解　説　コンピュータ・グラフィックスの基本テクノロジーと最近の話題

4. 3次元グラフィックスのハードウェア

3 D Graphics Hardware by Kei KAWASE, Takao MORIYAMA, Fusashi NAKAMURA and Kazuya SHIMIZU (Tokyo Research Laboratory, IBM Japan).

川瀬　桂[1]　森山孝男[1]
中村英史[1]　清水和哉[1]

1 日本アイ・ビー・エム(株)東京基礎研究所

1. はじめに

ランダムスキャンのディスプレイを用いた線画表示システムから始まったコンピュータ・グラフィックスは，1960年代初頭に登場したラスタスキャン・ディスプレイの一般化とともに急速に進歩し，現在では3次元図形の描画をハードウェアで対話的に実現するようになってきた．これにともなって，3次元グラフィックスの応用範囲も，コンピュータゲームや，VRMLによるインターネット上での3次元表示といったパーソナルコンピュータ上での利用まで普及しつつある．

3次元描画法は多くの手法が考案されているが，インタラクティブな描画に限定した場合，OpenGLやPHIGS PLUSなどのアプリケーション・プログラム・インタフェース（API）を通してZバッファ法とグーロー・シェーディングといった手法を用いることが一般的である．

本稿ではこういった手法を用いたグラフィックスのハードウェアに関する解説を行う．

2. 処理の流れ

Zバッファ法を用いてグーロー・シェーディングを行う場合の一般的な処理の流れを以下に示す．

1. モデル生成（Model generation）
   ユーザはまず何らかの方法（たとえばモデラや3次元スキャナ）で3次元モデル（階層構造をもったデータベースになっていることが多い）を構築する．
2. トラバーサル（Traversal）
   上記のモデルを順次解釈して描画属性と描画命令の列に直す．
3. ジオメトリ処理（Geometry processing）
   与えられた描画情報の列から表示される図形の画面上座標とその各頂点の奥行き，色，テクスチャ座標の値を生成する．
4. ラスタ処理（Raster processing）
   与えられた図形をフレームメモリ上で表現するために必要とされる点列を作成し，その各点における奥行き，色の値とテクスチャ座標を頂点での各々の値を補間して求める．さらにこの補間されたテクスチャ座標を用いてテクスチャリングを行い，フレームメモリ上に現在あるピクセル値との間でピクセル演算を行った後フレームメモリに書き戻す．
5. 表示（Display）
   フレームメモリよりピクセル値を読みだし表示する．

モデル生成は通常はオフラインで行うので本稿ではとくに触れないものとする．また，トラバーサルも最近のワークステーションでは，ワークステーション側のプロセッサで行うことが多いので触れないものとする．**図-1**にジオメトリ処理以降の処理をもう少し詳しく示す．

2.1 ジオメトリ処理

ジオメトリ処理部は与えられた3次元データに対して，座標変換，ライティング，クリッピングといった処理を行い，フレームメモリ上での2次元座標（x, y）と奥行きなどの値をもった頂点の列への変換を行う処理である．それら処理はほとんどが浮動小数点演算である．詳細についてはそれぞれのAPIごとに多少定義が違うが，おおざっぱにいって最近のRISCプロセッサ上で処理すると1頂点あたり数百クロック程度[8]を消費す

