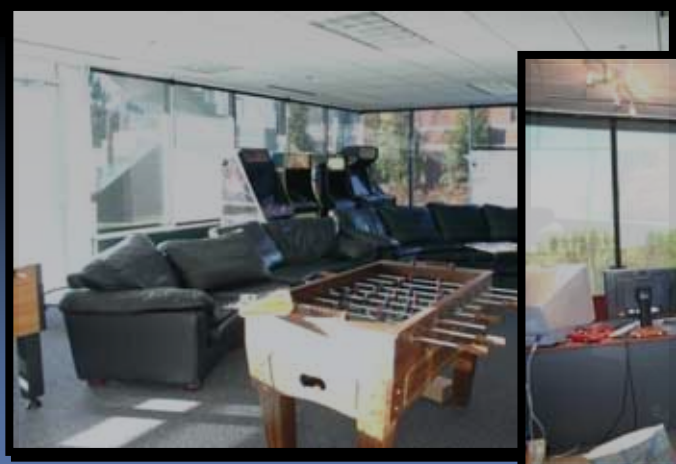
Tim Sweeney
Founder, Programmer, CEO
Epic Games
tim@epicgames.com

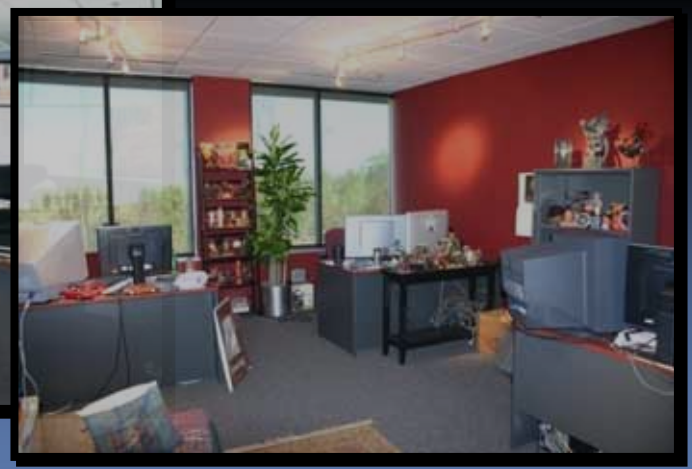
GAME DEVELOPMENT 2012-2020

Background:
Epic Games

Epic Gamesの紹介

- 独立ゲーム開発会社
- 米国ノースキャロライナ州ラリー
- 1991年設立
- 30以上のゲームをリリース
  - Gears of War
  - Unreal シリーズ
- ゲームエンジンのトップメーカー





History:
Unreal Engine

Unreal Engine 1
1996-1999

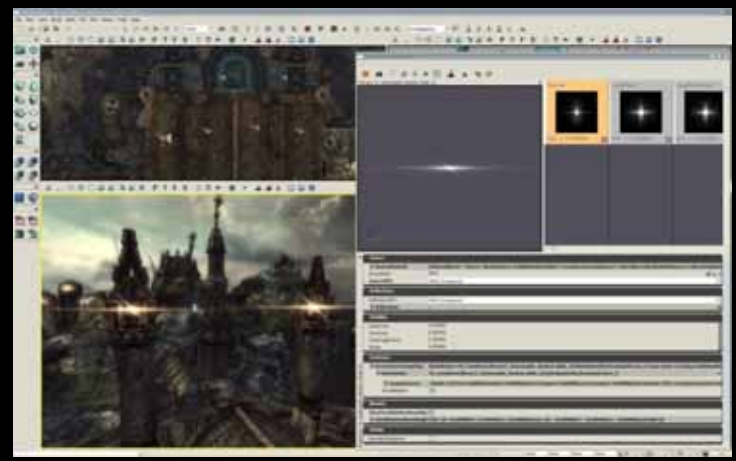
- 最初の現代的ゲームエンジン
  - オブジェクト指向
  - リアルタイム、ビジュアルツールセット
  - スクリプト言語
- 最後の大規模ソフトウェアレンダラー
  - ソフトウェアテクスチャマッピング
  - カラーライティング、シャドウ
  - ボリュームライティング &フォグ
  - ピクセル単位カリング
- 25 本のゲームを出荷

Unreal Engine 2
2000-2005

- PlayStation 2, Xbox, PC
- DirectX 7 グラフィックス
- シングルスレッド
- 40 本のゲームを出荷

Unreal Engine 3
2006-2012

- PlayStation 3, Xbox 360, PC
- DirectX 9 **グラフィックス**
  - **ピクセルシェイダ**
  - **高度なライティングとシャドウ**
- **マルチスレッド** (6 threads)
- **高度な物理**
- **もっと多くのビジュアルツール群**
  - **ゲームスクリプティング**
  - **マテリアル**
  - **アニメーション**
  - **シネマティクス…**
- 150+ **本のゲームの開発で使用されている**



Unreal Engine 3 Games

Mass Effect (BioWare)

Army of Two (Electronic Arts)

Undertow (Chair Entertainment)

BioShock (2K Games)

Game Development: 2008

Gears of War 2: プロジェクト概要

- プロジェクト規模
  - 15 人のプログラマ
  - 45 人のアーティスト
  - 2 年のスケジュール
  - 1200万ドル (　13億円) の予算
- ソフトウェア依存性
  - 1 つのミドルウェアゲームエンジン
  - 20 くらいのミドルウェアライブラリ
  - プラットフォームライブラリ

Gears of War 2: ソフトウェア依存性

Gears of War 2
ゲームプレイコード
25**万行くらいの**C++, **スクリプトコード**

Unreal Engine 3
ミドルウェアゲームエンジン
200**万行くらいの**C++**コード**

DirectX,
OpenGL
Graphics

OpenAL
Audio

Speed
Tree
Rendering

FaceFX
Face
Animation

Bink
ムービー
Codec

ZLib
データ
圧縮

...

