アトリビュートプロセッサを用いて効率化した三次元グラフィクスのジオメトリ演算の並列処理

川瀬 桂[†] 森山孝男[†]

PHIGS や **OpenGL** 等の対話型の三次元グラフィクスインタフェースのジオメトリ演算をマルチプロセッサを用いて処理する場合，描画アトリビュートの処理が高速化のために重要である．本稿では汎用マイクロプロセッサを使用したマルチプロセッサシステムにおいて，描画アトリビュートを効率良く処理する方法としてアトリビュートプロセッサを提案し，実際のアプリケーションの描画命令列のトレースを用いてシミュレーションによる比較を行い，提案する方式の優位性を示した．

Attribute Processor: An Efficient Parallel Processing Method for 3D Graphics Geometry Calculation

KEI KAWASE[†] and TAKAO MORIYAMA[†]

Efficient handling of drawing attributes is one of the most important design points of parallel rendering of interactive 3D graphics such as **PHIGS** and **OpenGL**. We invented Attribute Processor method for such applications. In this paper we present our method along with several alternatives, and describe the superiority of our method by simulation using real application traces.

1. はじめに

三次元グラフィクスはその計算量の多さから，マルチプロセッサを用いて高速化を図る試みが多くなされてきた．筆者らは対話型のグラフィクスインタフェースである **PHIGS**[3] や **OpenGL**[8] を採用する場合，モデリングデータの構造からくる制約が性能上の問題点となることを指摘してきた[6]．

描画プリミティブ（以降単にプリミティブと呼ぶ）の処理は並列化により容易に性能を向上させることができるが，描画アトリビュート（以降単にアトリビュートと呼ぶ）の処理は並列化が困難である．プリミティブだけを並列処理した場合，アトリビュートの処理コストが相対的に増加し，プロセッサの処理能力の半分以上がアトリビュートの処理に費やされる場合も生じる．したがって，アトリビュートの処理をいかに高速化するかがシステム全体の性能を上げる鍵である．

本稿では実際の CAD アプリケーションの例としてダッソーシステムズ社の機械設計 CAD CATIA を，そして Standard Performance Evaluation Corporation のグラフィクスベンチマーク OPC Viewperf[9] の中から CAD アプリケーションのデータを用いた CDRS-03 とビジュアリゼーション（可視化）アプリケーションのデータを用いた DX-03 を選び，それらが生成する描画命令列のトレースを採取し，そのトレースデータを用いたシミュレーションを行い，複数のアトリビュート処理方式の実行効率の評価を行った．

なお，本稿では座標変換，照度計算等のジオメトリ演算の処理のみを扱い，ホストシステムからのデータ転送やラスタ処理部は隘路にはならないという仮定を置いている．またラスタ処理部の構成法についても触れない．ジオメトリ演算の詳細については各 API の仕様[3),8)] を参照されたい．

2. アトリビュート

PHIGS や **OpenGL** におけるアトリビュートに関して簡単に触れておく．表示するための三次元データには，各種のプリミティブ（点，直線，多角形等）の描画を指示するものとアトリビュート（プリミティブの色，光の拡散率，変換行列等）の設定を指示するものがある．

プリミティブの処理中に必要とされるアトリビュートの種類はプリミティブの種類によって異なるが，多

† 日本アイ・ビー・エム株式会社東京基礎研究所
IBM Research, Tokyo Research Laboratory

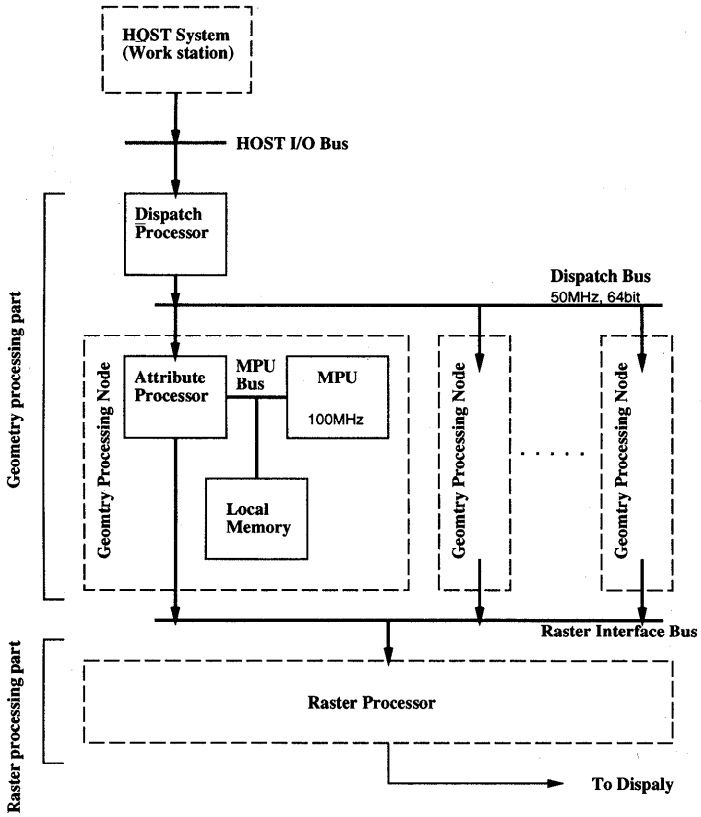


図1 システムの構成
Fig. 1 Overview of the system.