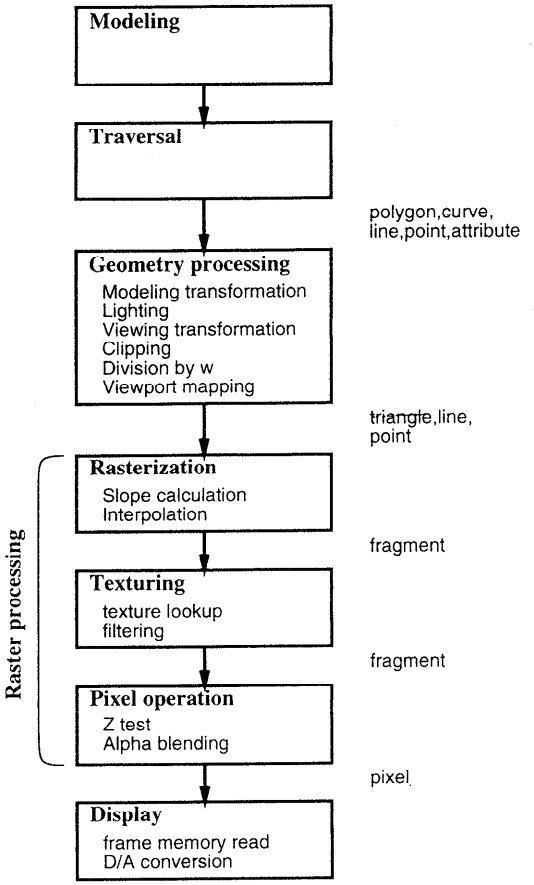


図-1 処理の流れ

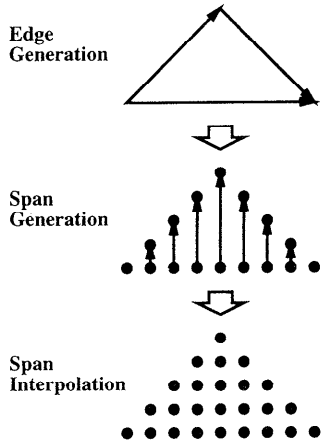


図-2 ラスタライズ

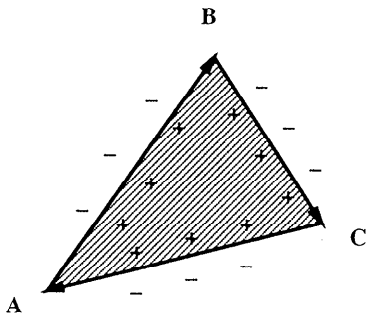


図-3 エッジ関数

る（当然光源の種類とその数に依存する）．計算の詳細は各APIごとにきめられているので本稿ではこれ以上は触れない．

2.2 ラスタ処理

ラスタ処理部は大きく分けてラスタライジング，テクスチャリング，フォギング，ピクセル演算の処理からなる．

ラスタライジング

ラスタライジングとは頂点の列だけを与えられた三角形，線分といった図形集合から，それらを表現する2次元座標（x, y）と，奥行き座標値（z）と色の値（r, g, b, a）（r, g, bは色を赤，緑，青の成分に分解したときのそれぞれの明るさ．aは透明度などを表すのに使われる）をもったフレームメモリ上のピクセルに対応する点（フラグメント）の集まりに変換する処理である．以下に例として三角形をグーロー・シェーディングする場合についてその計算方法を説明する．

2次元座標（x, y）と値（z, r, g, b, a）をもつ3つの頂点が与えられたとき，その頂点に囲まれる三角形領域内部に含まれるピクセルに対応する点を求め，それぞれの点での値を求めなければならない．これは三角形領域内部に含まれるピクセルに対応する点の座標（x, y）をすべて求めることと，それらの点での値（z, r, g, b, a）を計算することに分けられる．

3頂点で囲まれる領域を求める方法はいろいろとあるが，三角形の辺を含む直線を発生させ，その間の縦スパンを求め，そのスパンごとに補間していく方法（**図-2**）や，3辺を表す辺関数（Edge function）を利用することによって任意の点について三角形の内外の判定を行う方法[4]（**図-3**）が一般的である．

フレームメモリ上の各ピクセルは整数の2次元座標をもつが，描画をより忠実に行うために，頂点の座標は固定小数点（実装によっては浮動小数

点）で指定される．またピクセルに対応する点が三角形内部か外部かの判定は通常ピクセルの中心座標で行うが，中心座標が辺上にきてしまった場合は何らかの排他的な割り振りのルールを用いて判定しなければならない．また中心座標のみでなくピクセルが面積をもつと考え，その一部でも三角形の内部に入った場合は，その面積の割合で色を決めることで三角形の周辺部に起きるギザギザを軽減するアンチエリアシングを行うこともある．（ただし複数の三角形が重なる場合は，面積だけの情報では正確なアンチエリアシングは行えない．）

三角形内部の点の値を求めるには3次元空間での平面方程式を解くことにほかならない．たとえばzについて，与えられた3頂点P_0（x_0, y_0, z_0），P_1（x_1, y_1, z_1），P_2（x_2, y_2, z_2）からその内部の点の値z（x, y）を求めるには以下の式を解けばよい．色r, g, b, aについても通常は同様に行う．厳密にいうと透視変換を行っているときは，色を画面上で線形補間することは正しくない．後に述べるテクスチャ座標の補間と同様の注意が必要であるが，実用上ここに述べた線形補間で十分である．

$$z(x,y)=\frac{A}{C}(x_0-x)+\frac{B}{C}(y_0-y)+z_0$$

where,

$$A=(z_2-z_0)(y_1-y_0)-(z_1-z_0)(y_2-y_0)$$
$$B=(x_2-x_0)(z_1-z_0)-(x_1-x_0)(z_2-z_0)$$
$$C=(y_2-y_0)(x_1-x_0)-(y_1-y_0)(x_2-x_0)$$

