

ストリーミング SIMD 拡張命令 2(SSE2)を使用した 3D トランスフォーメーション

バージョン 2.0

2000 年 7 月

資料番号 248604J-001

【輸出規制に関する告知と注意事項】

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

【資料内容に関する注意事項】

- ・本ドキュメントの内容を予告なしに変更することがあります。
 - ・インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。
 - ・インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。
 - ・本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。
- インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。
- ・いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複製することは禁じられています。

本資料の内容についてのお問い合わせは、下記までご連絡下さい。

インテル株式会社 資料センター

〒305-8603 筑波学園郵便局 私書箱115号

Fax: 0120-47-8832

*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。

Copyright © Intel Corporation 1999, 2000

目次

1	はじめに	5
2	倍精度 3D トランスフォーメーション	5
2.1	倍精度 3D トランスフォーメーションのアプリケーション	6
2.2	倍精度 3D トランスフォーメーションの基礎知識	6
2.3	倍精度 3D トランスフォーメーションのコーディング	7
2.3.1	最適化手法	8
2.3.2	ヒントと要領	9
3	パフォーマンス	11
3.1	パフォーマンスの向上/改善	11
3.2	考察	12
4	結論	13
5	C/C++を使用したコード例	14
6	SSE2 C++ベクトル・クラスを使用したコード例	16
7	SSE2 組み込み関数を使用したコード例	17
8	SSE2 アセンブリを使用したコード例	20
9	コンパイラのベクトル化を使用したコード例	25
付録 A	パフォーマンス・データ	A-1
	パフォーマンス・データの改訂履歴	A-1
	テスト・システムの構成	A-3

改訂履歴

改訂番号	改訂履歴	改訂時期
2.0	インテル® Pentium® 4 プロセッサに関する更新	2000年7月
1.0	初版	1999年9月

参考資料

このアプリケーション・ノートでは次の資料を参考にした。これらの資料には、本書で取り上げた事項を理解するための基礎知識が含まれている。

Alan Watt, *3D Computer Graphics 2nd Edition*, Addison Wesley, 1993.

James Foley, Andries van Dam, Steven Feiner, John Hughes, *Computer Graphics: Principles and Practice, 2nd Edition*, Addison Wesley, 1990.

インテル、『C++ SIMD 命令クラス・ライブラリ・リファレンス・マニュアル』、資料番号 693500J、1999年

インテル、『インテル® C/C++ コンパイラ・ユーザズ・ガイド』、資料番号 741901J、1999年

『ストリーミングSIMD 拡張命令2(SSE2)を使用した SAXPY/DAXPY』、インテル・アプリケーション・ノート AP-935、Copyright 2000

1 はじめに

ストリーミング SIMD 拡張命令 2(SSE2、Streaming SIMD Extensions 2)では、SIMD(Single Instruction Multiple Data)倍精度浮動小数点命令、および SIMD 整数命令が IA-32 インテル® アーキテクチャに新しく導入された。倍精度 SIMD 命令による機能拡張の方法は、ストリーミング SIMD 拡張命令(SSE)で導入された単精度 SIMD 命令による機能拡張とよく似ている。128 ビットの整数 SIMD 拡張命令は、64 ビットの整数 SIMD 拡張命令の完全なスーパーセットで、より多くの整数データ型、整数と浮動小数点間のデータ型変換、キャッシュとシステム・メモリの効果的使用をサポートする命令が追加されている。これらの命令は、3Dグラフィックス、リアルタイムの物理的な現象、空間的(3D)オーディオ、ビデオ・エンコーディング/デコーディング、暗号化、および科学計算アプリケーションによく使用される演算を高速化する。このアプリケーション・ノートでは、倍精度 3D ジオメトリ変換の実装手法と、SSE2 を利用したコードの例について説明する。

コード最適化手法の中には、128 ビット XMM レジスタを使用して、パックド倍精度浮動小数点データとそれに関連する SSE2 を処理するものがある。開発者は、これらの最適化手法を、自分で作成した倍精度 3D トランスフォーメーションのコードを最適化するためのツールとして利用できる。また、3D トランスフォーメーション・アルゴリズムのさまざまな実装手法を含むソース・コードも示す。

2 倍精度 3D トランスフォーメーション

3D トランスフォーメーションは、すべての 3D ジオメトリ・エンジンの不可欠の構成要素であるが、3D トランスフォーメーション・アルゴリズムの倍精度版は、通常は数学的に高精度のモデリングが必要なアプリケーションにのみ使用される。この変換の目的は、オブジェクトの頂点の値を、ローカル座標空間(特定のオブジェクトだけを含む座標系)から、ワールド座標空間(特定の 3D シーン内のすべてのオブジェクトを含む座標系)に変換することである。変換それ自体は、ローカル空間内の 4×1 の頂点ベクトルに 4×4 の変換マトリックスを掛けて、ワールド空間内の新しい 4×1 の頂点ベクトルを得る簡単な乗算である(図 1 を参照)[Watt, 45-49]。

$$\begin{pmatrix} \text{変換} \\ \text{マトリックス} \end{pmatrix} * \begin{pmatrix} \text{ローカル} \\ \text{空間} \\ \text{ベクトル} \end{pmatrix} = \begin{pmatrix} \text{ワールド} \\ \text{空間} \\ \text{ベクトル} \end{pmatrix}$$

図 1: 3D トランスフォーメーションとは 4×1 のベクトルに 4×4 のマトリックスを掛ける乗算である

2.1 倍精度 3D トランスフォーメーションのアプリケーション

3D グラフィックスは、3D ゲームから惑星間引力の研究まで、きわめて広範囲にわたるアプリケーションに使用される。3D トランスフォーメーションの必要条件は、アプリケーションによって異なる。例えば、3D ゲームでは、主にリアルタイム・レンダリングの性能が重視される。パフォーマンス上の理由で、ゲームでは、通常は、画面上に描かれる画像が人間の目からほぼ正確に見えるように、単精度浮動小数点値だけを使用するグラフィックス・システムが使用される。