くのものは後続するプリミティブ間で共通している場合が多い．たとえば，1つの物体をいくつかの多角形で表現する場合，変換行列は共通であるし，均一な物質からできていれば色や光の拡散率は変わらない．**PHIGS** や **OpenGL** では，一度設定したアトリビュートの値は同じアトリビュートが再設定されない限り有効である☆．実行時にはアトリビュート設定指示に従ってある領域にアトリビュートを保存しておき，プリミティブを処理する間にはその領域からアトリビュートを読み出して使用する．このアトリビュートを保存する領域を今後 ASL（Attribute State List）と呼ぶことにする．

PHIGS の定義では ASL の大きさは可変長であるが，実装時に大きさを制限することも許されており，通常は 1K ないし 2Kbyte 程度の固定長で実装する．**OpenGL** も同様である．

☆ アトリビュートは Push/Pop 可能なので Pop 時にもアトリビュートの値は変更されうる．

3. システムの構成と処理の流れ

プロセッサの配置のしかたとしては，直列に並べて1つのプリミティブをパイプラインで処理する**パイプライン方式**と，並列に並べて，手の空いたプロセッサにそれぞれ1つのプリミティブのジオメトリ計算を順次割り当てる**順次振り分け方式**があるが，後者を用いた方が負荷分散の面での優位であることが示されている[2),4),6),10)]のでそれを採用することとした．

また，中小規模のマルチプロセッサシステムを構築する際，共有バス型の共有メモリを実装することは汎用性の高いシステムにできる反面，プロセッサ数が増えてくると実装にかかるコストが大きくなる．本稿では実装コストを重視し，共有バス型の共有メモリを持たない方針をとり，図1に示すシステム構成で検討を行った．

演算ノードは Dispatch Bus，Raster Interface Bus および MPU Bus 間のブリッジ機能とアトリビュート処理機能（後述）を持つアトリビュートプロセッサ，ジオメトリ演算を行う汎用マイクロプロセッサ（MPU），

そしてLocal Memoryからなる.

大まかな処理の流れは以下のとおりである.

(1) プリミティブとアトリビュートからなる描画命令列はホストシステムの主記憶内にあり，HOST I/O Busを通してジオメトリ処理部のディスパッチプロセッサ（Dispatch Processor）に送られる.

(2) ディスパッチプロセッサはDispatch Busを通して描画命令列を順次各演算ノード（Geometry Processing Node）に分配する.

(3) 演算ノードは配られたプリミティブに対して座標変換や照度計算といったジオメトリ処理を行い，結果をRaster Interface Busを通してRaster Processorに送る.

PHIGSや**OpenGL**では，描画命令の発行順序を維持したまま描画を行うことを規定しているので，並列にジオメトリ処理をした描画命令を発行順にRaster Interface Bus上で並べ直してRaster Processorに渡す方式（Sort-middle方式[7]）を使う.

順次振り分け方式では，ディスパッチを行うプロセッサの性能が隘路になりやすいことと，アトリビュートの処理の効率良い扱いが重要であることが指摘されている[5]. 前者の問題に対しては，ディスパッチのための支援機構を付加した専用プロセッサを用いることで，隘路になることを回避することにした.

4. アトリビュート処理

4.1 アトリビュート処理の問題点

順次振り分け方式で並列処理する場合，演算ノードがプリミティブの処理をしている間，ASLのスナップショットを演算ノードごとに個別に保持しなければならない. なぜならば，すべての演算ノードで共通なASLを設けたとすると，ある演算ノードが1つのプリミティブの描画のためにASLを参照している間はASLの内容を固定しておく必要があるので，そのプリミティブの描画の指示に続くアトリビュート設定指示を実行することができないからである.

ASLの参照の衝突を避けるために，プリミティブとともにASLの内容を複製して各演算ノードに転送する方法が考えられるが，一般的にプリミティブ転送のコストに対してASLの内容のすべてを複製・転送するコストは無視できず実用的でない.

4.2 従来の方法

ジオメトリ演算を効率良く並列処理を行うためには，アトリビュートを効率良く演算ノードに割り振ることが重要である. この問題に対して支援機構の導入による解決方法の例をいくつか示す.

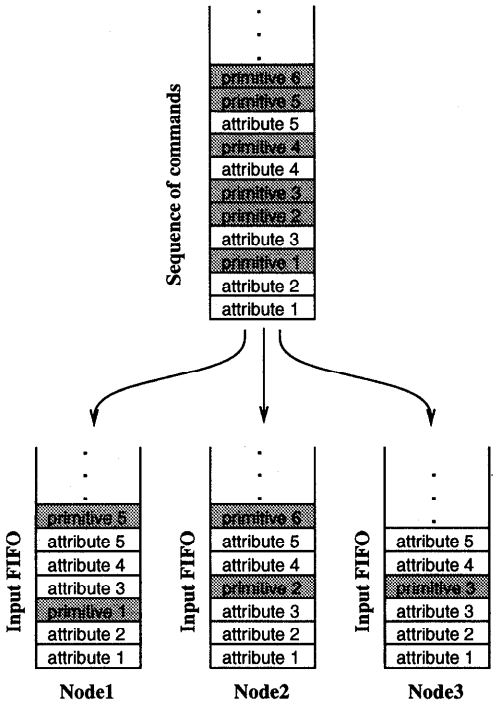


図2 Broadcast方式
Fig. 2 Broadcast method.