$-A/C$ と $-B/C$ は平面のx, y軸に沿った傾き $\partial z/\partial x$ と $\partial z/\partial y$ にほかならない．この傾きは三角形ごとに求めなければならないが，割り算が必要なのでこれをどのように実装するかが設計のポイントになる．内部の値を求めるのに掛け算を使わず順次計算する方法[2]はあるが，傾きを求めるための割り算はやはり避けられない．

テクスチャリング

テクスチャリングは生成されたフラグメントに対してテクスチャ（模様）を張りつける処理（テクスチャ・マッピング）を行うことである．張りつけられるテクスチャを描画時に生成するプロシージャルテクスチャとあらかじめイメージ（当然のことながら離散的な画像，各要素をテクセルという）としてをメモリに入れておくイメージテクスチャがあるが，本稿では一般的に多用されているイメージテクスチャについてのみ触れる．テクスチャリングは三角形の各頂点のテクスチャ座標をz値と同じくラスタライズしてフラグメント単位に補間した後，その座標を用いてテクスチャを格納してあるメモリから該当するテクセルの値を読み出し，適切なフィルタ処理を施してフラグメントの色と適宜混ぜ合わせてフラグメント値の更新を行う．

各頂点のテクスチャ座標をラスタライズしてフラグメント単位に計算することはzの値を求めるのと違ってかなりコストがかかる．透視変換を行って画面に表示を行う場合，テクスチャ座標自体は透視変換を行う前の座標系（世界座標系もしくは視線座標系）で直線補間されなければならない．同次座標系である視点座標系での2頂点P_0（x_{e0}, y_{e0}, z_{e0}, w_{e0}），P_1（x_{e1}, y_{e1}, z_{e1}, w_{e1}）でのテクスチャ座標がそれぞれ（s_0, t_0），（s_1, t_1）である場合，これらを画面座標系で補間するには以下の式で行わなければならない．

$$s(k)=\frac{(1-k)(s_0/w_{e0})+k(s_1/w_{e1})}{(1-k)(1/w_{e0})+k(1/w_{e1})}$$
$$t(k)=\frac{(1-k)(t_0/w_{e0})+k(t_1/w_{e1})}{(1-k)(1/w_{e0})+k(1/w_{e1})}$$

これは1つのフラグメントを生成するごとに，テクスチャ座標の次元あたり1回の割り算をしなければならないことを意味する．この割り算を避けるために近似を行う方法や局所性を利用して漸近的に計算する方法も提案されているが，正確さや並列処理との適合性などの問題がある．

また表示画像の品質を上げるために，補間されたテクスチャ座標からイメージを参照するのに，最近傍のテクセルの値だけ取り出すのではなく，近傍数個のテクセルの値を取り出してこれらをフィルタリングして用いる．たとえば2次元のイメージに対してトリリニア（tri-linear）という手法を用いる場合，1フラグメントあたり8個のテクセル参照が必要になる．

したがって，たとえば100 M fragment/secの早さで32 bit/texelの2次元イメージをtri-linearでテクスチャリングしたフラグメントを発生させるためには，200 M div/sec，3.2 GB/secの割り算とメモリバンド幅が必要とされる．

フォギング

遠近感を出すため視点から遠い物体を霞んでみえるようにするフォギング（デプスキューと呼ばれるものも効果は同じ）の施す場合もある．これはフラグメントの色 C_r に対して視点からの距離 distance（APIによってはz座標の値）の関数で，フォグの色 C_f（通常は白または黒が使われることが多い）と混ぜ合わせる演算によって行われる．

$$f=Func(distance)$$
$$C_n=f\,C_r+(1-f)C_f$$

関数 Func() は通常折れ線関数や指数関数などが使われる．指数関数は表引きを用いて実装されることが多い．

ピクセル演算とフレームメモリ

前述のように生成・処理されたフラグメントは**図-4**に示すようにフレームメモリから読み出されたデータとの間でz値の比較やアルファ・ブレンディングなどの演算が行われた後，再びフレームメモリに書き戻される．

