

Pentium® III プロセッサのパフォーマンスを最大限に活かす 3D ソフトウェア・スタック・アーキテクチャ

Paul M. Zagacki, Deep Buch,
Emile Hsieh, Daniel Melaku, Vladimir Pentkovski:
インテル社、マイクロプロセッサ・プロダクツ・グループ(MPG)
Hsien-Hsin Lee, ミシガン大学(アナーバー)、EECS-ACAL

インデックス・ワード: 3D、グラフィックス、パフォーマンス、Pentium® III、ドライバ

摘要

この記事では、従来の 3D グラフィックス・ソフトウェア・スタック(アプリケーション、ライブラリ、グラフィックス・ドライバ)に対するアーキテクチャ上の主要な改善がもたらす利点について解説します。ここでは、新たな 3D パイプライン・アーキテクチャを提示するのではなく、このアーキテクチャを実際に使用する際の効率を高めることに重点を置いています。3D ソフトウェア・スタックの特定の層で最適化を図ることにより、Pentium® III プロセッサおよびインターネット・ストリーミング SIMD 拡張命令でのパフォーマンスが向上することは明らかです。しかし、この記事では、3D ソフトウェア・スタックのカーネル層の最適化によって、アプリケーション・レベルで Pentium III プロセッサの持つ潜在能力を最大限に引き出す方法を示すことが目的です。ケース・スタディとして、ジオメトリ・パイプラインの実装、Pentium III アーキテクチャ・チームの開発による Architecture Geometry Engine (ArchGE)、3D シーン・マネージャを取り上げて説明します。この記事では、計測結果に基づくパフォーマンス・データを紹介し、アーキテクチャ構造の強化がもたらす利点を具体的に説明します。

はじめに

従来の 3D パイプラインでジオメトリや照明の計算に必要なアルゴリズムを適用するにはコストがきわめて高く、そのために 3D の利用はハイエンド・ワークステーション分野だけに限られていました。図 1 は、複数の主要なコンポーネントで構成される典型的な 3D パイプライン構造を示しています。まず、ジオメトリ/照明の計算がシステムのホスト・プロセッサで行われます¹。アプリケーションの

¹ この記事は、3D グラフィックスに関する基礎知識を前提に記述されています。この分野の詳細については[1]を参照してください。

3D モデルは独自のバーチャル・ワールドへと変換され、照明に関する情報が生成されます。これらの計算は、一般的な 3D ライブラリ(OpenGL*、Microsoft Direct3D*など)がアプリケーションによって行われます。ここで生成された情報は、別のコンポーネントである 3D グラフィックス・コントローラに渡され、コンピュータの画面に表示できるようラスタ化(画像データの 2D ピクセル表示への変換)されます。高性能 3D エンジン開発においては、この 2 つのコンポーネントのバランスを取ることが最も重要な課題となります。

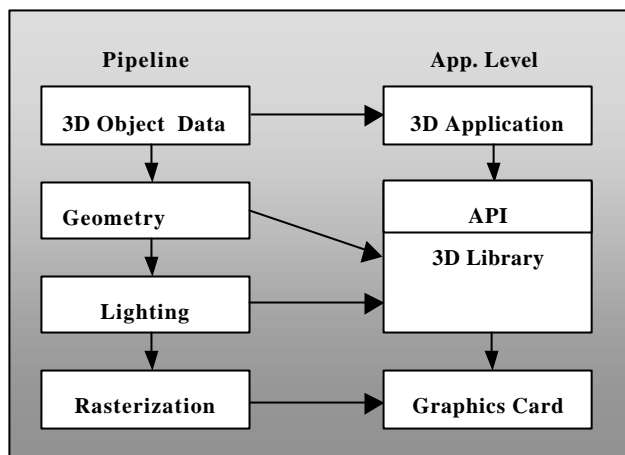


図 1: 典型的な 3D パイプライン構造とそれに関連するアプリケーション・レベル・コンポーネント

グラフィックス・コントローラのパフォーマンスを高めるには、ジオメトリ情報をカードに伝えるための 3D ライブラリのパフォーマンスも向上させなければ、双方の機能のバランスを維持できません。インターネット・ストリーミング SIMD 拡張命令は、ジオメトリ/照明の計算での効率とスループットを高め、システム・パフォーマンスを向上させることに配慮した設計がなされています。しかし、

Pentium® III プロセッサで最高の 3D パフォーマンスを達成するには、カーネル・レベルのコンポーネントを最適化するだけでは十分とは言えません。

Pentium III プロセッサで 3D のパフォーマンスを最大限に引き出し、使用モデルを提示することを目的として、我々は、3D ライブラリ層(図 1)の部分に相当する Architecture Geometry Engine (ArchGE)を開発しました。ArchGEには、ジオメトリ/照明アルゴリズムのパフォーマンスを高めるインターネット・ストリーミング SIMD 拡張命令が組み込まれているだけでなく、汎用 3D ライブラリの機能を拡張する 2 つのアーキテクチャが追加されています。

1. マルチプリミティブ API エクステンション
2. 「オンライン」ドライバ・モデル

インターネット・ストリーミング SIMD 拡張命令の利用による x87 コードからのカーネル・レベルの高速化の度合いは、光源のタイプと数(無限、ローカル、鏡面反射など)、クリッピングの量、プリミティブの種類、その他の内容や変数²によって異なります。一般的な変換/照明カーネルで使用すると、1 つの光源の処理につき、最適化された x87 浮動小数点コードに比べてカーネル・レベルで 1.4~2.0 倍のパフォーマンス向上が見込まれます。複数の光源やより大きなサイズのプリミティブを扱う場合は、最適化された x87 コードに比べて 2.0 倍以上のパフォーマンス向上が見込まれます。さらに、最適化を極めたカスタム・エンジンを使って計測したところ、カーネル・レベルで 2.5~2.75 倍のパフォーマンス結果が出ています。しかし、こうしたカーネル・レベルのパフォーマンスも、ユーザにとって重要な基準となるアプリケーション・レベルのパフォーマンスにはほんのわずかしが反映されていないのが現状です。アプリケーションのパフォーマンス向上は、*アムダールの法則*に則って、通常はカーネル・レベルの高速化ほどの数字が達成されないため、ソフトウェア・スタックで別の最適化を施す必要があります(参考資料[2])。

マルチプリミティブ API エクステンションを利用すると、アプリケーションでは、同じレンダリング・ステート情報を持つプリミティブ(ストライプ、ファン、頂点バッファなど)をまとめて 1 回の API 呼び出しでライブラリに渡すことができます。マルチプリミティブの利用による最適化では、従来の(1 回の呼び出しで 1 個ずつプリミティブを処理する)手法に比べて、17%~40%のパフォーマンス向上が見られます。このパフォーマンス向上は、関数呼び出しのオーバーヘッドを減らし、多数の頂点データを処理する際の

² ここでのカーネル・レベルのパフォーマンスの定義には、変換、カリング、鏡面反射、SIMD フォーマットからグラフィックス・コントローラの頂点データ編成への置き換え、処理済みの頂点データの AGP メモリへの格納といった処理が含まれます。

