C, D の 4 点である．ここで，傾き α $(-1 \le \alpha \le 1)$ の直線を考える．それぞれの点を通過する直線の y 軸との切片は α の一次式として表現され，α を横軸にとったグラフは図 7 の右図のようになる．この場合，それぞれの α について，値が最大および最小のものが実際に凸包と接する直線と対応する．したがって，このような直線を実際に点固有の色で 3 次元空間にそれぞれ描画して上方向または下方向から眺めた画像を構成すれば，その方向の凸包を構成する頂点が見える．また，画像のなかで，ある色から別の色に変化する境界は凸包上の辺に対応することがわかる．ある円盤中に一様に分布した 10 万点の 2 次元凸包を計算する際のイメージを図 8 に示す．このイメージは 360 度を 90 度ずつ 4 つの方向に分け，それぞれの方向について計算しているので，4 つの帯から成っている．このイメージで隣り合う色を列挙すれば，2 次元凸包を計算したことになる．

2 次元凸包を計算するこの方法は 3 次元凸包にそのまま拡張することができる．この場合，2 次元凸包の計算での 1 次元的な帯は 3 次元凸包ではボロノイ図のような 2 次元的な模様となる．ドローネ三角形分割を計算するためには，平面上に与えられた母点 (x, y) を 3 次元空間の点 $(x, y, x^2 + y^2)$ へ射影し（放物面上への射影になっている），これらの点の 3 次元凸包を計算して，凸包を構成する辺を 2 次元平面へ射影すれば，それがドローネ三角形分割になっていることが知られている．したがって，3 次元凸包が計算できれば，それを用いてドローネ三角形分割を計算することは容易である．この方法を用いれば，ボロノイ図のイメージを直接調べるのとは異なり，すべての隣接関係が得られる．

次に，2 次元凸包の場合に膨大なピクセルから色の境界を見つけ出す方法について述べる．この方法は 3 次元凸包の場合にも拡張することができる．2 次元の場合，直線状にピクセルが並んでいるので，左から順にスキャンすれば境界を見つけることができる．しかし，この場合，ピクセル数に比例する計算量が必要となり，高精度に問題を解こうとした場合には計算量が爆発する．この場合，それぞれの色の領域は連結している．これは，3 次元空間中に直線を描画してある方向から見ていることから明らかである．この性質を用いることによって，以下の再帰的アルゴリズムによって，$O(m \log M)$ ですべての境界を列挙することができる [12]．ただし，m は凸包上の点の数であり，M はピクセル数である．

アルゴリズム 1 ([12]) 区間 $[x, y]$ $(0 \le x < y \le 1)$ の中のすべての色の境界を列挙する関数 $E(x, y)$.

1 **function** $E(x,y)$
2 **begin**
3 **if** $c(\varphi(x)) = c(\varphi(y))$ **then return** $\emptyset$
4 **else if** $\varphi(x) + 1 = \varphi(y)$ **then**
5 **return** $\{(c(\varphi(x)), c(\varphi(y)))\}$
6 **else return** $E\left(x, \frac{x+y}{2}\right) \bigcup E\left(\frac{x+y}{2}, y\right)$
7 **end**

狭い領域に境界が多数集まることによって，本来は凸包の一部なのにイメージ上に出現しない色が出てくる可能性がある．ここでは，出現しない色が発生しないくらい細かく描画が行われると仮定する．この仮定の下では常に正しい凸包が出力される．取り出した画像情報を利用する場合にも，常に同様の問題が存在し，ある程度の細かさで画像を生成しない限り正しい情報を得ることができない．３次元凸包問題に関しても同様の問題が起こる．３次元凸包の場合には，得られるパターンの中で３つの領域が隣り合う点を数え上げることが，３次元凸包の表面三角形を数え上げることと同値である．実は３次元凸包の場合には，問題はさらに深く，いくら細かく見ても同じ領域の色が飛んだように見えるケースが存在する．飛んだ領域については，ある程度広い範囲で領域を見ることができれば数え上げることができるが，現実には画面上の描画できる領域の広さは限られており，正しい計算結果が得られない可能性がある．しかし，ボロノイ点でそれほど多くない個数の領域（３領域でなくても良い）が接するケースでは，ある程度の精度でイメージを描画すれば問題なく３次元凸包が計算される．