4.2.1 Broadcast方式

すべてのアトリビュートを演算ノードに対して全放送してしまう方法である[10]. この方法は全放送の機構と各演算ノードの入力FIFO（Input FIFO）だけを必要とするので実装は容易である（図2）.

しかし，この方法ではアトリビュートの処理に対して並列処理の効果が出ないので，プリミティブあたりのアトリビュートが多いときや演算ノードの数が増えたときは非常に効率が悪くなる.

各演算ノードでのアトリビュートの処理は以下の手順で行う.

(1) プロトコルヘッダを読み込み，解析する.

(2) アトリビュートの格納先を計算する.

(3) アトリビュートを入力FIFOから読み込む.

(4) アトリビュートを格納する.

これらの処理を汎用プロセッサで行った場合，上記(1)と(2)の処理はアトリビュートデータのサイズによらず10クロック，(3)と(4)の処理にはアトリビュートデータ1ワードあたり2クロックかかると見積もると，N-wayの演算ノードではプロトコルサイズがmのアトリビュートを処理するたびに，全体で$(N-1)\times(10+2(m-1))$クロックの無駄が生じる.

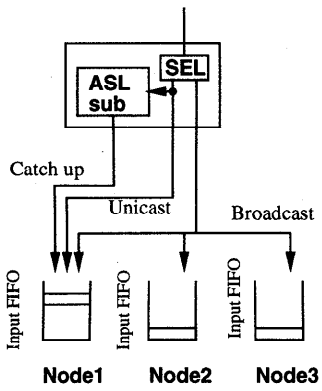


図3 Broadcast-Unicast 方式
Fig. 3 Broadcast-Unicast method.

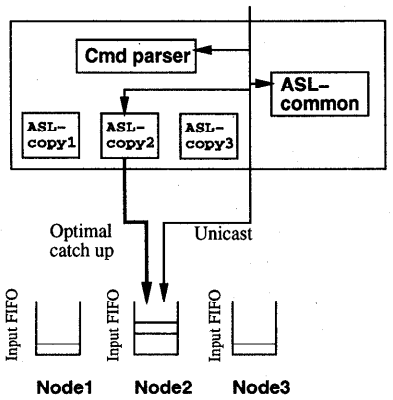


図4 アトリビュート転送の最適化
Fig. 4 Optimized attribute transfer.

4.2.2 Broadcast-Unicast 方式

この方法はまずアトリビュートを頻繁に変更される種類とそうでないものの2種類に分類し,「頻繁に変更されるアトリビュート」を格納するためのバッファ(**図3** の ASL sub)を装備する.「頻繁に変更されるアトリビュート」はディスパッチ対象のノードに対して送ると同時に ASL sub にも格納しておく.ディスパッチの対象ノードを切り替える場合には ASL sub の内容をすべて対象のノードに送る.これを catch-up 動作と呼ぶ.「頻繁に変更されないアトリビュート」は Broadcast 方式と同じ方法を用いて各演算ノードで処理を行う[1].履歴が問題になるような処理は「頻繁に変更されないアトリビュート」に分類する.

「頻繁に変更されるアトリビュート」のサイズの総和,すなわち ASL sub のサイズが小さいほど catch-up にともなうオーバヘッドが少なくなる.ただし,実際に頻繁に変更されるアトリビュートであるにもかかわらず「頻繁に変更されるアトリビュート」に分類されないものがあると,頻繁に全放送が起き効率が低下する.

この方法は **OpenGL** には有効であるが,**PHIGS** では「頻繁に変更されるアトリビュート」の総量が大きいため効率が上がらない.また NAME SET 等のように「頻繁に変更されるアトリビュート」の中にも履歴が問題になるものがあるので処理が困難である.

5. 提案する解決方法

5.1 アトリビュート転送の最適化

従来の方式の問題を回避するために演算ノードに対するデータ転送量を最小にすることを考える.そのためにディスパッチプロセッサ内部に完全な ASL の複製(**図4** 中の ASL-common)とともに,それぞれの演算ノードに対するアトリビュートの複製(同図 ASL-copy1, ASL-copy2, ASL-copy3)を装備する.ASL-copy の各エントリには ASL-common との差分を表すフラグ(変更フラグ)がついている.

アトリビュートが来ると,ディスパッチプロセッサは ASL-common の値を更新すると同時にディスパッチ対象のノードに送る.ASL-copy の処理はディスパッチ対象かどうかで異なる.ディスパッチ対象のノードに対しては ASL-copy のエントリを更新するとともに変更フラグをクリアする.それ以外のノードに対しては該当する ASL-copy の値と ASL-common の値を比較し,その結果を変更フラグに反映させる.

ディスパッチ対象のノードを切り替える場合は,新しいノードの ASL-copy の変更フラグを調べ,値の変更のあったアトリビュートのみを対象ノードに送出する.それと同時に該当する変更フラグをクリアする.これによって Dispatch Bus の転送量をおさえることができる.