プリフェッチに伴うコストを軽減することによって達成しています。「スタートアップ」コストでの時間が短縮されると、より多くの時間を重要なジオメトリや照明の計算に充てることができます。また、これらの計算もインターネット・ストリーミング SIMD 拡張命令の利用により高速化が図れます。

ArchGE に取り入れられている 2 つめの拡張機能は「オンライン」ドライバ(OLD)です。OLD のメカニズムでは、ジオメトリ・パイプラインに対し、グラフィックス・コントローラのドライバが、変換/照明処理された頂点データを格納する最終的なターゲット・バッファを提供します。一般に、汎用 API では、すべての頂点データについて変換/照明処理が行われた後に、ライブラリによる制御のもとにバッファに格納されます。バッファの準備が整い情報を安全に渡せる状態になると、ライブラリはグラフィックス・コントローラに通知します。バッファが準備された後に、デバイス・ドライバでは、ライブラリのバッファからコントローラのメモリ(通常は AGP メモリ内の領域)へデータをコピーしなければなりません。この方法には 3 つの問題があります。第 1 に、プロセッサ・バスが、計算処理ではなく、大きなブロックの変換/照明処理済みの頂点データをライブラリとグラフィックス・メモリ間で移動することに占有されるため、リソースの使用効率の低下につながります。第 2 に、このプロセスでは、通常キャッシュの書き換えが頻繁に発生します(変更されたラインを小さく高速なキャッシュ・レベルから大きく低速なキャッシュ・レベルまたはメモリに移動するため)。このため、プロセッサ・バスでのロードの負荷が高くなり、キャッシュ階層やプリフェッチ命令の効率が下がり、ジオメトリ計算のスループットが低下します。最後に、従来型のデバイス・ドライバではデータのコピーやフォーマット処理が多く、ドライバを実行するのに OLD アプローチの 10 倍もの時間がかかります。この時間はデータの移動に使われるものであり、意味のある計算に使われるわけではありません。OLD ならば、こうした問題を解決できます。グラフィックス・パイプラインでは、計算を行いながら、変換/照明処理済みの頂点データをただちにグラフィックス・コントローラのローカル・メモリに送ります。頂点データを「直送」することにより、ジオメトリ計算や照明処理の計算(大量の計算を必要とする)と、結果の保存(バスの使用率が高い)の並行実行性が高まります。この並行実行性の向上によって、標準的なオフライン・ドライバを使用した場合に比べてアプリケーション・レベルで 30%~80%の高速化を達成します。

この後、次のような 3D ソフトウェア・スタックの最適化手法(図 1 参照)を紹介し、こうした最適化がアプリケーション・レベルのパフォーマンスにどのような効果をもたらすかについて説明します。

- 3D ライブラリ/API 層: 1 つのコマンドによる複数のプリミティブの一括処理、シングルパス・ジオメトリ・パイプラインとマルチパス・パイプラインの比較

- デバイス・ドライバグラフィックス・コントローラ層: オンライン・ドライバによる処理済みの頂点データの転送
- 3D アプリケーション層: オブジェクト・レベルのクリッピング、レンダリング・ステートの分類

こうしたすべての最適化手法を取り入れ、ArchGE を使用すれば、同様の構成のマシンで同じワークロードを実行した場合に、最適化された x87 浮動小数点パイプラインに比べてアプリケーション・レベルで 2 倍近くの高速度を達成できます。既存の 3D ライブラリやデバイス・ドライバでもリアルタイム 3D グラフィックスで必要とされる計算を行うことができますが、そのような場合にも、ここで紹介する手法を取り入れればアプリケーション全体のパフォーマンスが格段に向上します。

アムダールの法則

アムダールの法則は、カーネル・レベルの高速度がどの程度アプリケーション・レベルのパフォーマンスに反映されるかの基準を示しています。簡単に言うと(3D パイプラインを例に取ると)、アムダールの法則によると、変換/照明処理の最適化がもたらすアプリケーション・レベルの高速度率は、最適化されたコードをソフトウェアで実行する時間の比率に左右されるということです

図 2 に、アムダールの法則を 3D プログラムに適用した場合の例を示します。ここでは、3D アプリケーション・スタック内の変換/照明処理で、Pentium® III プロセッサのインターネット・ストリーミング SIMD 拡張命令を利用しています。カーネル・レベルでの高速度を 2 倍とし、これらの 3D ジオメトリ・ルーチンの実行に要する時間の比率を 50% とします。

$$\begin{aligned}
 \text{Speedup}_{\text{overall}} &= \frac{\text{ExecutionTime}_{\text{old}}}{\text{ExecutionTime}_{\text{new}}} \\
 &= \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} \\
 &= \frac{1}{(1 - .5) + \frac{.5}{2.0}} = 1.33x \text{ Speedup}
 \end{aligned}$$

図 2: アムダールの法則による、変換/照明処理の最適化から得られるアプリケーション・レベルのパフォーマンス向上

図 2 で言うと、変換/照明処理(図 1 の 3D ライブラリ層)のパフォーマンスが 2 倍に向上しても、このコードを実行する時間が全体の 50% に過ぎなければ、アプリケーション全体としての高速度は 1.33 倍になります。1.33 倍のパフォーマンス向上は悪くない数字ですが、カーネル・レベルでの 2 倍の高速度からはかけ離れています。このことからわかるように、Pentium III プロセッサのパフォーマンスを最大限に引き出すには、変換/照明ルーチンをインターネット・ストリーミング SIMD 拡張命令に移植するだけでなく、アプリケーション・スタックの別の部分でも最適化を図る必要があります。

この後のセクションで取り上げる数々の最適化手法は、「強化された」コード・セグメントを実行する時間の比率を増やすことによって、アムダールの法則が示すパフォーマンスの制約を軽減することに主眼を置いています。

3D ライブラリと API の最適化

3次元オブジェクトを操作し、2D のモニタ画面に表示するためのさまざまな処理を行う 3D ライブラリやカスタム・エンジンは数多く存在します。一般に、既存の 3D ライブラリのアーキテクチャでは、Pentium® III のようなプロセッサのパフォーマンスをアプリケーションで十分に活かすきれない場合があります。

マルチパス頂点処理

現在の 3D ライブラリは、通常、取り込まれた頂点情報をマルチパス構造で処理します。マルチパス・ジオメトリ・パイプラインでは、複数の別々のループを通じて頂点データを処理します。各ループでは、パイプラインに取り込まれた頂点データに対し、変換やバックフェイス・カリング(表面が視点に向いていない三角形を取り除くプロセス)を行い、さらに照明などの処理を行います(図 3 の MP の箇所を参照)。このアプローチには、2つの問題があります。

1. キャッシュ管理コードが複雑
2. メモリと演算命令をインターリーブする基本コード・ブロックのサイズが小さい

マルチパス処理はキャッシュ管理に大きく依存しており、場合によっては、変換や照明処理のための頂点データをキャッシュ・サイズの複数のブロックに分割します。パイプラインでは、変換フェーズの間に生じたキャッシュ・ミスすべてを吸収した後に、カリングや照明でのミスを処理する必要があります(L2 キャッシュに格納されたデータにアクセスする場合にもペナルティを抑えられます)。