Hardware:
History

コンピュータの歴史

1985 Intel 80386: **ｽｶﾗｰ, ｲﾝｵｰﾀﾞｰ** CPU

1989 Intel 80486: **ｷｬｯｼｭ!**

1993 Pentium: **ｽｰﾊﾟｰｽｶﾗｰ実行**

1995 Pentium Pro: **ｱｳﾄｵﾌﾞｵｰﾀﾞｰ実行**

1999 Pentium 3: **浮動小数点ベクター**

2003 AMD Opteron: **マルチコア**

2006 PlayStation 3, Xbox 360: **“メニイコア”**

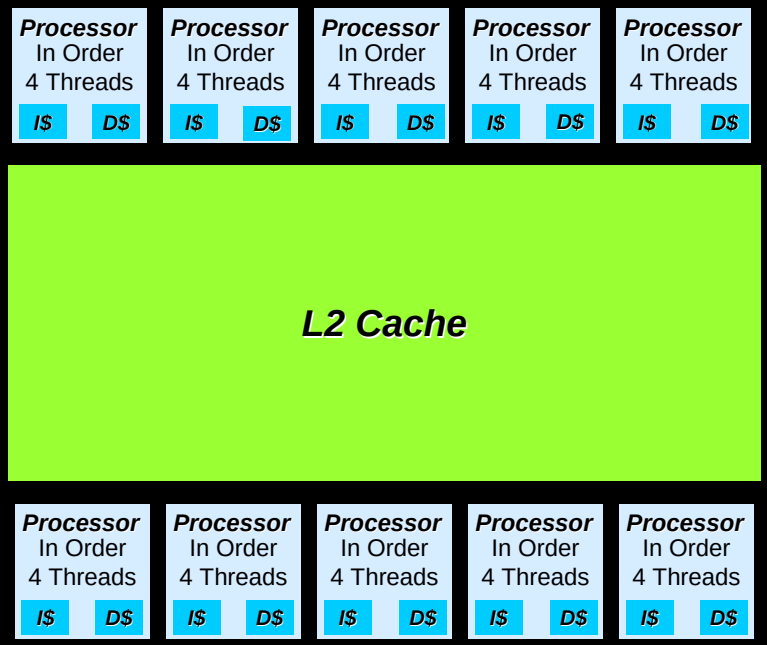
…そして、インオーダー実行に逆戻り

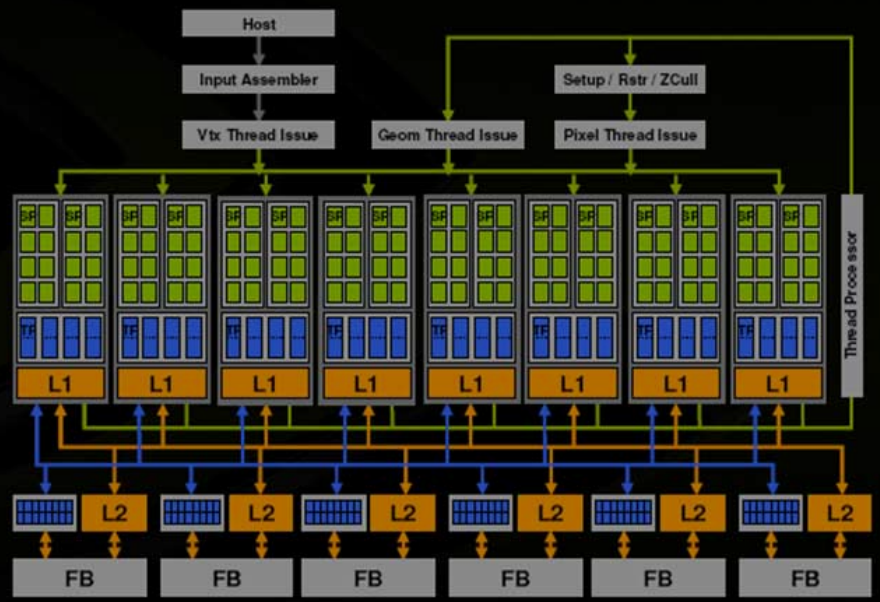
グラフィックスの歴史

1984	3D **ワークステーション** (SGI)
1997	GPU (3dfx)
2002	DirectX9, **ピクセルシェイダー** (ATI)
2006	**ﾌﾙﾌﾟﾛｸﾞﾗﾐﾝｸﾞ言語サポートの**GPU (NVIDIA GeForce 8)
2009?	x86 CPU/GPU **複合型** (**インテル** Larrabee)

Hardware:
2012-2020

ハードウェア: 2012-2020





インテル Larrabee

- x86 CPU-GPU **複合型**
- C/C++ **コンパイラー**
- DirectX/OpenGL
- **メニイコア, ﾍﾞｸﾀｰｱｰｷﾃｸﾁｬ**
- Teraflop**級性能**