さらに最適化を行う場合には,プリミティブの処理に必要なアトリビュートのセットをあらかじめ表としてディスパッチプロセッサ内部に持っておき,以下の2つの条件を満たすものだけを送る方法が考えられる.

- これから割り当てようとしているプリミティブの処理に必要なもの.
- 前回の割り当て時から値が変更されたもの.

この処理により catch-up にともなうアトリビュートの転送量を削減できるが,以下の問題が残る.

- ディスパッチプロセッサ内部にすべての演算ノードに対応するだけの ASL-copy を持たなければならないので複雑化する.
- 演算ノードを増やすにはディスパッチプロセッサを作り直さなければならない.

- 個々の演算ノードから見れば最適な転送であるが，Dispatch Bus には同じアトリビュートが複数回流れることとなり，演算ノード数の増加とともに Dispatch Bus が飽和する．
- ディスパッチプロセッサ内部での catch-up 処理は並列処理ではないので演算ノードが増加するとその処理が隘路になってくる．

そこで，ディスパッチプロセッサ内部の ASL-copy を各演算ノード内部に移すことによりディスパッチプロセッサの複雑化をおさえ，スケーラビリティを確保し，さらに Dispatch Bus のデータ転送量を最小におさえる方法として Broadcast-AP 方式を提案する．

5.2 Broadcast-AP 方式

Broadcast-AP 方式ではアトリビュートはつねに全放送され，各演算ノードでアトリビュートの複製を更新する．

そのため，アトリビュート処理用の支援機構であるアトリビュートプロセッサと MPU を組み合わせることを提案する（図 1 および**図 5**）．MPU には高性能な浮動小数点演算器とスヌープキャッシュを持つものを選択し，アトリビュートプロセッサとの間を共有バスでつなぐ．

アトリビュートプロセッサは MPU とは並列に動作し，アトリビュート設定指示を解釈し処理する．プリミティブディスパッチ時は，前回のディスパッチからのアトリビュートの差分を検出し，変化が起こったアトリビュートが MPU 内部にキャッシュされている場合はスヌープキャッシュの機能を利用してそれを無効化する．したがって，MPU はディスパッチされたプリミティブの処理に必要でなおかつ前回のディスパッチから変化したアトリビュートのみを共有バスを通して参照すればよいので，アトリビュートプロセッサと MPU の間の通信量は最適化され，ジオメトリ演算に専念できる．

アトリビュート処理機能を内蔵した専用のジオメトリ演算用プロセッサを開発することも考えられる[6]が，そのような専用の高性能数値演算用プロセッサを新たに開発することは，高性能な浮動小数点演算機能を持つ汎用 MPU が安価に入手できるようになってきたことから，コストおよび開発期間の面で不利である．

5.3 アトリビュートプロセッサの概要

ディスパッチプロセッサによって発行されたプロトコルは Dispatch Bus を通してアトリビュートプロセッサに渡される．アトリビュートプロセッサは decoder によってプロトコルをアトリビュートとプリミティブに分けて，それぞれアトリビュート FIFO およびプリミティブ FIFO（以降それぞれ A-FIFO，P-FIFO と呼ぶ）に格納する．

A-FIFO に入ったアトリビュートは ASL Controller（以降 ASLC と呼ぶ）によって処理される．**図 6** に ASLC の構成を示す．ASLC には最新の ASL の値（Current ASL）と任意の時点での ASL のスナップショットをとる機能があり，そのスナップショット（Locked ASL）は MPU からはメモリマップド I/O として見える．

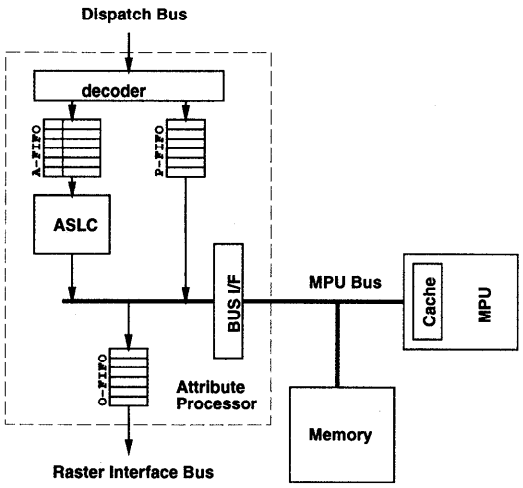


図 5 アトリビュートプロセッサ
Fig. 5 Attribute processor.

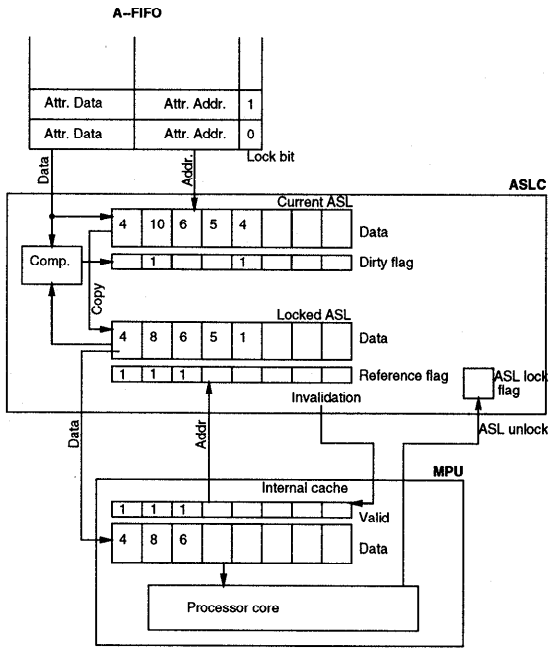


図 6 ASLC の構成
Fig. 6 Structure of ASLC.