マルチパス処理の場合、キャッシュの使用効率を直接管理するための特別な配慮が必要であるだけでなく、基本コード・ブロックのサイズが小さいために、メモリ・アクセスと演算命令を効率的にインターリーブすることが難しくなります。理想を言えば、同じループ内で、メモリのリードオフ/レイテンシは、計算に要する時間とバランスを取る

必要があります。実際には、マルチパス・パイプでは、ループの反復ごとに、ロード/ストアの要求とのバランスをとるのに十分な計算が行われるケースはほとんどありません。このため、主要なコード・セクションでは大量のメモリを必要とし、プロセッサ・コアのクロックが上がってもそれに見合うパフォーマンスの向上が実現できません(一般にシステム・メモリのパフォーマンスはプロセッサのパフォーマンスに比べて低いため)。

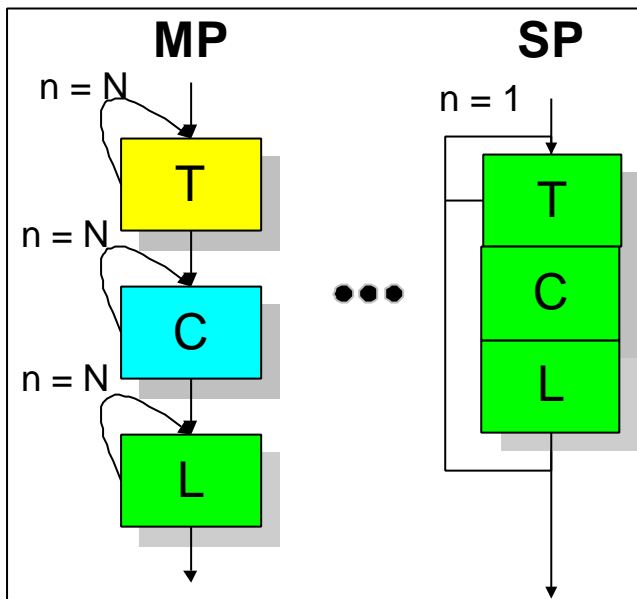


図 3: マルチパス(MP)とシングルパス(SP) ジオメトリ・パイプの比較
(T=変換、C=バックフェイス・カリング、L=照明、n=反復回数、N=頂点の数)

シングルパス頂点処理

こうしたマルチパス・パイプライン構造の問題を回避するために、ArchGEでは図3のようなシングルパス(SP)手法を取り入れています。このアプローチでの主な違いは、複数の頂点データの変換、カリング、照明、グラフィックス・コントローラのメモリへの書き込みまでを、1つのループで処理するという点です³。このため、キャッシュの使用率を細かく管理するためのコードを追加する必要がなくなり、使用できる基本ブロックのサイズもかなり大きくなります。インターネット・ストリーミング SIMD 拡張命令に含まれる PREFETCH 命令を使うと、パイプで実行される計算でメモリ・レイテンシをカバーできるようになります。現在の変換処理を行う間に、次のループ反復での変換に必要なデータ(x、y、zの座標情報)がキャッシュにロードされます。照明の計算で使われる法線やテクスチャの座標値にも、同じ方法論が適用されます。

³ ArchGEの場合、実際には4つの頂点データを同時に処理します。この数値は、Pentium® III プロセッサのインターネット・ストリーミング SIMD 拡張命令のレジスタの幅と一致しています。

ArchGEでは、PREFETCH 命令の使用と大きな基本ブロックの利用により、メモリ・アクセスと計算の並行実行性を高めることができます。研究の結果、シングルパス・パイプラインは、この並行実行性の向上により、最適化されたマルチパス・パイプラインに比べても 20%~30% 高速であることがわかっています。ArchGE パイプはマルチパス・パイプに比べて、より多くの計算を処理できるため、プロセッサのクロックに見合ったパフォーマンス向上が期待できます。

マルチプリミティブ API エクステンション

頂点データをどのようにジオメトリ・パイプに渡すかという問題は、データをどのように処理するかと同じくらい重要です。ほとんどの 3D ライブラリは、アプリケーション・プログラミング・インターフェース(API⁴)を通じて、頂点データをアプリケーションに渡すためのさまざまな方法をサポートしています。頂点データは、アプリケーションによってプリミティブにグループ化され(通常はトライアングル・ベース)、ライブラリに渡されて変換や照明の処理が行われ、その後、グラフィックス・コントローラでラスタ処理が行われます。OpenGL*などの場合は、個々の点という単純なものから複雑な四辺形まで、10種類のプリミティブをサポートしています(参考資料[4])。大抵のグラフィックス・コントローラはトライアングル・ベースのフォーマットをサポートしているため、現在はこのプリミティブ・タイプが最も一般的に使われています。よく使われている3種類のプリミティブを図4に示します。

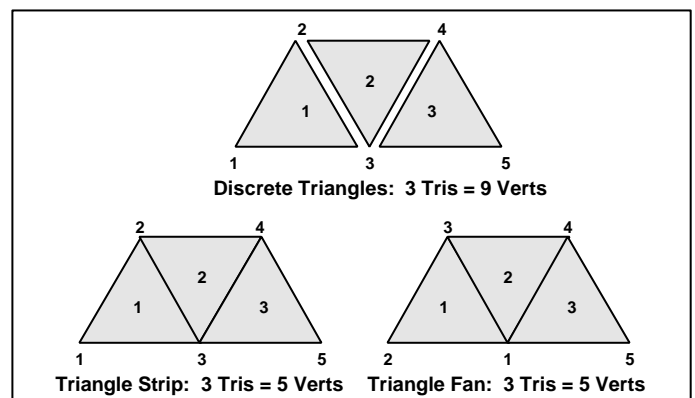


図 4: トライアングル・プリミティブの種類と、描画に必要な頂点の数

既存のグラフィックス・ライブラリのほとんどでは、アプリケーションから渡されるプリミティブは1度の処理につき1個だけに限定されます。つまり、アプリケーションで

⁴ API は、ライブラリとのやり取りのためにプログラムが呼び出せる関数をまとめたものです。

は、トライアングル・ストライプ、トライアングル・ファン、インデックス・リスト(あるいはライブラリがサポートするその他のプリミティブ)いずれの場合でも、1回の関数呼び出しにつき1つずつ処理することになります。ほとんどのプリミティブは比較的少数の頂点で構成されているため、個々のプリミティブを処理する関数を呼び出すだけで、かなりのオーバーヘッドが生じます⁵。