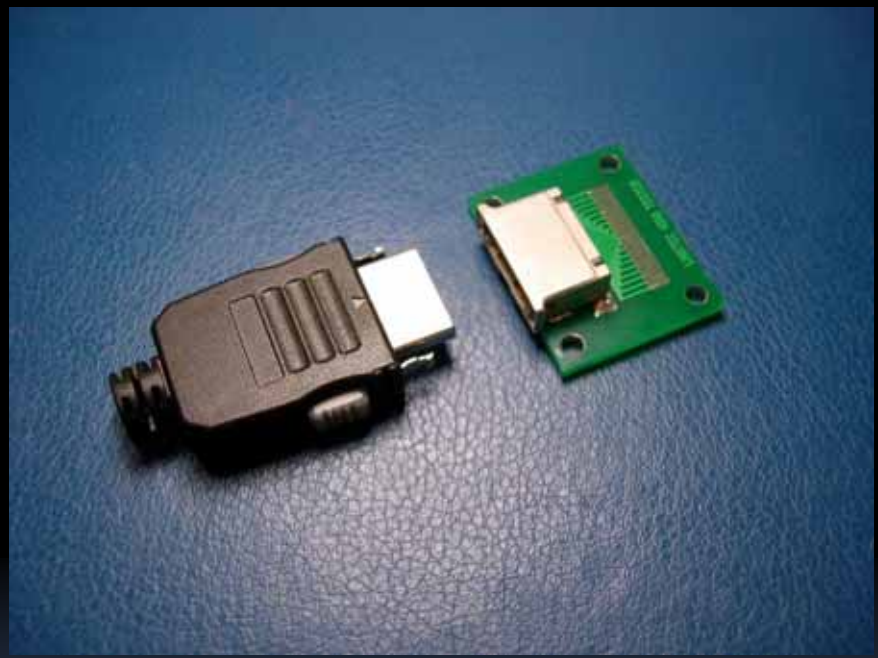
NVIDIA GeForce 8

- **汎用** GPU
- CUDA "C" **コンパイラー**
- DirectX/OpenGL
- **メニイコア, ﾍﾞｸﾀｰｱｰｷﾃｸﾁｬ**
- Teraflop**級性能**

ハードウェア: 2012-2020

つまり
CPUとGPUのアーキテクチャは収束しつつある

未来のグラフィックスハードウェア



他は全て*計算のみで!*

2012-2020年に登場し得るハードウェア:
計算とグラフィックスを統合したアーキテクチャ

ハードウェアモデル

- 性能の三つの軸
  - 周波数
  - コアの数
  - ベクトルの幅
- 二種類のコードが実行される:
  - スカラーコード (x86やPowerPCのような)
  - ベクターコード (GPUｼｪｲﾀﾞや SSE/Altivecのような)
- 多少の固定機能ハードウェア
  - テクスチャサンプリング
  - ラスタライゼーション?

Game Development Technology:

2012-2020

ゲーム開発: 2012-2020

- **プログラミング**
  - **どうやって、50 コア用のコードを書けばいい?**
- **グラフィックス**
  - DirectX / OpenGL**を超える可能性とは?**
- **学んだこと**

PROGRAMMING: 2012-2020

プログラミング: 2012-2020

"コンピューティングの中核になる挑戦は、いかにヘマをしないかだ"

- エドガー・ダイクストラ, 1972

プログラミング: 2012-2020
重要なこと

- **開発者の生産性**
  - **プログラマーの時間は貴重**
  - **生産性は非常に重要!**
  - **マルチコアのプログラミングを簡単にしなければならない!**
- **性能**
  - **マルチスレッドで "メニイコア" をサポートする**
  - **"ベクター命令セット" にスケールする**

Multithreading: 2012-2020

Unreal Engine 3 でのマルチスレッド: “タスク並列”

- ゲームプレイスレッド
  - AI, スクリプト
  - 数千の相互作用するオブジェクト
- 描画スレッド
  - シーンのトラバーサル, オクルージョン
  - Direct3D コマンド実行
- その他の仕事のために、ヘルパースレッドをプール
  - 物理ソルバー
  - アニメーションの更新

4 コアなら良い。
40 コアには良くない!

“共有ステート並列”
C++ や Java の標準的なスレッドモデル

- **多くのスレッドが動いている**
- 512MB **のデータ**
- **どのスレッドも、任意のデータを、好きな時に変更できる**
- **全ての同期は明示的に手動で行われる**
  - **参照**: LOCK, MUTEX, SEMAPHORE
- **正確性の検証をコンパイル時に行う事はできない**:
  - **デッドロックが起きないか**
  - **競合が起きないか**
  - **不変性**

Multithreaded Gameplay Simulation

ﾏﾙﾁｽﾚｯﾄﾞ化されたｹﾞｰﾑﾌﾟﾚｲｼﾐｭﾚｰｼｮﾝ

- 千以上のゲームオブジェクト
- それぞれのオブジェクトは:
  - 変更可能
  - 毎フレーム更新
    - 各更新は、5-10の他のオブジェクトに影響
    - 更新はオブジェクト指向なので、
      制御フローを静的に知る事ができない
- コードは数十人のプログラマが書く
  - コンピュータサイエンティストでは無い!