基本的な動作は以下のとおりである．

(1) ディスパッチプロセッサはプリミティブを割り当てるのに先立ち，割り当て先の演算ノードに対してASLのlock命令を発行する．ASLCはA-FIFOを通してlock命令を受信するとその時点のASLのスナップショットをLocked ASLにとる．

(2) MPUはP-FIFOからプリミティブを読み込んで処理を行う．このとき参照するASLはLocked ASLである．Locked ASLの参照に際しlock命令が完了していないとき，つまりASLCがスナップショットをとり終えていないときは，その完了を待ってから読み込みを行う．

(3) MPUはプリミティブの処理が終るとASLCに対してLocked ASLの参照が完了したことを告げる．

5.4 ASLCの詳細

ASCLの機能は以下のとおりである．

- Locked ASLの内容とMPUのキャッシュの一貫性をとる．
- A-FIFOからのlock命令を受け，Current ASLの内容の複製をLocked ASLにとる．
- Locked ASLに有効な複製データがあることをMPUに知らせる．

以下順に説明する．

最近の高性能マイクロプロセッサはキャッシュヒットが最高性能を出すための前提となっているので，なるべくASLをキャッシュに保持するように工夫する必要がある．Current ASLとLocked ASLのそれぞれにMPUのキャッシュラインサイズ（たとえば32 byte）ごとに1 bitのフラグを設ける．Current ASLに付随しているフラグ（Dirty flag）はCurrent ASLとLocked ASLのそのブロックの内容が一致していない場合dirtyとなり，一致している場合はcleanとなる．Locked ASLに付随しているタグ（Reference flag）はMPUから参照された場合sharedになる．参照される前の状態ではprivateである．Locked ASLが有効な値を保持していることを示すためASL lock flagというフラグを設ける．ASL lock flagはlockedもしくはunlockedの値をとる．

ALS lock flagの初期状態がunlockedであるとしてASLCの動作を説明をする（図6参照）．

A-FIFOの中にはASL内に格納すべきアトリビュートとそのアトリビュートのASL内でのアドレスが入っている．ASLCはA-FIFOの先頭を読み出し，そのアトリビュートをCurrent ASLの該当するアドレスに格納する．このときLocked ASLの該当アトリビュートの値と比較し，その結果をDirty flagに反映させる．ここで注意すべき点としては，変更によって一度dirtyになったものが，次の変更でまたcleanにもどる可能性があるということである．

ASLCへのlock命令はA-FIFOを通して送られる．lock命令はA-FIFOの中のlock bitで判別される．ASLCはlock命令を受けると，まずASL lock flagを見にゆく．ASL lock flagは1ビットのセマフォであり，すでにlockedのときはunlockedの状態になるまで待つ．次にCurrent ASLのDirty flagを走査して，dirtyなブロックをさがす．dirtyなブロックを見つけると，そのブロックの内容をLocked ASLに複製してDirty flagをcleanにする．また，複製された先のブロックのReference flagがshared場合はMPUの内蔵キャッシュに古い値がキャッシュされている可能性があるので，MPU Busにinvalidate messageを流して，MPUのキャッシュの該当ラインを無効化する．その後Reference flagをprivateにする．逆に，Reference flagがprivateである場合，前回ASLがロックされてから，そのブロックが参照されなかったことが分かるので，invalidate messageを送出する必要はない．Dirty flagがすべてcleanになり，複製およびMPU Busへのinvalidate messageの送出が完了したら，ASL lock flagをlockedにする．

MPUはASL lock flagをポーリングしており，lockedになればLocked ASLを参照してプリミティブの処理を進める．必要なアトリビュートの参照を終えたら，ASL lock flagをunlockedにして次のプリミティブのディスパッチにそなえる．ディスパッチプロセッサはプリミティブの終了を待たずに次のプリミティブおよびASL lock命令を先行して，P-FIFOおよびA-FIFOにつめておくことができる．

6. トレースデータ

実際のアプリケーションがどのようなデータを用いているかを調べるために3種のアプリケーションの実行トレースをとった．

まず，三次元機械設計CADであるCATIAを用い，実行されるPHIGSのエレメントタイプのトレースをとった．エレメントタイプごとの出現率順に並べたものの上位を**表1**に示す．一番左の列は1画面を描画する間の出現回数である．この表よりプリミティブ（GDP DISJOINT POLYLINE3）に対して，6倍の個数のアトリビュートが発行されていることが分かる．一般的にCADではプリミティブ1つに対して多くの

表1 CATIAのトレースデータの統計
Table 1 Statistics of CATIA trace data.