4. 描画の効率について

ユークリッド平面上のボロノイ図を本稿で述べる方法で計算した場合，プログラムを作る手間は普通のアルゴリズムに比べて比較にならないほど小さい．また，どのように計算すればよいかはっきりとしないボロノイ図でも定義に従って描画することができる．一方，効率については，必ずしも高いとはいえない．現在のグラフィックスハードウェアの描画速度は非常に高速であり，その内部では部分的に並列に動作している．それにもかかわらず，大規模なボロノイ図をここで述べるような方法でそのまま描画する場合，速いグラフィックスハードウェアを用いてもそれほど高速に描画できるわけではない．

図９にグラフィックスハードウェアの大雑把なブロック図を示す．図中で頂点プロセッサ (vertex processor) は，座標変換やクリッピングなど与えられた頂点情報に関する処理を行う部分である．CPU で計算された図形の情報は，まず，このプロセッサで処理されたあと，ラスタライザ (rasterizer) によってピクセル情報に変換される．このピクセル情報 (それぞれのピクセルの色情報，深さ情報，テクスチャマッピングの座標などが含まれる）に対し，フラグメントプロセッサ (fragrment processor) によって隠面処理や光源やテクスチャーに関する計算などが行われ，実際に画面に表示される．

グラフィックスハードウェアを用いてボロノイ図を描画する場合，それぞれのピクセルに関する隠面処理の計算が本質であって，この部分の機能が集中的に使われる．もちろん，頂点プロセッサによって，与えられた多角形の画面座標への変換などを計算する必要はあるが，フラグメントプロセッサに比べれば，負荷はそれほど重くない．

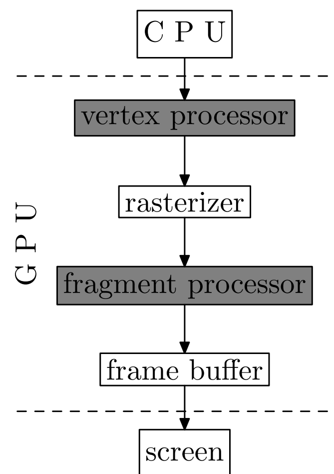

図 9: グラフィックスハードウェアを構成する主なプロセッサ.

大規模なボロノイ図をグラフィックスハードウェアで描く場合，フラグメントプロセッサに対する負荷は大変なものである．たとえば，1000 母点のボロノイ図を 600×600 のウィンドウに描画することを考えよう．それぞれの母点について，$360,000$ 点のピクセルのすべてについて，それまでに図形が描かれることによって変化した深さと，それぞれのピクセルに対応する深さバッファの値との比較を行わなければならない．したがって，合計で 3.6×10^8 回の比較演算が行われることになる．もちろんフラグメントプロセッサはその内部にある複数のプロセッサによって並列に比較を行うが，その並列度には限界がある．したがって，現実にはボロノイ図はそれほど高速に描画できないことになる．

ユークリッド平面上の通常のボロノイ図を考えたときには，この状況を改善することができる．実際，描画面のある部分を眺めたとき，ほとんどの面は一番上の幾つかの面によって隠されてしまう．したがって，フラグメントプロセッサによって扱われる情報のほとんどは描画に寄与していない．もちろんどの面が表面に出てくるか否かは，実際に比較してみなければわからないということになるし，どの面もいずれかの位置にその色が出現するので，ある母点について描画しないわけにはいかない．しかし，描画する場所に制限を加えた場合には状況が変わる．たとえば，山本 [13] は，ボロノイの描画面をいくつかの小さな領域に分けて，それぞれの領域で出現する色をあらかじめ調べることによって，ボロノイ図の描画が加速できることを示した．これは，図 9 のブロック図で見ると，描画の負荷をフラグメントプロセッサから頂点プロセッサや CPU に移していることに他ならない．この方法によって，描画速度が 10 倍から 20 倍になることもある．この方法の性能は使用するグラフィックスハードウェアの構成や内部プロセッサの速度比などによって色々に変化する．また，この手法は通常のボロノイ図ばかりではなく，色々なボロノイ図に適