ﾏﾙﾁｽﾚｯﾄﾞ化されたｹﾞｰﾑﾌﾟﾚｲｼﾐｭﾚｰｼｮﾝ

問題点:

- **ゲームは“メニイコア”にスケールしなければならない** (20-100)
- **全てのスレッドのボトルネックを解消しなければならない**

解決策:

- **“共有ステート並列”**
- **“メッセージパッシング並列”**
- **“ソフトウェアトランザクショナルメモリ”**
- **“純関数型プログラミング”**

ﾏﾙﾁｽﾚｯﾄﾞ化されたｹﾞｰﾑﾌﾟﾚｲｼﾐｭﾚｰｼｮﾝ：手動同期

アイデア:

- **複数スレッドでオブジェクトを更新**
- **それぞれのオブジェクトはロックを含む**
- **"オブジェクトを使う前に、とにかくロック"**

問題点:

- **"デッドロック"**
- **"データ競合"**
- **デバッグが難しい/時間がかかる**

ﾏﾙﾁｽﾚｯﾄﾞ化されたｹﾞｰﾑﾌﾟﾚｲｼﾐｭﾚｰｼｮﾝ：“メッセージパッシング”

アイデア:

- **複数スレッドでオブジェクトを更新**
- **それぞれのオブジェクトは、自身のみ更新可能**
- **他のオブジェクトとは、メッセージを送る事で通信**

問題点:

- **千ものメッセージプロトコルを書かねばならない**
- **それでも同期は必要**

ﾏﾙﾁｽﾚｯﾄﾞ化されたｹﾞｰﾑﾌﾟﾚｲｼﾐｭﾚｰｼｮﾝ：ソフトウェアトランザクショナルメモリ

アイデア:

- **複数スレッドでオブジェクトを更新**
- **各スレッドは、トランザクションブロックの中で動作し、それぞれの "ローカル" なメモリ変更についてアトミックなビューを持つ**
- **C++のランタイムは、トランザクション間の衝突を検出する**
  - **衝突の無いトランザクションは、"グローバル" メモリに適用される**
  - **衝突したトランザクションは "ロールバック" されて、再実行される**

完全にソフトウェアで実装される。特別なハードウェアは不要

問題点:

- **"オブジェクト更新" コードに副作用があってはいけない**
- **C++ ランタイムのサポートが必要**
- **30% ほどの性能を消費する**

See: "Composable Memory Transactions"; Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. ACM Conference on Principles and Practice of Parallel Programming 2005

マルチスレッド化されたゲームプレイシミュレーション：結論

手動同期:

- **とても難しい、エラーの原因**

メッセージパッシング

- **難しい、エラーの原因**

トランザクショナルメモリ

- **簡単! シングルスレッド並**
- **分析によると負荷も合理的, 衝突率も低い** (1-5%)

Claim: Transactional memory is the only productive, scalable solution for multithreaded gameplay simulation.

Pure Functional Programming

純関数型プログラミング

"純関数" プログラミングスタイル:

- **共有メモリへの書き込みも、I/O操作も行わないアルゴリズムを定義する**

  (作用は戻り値のみ)

例:

- **衝突判定**
- **物理ソルバー**
- **ピクセルシェイディング**

純関数型プログラミングの例: 衝突判定

線分に沿って動く点が、いつ、どこで、(固定した)ジオメトリと当たるかを調べる衝突判定

```
struct vec3
{
    float x,y,z;
};
struct hit
{
    bool  DidCollide;
    float Time;
    vec3  Location;
};
hit collide(vec3 start,vec3 end);
```

(これは、共有メモリを変更しない)

純関数型プログラミング

"副作用の無い関数の中では、下位計算は、任意の順序で行う事ができる。
並列でも構わない。
これは関数の結果に影響を与えない"

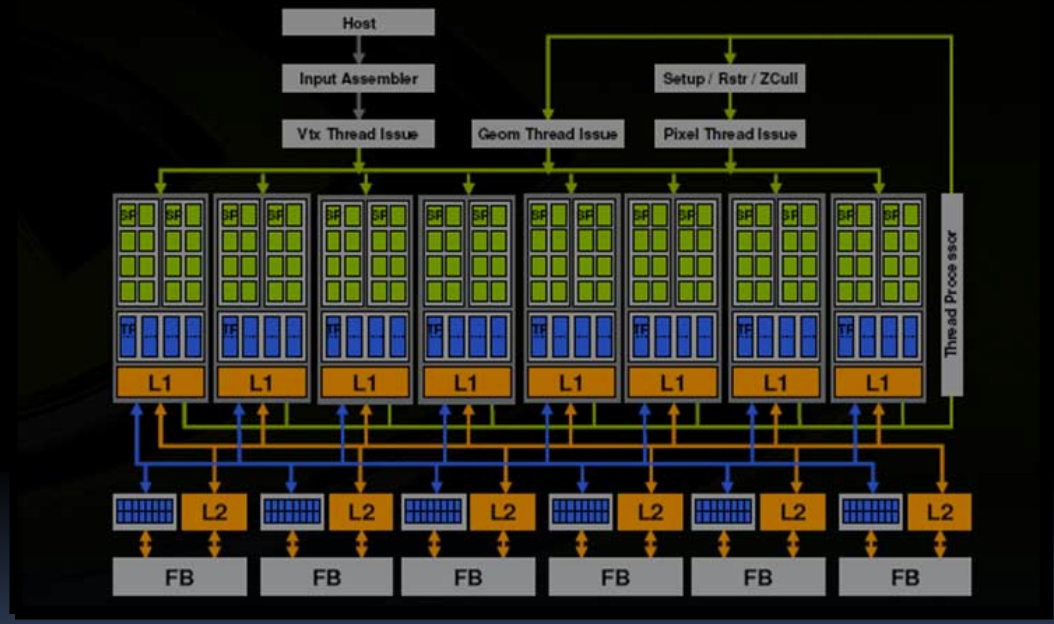
この性質により:

- プログラマは、明示的なマルチスレッドコードを安全に書く事が出来る
- 将来のコンパイラは、コードを**自動的に**マルチスレッド化することが安全に行える

See: "**Implementing Lazy Functional Languages on Stock Hardware**"; Simon Peyton Jones; Journal of Functional Programming 2005

ベクトル化

“ベクトル命令セット”を効率的にサポート



NVIDIA GeForce 8:

- 8〜15 **コア**
- 16**幅ベクトル**

ベクトル化

C++ や Java コンパイラは “スカラー” コードを生成する

GPU シェイダーコンパイラは、“ベクトル” コードを生成する

- 自由なベクトルサイズ (4, 16, 64, …)
- N幅ベクトルで、N幅の高速化を得られる

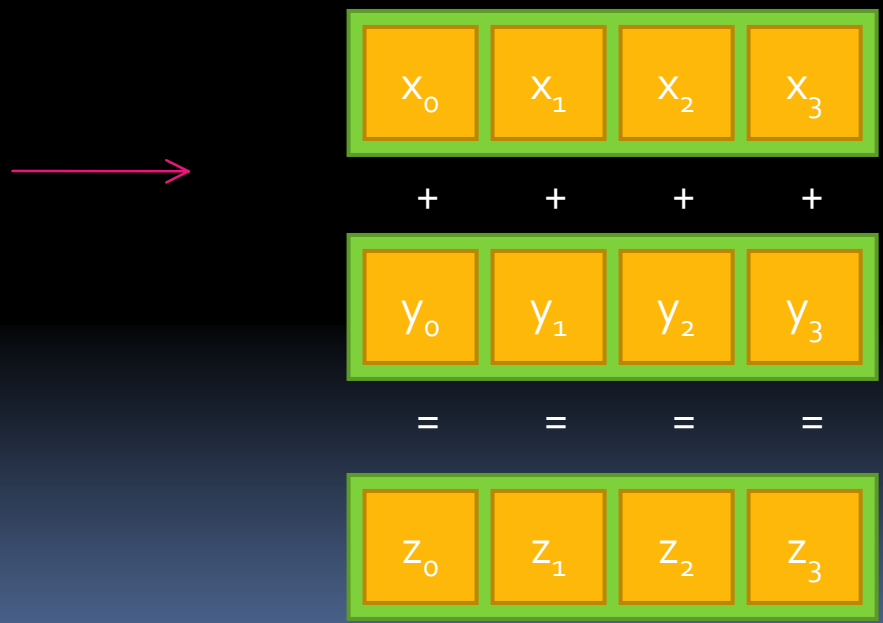
ベクトル化：“昔の方法”

- “**古いベクトル**” (SIMD):
  インテル SSE, **モトローラ** Altivec
  - 4**幅ベクトル**
  - 4**幅演算操作**
  - **ベクトルのロード**
    メモリに格納されたベクトルを、ベクトルレジスタにロード
  - **ベクトルのスウィズルとマスク**

将来のプログラミングモデル：ベクトル化

- **“古いベクトル”**
  インテル SSE, **モトローラ** Altivec

```
vec4 x,y,z;
...
z = x+y;
```



ベクトル化：“新しいベクトル”

(ATI, NVIDIA GeForce 8, **インテル** Larrabee)

- 16**幅ベクトル**
- 16**幅演算**
- **ベクトルロード/ストア**
  - 16**の*独立した*メモリアドレスから得られるスカラーを、16幅のベクトルレジスタにロード。アドレスは、別のベクトルに格納されている!**
  - **類似**: DirectX **のレジスタインデックスされたコンスタントアクセス**
- **条件付きベクトルマスク**

“新しいベクトル” は “古いベクトル” より、良い

- **“古いベクトル” は、ベクトル的なデータ型の場合のみ有用だった:**
  - **グラフィックスの “XYZW” ベクトル**
  - **4x4 行列**
- **“新しいベクトル” はもっと強力:**

  内部が順列依存関係のない静的なコールグラフであれば、どんなループでも “ベクトル化” できるし、等価な16幅ベクトルプログラムにコンパイルできる。しかも、16倍速い!

“新しいベクトル” は、汎用的

```
int n;
cmplx coords[];
int color[] = new int[n]

for(int i=0; i<n; i++) {
    int j=0;
    cmplx c=cmplx(0,0)
    while(mag(c) < 2) {
          c=c*c + coords[i];
          j++;
    }
    color[i] = j;
}
```

(マンデルブロート集合の生成)

このコードは…

- 順列依存がない
- 静的なコールグラフ

なので、これを機械的に等価な並列コードフラグメントに変換できる

“新しいベクトル” 変換

```
for(int i=0; i<n; i++) {
    …
}
```

```
for(int i=0; i<n; i+=N) {
    i_vector={i,i+1,..i+N-1}
    i_mask={i<n,i+1<N,i+2<N,..i+N-1<N}
    …
}
```

標準的なデータ並列ループセットアップ

注: (このループを呼び出す) ループ外のコードは、全てスカラーである必要がある!