描画途中の状態がみえるとアニメーションがみずらくなるので，フレームメモリには色表示用のバッファを二重にもち（ダブルバッファ），それらに対して交互に描画を行い，現在描画を行っていない側を表示することで描画途中の状態がみえないようにしている．さらに，Zバッファやステンシル・バッファ，メニューなどを表示するオーバーレイ・プレーン，ウィンドウシステムを支援するためのプレーンなどが必要で，合計128 bit/pixel程度必要になる．128 bit/pixelで1280×1024の画面を構成すると20 Mbyteのフレームメモリが要求される．

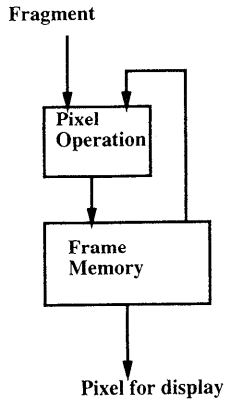


図-4 ピクセル演算

バンド幅について考えると，前述のように1ピクセル描画には2回のアクセス（読出し1回，書込み1回）が必要なので，たとえば128 bit/pixelで100 M pixel/secの描画を行うためには，3.2 GB/secのメモリバンド幅が必要であり，さらに表示用に150 MHzで64 bitを読み出すとすると，合計4.4 GB/secのバンド幅が要求される．ただし不要な部分への読み書きを行わないなどの工夫によってアクセスを軽減できる可能性はある．

3. 実 装 例

描画の高速化に際しては，演算の量とハードウェアの各コンポーネント間のバンド幅が問題になる．半導体技術の進歩とともにこれらの問題は軽減されつつあるが，どのようにハードウェア・コンポーネントを設計し配置するかは重要である．たとえばフレームメモリのバンド幅に関しては，メモリをインタリーブして同時に多数のバンクに対してアクセスすることによって，バンド幅を確保する工夫がなされてきた．また演算量の問題はパイプライン処理や並列処理を行うことによって解決されてきた．これまでに行われてきたいくつかの実装例を以下に紹介する．

ジオメトリ演算を専用のパイプラインプロセッサで処理する方法は1980年代初頭から用いられてきた．**図-5**にジオメトリ演算をパイプラインで行い，ラスタ処理をスパンごとにインタリーブして並列処理で行う例を示す．図中G，R，T，P，Dは各々Geometry Processing, Raster Processing, Texturing, Pixel Operation, Displayを表す．このような実装例としては，Silicon Graphics社の4 D/240 GTX[5]や4 D/480 VGXがある（両者ともラスタ処理の前半部のスパン生成部までパイプライン処理で行いその後を並列処理している．また4 D/240 GTXはテクスチャリングの機能はない）．

当初は座標変換とクリッピング判定のように処理がパイプラインのステージに静的に割り当てられるものだけを考えていたので問題にならなかったが，ライティングのように複雑で計算量が動的に変化するような処理が入りだすとパイプラインプロセッサではロードバランスを適切に保つことが難しくなり，性能を十分に発揮できなくなって