1個のプリミティブを処理する際に生じるオーバーヘッドの要因は、ジオメトリ計算以外の追加命令と、メモリ・パイプの分断という2つのパートに分けられます。追加作業が発生する要因となるのは、パラメータのプッシュ/ポップに必要な命令とサイクルの追加、変換マトリクスおよび照明情報のセットアップ、パラメータの確認といった処理があります。よく使われているライブラリのいくつかでこれを計測したところ、1回の呼び出しにつきおよそ1,000サイクルという結果が出ています。1回の呼び出しで少数の頂点データを処理する場合、オーバーヘッドはかなりの量になります。

理念上は、Pentium III プロセッサのインターネット・ストリーミング SIMD 拡張命令を利用すると、メモリ・アクセスと計算をほぼ同時に実行できます。PREFETCH 命令を使ってメモリ・アクセスを完全にパイプライン化することにより、これを実現します(図5の下段)。

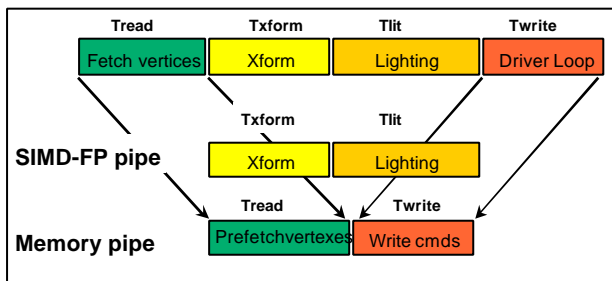


図5: 3Dパイプライン内でメモリと計算の並行実行性が向上(理想的なケース)

図5の各ボックスは、簡素化された3Dパイプラインでの処理時間ブロックを表しています。この図の一番上の段は従来型のパイプを表しており、メモリ・アクセスと計算処理が直列に行われます(古いプロセッサ・ファミリでもメモリと計算の並行処理がわずかながら可能であるためパイプラインは多少簡素化されています)。図5の下半分は、Pentium III プロセッサの PREFETCH 命令とストリーミング・ストア命令を利用した場合のパイプを表しています。しかし、現実には、プリミティブの境界で「メモリ・パイ

⁵ この所見は、いくつかの既存のゲームおよびベンチマークで実験した結果に基づいています。

プの分断」と呼ばれる現象が発生するため、総合すると理論上の処理時間よりもいくぶん長くなります(参考資料[8])。たとえば、プリミティブの最初のいくつかの頂点データをプリフェッチする際に、データを待ちながら計算するため「スタートアップ・コスト」が生じる場合があります。ネスト・ループの場合、内側のループの最後に来る反復と、その外側にあるループの次の反復の間でメモリ・パイプの分断が生じることがあります。

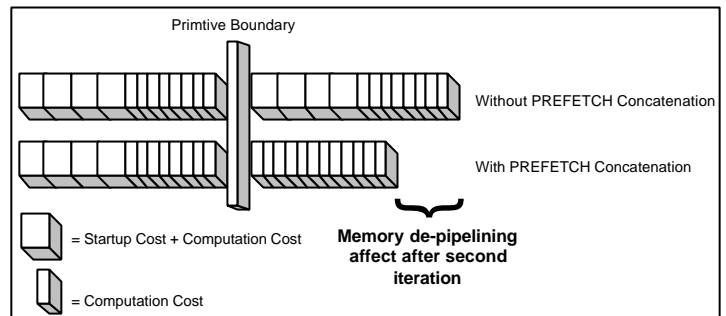


図6: 2つの短いプリミティブの間で起こるメモリ・パイプの分断

図6は、メモリ・パイプの分断による影響を図で表したものです。この図で、大きなボックスは、通常の計算を行う時間と最初の PREFETCH 命令がデータをキャッシュに返すまでにかかる時間を足した量を表しています(このため、ジオメトリ処理で最初のいくつかの反復では遅延が生じます)。小さなボックスは、定常状態において計算を完了するのに必要な時間の量を表しています。

メモリ・パイプの分断によるパフォーマンスへの影響を緩和する有効な手法として、「プリフェッチの連結」が挙げられます。連結を使うと、PREFETCH 命令で次の外部ループの反復を「先読み」することにより、内側のループとそれに関連する外側のループの間に生じる実行パイプラインの隙間を埋めることができます。図6の例では、ジオメトリ・パイプラインがプリミティブの境界をまたいで「先読み」しています。しかし、APIの都合で呼び出しのたびに1個ずつプリミティブを処理するような場合、メモリのスタートアップ・コストを相殺しようとしても、個々に渡されるプリミティブの境界でこの手法を使うことはできません。特に、比較的少ない数(100個未満)の頂点で構成されるプリミティブを扱う場合には、このことが特に重要です。

アプリケーションが API 層を介して各プリミティブを呼び出すことの影響と、メモリ・パイプの分断による影響を軽減するために、ArchGE ではプリミティブをジオメトリ・エンジンに渡す方法としてマルチプリミティブ手法を採用しています。この手法では、アプリケーションは、1回の呼び出しでプリミティブのリストとそれに対応するプ

MultiPrimitive Application Speedup vs. Primitive Length

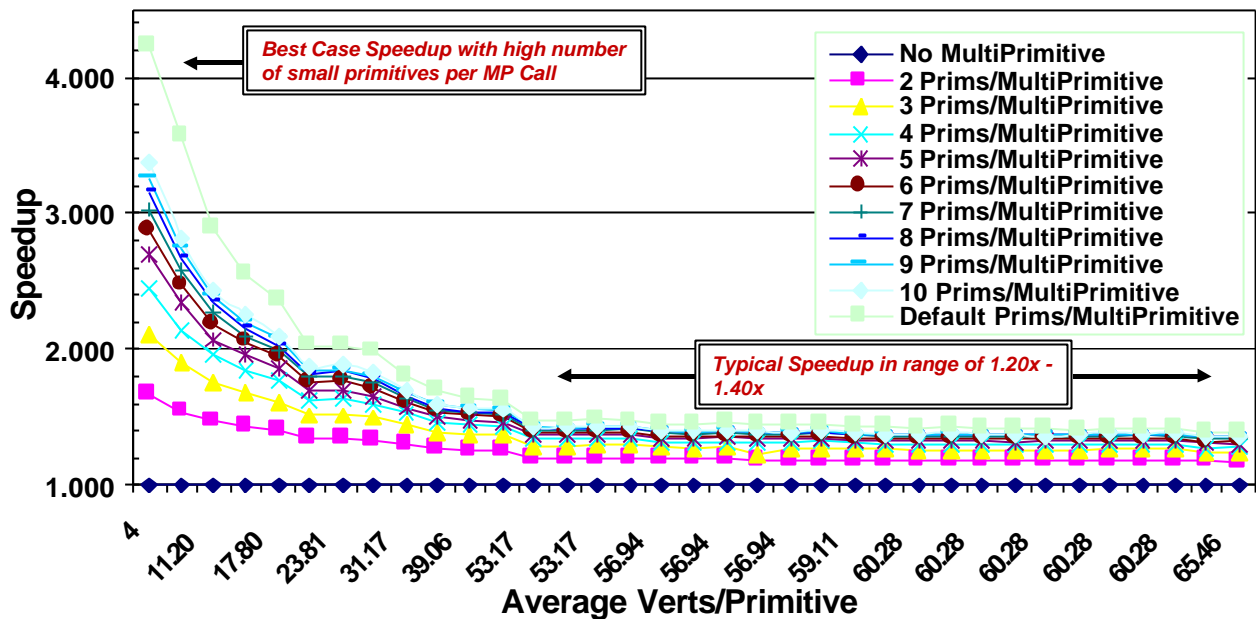


図7: プリミティブのサイズとバッチ処理できるプリミティブの数に対する、マルチプリミティブの実行速度への影響(x軸上の数値は非線形)