“新しいベクトル” 変換

```
int n;
cmplx coords[];
int color[] = new int[n]

for(int i=0; i<n; i++) {
     int j=0;
     cmplx c=cmplx(0,0)
     while(mag(c) < 2) {
            c=c*c + coords[i];
            j++;
     }
     color[i] = j;
}
```

注: (このループを呼び出す) ループ外のコードは、全てスカラーである必要がある!

```
int n;
cmplx coords[];
int color[] = new int[n]

for(int i=0; i<n; i+=N) {
     int[N] i_vector={i,i+1,..i+N-1}
     bool[N] i_mask={i<n,i+1<N,i+2<N,..i+N-1<N}

     complx[N] c_vector={cmplx(0,0),..}

     while(1) {
            bool[N] while_vector={
                        i_mask[0] && mag(c_vector[0])<2,
                        ..
            }
            if(all_false(while_vector))
                        break;
            c_vector=c_vector*c_vector + coords[i..i+N-1 : i_mask]
     }
     colors[i..i+N-1 : i_mask] = c_vector;
}
```

ループインデックスベクトル

ループマスクベクトル

ベクトル化されたループ変数

ベクトル化された条件:
ループマスクを局所条件に伝播

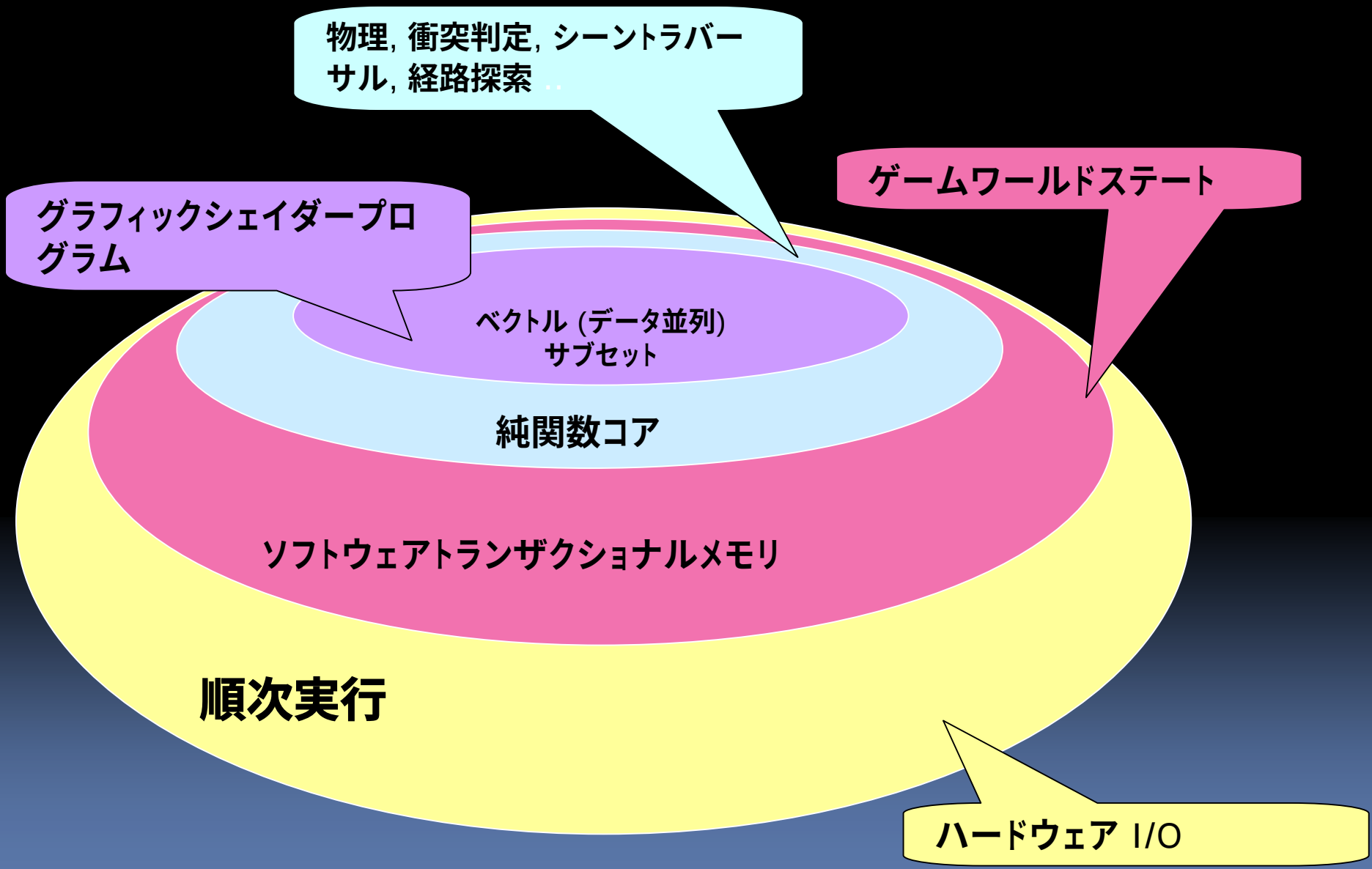
マスクを用いたベクトル書き込み

マスクを用いたベクトル読み出し

ベクトル化のトリック

- ループのベクトル化
  - ループ変数と独立の部分式はスカラーであり、ループの外に出す事ができる
  - ループ変数に依存する部分式はベクトル化する
  - 各ループで、Nコンポーネントのサブセット計算を有効化する"有効マスク" を計算
- 関数呼び出しのベクトル化
  - 全てのスカラー関数について、N幅 "有効マスク" を受け取るN幅ベクトル版を生成する
- 条件のベクトル化
  - N幅の条件を評価して、現在の有効マスクを用いて統合
  - マスク化条件のどれかが真ならば、"true" 節を実行
  - マスク化条件のどれかが偽ならば、"false" 節を実行
  - 両方の節が実行されることが多い