倍精度 3D 変換は、オブジェクトの回転、移動、およびスケーリングの正確なモデリングが必要とされるモデリング・システムおよびアプリケーションで主に使用される。倍精度計算は、ハイエンドの 3D モデリング、アニメーション、科学研究アプリケーション、CAD (Computer Aided Design)、および医療用のグラフィックス・アプリケーションに効果的である。このアプリケーション・ノートで説明する最適化手法は、ほとんどの倍精度グラフィックス・システムに適用できるが、各システム固有のニーズに対応するには、多少の修正が必要である。

2.2 倍精度 3D トランスフォーメーションの基礎知識

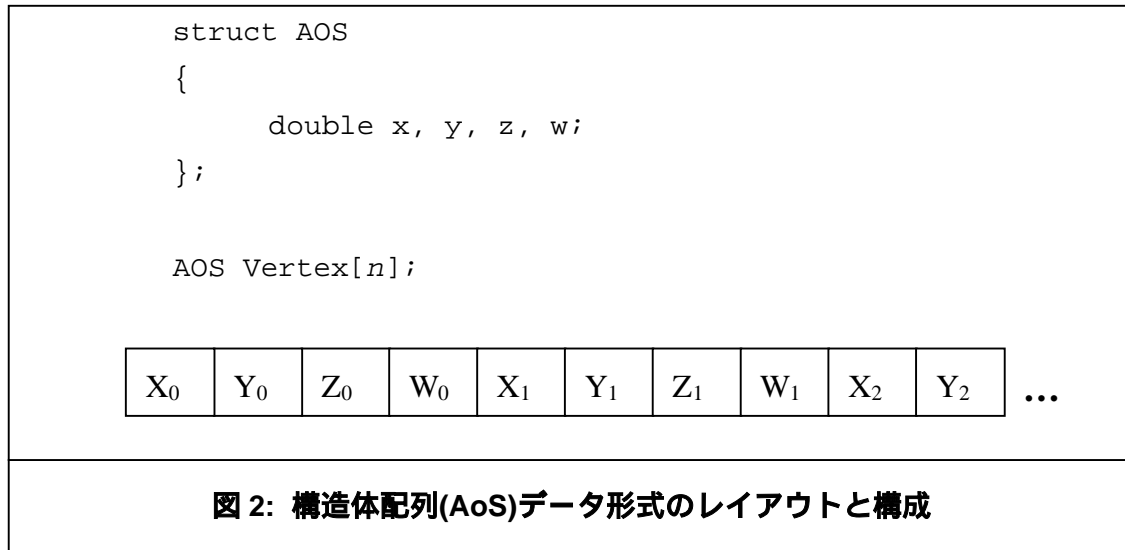
3D グラフィックスの分野では、従来、ポリゴン・モデリングを使用して、シーンと呼ばれる 3D 環境内で相互作用する 3 次元オブジェクトを作成してきた。ポリゴン・モデリングでは、グラフィック・オブジェクトは、オブジェクトの形状を構成する頂点のリストによって定義される。ジオメトリ・エンジンの役割は、頂点リストの演算を実行し、オブジェクト間の相互作用、オブジェクトと光源の間の相互作用、シーンの境界、および物理学の法則のリアルなモデリングを可能にすることである。

ルネ・デカルトやアイザック・ニュートンなどの数学者は、17 世紀には既に 3D グラフィックスの基礎原理を公式化していた。1960 年代および 1970 年代に、初めてベクトル表示とラスタ表示が開発され、マイクロコンピュータによるポリゴン・モデリングが実現された。現在のところ、コンピュータ・グラフィックスに関する最も詳しい総合的な研究は、おそらく

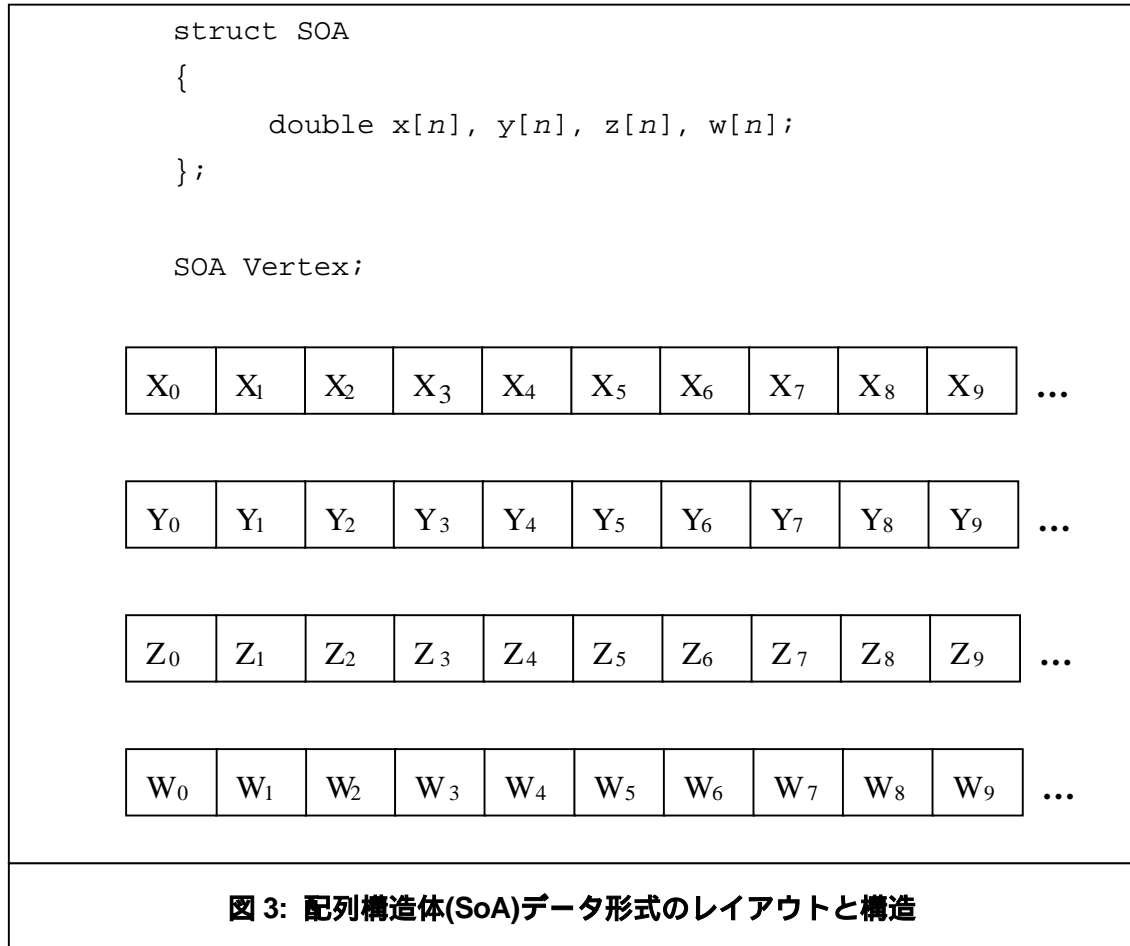
『*Computer Graphics: Principles and Practice, 2nd Edition*』James Foley, et al [Foley]である。