リミティブの長さのリストを ArchGE に渡します。ArchGE/Scene Manager ソフトウェア・スタックでマルチプリミティブを利用することにより、アプリケーション・レベルで 40% のパフォーマンス向上を達成します。図 7 は、プリミティブのサイズとバッチ処理できるプリミティブの数に対する、マルチプリミティブの影響を詳細に表したものです。マルチプリミティブを利用して、一度の呼び出しでサイズの小さいプリミティブを多数まとめて処理すると、ベスト・ケースではシングル・プリミティブ API の 400% 以上ものパフォーマンスを達成します。グラフの最も低い部分については、サイズが非常に大きいプリミティブ (65 ~ 120 個の頂点を含むプリミティブ) を 2 個まとめて処理するだけでも、マルチプリミティブの効果により、アプリケーション・レベルで 20% のパフォーマンス向上が見られます。既存のゲームやベンチマークを研究した結果、標準的なワークロードでこの機能を利用することにより、アプリケーション・レベルで 30% ~ 40% の高速化が見込めることがわかっています。

図 7 は、各プリミティブに含まれる頂点の最大数とバッチ処理するプリミティブの数という 2 つの変数をさまざまな数値で実験した結果をグラフにしたものです。この図では、プリミティブに含まれる頂点の最大数が増えるに従って、「プリミティブごとの頂点の平均数」の間隔(x軸の目盛)は必ずしも等間隔ではありません。この実験で使用したシーンの本来の構造では、プリミティブに含まれる頂点の数の平均値は 66 個です。プリミティブ・サイズが本来の平均値に近づくにつれて(図 7 で言うと x 軸に沿って右に移動するにつれて)、x 軸のスケールが圧縮されていますが、

これは、プリミティブごとの頂点の数をこれ以上増加しても最終的な平均値にはなんら影響がないためです。

デバイス・ドライバの最適化

従来型の 3D ライブラリには、オフライン・ドライバ・モデルを採用しているものがあります。このモデルでは、頂点データは変換・照明処理が行われた後、キャッシュ内の利用可能な領域に一時的に格納されます。その後、ジオメトリ・エンジンがグラフィックス・コントローラのドライバに対してバッファの準備が整ったことを知らせると、デバイス・ドライバは、一時的なキャッシュ領域に置かれた情報を、ローカル・メモリ・コントローラのメモリ(通常は書き込みの連結が可能でキャッシュ不可能なメモリ領域を使用⁶)へと転送し始めます。図 5(前のセクションを参照)の一番上の段を見るとわかるように、この処理方法では、メモリ・アクセスと計算処理を並行して行うには支障があります。この図では、従来型のドライバで要する時間は Driver Loop ブロックで表されています。

従来型のジオメトリ・パイプラインでの並行実行性が低いことに加え、オフライン・ドライバ・モデルでは外部バス(CPU コアとメモリをつなぐバス)の負荷が高くなる傾向があります。プリミティブの頂点に関するすべての情報(変換/照明処理済みの情報)とコマンドを格納するまでの過程

⁶ Pentium® III プロセッサのメモリ・タイプの定義については、参考資料[9]を参照してください。

で、キャッシュ・ラインの内容が何度も更新されます。この場合、アプリケーションや変換/照明ルーチンで慎重にキャッシュ効率を高める努力をしても、予期しないデータがキャッシュに書き込まれることにより、すべてが無効になってしまいます。アプリケーションでは、変更されているキャッシュ・ラインの内容を書き出してから、計算に必要なデータが格納されている新しいキャッシュ・ラインにアクセスしなければなりません。キャッシュ・ラインの頻繁な書き換えは内部バスと外部バスの負荷を増やし、アルゴリズムのパフォーマンスを大幅に低下させます。

ArchGE では、別のドライバ・モデルを実装することにより、こうした並行実行性の低さと非効率なキャッシュ使用という問題を一挙に解決します。従来型のドライバ・モデルとは異なり、オンライン・ドライバ・モデル(OLD)は変換/照明処理済みの頂点データを格納するための大きな一時バッファ領域を必要としません。OLD の場合、頂点データの変換/照明処理(独自のシングルバス・パイプにより一度に 4 つのデータを処理)が完了すると、グラフィックス・コントローラが提示するメモリ領域に直接データを格納します。ArchGE には、市販の高性能グラフィックス・コントローラに対応する OLD メカニズムが実装されています。ArchGE のジオメトリ・パイプラインでは、一度に 4 つの頂点データを変換し、照明(表示される箇所のみ)処理を行い、そのデータを直接グラフィックス・コントローラのメモリに格納します。頂点データとコマンド情報を小さなブロックごとに直接メモリに格納するため、変換や照明の計算処理との並行実行性が向上するとともに、キャッシュ領域の頻繁な書き換えによる影響を軽減できます。

デルがもたらすアプリケーション・レベルでの高速化です。図 8 は、オンライン・ドライバ・モデルの利用によるパフォーマンス向上の見込みを示しています。このデータは、複雑さの異なる 3 つのシーンを使って計測した結果に基づくものです。グラフ上の Scene 1 では、ArchGE でオフライン・ドライバ・モデルを使って実行した場合に比べて 1.8 倍の高速化を達成しています。パフォーマンスが大きく向上した要因としては、このシーンのデータ構造が大量のジオメトリ計算を必要とする(1 フレームにつきおよそ 82,000 個の頂点データの処理が必要)という点と、グラフィックス・コントローラのフィル・レートの制約を受けない(大部分が小さなトライアングルで構成されているため)という点が挙げられます。つまり、Scene 1 はプロセッサの能力と利用できるバス帯域幅に依存する部分が多いこととなります。このワークロードでは、外的な制約がなければ、OLD の利用によりピーク性能に近い数値を達成できます。

2 つめの Scene 2 は Scene 1 をクローズアップしたシーンであり、グラフィックス・コントローラのフィル・レートの影響をやや受けやすい構成です。従来型のドライバ・モデルに比べて 1.3 倍の高速化が見られ、フィル・レートの影響により(Scene 1 での 1.8 倍と比較すると)高速化の度合いが少なくなっています。3 つめの Scene 3 では、オフライン・ドライバの 1.17 倍のパフォーマンス向上が見られます。Scene 3 は ArchGE の利用による効果が最も少ないタイプのコンテンツであると言えます。ジオメトリ計算の量が比較的少なく、フィル・レートの制約が大きいため、グラフィックス・コントローラがパフォーマンスのボトルネックになっています。