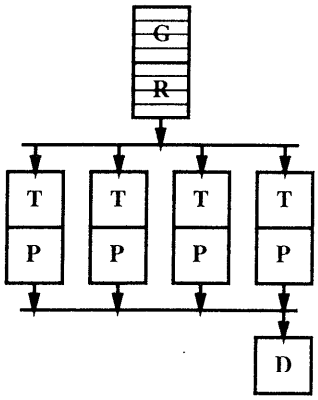


図-5 GR-TPD 方式

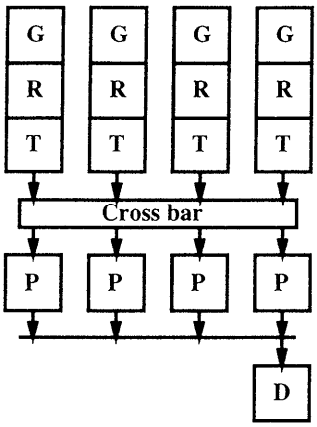


図-7 GRT-PD 方式

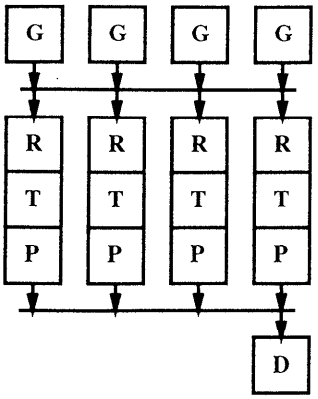


図-6 G-RTPD 方式

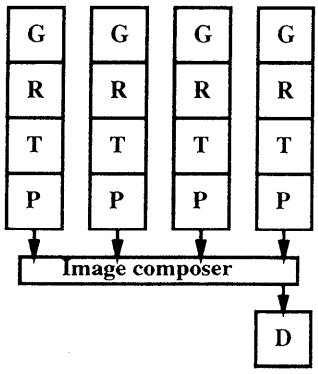


図-8 GRTP-D 方式

きた．

そこで図-6のようにジオメトリ演算を描画プリミティブ（三角形や線分）ごとに1つのノードに割り当て処理を行う方法（オブジェクトパラレル）が考案された[3]．RealityEngine Graphics[7]では，ジオメトリ処理部とラスタ処理部の接続をバス構造にして，並列にジオメトリ処理された描画プリミティブをこのバス上で回収している．

ジオメトリ処理部とラスタ処理部の接続をバスで接続するとそこがボトルネックになる．これに対して Evans & Sutherland 社の Freedom やクボタコンピュータの Titan II では図-7のようにラスタライズまでをオブジェクトパラレルで行い，その後をクロスバーでピクセル処理部に渡すということをしている．

PIXEL-PLANES[1]や PixelFlow[6]では図-8にあるようにすべてを並列化処理している．描画プリミティブごとは各ノードでオブジェクトパラレルに処理を行い，それぞれのノードがもつ部分的な画像を合成する（Image composition）ことによって全体の画像を作る．この構成は描画プリミティブの分配さえ行えればほかにボトルネックとなるところがないのでスケーラブルな並列度が得られるが，画像合成を高速に行う回路が必要な上，各ノードはそれぞれローカルに表示画面サイズと同じ大きさのフレームメモリをもたなければならないのできわめて多量のリソースが要求される．Pixel-Planes 5 以降はノードごとのローカルなフレームメモリは 128×128 pixel にして，表示画面を 128×128 ごとに分割して複数回に分けて描画する方針（tiling）に変更している．またローカルにはフレームバッファの一部をキャッシュとしてもつ方法も開発されている[10]．

4. 並列処理にともなう問題

並列処理を行う場合，描画結果がシングルプロセッサ実行したのものと同一になるのかという問題がある．主な問題としては API を通して発行

された描画命令の解釈の問題と，発行された描画コマンドがフレームメモリに反映される順序の問題がある．

4.1 アトリビュート処理

描画コマンドは図形要素（プリミティブ：点，線分，ポリゴンなど）の描画を指示するものと属性（アトリビュート：座標変換マトリックス，材質，光源など）の設定指示を行うものとに大別される．アトリビュートはステートであり，アトリビュート設定指示が発行されるたびに随時更新される．プリミティブの処理はその描画を指示するコマンドが発行されたときのアトリビュートを用いて描画を行わなければならない．

マルチプロセッサで並列処理を行う最も原始的な方法は，描画コマンドを図-9のようにアトリビュート設定指示はブロードキャスト，プリミティブ描画命令は1つのプロセッサを選んでユニキャストするという方法である．この方法では各プロセッサは自分に割り振られたコマンドを順次処理するだけで正しいアトリビュートでプリミティブ描画命令を処理することができる．しかしこの方法ではアトリビュート設定指示命令の処理自体は並列効果が出ない（すべてのプロセッサで同じ処理をしている）ので，プロセッサの台数が多くなると効率が上がらなくなってしまう．

ある瞬間のアトリビュートの値（スナップショット）をまとめてすべてのプリミティブ描画命令の前につけてプロセッサに割り当てることは，アトリビュートの大きさが小さくない（通常1 Kbyte以上）ので汎用のプロセッサで行うとコストがかかりすぎて現実的ではない．

アトリビュートのスナップショットを専用ハードウェアを使って高速化する手法[9]や，OpenGLのように頻繁に変化するアトリビュートの種類が少ないものに関して，アトリビュートの種類を「頻繁に変更される部分」と「頻繁には変更されない部分」の2種類に分類して，「頻繁に変更される部分」のみスナップショットを作ってプリミティブ描画命令の前につけてプロセッサに割り当て，それ以外はブロードキャストするという方法も開発されている[7]．