2.3 倍精度3Dトランスフォーメーションのコーディング

このアプリケーション・ノートでは、頂点データを構造化する2種類の方法を取り上げる。第1の方法は、標準的な構造体配列(AoS)形式である。この形式では、4つの倍精度値 x 、 y 、 z 、 w で構成される構造体が定義され、このような構造体の数 n を含む配列が宣言される(図2を参照)。開発者はこの形式によく慣れており、比較的少ない労力でかなり高いパフォーマンスが得られるため、おそらくこれが最も簡単な方法である。開発期間の短縮が重視されるアプリケーションで、最大限のパフォーマンスが必要でない場合は、この方法が最適である。



第2の(一般的でない)データ構造は、配列構造体(SoA)形式である。この形式では、頂点データの4つの配列が「2次元」形式でレイアウトされる。第1の配列は、一定の数(n)の X 値で構成される。第2の配列は n 個の Y 値、第3の配列は n 個の Z 値、第4の配列は n 個の W 値で構成される(図3を参照)。この構造は、通常は最初にデータを再フォーマットする必要があるが、不要なロードおよびキャッシュ排出操作の数が減少するため、構造体配列(AoS)形式より高速で処理できる。このデータ構造は、配列構造体(SoA)と呼ばれる。この構造によって、開発者は、一度に2つの倍精度データ要素を操作でき、SSE2の並列性をフルに利用できる。データ構造の例については、付属のソース・コードを参照のこと。



2.3.1 最適化手法

3Dグラフィックス・エンジンは、通常は頂点座標 X、Y、Z、および W を使用する。ローカル座標空間では、W は常に 1.0 と見なされる。これらのエンジンに一般的な最適化手法は、修正されたマトリックス乗算を実行し、X、Y、および Z に、それぞれに対応するマトリックス要素を掛けることである。変換マトリックスの第 4 列の要素は、1.0 を掛けても値は変わらないため、乗算を省略して結果に追加される。この手法の例は、このアプリケーション・ノートで示すすべてのコード例に使用されている。ジオメトリ・パイプラインを最適化する際は、この手法を考慮する必要がある。

最適化された 3D トランスフォーメーションは、128 ビット・レジスタを使用して、1 回のループ当たり 2 回の変換を実行する。配列構造体(SoA)データ構造を使用する場合は、`movapd` 命令を使用して、連続する 2 つの頂点値を同時にロードできる(つまり、 X_0 と X_1 はメモリ内で連続しているため、1 回のメモリ参照でアクセスできる)。構造体配列(AoS)形式を使用する場合は、`movhpd` 命令と `movlpd` 命令を使用して、2 つの頂点値を別々にロードしなければならない。この場合、 X_0 と X_1 はロードされるが、メモリ内で連続していないため、個々にアクセスしなければならない。SoA 手法の方が明らかに高速であるが、AoS 手法でもパフォーマンスがかなり向上するため、AoS 手法も検討するべきである。

2.3.2 ヒントと要領

3Dトランスフォーメーションは、X、Y、およびZの各入力値を使用して、4つの新しい頂点値X'、Y'、Z'、およびW'を計算する。一方、マトリックスの要素は、ロードされ、1ループ当たり1回使用され、その後は不要になる。この変換ではレジスタにかかる圧力は問題にならないため、ループ全体が終了するまでレジスタ内に頂点値を保持することによって、パフォーマンスを改善できる。マトリックスの値が入っているレジスタを各操作のデスティネーションにすれば、頂点のデータはループの始めに一度だけロードされ、次のループの始めに上書きされる。図4のアセンブリ・コード・リストの抜粋に、この手法の例を示す。

```

movapd   xmm0, [ebx+eax]           ; load x+1|x
movapd   xmm1, [esi+0]            ; load m00|m00
mulpd    xmm1, xmm0

movapd   xmm2, [ecx+eax]           ; load y+1|y
movapd   xmm3, [esi+16]           ; load m01|m01
mulpd    xmm3, xmm2

movapd   xmm4, [edx+eax]           ; load z+1|z
movapd   xmm5, [esi+32]           ; load m02|m02
mulpd    xmm5, xmm4
addpd    xmm3, xmm1

movapd   xmm7, [esi+48]           ; load m03|m03
addpd    xmm5, xmm3
addpd    xmm7, xmm5
movapd   [ebx+eax],  xmm7         ; store (x+1)'|x'

/* x, y, and z values are still available in their original registers */
movapd   xmm1, [esi+64]           ; load m10|m10
mulpd    xmm1, xmm0

movapd   xmm3, [esi+80]           ; load m11|m11
mulpd    xmm3, xmm2

...