図 8 を見ると、オンライン・ドライバ・モデルの実装により得られるパフォーマンス向上の度合いにはばらつきがあることがわかります。既存のゲームやベンチマークの内容について実験した結果、パフォーマンス向上の見込みは、1.8 倍をピークに 1.8 倍～1.3 倍の範囲内に収まります。Scene 3 についても、コンテンツをチューニングすればこの範囲内の数値が得られるはずですが。

オンライン・ドライバでのもう 1 つの利点は、より多くの時間を頂点データの変換/照明処理に充てられるという点です。この時間を Pentium® III プロセッサのインターネット・ストリーミング SIMD 拡張命令に最適化されたコードを実行することに充てれば、理論上、変換/照明の処理で、カーネル・レベルの最適化による高速化率(前述のアムダールの法則による)にかなり近い数値を達成できます。図 9 を見ると、オンライン・ドライバのケースでは、より多くの時間が有効に計算に使用されていることがわかります。図 9 の左の円グラフでは、90%の時間が ArchGE ライブラリでの頂点データの変換/照明処理に使われています。これに対し、右の円グラフでは、変換/照明の処理に使われる時間は 50%に過ぎず、デバイス・ドライバのコードで使用する時間が 46%にのぼっています(頂点データをグラフィックス・コントローラのローカル・メモリにコピーす

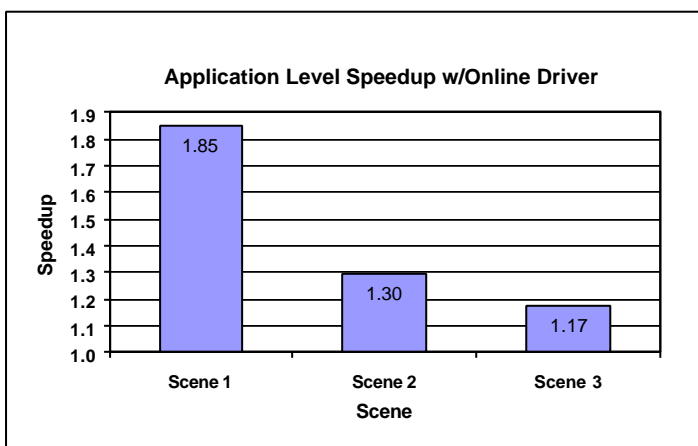


図 8: アプリケーション・レベルのパフォーマンス向上(オンライン・ドライバ・モデルと従来型ドライバ・モデルの比較)

ArchGE で OLD を利用することには、2 つの大きな効果があります。1 つめの効果は、このデバイス・ドライバ・モ

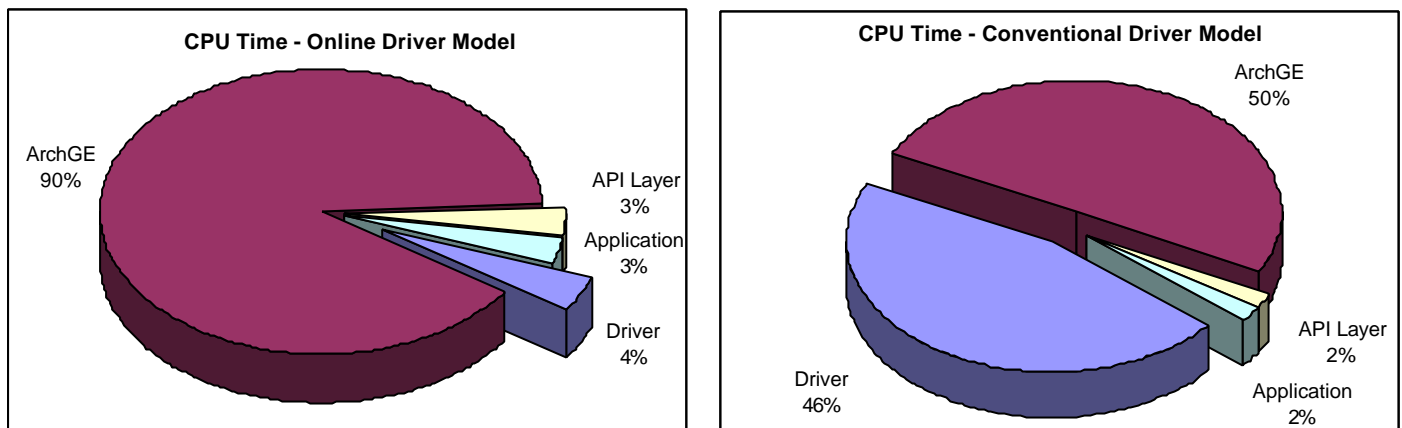


図9: オンライン・ドライバ・モデルの効果(計算に使われる時間と、データのフォーマットや移動に使われる時間の比較)

るため)。このグラフは、非常に複雑なシーン(グラフィックス・カードのフィル・レートの制約を受けない)について、ArchGEを使って計測した結果に基づいて作成したものです。

オンライン・ドライバ・モデルを利用すると、より多くの時間を頂点データの変換/照明処理に使用でき、キャッシュ階層とのデータのやり取りに要する時間を減らすことができます。変換/照明処理の高速化(Pentium III プロセッサに対する最適化と併せて)は、ハイレベルのコンテンツを駆使したアプリケーションの実現につながり、よりリアルで豊かなユーザ体験を提供できるようになります。オンライン・ドライバ・モデルでは、従来型モデルの1.8倍~1.3倍の高速化が達成されるという計測結果が出ています。

3D アプリケーション層の最適化

3D アプリケーション・スタックの最適化は、一番上の層であるアプリケーション・コードとコンテンツ自体を最適化することがその第一歩です。コンテンツの構造(プリミティブのタイプ、シーンごとの頂点の数、テクスチャの量など)はアプリケーションのパフォーマンスを大きく左右します。また、このコンテンツが3D ライブラリ層に渡される方法も非常に重要です。この研究に使用したシーン・マネージャには、パフォーマンスを大きく向上させる鍵となるいくつかの最適化手法が取り入れられています。

レンダリング・ステートの分類

シーン・マネージャでの3D モデルを画面表示用の画像に変換するには、レンダリング・ステートが必要になります。レンダリング・ステートとは、ジオメトリ・エンジンとグラフィックス・コントローラに対して、受け取ったデータを処理(および画面表示)する方法を伝えるための一連の情報です。レンダリング・ステートにはさまざまな変換マトリックスの値からテクスチャ・マップ・アドレスまで幅広

い情報が含まれており、グラフィックス・カードではこれらの情報の中から選択してラスタ処理⁷を行います。標準的な3D ライブラリでレンダリング・ステートを一部でも変更しようとする、アプリケーションへの負荷が非常に高くなります。

レンダリング・ステートの変更に伴うコストは、第1に3D ライブラリ自体に原因があります。大抵の場合、3D ライブラリを呼び出して現在のステートを変更するには、非常に多くのプロセッサ・サイクルを要します。まず、ライブラリ・ルーチンが新しいステートを確認し、ステートの変数を操作する際に時間がかかります。また、通常はデバイス・ドライバも独自のプロセスでステートの確認とセットアップを行うため、かなりの時間を費やします。