4.2 描画順序

描画順序の問題とは描画プリミティブが発行された順にフレームメモリの上にその結果が反映さ

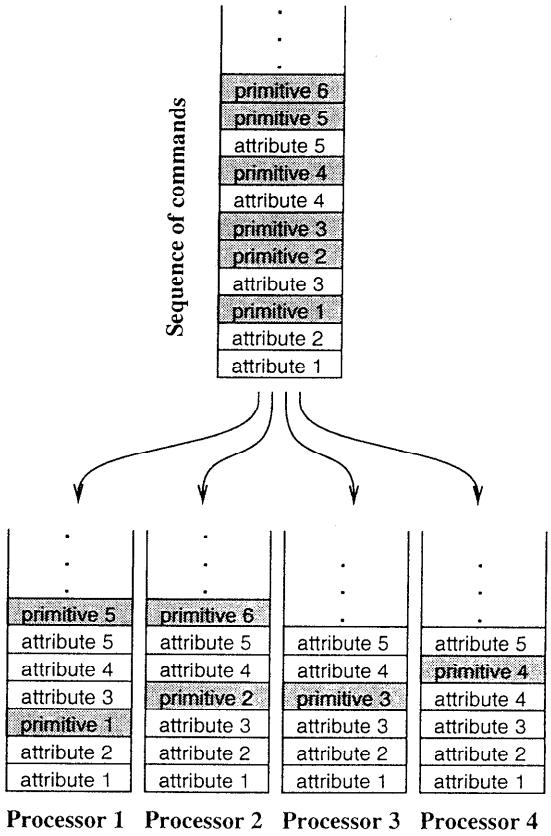


図-9 描画コマンドの分配

れるかどうかということである．Zバッファ法を用いて描画を行う場合，どのような順序で描画しても最終的に生成される画面は同じなので描画順序は保証する必要はないとしている報告もあるが，それは以下の理由で受け入れられない場合もある．

1. アルファ・ブレンディングやステンシル・バッファといった操作は描画順序に依存して結果が異なる．
2. 重なり合う複数の描画プリミティブのピクセル間でzの値が同一になる場合，そのピクセルの色は描画順序に依存する．（細かいことではあるが，このような違いを問題にするアプリケーションが存在する）

1番目の問題はアルファ・ブレンディングやステンシル・バッファといった操作がそもそも順序性を仮定した手法であるので，その部分については同期機構などを用意して順次処理をするようにすれば，そのほかのものについては順序を気にせ

ずに描画することも可能である．

2番目の問題まで気にするとそもそも描画順序はすべて守らざるを得なくなる．

前述の図-7や図-8といった切り分けでは，順序を守った描画を行うには描画プリミティブごとに同期をとらなければならなくなるのでハードウェアの利用効率は落ちてしまう．

5. 実装技術

フレームメモリのバンド幅の問題に対してメモリと同じチップ内にロジックを入れることが考えらている．これらはLogic in memory やSmart memoryと呼ばれている．同一チップ内ではより多くの配線を高速に動かすことが可能になるで多くのメモリバンド幅をとることが可能となる．古くはTexas Instruments 社が開発したVideo RAM（VRAM）があげられる．これはDRAMの1つの行（row）を丸ごとバッファしておき通常のリード/ライト・ポートとは別のポートから順次読み出せる．VRAMは画面表示のための読み出しのバンド幅を飛躍的に拡大し周辺回路の簡略化に貢献した．描画機能を内蔵したものとしては1次元スパン計算器を内蔵したScan Line Access Memory（SLAM）が1985年に発表されている．また1994年には3次元グラフィックスに必要なピクセルごとの演算機能を内蔵したFBRAMが発表されている．FBRAMではZバッファやアルファ・ブレンディングなどのread modify write がチップ内で行われるため，チップ外の描画のためのデータ転送が半減する．

これとは対照的にチップ間の通信を高速化することによってバンド幅の問題を解決しようという方向もある．これはLogic in memory が通常のDRAMに対して製造がより難しい（歩留まりを上げにくい）ことや，専用品であるために量産効果による生産コストの低下が見込みにくいということと，現在（および次世代）のテクノロジーではフレームメモリ全体を1つのチップに納めることができないため，画面内での画像の移動など依然チップ間の通信の高速化が必要な場合があることがあげられる．高バンド幅を目指したDRAMとしては，Rambus Inc. のRDRAMやMoSys Inc. のMDRAMなどがある．いずれも1チップあたり数百MB/secのバンド幅をもち将来的にはGB/secのオーダをねらっている．これらはバースト転送が基本なため性能を引き出すには工夫が必要である．