```

図4: X、Y、およびZ値を再使用する、最適化されたアセンブリ・リスト

組み込み関数を使用して倍精度データにアクセスするには、アクセスされる倍精度値へのメモリ・ポインタを使用する必要がある。既存の構造と address-of 演算子(&)を組み合わせるとループを書き直したくなるが、この場合、コンパイラは、同じデータ構造に対して複数のメモリ・ポインタが対応付けられていると見なす可能性がある。その結果、address-of 演算子がなければ実行されていたはずの最適化が無効にされることがある(図5を参照)。

```
AOS vertex[n];
    for (i = 0; i < n; i++)
    {
        __m128d tx, m00;
        tx = _mm_load_pd(tx, &(vertex[i].x));
        m00 = _mm_load_pd(m00, &(Matrix->m00));
        m00 = _mm_mul_pd(m00, tx);
    };
```

図5: address-of 演算子(&)の誤使用によって、組み込み関数の最適化が一部無効にされる

すべての最適化を有効にするには、組み込み関数によって最適化されるループを、address-of 演算子を使用せずにアドレス指定できるように修正する必要がある。簡単に address-of 演算子を取り除く方法は、double 型データ要素の配列へのポインタを、__m128d 型の配列へのポインタに型キャストすることである(図6を参照)。この手法は、データ構造が SoA である場合は非常に効果的であるが、AoS 形式にも使用できる。ただし、AoS 形式の場合は、データのシャッフルのために多少のオーバーヘッドが発生する。

```
AOS vertex[n];
__m128d *V = vertex;
__m128d *M = Matrix;
for (i = 0; i < n; i++)
{
    __m128d m00;
    m00 = _mm_mul_pd(M[0], V[i]);
};
```

図6: __m128d 配列に型キャストすると、組み込み関数による最適化が有効になる

3 パフォーマンス

変換の結果は、データ構造とコーディング方法によって異なる。この節では、3Dトランスフォーメーションの開発時間とパフォーマンスの向上の間のトレードオフについて簡単に説明する。ここでは、倍精度 SSE2 をコーディングする 3 種類の方法について検討する。最も基本的な方法は、インライン・アセンブリ・コーディングである。この方法では、プログラマがレジスタの割り当てと命令スケジューリングの責任を負う。その他の 3 つの方法(インテル[®] C/C++コンパイラ組み込み関数の使用、SIMD 演算用 C++クラス・ライブラリの使用、およびインテル C/C++コンパイラのベクトライザの使用)では、コンパイラがレジスタの割り当てと命令スケジューリングの責任を負うため、プログラマはより高水準のアルゴリズムの修正に専念できる。どの方法を選択した場合も、このアプリケーション・ノートの推奨事項に従えば、SSE2 を使用した倍精度 3D トランスフォーメーションのコーディングによって、パフォーマンスが大きく向上するはずである。

3.1 パフォーマンスの向上/改善

このテストでは、最初に標準的な C コードと AoS データ形式の組み合わせを基準に使用し、次に SoA 形式を使用して C コードで変換プログラムを書き換えた。14 ページの「5 C/C++によるコード例」に、2 種類のコード・セグメントを示す。次に、2.3.2 項で説明した手法とヒントを使用して、両方のデータ形式で組み込み関数をコーディングした。AoS 形式と SoA 形式の組み込み関数では、標準的な C コードに対して、パフォーマンスが大きく向上した。また、予想どおり、SoA 組み込み関数は AoS 組み込み関数より高いパフォーマンスを示した。3D トランスフォーメーションの組み込み関数を使用したコード・リストについては、17 ページの「7 SSE2 組み込み関数を使用したコード例」を参照のこと。

C++を使用する開発者のために、SIMD 演算用 C++クラス・ライブラリを使用して、SoA 形式のコードが作成された。このクラス・ライブラリは、他の C++クラスと同じように簡単に使用できるが、実際には組み込み関数を抽象化したものである。ベクトル・クラスを使用すれば、コーディング上の多くのメリットが得られる。このコードは、C++コードと同じくらい簡単に開発でき、組み込み関数版のコードと同等のパフォーマンスを持ち、元の C コードと同じくらいわかりやすい。第 6 節に、C++ベクトル・クラスのコード例を示す。

もう 1 つの方法は、インテル C/C++コンパイラのベクトル化機能を使用して、SIMD 命令向けに C/C++コードをコンパイルするものである。SoA 形式の 3D トランスフォーメーションのコードは、ベクトル化に適している。ベクトル化についての詳細は、アプリケーション・ノート『AP-935、ストリーミング SIMD 拡張命令 2(SSE2)を使用した SAXPY/DAXPY』を参照のこと。SAXPY/DAXPY アプリケーション・ノートには、ベクトライザの使用方法についての簡単な説明がある。また、ベクトライザの使用に関するより一般的な説明として、『インテルコンパイラ・ユーザ・ガイド』を参照のこと。第 8 節に、ベクトライザのコードを示す。

最後に、パフォーマンスの比較のために、アセンブリ・コードを手作業で作成した。これらの小さなテストでは、アセンブリ・コードのパフォーマンスは組み込み関数コードをわずかに上回った。ただし、アプリケーション全体にわたってインライン・アセンブリを使用すると、コンパイラがプロシージャ間の最適化を実行する機会が失われ、手作業でコーディングされた関数とその他のコード・セグメントのインターリーブとインライン展開のコストが増加する。多くの場合、アセンブリの最適化によってパフォーマンスを限界まで数%向上させようとする、コンパイラの最適化のうち数%が失われ、コーディングの労力が大幅に増える。開発者は、イ

ンライン・アセンブリのコーディングにかかる時間と、それによって得られるパフォーマンスの向上を比較した上で決定を下す必要がある。パフォーマンスの向上が数%以内であれば、通常は組み込み関数を使用する方がよい。手作業で最適化されたアセンブリ命令については、第8節のアセンブリ言語を使用したコード・リストを参照。

3.2 考察

テストの結果は、SoAデータ形式が最適な構造であることを示している。ただし、この文書では検討していないデータ構造がある。一定の条件下では、この構造の方が高いパフォーマンスが得られる可能性がある。SoA入力データの配列のサイズが非常に大きい場合を考える。配列のサイズが非常に大きいため、X、Y、およびZの値がそれぞれメモリの別々のページに置かれる場合は、頂点の成分にアクセスするたびにページ・ミスが発生する。頂点の成分が同じページ上に置かれるようにするには、ハイブリッドSoA形式と呼ばれる形式で、データのパックとインターリーブを実行すればよい(図7を参照)。ハイブリッド・データ・オーダは、メモリ上の類似のデータ要素に同時にアクセスでき、2次元データの大きな配列で発生するページ・ミスの問題を回避できるため、他のデータ形式より高い性能を発揮する。他の方法で期待したメモリ・パフォーマンスが得られない場合は、代替手法としてハイブリッド構造を検討するとよい。

```
struct Hybrid_SOA
{
    double x[2], y[2], z[2], w[2];
};

Hybrid_SOA Vertex[n/2];
```