Count	Total words	Ave. words	Element type
5796	11592	2.000	ADD NAMES TO SET
5737	11474	2.000	REMOVE NAMES FROM SET
5683	11366	2.000	SET HIGHLIGHTING INDEX
5672	11344	2.000	SET LINE WIDTH SCALE FACTOR
5672	11344	2.000	SET LINETYPE
5672	11344	2.000	GSE SET FRAME BUFFER PROTECT MASK
5671	285190	50.289	GDP DISJOINT POLYLINE3
143	429	3.000	SET PICK IDENTIFIER

表2 CATIAのトレースデータの一部分
Table 2 A portion of CATIA trace data.

Seq	Element type	Value
+00	ADD NAMES TO SET	00000001
+01	SET HIGHLIGHTING INDEX	00000007
+02	GSE SET FRAME BUFFER PROTECT MASK	FF000000
+03	SET LINE WIDTH SCALE FACTOR	3F800000
+04	SET LINETYPE	00000001
+05	GDP DISJOINT POLYLINE3	.....
+06	REMOVE NAMES FROM SET	00000004
+07	ADD NAMES TO SET	00000001
+08	SET HIGHLIGHTING INDEX	00000007
+09	GSE SET FRAME BUFFER PROTECT MASK	FF000000
+10	SET LINE WIDTH SCALE FACTOR	3F800000
+11	SET LINETYPE	00000001
+12	GDP DISJOINT POLYLINE3	.....
+13	REMOVE NAMES FROM SET	00000004

アトリビュートが付く．

このトレースデータのプリミティブとアトリビュートの並びをもう少し詳しく見てみる．**表2**はトレースデータの一部分を切りだしたものである．

ここで注目すべき点は，1つのプリミティブに対して6個のアトリビュート設定指示がなされているという点と，同種のアトリビュートについては指定されている属性値が同じであるということである．CADアプリケーションではモデリングデータの編集を簡単にするために，たとえ同じアトリビュートの値であってもプリミティブの発行時に再度アトリビュートを設定することがある．

次に，**OpenGL**のデータとしてOPC Viewperf CDRS-03とDX-03の実行トレースをとり，それぞれ出現率順に並べたものの上位を**表3**と**表4**に示す．CDRS-03には7種のテストが含まれるが，そのうち最もアトリビュートの指定が多いNo.4のテストを採用した．また，DX-03はNo.3のテストを採用した．

CDRS-03ではglBegin（GL_TRIANGLE_STRIP）とglEnd()の間にglTexCoord2fv()，glNormal3fv()，glVertex3fv()の列が最低4回から最高2967回繰り返されていた．平均繰返し回数は442.0回である．

DX-03ではglBegin（GL_TRIANGLE_STRIP）のとglEnd()の間にglColor4fv()，glNormal3fv()，glVertex3fv()の列が最低22個から最高100回繰り返されていた．平均繰返し回数は95.8回である．

表3 CDRS-03のトレースデータの統計
Table 3 Statistics of CDRS-03 trace data.

Count	Total words	Element type
31380	125520	glVertex3fv()
31380	125520	glNormal3fv()
31380	94140	glTexCoord2fv()
71	355	glColor4fv()
71	71	glBegin(GL_TRIANGLE_STRIP)
71	71	glEnd()

表4 DX-03のトレースデータの統計
Table 4 Statistics of DX-03 trace data.

Count	Total words	Element type
93536	374144	glVertex3fv()
93536	374144	glNormal3fv()
93536	467680	glColor4fv()
976	976	glBegin(GL_TRIANGLE_STRIP)
976	976	glEnd()

7. シミュレーション

本稿では4.2節で述べたBroadcast方式およびBroadcast-Unicat方式，そして今回5.2節で提案したBroadcast-AP方法についてシミュレーションによる性能評価を行った．ただし，4.2.2項で述べたとおり，Broadcast-Unicast方式をPHIGSに対して適用するのは困難であるため，CDRS-03とDX-03のみに適用した．

7.1 シミュレーションの方式

今回のモデルでは共有メモリ等がないため，シミュレーションの効率を考慮してサイクルシミュレータをC++のTask Libraryで記述した．

アトリビュートおよびプリミティブの種類ごとにメモリの参照コストや計算コストを表として持ち，シミュレーション時にはそのテーブルの内容とプロトコルの内容を参照しながら計算を行うようにした．

7.2 パラメータの設定

シミュレータの設定可能なパラメータのうち，シミュレーションにおいて仮定した主なパラメータの設定内容を以下に列挙する．

MPUは100 MHzで動作し，単精度浮動小数点の積和を毎クロック実行できるものとする．Dispatch Busはバス幅64 bitで50 MHzで動作する．

演算ノードでの処理コストを**表5**に示す．表中のmはプロトコルのワード数である．アトリビュートの入力はMPUのキャッシュを通して行うものとし，キャッシュのfill-inは演算と並行して実行されるので計算コストには現れないものとした．

表5 演算ノードの処理コスト
Table 5 Calculation cost.

Item	Cost (clock)
アトリビュート処理コスト	$10+2(m-1)$☆
プリミティブ入力コスト	m
プリミティブ計算コスト	109 (CATIA 付録 A.1 参照) 157 (CDRS-03, DX-03 付録 A.2 参照)
プリミティブ出力コスト	8/頂点

表6 頻繁に変更されるアトリビュートの設定
Table 6 Frequent attribute set.