ステートの変更に伴うコストの2番目の要因は、3D グラフィックス・カードにあります。グラフィックス・カードのクロックとパフォーマンスが高まれば、ラスタライズ・パイプラインの深度も高くなります。通常は、ラスタ・パイプから既存のプリミティブを一扫し、その後で再開する必要があります。この操作によってラスタライズ・パイプラインに多くの隙間が生じ、帯域幅の利用効率が下がるためピクセル・フィル・レートやトライアングル・セットアップ・レートも低下します。

モデル/シーン・グラフを作成する時点で、使用するレンダリング・ステートのタイプを特定し、ステート設定の特性ごとにプリミティブを分類しておけば、アプリケーションからこの種のオーバーヘッドを排除することができます。

⁷ ラスタ処理とは、グラフィックス・コントローラがジオメトリ情報をコンピュータ・モニタで表示するピクセルの位置や色の情報に変換するプロセスのことです。

オブジェクト・レベルのクリッピング

3D アプリケーションで表示されないジオメトリの処理を回避するには、さまざまな方法があります。我々のアプリケーションでは、シーン内の処理中の部分を境界ボックスで囲んで、プリミティブに対してその後の処理を行うかどうかを細かく確認します。シーン・マネージャでは、境界ボックスの角にあたる点をビューポートの範囲と比較します。この比較で得られる結果には、次の3種類があります。

1. 完全にビュー・フラスタムの範囲外 -- その後の処理を拒否。
2. 完全にビュー・フラスタムの範囲内 -- 処理を続行する。クリッピングの必要がないことを示す。
3. 境界ボックスの角がビュー・フラスタムの境界にまたがっている -- 処理を続行する。クリッピングが必要であることを示す⁸。

境界ボックスの8つの角を使ってこの比較を行うと、処理時間を大幅に短縮できます。比較の計算では、プリミティブのすべての頂点について、独自のオブジェクト・スペースからワールドまたは画面スペースに変換する必要があります。境界ボックスを利用すると、8個の点をチェックするだけで済みます。隣接する平面に対して個々の点をチェックする場合、少なくとも4回の乗算と3回の加算演算が必要になります。これらの演算を、ビュー・フラスタムの頂部、底部、右側面、左側面、前面、裏面(合計6つの平面)について行わなければなりません。各平面を合わせると、1個の点ごとに、合計24回の乗算と18回の加算演算が必要になります(参考資料[3])。

3D モデルを境界ボックスで囲むことにより、頂点データに対してその後の処理を行うかどうかを確認するのに必要な計算の量を最小限にすることができます。この最適化によるアプリケーション・レベルのパフォーマンスへの影響を、図10に示します。このグラフを見ると、オブジェクト・レベルのクリッピングにおいて、境界ボックスの使用によりアプリケーション・レベルのパフォーマンスが明らかに向上していることがわかります。この結果は、ジオメトリの複雑さが異なる3種類のシーンを使って、アプリケーション内でオブジェクトの表示/非表示を切り替えて計測したものです(含まれる頂点の数が最も多いのが Scene 1、最も少ないのが Scene 3)。

境界ボックスの使用に関する研究の過程で、実際に、1つのシーン内で数多くの境界ボックスをさまざまな要素に使用できることがわかりました。境界ボックスに含める頂点の数の最小値は状況によって異なるため、特定の3D ソフトウェア・スタックで測定しながら最も効果的な数を割り出す必要があります。

アプリケーション層にレンダリング・ステートの分類とオブジェクト・レベルのクリッピングを実装することにより、最適化された ArchGE エンジンでの飛躍的なパフォーマンス向上が見込まれます。オブジェクト・レベルのクリッピングではアプリケーション・レベルのパフォーマンスが16%~35%向上しており、ArchGE でも同様の向上が見込まれることから、Pentium® III プロセッサのパフォーマンスを最大限に引き出すために貢献します。

まとめ

最近のソフトウェア・アプリケーションでは、これまでになく表現力豊かな3D グラフィックスが多用されるようになってきました。Pentium® III プロセッサおよびインターネット・ストリーミング SIMD 拡張命令を利用することにより、3D 変換/照明の処理において、最適化された従来の浮動小数点命令の2倍以上のパフォーマンスを達成できます。しかし、こうしたカーネル・レベルのパフォーマンスがそのままアプリケーション・レベルのパフォーマンスに反映されるとは限りません。

この記事では、3D アプリケーション・ソフトウェア・スタックのさまざまな層で根本的な最適化を図るための数々の手法を取り上げてきました。これらの最適化手法を取り

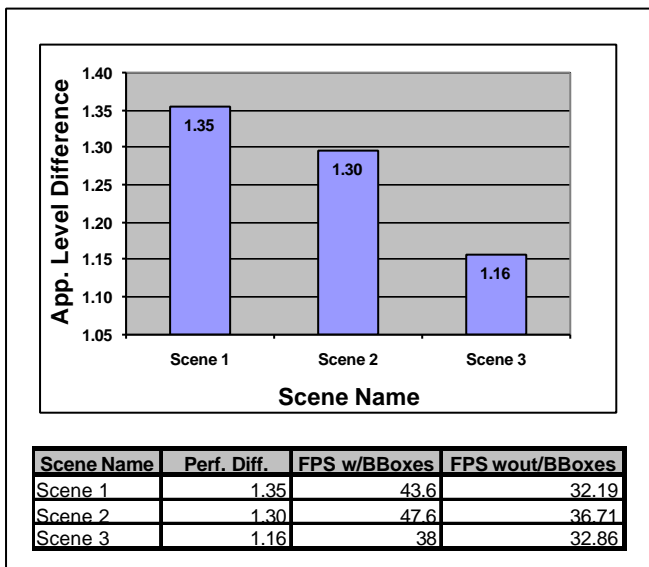


図 10: アプリケーション・レベルでのパフォーマンスの違い(境界ボックスを使用した場合)

⁸ プリミティブのクリッピングの詳細については、参考資料[5]および[6]を参照してください。

入れることにより、アプリケーションでは、一般的な 3D グラフィックス・ワークロードにおいて、Pentium III プロセッサのパフォーマンスを最大限に引き出すことが可能になります。Pentium® II プロセッサの場合でも、最適化されたシーン・マネージャと ArchGE ジオメトリ・エンジンを利用することにより、同じ動作クロックの Pentium III プロセッサで汎用 3D ソフトウェア・スタックを使用する場合に比べて、アプリケーション・レベルで 2 倍近くの高速度を実現しています。