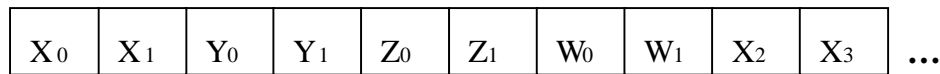


図7: 「ハイブリッド」SoAデータ形式が最適な場合もある

4 結論

構造体配列(AoS)と配列構造体(SoA)のどちらのデータ形式を使用した場合でも、SSE2は、倍精度3Dトランスフォーメーションのパフォーマンスの大幅な向上を実現する。このアプリケーション・ノートのテストでは、SoAデータ構造に対してSSE2を使用すると、AoS形式に対してSSE2を使用した場合と比べて、かなりのスピードアップが見られた(ただし、いずれの場合も、標準的なCコードよりはるかに高速である)。2次元データ構造への移行が困難なアプリケーションでも、SSE2の使用によってパフォーマンスが大きく向上する可能性があるため、SSE2の使用をお勧めする。

アプリケーションによっては、アセンブリ命令プログラミングによってパフォーマンスを限界まで向上させる必要があるが、ほとんどの場合は、SSE2組み込み関数による標準的なCコードの最適化、SIMD演算用C++クラス・ライブラリによる標準的なC++コードの最適化、またはベクトル化の使用によって、最適な投資効果が得られる。3Dトランスフォーメーションの必要条件に合わせて、このアプリケーション・ノートに示した最適化手法を修正すれば、少ない労力で十分な効果が得られるはずである。

5 C/C++によるコード例

```
struct Matrix
{
    DBL m00, m01, m02, m03;
    DBL m10, m11, m12, m13;
    DBL m20, m21, m22, m23;
    DBL m30, m31, m32, m33;
} M44;

struct Soa
{
    DBL *x, *y, *z, *w;

    void Initialize()
    {
        x = _mm_malloc(NUM_ELEMENTS*sizeof(DBL), 32);
        y = _mm_malloc(NUM_ELEMENTS*sizeof(DBL), 32);
        z = _mm_malloc(NUM_ELEMENTS*sizeof(DBL), 32);
        w = _mm_malloc(NUM_ELEMENTS*sizeof(DBL), 32);
    }
} SOA_Vertex;

__declspec(align(32))
struct Aos
{
    DBL x, y, z, w;
} AOS_Vertex[NUM_ELEMENTS];

int AOS_Transform ()
{
    Aos *vertex = AOS_Vertex;

    for (int i=0; i < length; i++)
    {
        DBL tx, ty, tz, tw;

        tx = M44->m00*vertex[i].x + M44->m01*vertex[i].y + M44->m02*vertex[i].z
+ M44->m03;
        ty = M44->m10*vertex[i].x + M44->m11*vertex[i].y + M44->m12*vertex[i].z
+ M44->m13;
        tz = M44->m20*vertex[i].x + M44->m21*vertex[i].y + M44->m22*vertex[i].z
+ M44->m23;
```

```
    tw = M44->m30*vertex[i].x + M44->m31*vertex[i].y + M44->m32*vertex[i].z
+ M44->m33;

    vertex[i].x = tx; vertex[i].y = ty; vertex[i].z = tz; vertex[i].w = tw;
}
}

int SOA_Transform()
{
    for (int i=0; i < length; i++)
    {
        DBL tx, ty, tz, tw;

        tx = M44->m00*SOA_Vertex.x[i] + M44->m01*SOA_Vertex.y[i] +
            M44->m02*SOA_Vertex.z[i] + M44->m03;
        ty = M44->m10*SOA_Vertex.x[i] + M44->m11*SOA_Vertex.y[i] +
            M44->m12*SOA_Vertex.z[i] + M44->m13;
        tz = M44->m20*SOA_Vertex.x[i] + M44->m21*SOA_Vertex.y[i] +
            M44->m22*SOA_Vertex.z[i] + M44->m23;
        tw = M44->m30*SOA_Vertex.x[i] + M44->m31*SOA_Vertex.y[i] +
            M44->m32*SOA_Vertex.z[i] + M44->m33;

        SOA_Vertex.x[i] = tx; SOA_Vertex.y[i] = ty; SOA_Vertex.z[i] = tz;
        SOA_Vertex.w[i] = tw;
    }
}
```

6 SSE2 C++ベクトル・クラスを使用したコード例

```
struct WMatrix
{
    union
    {
        struct
        {
            DBL m00, m00p, m01, m01p, m02, m02p, m03, m03p;
            DBL m10, m10p, m11, m11p, m12, m12p, m13, m13p;
            DBL m20, m20p, m21, m21p, m22, m22p, m23, m23p;
            DBL m30, m30p, m31, m31p, m32, m32p, m33, m33p;
        };
        struct
        {
            __m128d dm00, dm01, dm02, dm03;
            __m128d dm10, dm11, dm12, dm13;
            __m128d dm20, dm21, dm22, dm23;
            __m128d dm30, dm31, dm32, dm33;
        };
    };
} WM44;

int VectorClass_SOA()
{
    F64vec2 *x = SOA_Vertex.x; F64vec2 *y = SOA_Vertex.y; F64vec2 *z =
    SOA_Vertex.z;
    F64vec2 *w = SOA_Vertex.w; F64vec2 *WM = WM44;

    for (int i=0; i < length/2; i++)
    {
        F64vec2 tx, ty, tz, tw;
        tx = x[i] * WM[0] + y[i] * WM[1] + z[i] * WM[2] + WM[3];
        ty = x[i] * WM[4] + y[i] * WM[5] + z[i] * WM[6] + WM[7];
        tz = x[i] * WM[8] + y[i] * WM[9] + z[i] * WM[10] + WM[11];
        tw = x[i] * WM[12] + y[i] * WM[13] + z[i] * WM[14] + WM[15];
        x[i] = tx; y[i] = ty; z[i] = tz; w[i] = tw;
    }
    return EXIT_SUCCESS;
}
```