Set	Frequent attribute set
A	glNormal, glTexCoord
B	glNormal, glTexCoord, glColor, glMaterial, glEdgeFlag
C	glNormal

また，ディスパッチプロセッサがプリミティブを演算ノードに割り振る際にはラウンドロビン方式を用いるようにした．**OpenGL** では文献 1) に見られるような手法を用いて glBegin (GL_TRIANGLE_STRIP) を 12 頂点ごとに分割してディスパッチ対象を切り替えるようにした．Broadcast-Unicast 方式においては「頻繁に変更されるアトリビュート」の設定を**表 6** のように 3 種類とした．catch-up 動作時には各アトリビュートに 1 ワードのヘッダが付加され，さらに 64 bit 単位に丸め上げられて転送されるものとした．

8. 結　　果

シミュレーションによる実験結果を以下に示す．

8.1 CATIA

CATIA の例では，アトリビュートが多いので 2 方式間で性能に大きな差が出ている (**図 7**)．Broadcast 方式ではアトリビュート処理に並列処理の効果が出ないため，演算ノード数を増加した場合の性能向上率が次第に低下している．

一方，Broadcast-AP 方式では演算ノード数に対して線形な性能向上が得られている．演算ノード数が 28 以上で性能が頭打ちになっている領域では Dispatch Bus が飽和していることがシミュレータ上で観測された．

8.2 CDRS-03

次に CDRS-03 の結果であるが，Broadcast-AP 方式と Broadcast 方式では CATIA のケースと同様な傾向が見られる (**図 8**)．Broadcast-AP 方式および Broadcast-Unicast 方式の設定 A と設定 B では，ある演算ノード数を超えた時点でまったく性能向上が見られなくなる．これらは CATIA の例で見られたように Dispatch Bus の飽和が原因である．

Broadcast-Unicast 方式では catch-up の対象とするアトリビュートの設定の仕方で結果が大きく異なる．表 3 から分かるように，CDRS-03 では glTexCoord と glNormal のみが「頻繁に変更されるアトリビュート」であるので，それらのみを catch-up するようにした設定 A が最も効率が良い．しかしながら，その場合においても catch-up のためにプロセッサの処理を要するため Broadcast-AP 方式に比べて，プロセッサあたりの処理能力の面で劣っている．また，バスの飽和領域においても catch-up のための余分なバス転送が必要なため Broacast-AP 方式に比べて性能が劣っていることが分かる．

Broadcast-Unicast 方式の設定 B では描画中に変更されることのない glMaterial や glEdgeFlag 等のアトリビュートも catch-up の対象としているため，演算ノードあたりの性能が設定 A に比べさらに低下してお

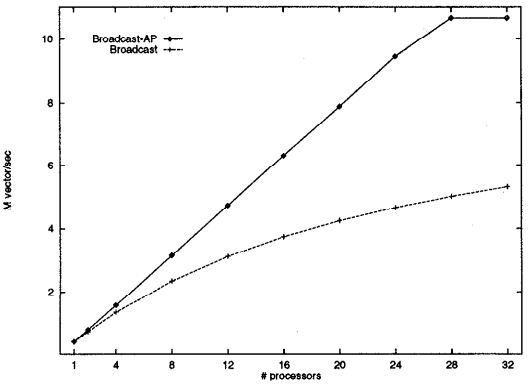


図7 CATIA のトレースを用いたシミュレーション結果
Fig. 7 Simulation result of CATIA.

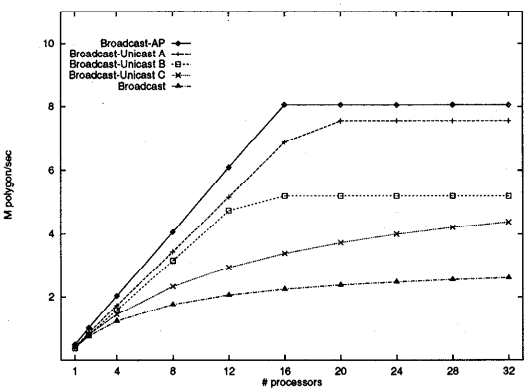


図8 CDRS-03 のトレースを用いたシミュレーション結果
Fig. 8 Simulation result of CDRS-03.

☆ 算出根拠は 4.2.1 項を参照のこと．また Broadcast-AP 方式の場合はアトリビュートプロセッサ内部で処理されるので見かけ上のコストは 0 である．

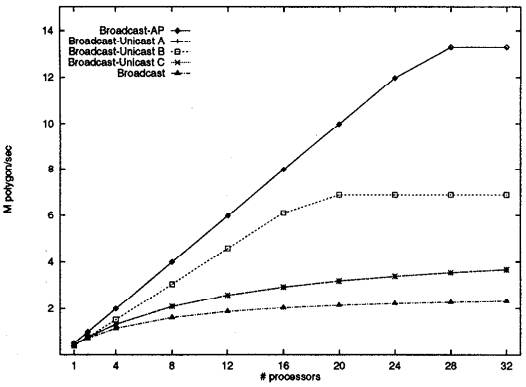


図9 DX-03のトレースを用いたシミュレーション結果
Fig. 9 Simulation result of DX-03.