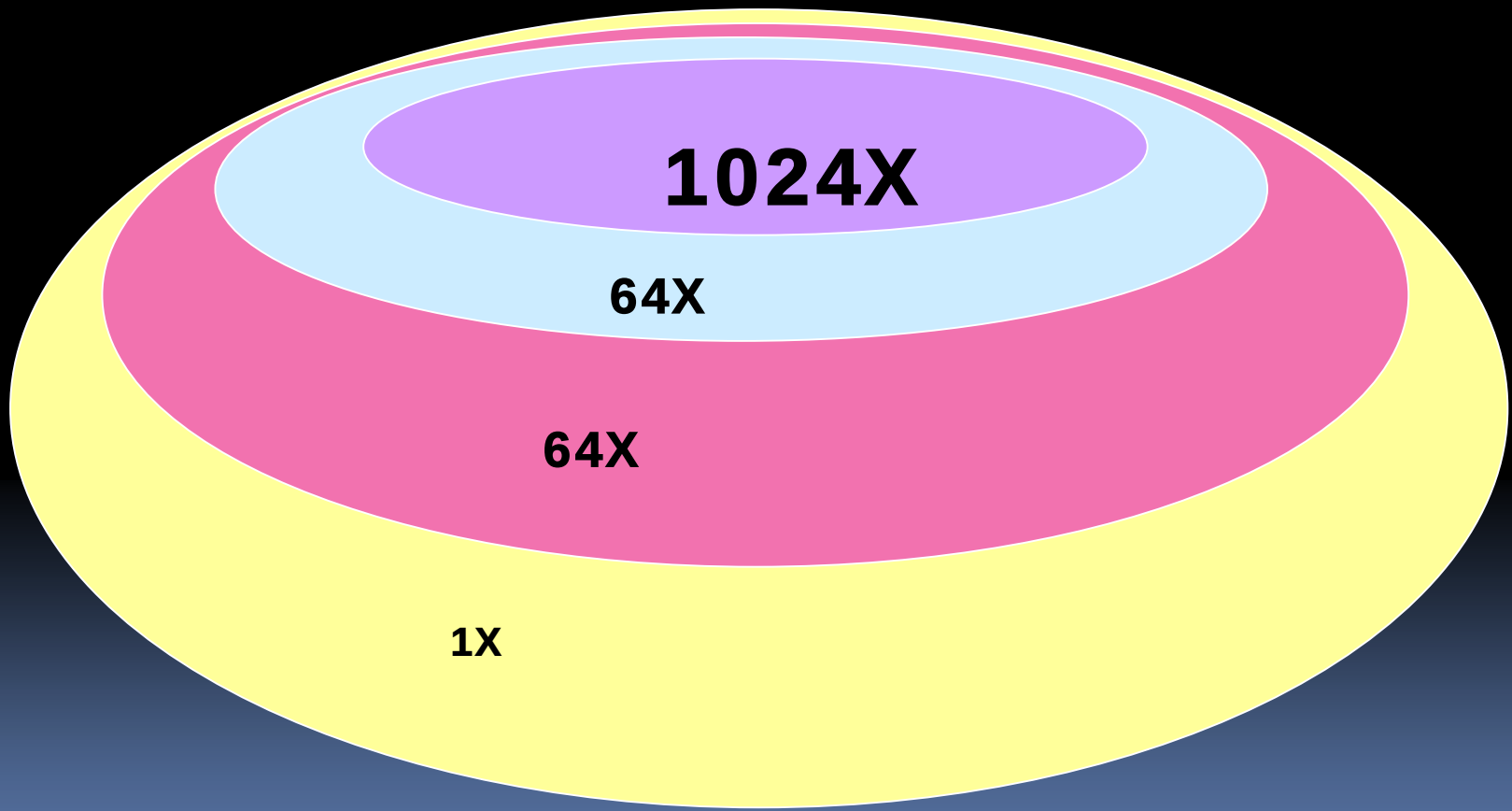
層： マルチスレッド & ベクトル



性能向上の可能性*: 2012-2020

最高で…

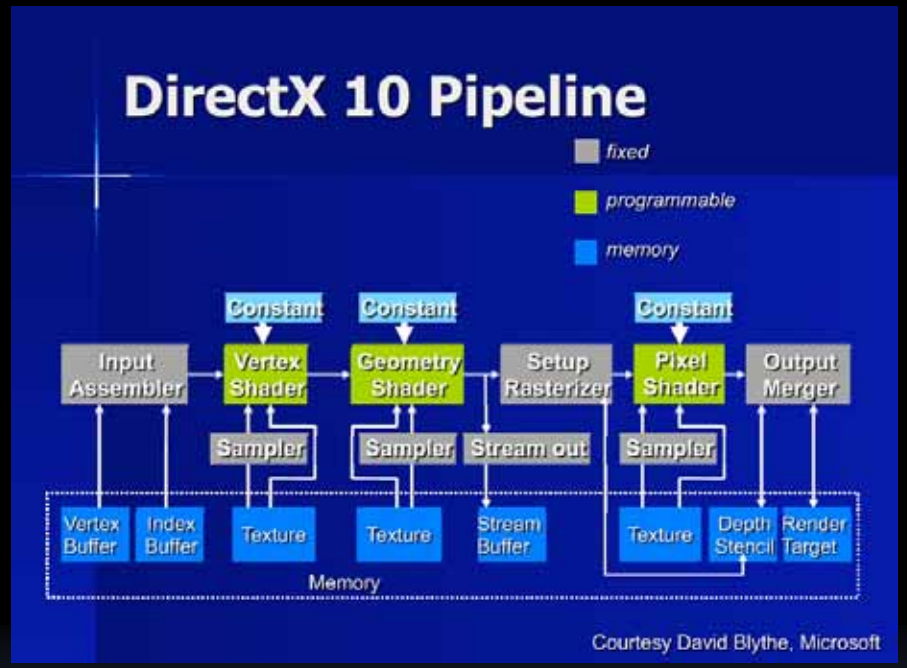
- **マルチスレッドについて64倍**
- **マルチスレッド＋ベクトルについて1024倍!**



*** この推測はムーアの法則が成り立つという仮定に基づく**

GRAPHICS: 2012-2020

2008年の GPU プログラミング: モデル



- 大きなフレームバッファ
- 複雑なパイプライン
- これは固定機能
- しかし、
  シェイダープログラム
  を、いくつかのパイプラインステージで実行できる

現在の GPU プログラミング：シェイダープログラムの制限

- ランダムなメモリ書き込みができない
  - ﾌﾚｰﾑﾊﾞｯﾌｧの現在のﾋﾟｸｾﾙには書き込める
  - データ構造を作成できない
- データ構造をトラバースできない
  - テクスチャアクセスを使ったハックは可能
- メインプログラムとシェイダープログラム間でデータを共有するのが難しい
- 奇妙なプログラミング言語
  - C/C++ではなくHLSL

現在の GPU プログラミング：問題点

- **全てのゲームが同じように見える**
  - **固定機能パイプラインステージのせい**
- **“シェイダーALUの停滞”**