7 SSE2 組み込み関数を使用したコード例

```

int AOS_Intrinsics()
{
    DBL *vertex = AOS_Vertex;

    for (int i=0; i < length; i+=2)
    {
        __m128dtx, ty, tz, tw;
        __m128dmx0, mx1, mx2, mx3;

        // Compute (x) and (x+1)
        tx = _mm_loadl_pd(tx, vertex+i*4+0);
        tx = _mm_loadh_pd(tx, vertex+i*4+4);
        mx0 = _mm_mul_pd(tx, WM->dm00);

        ty = _mm_loadl_pd(ty, vertex+i*4+1);
        ty = _mm_loadh_pd(ty, vertex+i*4+5);
        mx1 = _mm_mul_pd(ty, WM->dm01);

        tz = _mm_loadl_pd(tz, vertex+i*4+2);
        tz = _mm_loadh_pd(tz, vertex+i*4+6);
        mx2 = _mm_mul_pd(tz, WM->dm02);
        mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm03)));
        _mm_storel_pd(vertex+i*4+0, mx0);
        _mm_storeh_pd(vertex+i*4+4, mx0);

        // Compute (y) and (y+1)
        mx0 = _mm_mul_pd(tx, WM->dm10);
        mx1 = _mm_mul_pd(ty, WM->dm11);
        mx2 = _mm_mul_pd(tz, WM->dm12);
        mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm13)));
        _mm_storel_pd(vertex+i*4+1, mx0);
        _mm_storeh_pd(vertex+i*4+5, mx0);

        // Compute (z) and (z+1)
        mx0 = _mm_mul_pd(tx, WM->dm20);
        mx1 = _mm_mul_pd(ty, WM->dm21);
        mx2 = _mm_mul_pd(tz, WM->dm22);
        mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm23)));
        _mm_storel_pd(vertex+i*4+2, mx0);
        _mm_storeh_pd(vertex+i*4+6, mx0);
    }
}

```

```

        // Compute (w) and (w+1)
        mx0 = _mm_mul_pd(tx, WM->dm30);
        mx1 = _mm_mul_pd(ty, WM->dm31);
        mx2 = _mm_mul_pd(tz, WM->dm32);
        mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm33)));
        _mm_storel_pd(vertex+i*4+3, mx0);
        _mm_storeh_pd(vertex+i*4+7, mx0);
    }
}

```

```

int SOA_Intrinsics()
{
    Soa vertex = SOA_Vertex;

    for (int i=0; i < length; i+=2)
    {
        __m128dtx, ty, tz, tw;
        __m128dmx0, mx1, mx2, mx3;

        // Compute (x) and (x+1)
        tx = _mm_load_pd(vertex.x+i);
        mx0 = _mm_mul_pd(tx, WM->dm00);
        ty = _mm_load_pd(vertex.y+i);
        mx1 = _mm_mul_pd(ty, WM->dm01);
        tz = _mm_load_pd(vertex.z+i);
        mx2 = _mm_mul_pd(tz, WM->dm02);
        mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm03)));
        _mm_store_pd(vertex.x+i, mx0);

        // Compute (y) and (y+1)
        mx0 = _mm_mul_pd(tx, WM->dm10);
        mx1 = _mm_mul_pd(ty, WM->dm11);
        mx2 = _mm_mul_pd(tz, WM->dm12);
        mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm13)));
        _mm_store_pd(vertex.y+i, mx0);

        // Compute (z) and (z+1)
        mx0 = _mm_mul_pd(tx, WM->dm20);
        mx1 = _mm_mul_pd(ty, WM->dm21);
        mx2 = _mm_mul_pd(tz, WM->dm22);
        mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm23)));
    }
}

```

```
    _mm_store_pd(vertex.z+i, mx0);

    // Compute (w) and (w+1)
    mx0 = _mm_mul_pd(tx, WM->dm30);
    mx1 = _mm_mul_pd(ty, WM->dm31);
    mx2 = _mm_mul_pd(tz, WM->dm32);
    mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm33)));
    _mm_store_pd(vertex.w+i, mx0);
}
}
```

8 SSE2 アセンブリを使用したコード例

```

int AOS_Asm()
{
    int len = length;
    Aos *vertex = AOS_Vertex;
    WMatrix *WM = WM44;

    __asm
    {
        xor     eax, eax
        mov     edi, len
        imul   edi, 32
        sub     edi, 8
        mov     ecx, vertex
        mov     edx, WM

LOOP:
        cmp     eax, edi
        jge     END_LOOP

        // Compute (x) and (x+1)
        movlpd  xmm0, [ecx+eax+0]      ; load x+1|x
        movhpd  xmm0, [ecx+eax+32]
        movapd  xmm1, [edx+0]         ; load m00|m00
        mulpd   xmm1, xmm0

        movlpd  xmm2, [ecx+eax+8]     ; load y+1|y
        movhpd  xmm2, [ecx+eax+40]
        movapd  xmm3, [edx+16]        ; load m01|m01
        mulpd   xmm3, xmm2
        movlpd  xmm4, [ecx+eax+16]    ; load z+1|z
        movhpd  xmm4, [ecx+eax+48]
        movapd  xmm5, [edx+32]        ; load m02|m02
        mulpd   xmm5, xmm4
        addpd   xmm3, xmm1
        movapd  xmm7, [edx+48]        ; load m03|m03
        addpd   xmm5, xmm3
        addpd   xmm7, xmm5
        movlpd  [ecx+eax+0], xmm7     ; store X'
        movhpd  [ecx+eax+32], xmm7   ; store (X+1)'
    }
}