用できると考えられるが，これは今後の課題である．

5. おわりに

本稿ではグラフィックスハードウェアを用いた各種ボロノイ図の描画方法およびその周辺の話題について述べた．

グラフィックスハードウェアは CPU とは別のハードウェアであり，画像を描画するためのいくつかの特殊な演算に関しては非常に高速に動作する．その内部では，いくつかのプロセッサがパイプライン状に連結され，並列に動作する．また，画面のピクセルに対応する色，深さ，およびその他の情報を蓄えるためのメモリを持っている．これは，ソフトウェアの立場からみれば，巨大な２元配列とその配列を高速に操作するための専用ハードウェアがセットになったものと考えることができる．それぞれの操作は高速に動作するが，その機能を目的とする計算にうまく合わせることは，それぞれの演算が図形の描画に特化されていることからそれほど容易ではない．現在では，グラフィックスハードウェア内部の頂点プロセッサやフラグメントプロセッサなどについては，特殊なプログラムを与えることによってその動作を変更させることが可能になっている [2, 8]．単にボロノイ図を描画するためには，現在のところ，このような機能を用いる必要はないが，より高度な計算をさせるために，今後検討して行く必要があると考えられる．

ボロノイ図を画面に描画することだけが目的であれば，適当な図形を描画すれば良い．その画像データを別の計算に利用するためには，得られたデータの質に関する保証が必要となる．狭い場所に多くの母点やボロノイ点が集中したようなボロノイ図のイメージは，利用法にもよるが，情報としては不完全である．通常のアルゴリズムでは，退化の発生問題などについて，記号摂動法 [1, 14] などを利用することにより解決することができるが，グラフィックスハードウェアで画像を描画する場合には，退化が起こらない代わりに，ピクセルを見たときに前述のような「飛んだ点」が発生する可能性がある．この問題の現実的な解決は，今後の課題である．

参考文献

[1] Edelsbrunner, H., and Mucke, E. 1988. Simulation of simplicity – a technique to cope with degenerate cases in geometric algorithms. In *Proceedings of the 4th ACM Annual Symposium on Computational Geometry*, ACM, 118–133

[2] Fernando, R. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* Addison-Wesley. (2004)

[3] Fortune, S.: A sweepline algorithm for Voronoi diagrams. *Algorithmica* **2** , (1987) 153–174

[4] Hoff, K. E., Culver, T., Keyser, J., Lin, M., and Manocha, D.: Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of ACM SIGGRAPH Annual Conference on Computer Graphics*, ACM, (1999) 277–286

[5] Kilgard, M. J.: *OpenGL Programming for the X Window System.* Addison-Wesley. (1996)

[6] Ohya, T., Iri, M., and Murota, K.: Improvement of incremental method for Voronoi diagram with computational comparison of algorithms. *Journal of Operations Research Society of Japan* **27**, (1985) 306–336

[7] Okabe, A., Boots, B., and Sugihara, K.: *Spatial Tessellations – Concepts and Applications of Voronoi Diagrams.* John-Willey. (1992)

[8] Rost, R. J.: *OpenGL Shading Language*, Addison Wesley. (2004)

[9] Shamos, M. I., and Hoey, D.: Closest-point problems. In *Proceedings of Annual IEEE Symposium on Foundation of Computer Science*, IEEE, (1975) 151–162

[10] Woo, M., Neider, J., Davis, T., and Shreiner, D.: *OpenGL Programming Guide*, third edition ed. Addison Wesley. (1999)

[11] Yamamoto, O.: Fast display of Voronoi diagrams over planes and spheres using graphics hardware. *Transactions of the Japan Society for Industrial and Applied Mathematics* **12**, 3, (2002) 209–234 (in Japanese)

[12] Yamamoto, O.: Fast computation of 3-dimensional convex hulls using graphics hardware. *Proceedings of International Symposium on Voronoi Diagrams in Science and Engineering*, September 13-15, 2004, Tokyo, Japan, (2004) 179–190

[13] Yamamoto, O.: An acceleration technique for the computation of Voronoi diagrams using graphics hardware. *Proceedings of International Conference on Computation Science and its Applications, 2005*, O. Gervasi et al. (Eds.), May 9-12, 2005, Singapore, Tokyo, Japan, *Lecture Notes in Computer Science vol. 3480*, Springer-Verlag, Berlin, Heidelberg, (2005) 786–795

[14] Yap, C.-K. 1988. A geometric consistency theorem for a symbolic perturbation scheme. In *Proceedings of 4th ACM Annual Symposium on Computational Geometry*, ACM, 134–142