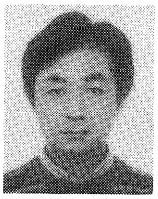
6. おわりに

ワークステーションに用いられてきた高性能な3次元グラフィックスのハードウェアについて紹介してきた．今後はVRMLの処理やアプリケーションレベルからの並列性を考慮した新しいアーキテクチャが登場してくる可能性がある．また本稿では触れなかったが，ゲーム機やパーソナルコンピュータのように正確さよりコストを重要視するような世界では違ったアーキテクチャが登場してきており，今後も変遷を続けていくものと思われる．

参考文献

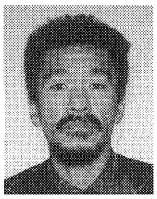
1) Poulton, J. et al.: PIXEL-PLANES: Building a VLSI-Based Graphics System, Proc. of 1985 Chapel Hill Conf. on VLSI, pp. 35-60 (1985).
2) Swanson, B. W. and Thayer, L. J.: A Fast Shaded-Polygon Renderer, Proc. SIG GRAPH'86, pp. 95-100 (Nov. 1986).
3) Torborg, J. G.: A Parallel Processor Architecture for Graphics Arithmetic Operaions, Proc. SIGGRAPH'87, pp. 197-204 (July 1987).
4) Pinedia, J.: A Parallel Algorithm for Polygon Rasterization, Proc. SIGGRAPH'88, pp. 17-20 (Aug. 1988).
5) Akeley, K. and Jermoluk, T.: High Performance Polygon Rendering, Proc. SIG GRAPH'88, pp. 239-246 (Aug. 1988).
6) Molnar, S., Eyles, J. and Poulton, J.: PixelFlow: High-Speed Rendering Using Image Composiotion ACM Computer Graphics, 26, 2 pp. 231-240 (July 1992).
7) Akeley, K.: RealityEngine Graphics, Proc. SIGGRAPH'93, pp. 109-116 (1993).
8) 松本 尚，川瀬 桂，森山孝男：PHIGSのジオメトリ演算のための並列処理方式の検討，情報処理学会論文誌，Vol. 35, No. 1, pp. 92-101 (Jan. 1994).
9) 川瀬 桂，森山孝男：アプリケーショントレースを用いたPHIGSの並列処理の実行効率評価，グラフィックスとCADシンポジウム論文集，pp 87-94 (Sep. 1994).
10) Nishimura, S. and Kunii, T.: VC-1: A Scalable Graphics Computer with Virtual Local Frame Buffer, Proc. SIGGRAPH'96, pp. 365-372 (1996).

（平成8年11月5日受付）



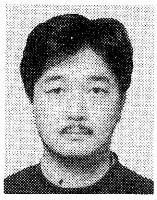
川瀬 桂（正会員）

1961年生．1985年早稲田大学理工学部機械工学科卒業．1987年同大学院理工学研究科博士前期課程修了．同年日本アイ・ビー・エム(株)東京基礎研究所に入社．3次元グラフィックスシステムの研究に従事．



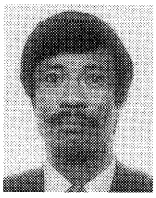
中村 英史（正会員）

1961年生．1992年東京大学大学院理学系研究科物理学専攻博士課程修了後，法政大学工学部非常勤講師を経て，1993年日本アイ・ビー・エム(株)東京基礎研究所に入社．3次元グラフィックスシステムの研究に従事．理学博士．



森山 孝男（正会員）

1962年生．1985年東京工業大学工学部情報工学科卒業．1987年同大学院修士課程修了．同年日本アイ・ビー・エム(株)東京基礎研究所に入社．並列マシンのオペレーティングシステム，グラフィックスの並列処理の研究に従事．



清水 和哉（正会員）

1952年生．1967年東京大学理学部物理学科卒業．同理学研究科情報科学専攻課程を経て，1972年日本アイ・ビー・エム(株)東京サイエンティフィックセンターに入社．以来グラフィックスの研究・開発に従事．現在同社東京基礎研究所に勤務．