```

```
/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
```

```
    // Compute (y) and (y+1)
    movapd   xmm1, [edx+64]      ; load m10|m10
    mulpd    xmm1, xmm0
    movapd   xmm3, [edx+80]     ; load m11|m11
    mulpd    xmm3, xmm2
    movapd   xmm5, [edx+96]     ; load m12|m12
    mulpd    xmm5, xmm4
    addpd    xmm3, xmm1
    movapd   xmm7, [edx+112]    ; load m13|m13
    addpd    xmm5, xmm3
    addpd    xmm7, xmm5
    movlpd   [ecx+eax+8], xmm7   ; store Y'
    movhpd   [ecx+eax+40], xmm7 ; store (Y+1)'
```

```
/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
```

```
    // Compute (z) and (z+1)
    movapd   xmm1, [edx+128]    ; load m20|m20
    mulpd    xmm1, xmm0
    movapd   xmm3, [edx+144]    ; load m21|m21
    mulpd    xmm3, xmm2
    movapd   xmm5, [edx+160]    ; load m22|m22
    mulpd    xmm5, xmm4
    addpd    xmm3, xmm1
    movapd   xmm7, [edx+176]    ; load m23|m23
    addpd    xmm5, xmm3
    addpd    xmm7, xmm5
    movlpd   [ecx+eax+16], xmm7 ; store Z'
    movhpd   [ecx+eax+48], xmm7 ; store (Z+1)'
```

```
/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
```

```
    // Compute (w) and (w+1)
    movapd   xmm1, [edx+192]    ; load m30|m30
    mulpd    xmm1, xmm0
    movapd   xmm3, [edx+208]    ; load m31|m31
    mulpd    xmm3, xmm2
    movapd   xmm5, [edx+224]    ; load m32|m32
    mulpd    xmm5, xmm4
    addpd    xmm3, xmm1
    movapd   xmm7, [edx+240]    ; load m33|m33
    addpd    xmm5, xmm3
```

```

        addpd    xmm7,  xmm5
        movlpd  [ecx+eax+24], xmm7    ; store W'
        movhpd  [ecx+eax+56], xmm7    ; store (W+1)'

        add     eax, 64
        jmp     LOOP
END_LOOP:
    }
}

int SOA_Asm()
{
    int len = length*8;
    Soa vertex = SOA_Vertex;
    WMatrix *WM = WM44;

    __asm
    {
        xor     eax, eax
        mov     ebx, vertex.x
        mov     ecx, vertex.y
        mov     edx, vertex.z
        mov     edi, vertex.w
        mov     esi, WM

LOOP:
        cmp     eax, len
        jge     END_LOOP

        // Compute (x) and (x+1)
        movapd  xmm0, [ebx+eax]    ; load x+1|x
        movapd  xmm1, [esi+0]     ; load m00|m00
        mulpd   xmm1, xmm0
        movapd  xmm2, [ecx+eax]    ; load y+1|y
        movapd  xmm3, [esi+16]    ; load m01|m01
        mulpd   xmm3, xmm2
        movapd  xmm4, [edx+eax]    ; load z+1|z
        movapd  xmm5, [esi+32]    ; load m02|m02
        mulpd   xmm5, xmm4

        addpd   xmm3, xmm1
        movapd  xmm7, [esi+48]    ; load m03|m03
        addpd   xmm5, xmm3

```

```

    addpd    xmm7,  xmm5
    movapd   [ebx+eax],  xmm7      ; store (x+1)'|x'

/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
    // Compute (y) and (y+1)
    movapd   xmm1,  [esi+64]      ; load m10|m10
    mulpd    xmm1,  xmm0
    movapd   xmm3,  [esi+80]      ; load m11|m11
    mulpd    xmm3,  xmm2
    movapd   xmm5,  [esi+96]      ; load m12|m12
    mulpd    xmm5,  xmm4
    addpd    xmm3,  xmm1
    movapd   xmm7,  [esi+112]     ; load m13|m13
    addpd    xmm5,  xmm3
    addpd    xmm7,  xmm5
    movapd   [ecx+eax],  xmm7     ; store (y+1)'|y'

/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
    // Compute (z) and (z+1)
    movapd   xmm1,  [esi+128]     ; load m20|m20
    mulpd    xmm1,  xmm0
    movapd   xmm3,  [esi+144]     ; load m21|m21
    mulpd    xmm3,  xmm2
    movapd   xmm5,  [esi+160]     ; load m22|m22
    mulpd    xmm5,  xmm4
    addpd    xmm3,  xmm1
    movapd   xmm7,  [esi+176]     ; load m23|m23
    addpd    xmm5,  xmm3
    addpd    xmm7,  xmm5
    movapd   [edx+eax],  xmm7     ; store (z+1)'|z'

/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
    // Compute (w) and (w+1)
    movapd   xmm1,  [esi+192]     ; load m30|m30
    mulpd    xmm1,  xmm0
    movapd   xmm3,  [esi+208]     ; load m31|m31
    mulpd    xmm3,  xmm2
    movapd   xmm5,  [esi+224]     ; load m32|m32
    mulpd    xmm5,  xmm4
    addpd    xmm3,  xmm1
    movapd   xmm7,  [esi+240]     ; load m33|m33

```

```
    addpd    xmm5,  xmm3
    addpd    xmm7,  xmm5
    movapd   [edi+eax], xmm7    ; store (w+1)'|w'

    add     eax, 16
    jmp     LOOP
END_LOOP:
    }
}
```