謝辞

この記事で取り上げた製品や研究結果は、次の方々および組織の多大なる努力と協力によって実現したものです。

- **BMD アーキテクチャ:** Ken Castro -- ラボでのサポートを通じて開発/計測プロセスのスムーズな進行に貢献。
- **MAP-PBA:** Shervin Kheradpir, Jeff Ma -- ArchGEをサポートするシーン管理ソフトウェアを開発し、さまざまな実験に協力。
- **PMD アーキテクチャ:** Tom Huff -- ArchGEの機能、コードのレビュー、改善の余地に関する議論に参加。
- **GCD:** Peter Doyle -- i740(R)の仕様定義、オンライン・ドライバの仕様定義に尽力。
- **PDD:** Gerry Blank, Brandon Fliflet -- i740(R)版オンライン・ドライバのプロトタイプ作成と製品化に尽力。
- **3dfx Interactive, Inc.:** Colyn Case, Andrew Hanson -- 記事で引用されている多くの実験データの収集に使用した Voodoo2™版の ArchGE用オンライン・ドライバの仕様定義と開発に協力。

参考資料

- [1] James D Foley, Andries van Dam, Steven K. Feiner, John F. Hughes 著 『Computer Graphics: Principles and Practice』、Morgan Kaufmann, San Francisco, CA, pp.29 ~ 31
- [2] David A. Patterson, John L. Hennessy 著 『Computer Architecture: A Quantitative Approach』 Addison-Wesley, Menlo Park, CA, pp.201 ~ 283
- [3] David A. Patterson, John L. Hennessy 著 『Computer Architecture: A Quantitative Approach』 Addison-Wesley, Menlo Park, CA, pp.868
- [4] Mason Woo, Jackie Neider, Tom Davis 著 『OpenGL® Programming Guide: Second Edition』 Addison-Wesley, Menlo Park, CA, pp.42 ~ 45

* 一般に、ブランド名または商品名は各社の商標または登録商標です。

- [5] Jim F. Blinn, Martin E. Newell 著 『Clipping Using Homogeneous Coordinates』 SigGraph 1978 Proceedings, pp.245 ~ 251
- [6] Jim F. Blinn 著 『Jim Blinn's Corner: A Trip Down the Graphics Pipeline』 Morgan Kaufmann, San Francisco, CA, pp.122 ~ 134
- [7] 『インテル・アーキテクチャ最適化リファレンス・マニュアル』 (日本語 PDF ファイル: 2,934KB)
<http://developer.intel.com/design/pentiumiii/manuals/>
- [8] インテル・アーキテクチャ最適化リファレンス・マニュアル』 (日本語 PDF ファイル: 2,934KB)、 pp.6-8 ~ 6-9
- [8] 『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、下巻: システム・プログラミング・ガイド』 (日本語 PDF ファイル: 6,062KB)、 pp.9-4 ~ 9-13
<http://developer.intel.com/design/pentiumiii/manuals/>
- [10] 『インテル・アーキテクチャ最適化リファレンス・マニュアル』 (日本語 PDF ファイル: 2,934KB)、 pp.6-4 ~ 6-5
- [11] Chia-Lin Yang, Barton Sano, Alvin R. Lebeck 著 『Exploiting Instruction Level Parallelism in Geometry Processing for Three Dimensional Graphics Applications』 Technical Report CS-1998-14, Computer Science Department, Duke University, 1998 年 9 月

著者紹介

Paul Zagacki、カリフォルニア州フォルサム支社、マイクロプロセッサ・プロダクト・グループ(MPG)のシニア・プロセッサ・アーキテクト。1994年にインテルに入社し、マイクロプロセッサ・アーキテクチャのハイレベル・パフォーマンス・モデリング、Pentium® III プロセッサのソフトウェア/ベンチマークの分析と最適化、3D グラフィックスの実装とパフォーマンス分析/最適化を担当。専門分野は、コンピュータ・アーキテクチャ/マイクロアーキテクチャ、3D グラフィックス、コンパイラ・パフォーマンス、ソフトウェア/ハードウェアのパフォーマンス解析など。ミシガン大学(アナーバー)でコンピュータ・サイエンスの理学士号を取得。

電子メール・アドレス: paul.zagacki@intel.com

Deep Buch、フォルサム支社、マイクロプロセッサ・プロダクト・グループ(MPG)のスタッフ・プロセッサ・アーキテクト。1993年にインテルに入社し、プロセッサ・アーキテクチャ、プラットフォーム・テクノロジー、3D グラフィックス関連の開発を担当。インテルに入社する以前は、インドのバンガロールにある Wipro Infotech R&D で、ハー

ドウェア・スペシャリストとして ASIC およびシステム・レベルの設計を担当。専門分野はコンピュータ・アーキテクチャ、マルチメディア、通信など。1989年にインド工科大学(ボンベイ)で電子工学の技術修士号を取得。

電子メール・アドレス: deep.k.buch@intel.com

Emile Hsieh、フォルサム支社、マイクロプロセッサ・プロダクト・グループ(MPG)のシニア・プロセッサ・アーキテクト。専門分野は、コンピュータ・アーキテクチャ、パフォーマンス・モデリング/分析、コンパイラ、グラフィックス、信号処理、通信など。国立台湾大学(台湾、台北)で電子工学の理学士号を、パーデュー大学(イリノイ州ウエスト・ラファイエット)で電子工学の理学修士号を取得。

電子メール・アドレス: emile.hsieh@intel.com

Hsien-Hsin Lee、現在はミシガン大学でコンピュータ・サイエンスとエンジニアリングの博士課程に在学。1995～1998年の間、フォルサム支社でマイクロプロセッサ・プロダクト・グループ(MPG)のシニア・プロセッサ・アーキテクトとして、Pentium® Pro、Pentium® II、Pentium III プロセッサの設計/パフォーマンス・モデリングに従事。国立清華大学(台湾)で電子工学の理学士号を、ミシガン大学でエンジニアリングの理学修士号を取得。専門分野は、マイクロアーキテクチャ、メモリ・システム設計、ILP の最適化、グラフィックス・アーキテクチャなど。

電子メール・アドレス: linear@eecs.umich.edu

Daniel Melaku、フォルサム支社、マイクロプロセッサ・プロダクト・グループ(MPG)のプロセッサ・アーキテクト。1997年にインテルに入社し、パフォーマンス解析、検証、ツールの開発を担当する。専門分野は、デジタル信号処理、コンピュータ・アニメーション、音声/画像認識、人工知能など。カリフォルニア州立大学(サクラメント)でコンピュータ・エンジニアリングの理学士号を取得。

電子メール・アドレス: daniel.melaku@intel.com

Vladimir Pentkovski、フォルサム支社、マイクロプロセッサ・プロダクト・グループ(MPG)の首席エンジニア。IA-32 アーキテクチャのインターネット・ストリーミング SIMD 拡張命令の定義を手がけた中心チームの一員。Pentium III プロセッサ・アーキテクチャの開発とそのパフォーマンス分析において主導的役割を果たす。以前は、ロシアで Elbrus マルチプロセッサ・コンピュータ用プログラミング言語向けのコンパイラ開発とソフトウェア/ハードウェア・サポートに従事。ロシアでコンピュータ・サイエンスとエンジニアリングの理学博士号および Ph.D. を取得。

電子メール・アドレス: vladimir.m.pentkovski@intel.com