り，バスの飽和領域においても性能低下が見られる．

Broadcast-Unicast方式の設定Cでは「頻繁に変更されるアトリビュート」であるglTexCoordがブロードキャストされているため，プロセッサの処理能力を浪費しており性能が線形に向上しなくなる．特性としてはBroadcast方式に近い．

8.3 DX-03

DX-03のシミュレーション結果を図**9**に示す．

Broadcast-Unicast方式の中では設定Bが最も高い性能を出している．設定Aは大幅に性能低下し，設定Cと同一の性能にとどまっている．設定Bは無駄なcatch-upを行ってはいるが，「頻繁に変更されるアトリビュート」であるglNormalとglColorが両者ともcatch-upの対象になっており，アトリビュートの全放送が起きていない．それに対し，設定AではglColorが全放送され，プロセッサの処理能力を大きく浪費していることが性能低下の原因である．

このようにBroadcast-Unicast方式では「頻繁に変更されるアトリビュート」の設定をアプリケーションの特性に合わせて最適に選択しないと性能が低下してしまう．異なる特性のアプリケーションを使用する場合や，1つのアプリケーションの中で特性が変わるような場合には，つねに最適な設定を選択しておくことは難しい．

一方，本提案のBroadcast-AP方式ではDispatch Busのデータ転送，および演算ノードでのアトリビュート処理がつねに最適に行われるので，アプリケーションの特性によらずつねに最大性能を引き出すことができる．

9. おわりに

本稿では汎用プロセッサにアトリビュート処理専用のアトリビュートプロセッサを組み合わせることで，並列三次元グラフィクスシステムのジオメトリ演算部を構築することを提案した．

アトリビュートプロセッサの導入により，ディスパッチバスの転送量とMPUの処理を最適化するとともに，スヌープキャッシュを利用してASLのデータのうち現在の処理に必要でかつ前回から変化した部分のみをMPUのキャッシュに送り込むことを可能とした．

また，アプリケーションのトレースデータを用いたシミュレーションにより，本方式が従来の方式に比べてアプリケーションの特性によらず優位であることを示した．

最近の動向としては，Toolkitを用いてAPIの呼び出しを減らすことによる最適化の手法も登場してきたが，機械設計CADのようなアプリケーションではそれらの手法をすぐに取り入れることは難しいと思われる．今後**OpenGL**でのCADが出回り出したところで，それらを含めたさらに多くのアプリケーションでの評価を行いたい．

謝辞 本研究に関しご指導と貴重なご助言をいただいた清水和哉氏，小野眞氏をはじめとする日本アイ・ビー・エム（株）の諸氏に感謝します．

参考文献

1) Akeley, K.: Reality Engine Graphics, *Proc. SIGGRAPH '93*, pp.109–116, ACM, Addison-Wesley (1993).
2) Horning, R., et al.: System Design for a Low Cost PA-RISC Desktop Workstation, *Proc. COMPCON91 SPRING*, pp.208–213, IEEE (1991).
3) ISO: ISO/IEC 9592-1:1989 (E), Information processing systems – Computer Graphics – Programmers Hierarchical Interactive Graphics System (PHIGS), Part 1 – functional description (1989).
4) Kirk, D. and Voorhies, D.: The Rendering Architecture of the DN10000VS, *ACM Computer Graphics*, Vol.24, No.4, pp.299–308 (1990).
5) 松本 尚，川瀬 桂，森山孝男：PHIGSの構造体を処理するジオメトリ演算部のマルチプロセッサ上での実行効率評価，情報処理学会論文誌，Vol.34, No.4, pp.732–742 (1993).
6) 松本 尚，川瀬 桂，森山孝男：PHIGSのジオメトリ演算部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 (1994).
7) Molnar, S., Cox, M., Ellsworth, D. and Fuchs, H.: A Sorting Classification of Parallel Rendering, *IEEE Computer Graphics and Applications*, Vol.14, No.4, pp.23–32 (1994).

8) Segal, M. and Akeley, K.: *The OpenGL Graphics System: A Specification (Version 1.1)* (1997).
9) Standard Performance Evaluation Corporation: Graphics Performance Characterization Group (http://www.specbench.org/gpc/).
10) Toroborg, J.G.: A Parallel Architecture for Graphics Arithmetic Operations, *Proc. SIGGRAPH '87*, pp.197–204, ACM, Addison-Wesley (1987).

付　　録

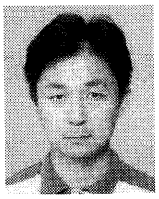
A.1 CATIA の計算時間の内訳

処理内容	時間（clock）
座標変換（MC → WC）	20
座標変換（WC → NPC）	42
クリッピングチェック	12
座標変換（NPC → DC）	15
後処理	20
合計	109

A.2 CDRS-03，DX-03 の計算時間の内訳

処理内容	時間（clock）
座標変換（Model → Eye）	20
照度計算	48
座標変換（Eye → Clip）	22
クリッピングチェック	12
座標変換（Clip → Window）	35
後処理	20
合計	157

(平成 9 年 9 月 1 日受付)
(平成 10 年 12 月 7 日採録)



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．三次元グラフィクスシステムの研究に従事．

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，グラフィクスの並列処理の研究に従事．