9 コンパイラのベクトル化を使用したコード例

```
int VecAOS ()
{
    Aos *vertex = AOS_Vertex;
    int loc_length = length;
    double m00 = M44->m00, m01 = M44->m01, m02 = M44->m02, m03 = M44->m03,
           m10 = M44->m10, m11 = M44->m11, m12 = M44->m12, m13 = M44->m13,
           m20 = M44->m20, m21 = M44->m21, m22 = M44->m22, m23 = M44->m23,
           m30 = M44->m30, m31 = M44->m31, m32 = M44->m32, m33 = M44->m33;

    #pragma ivdep
    #pragma vector aligned
    for (int i=0; i < loc_length; i++)
    {
        DBL tx, ty, tz, tw;

        tx = m00*vertex[i].x + m01*vertex[i].y + m02*vertex[i].z + m03;
        ty = m10*vertex[i].x + m11*vertex[i].y + m12*vertex[i].z + m13;
        tz = m20*vertex[i].x + m21*vertex[i].y + m22*vertex[i].z + m23;
        tw = m30*vertex[i].x + m31*vertex[i].y + m32*vertex[i].z + m33;

        vertex[i].x = tx; vertex[i].y = ty; vertex[i].z = tz;
        vertex[i].w = tw;
    }

    return EXIT_SUCCESS;
}
```

```
int VecSOA ()
{
    int loc_length = length;
    double m00 = M44->m00, m01 = M44->m01, m02 = M44->m02, m03 = M44->m03,
           m10 = M44->m10, m11 = M44->m11, m12 = M44->m12, m13 = M44->m13,
           m20 = M44->m20, m21 = M44->m21, m22 = M44->m22, m23 = M44->m23,
           m30 = M44->m30, m31 = M44->m31, m32 = M44->m32, m33 = M44->m33;
    double *x = SOA_Vertex.x, *y = SOA_Vertex.y, *z = SOA_Vertex.z,
           *w = SOA_Vertex.w;

    #pragma ivdep
    #pragma vector aligned
    for (int i=0; i < loc_length; i++)
    {
        DBL tx, ty, tz, tw;

        tx = m00*x[i] + m01*y[i] + m02*z[i] + m03;
        ty = m10*x[i] + m11*y[i] + m12*z[i] + m13;
        tz = m20*x[i] + m21*y[i] + m22*z[i] + m23;
        tw = m30*x[i] + m31*y[i] + m32*z[i] + m33;

        x[i] = tx;
        y[i] = ty;
        z[i] = tz;
        w[i] = tw;
    }

    return EXIT_SUCCESS;
}
```

付録 A - パフォーマンス・データ

パフォーマンス・データの改訂履歴

改訂番号	改訂履歴	改訂時期
2.0	Pentium® 4 プロセッサ 1.20 GHz のパフォーマンス・データによる更新	2000年7月
1.0	初版	1999年9月

表 1: 3D トランスフォーメーション・コードのパフォーマンス・データ

パフォーマンス・データ(nsec 単位)		
コード	Pentium® III プロセッサ (733 MHz)	Pentium 4 プロセッサ (1.2 GHz)
C コード AOS	60.1	29.9
C コード SOA	63.3	32.4
アセンブリ SSE2 AOS		18.5
アセンブリ SSE2 SOA		14.6
ベクトル・クラス SSE2 SOA		15.4
組み込み関数 SSE2 AOS		19.4
組み込み関数 SSE2 SOA		16.1
ベクトライザ SSE2 AOS		19.4
ベクトライザ SSE2 SOA		13.7

表 2: 表 1 のパフォーマンス・データに基づくスピードアップの比

コードおよびプラットフォーム	スピードアップ
Pentium 4 プロセッサ(ベクトライザ SSE2 SOA と C コード AOS)	2.18
Pentium 4 プロセッサ上の C コード AOS と Pentium III プロセッサ上の C コード AOS	2.01

パフォーマンスは、Pentium III プロセッサ 733 MHz と Pentium 4 プロセッサ 1.20 GHz を使用して測定された。テスト・システムについての詳細は、A-3 ページの「テスト・システムの構成」を参照のこと。

最初に、すべてのデータが 1 次キャッシュ内に保持されている状態(完全キャッシュ)でパフォーマンスを測定した。表 1 にテストの結果を示す。ベクトル長は、執筆時点でのグラ

フィックス標準で、オブジェクト当たりの平均ベクトル・サイズとして1000に設定された。ベクトルの各要素に対して、6回の浮動小数点演算(3回の乗算と3回の加算)を実行した。

倍精度3Dトランスフォーメーションでは、SSE2コードはスカラ・コードに対して2倍のパフォーマンスが得られると予測される。これは、レジスタの幅が2倍になるため、元のCコードと同じ命令数で2倍の処理を実行できるからである。構造体配列(AoS)形式のテストで、この予測が正しいことが実証された。このテストの結果、SSE2コードは、基準となるCコードより1.5~1.6倍高速であった。

予想どおり、配列構造体(SOA)コードはさらに高速であり、基準となるコードに対して2.18倍の速度が得られた。これは、メモリ構造の修正によって、より効率的なアクセスが可能になるからである。1回のデータ読み出しにつき2個のデータ要素にアクセスできるため、シャッフルのオーバーヘッドが不要になり、メモリ・レイテンシによるパイプライン型マイクロアーキテクチャのストールが減少する。また、SOAベクトライザを使用したコードはアセンブリ・コードより高速なことに注意する。このテストでは、ベクトライザ・コードは、SIMD命令を最も高速に実行できるだけでなく、最も高いパフォーマンスを達成した。

テスト・システムの構成

表 3: Pentium III プロセッサのシステム構成

プロセッサ	Pentium III プロセッサ 733MHz
システム	インテル® Desktop Board VC820
BIOS のバージョン	VC82010A.86A.0028.P10
2次キャッシュ	256KB
メモリ・サイズ	128 MB RDRAM PC800-45
Ultra ATA ストレージ・ドライバ	製品版候補 6.00.012
ハードディスク	IBM DJNA-371800 ATA-66
ビデオ・コントローラ/ バス	Creative Labs 3D Blaster [†] Annihilator [†] Pro AGP nVidia GeForce256 [†] DDR -32MB
ビデオ・ドライバの リビジョン	NVidia リファレンス・ドライバ 5.22
オペレーティング・ システム	Windows [†] 2000 ビルド 2195

表 4: Pentium 4 プロセッサのシステム構成

プロセッサ	Pentium 4 プロセッサ 1.20 GHz
システム	インテル Desktop Board D850GB
BIOS のバージョン	GB85010A.86A.0014.D.0007201756
2次キャッシュ	256KB
メモリ・サイズ	128 MB RDRAM PC800-45
Ultra ATA ストレージ・ ドライバ	製品版候補 6.00.012
ハードディスク	IBM DJNA-371800 ATA-66
ビデオ・コントローラ/ バス	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR -32MB
ビデオ・ドライバの リビジョン	NVidia リファレンス・ドライバ 5.22
オペレーティング・ システム	Windows 2000 ビルド 2195