  - **ピクセルシェイダーの性能が10倍になっても、ゲームは2倍良く見えるだけだろう**
- **アンチエイリアシングモデルが貧弱** (MSAA)

将来のグラフィックス：
100% “ソフトウェア” レンダリングへの回帰

- OpenGL/DirectX API **をバイパスする**
- 100% **のソフトウェアレンダラーを実装**
  - **全ての固定機能ﾊﾟｲﾌﾟﾗｲﾝﾊｰﾄﾞｳｪｱをﾊﾞｲﾊﾟｽ**
  - **イメージを直接生成**
  - **複雑なデータ構造を作成し、トラバース**
  - **無限の可能性**

これは、以下で実装できる…

- **インテル** CPU **上に** C/C++
- NVIDIA GPU **上に** CUDA (DirectX**は使わない**)

Unreal 1 のソフトウェアレンダリング（1998）

CPUで100%走っていた

GPU は必要なかった!

機能

- リアルタイムカラーライティング
- ボリュームフォグ
- タイルレンダリング
- 遮蔽検出

1998年と2012年のソフトウェアレンダリング

60 MHz Pentium **は、以下を実行できた**:

1ピクセルあたり16 命令

320x200, 30 Hz

2012**年の、**4 Teraflop **プロセッサは**:

1ピクセルあたり、16000 命令

1920x1080, 60 Hz

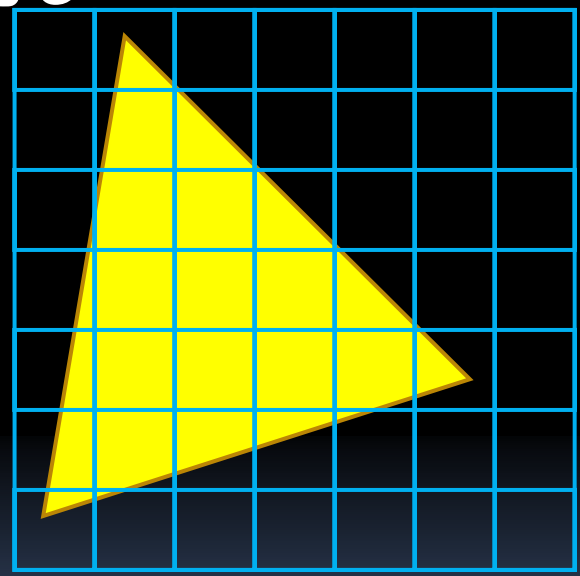
仮定: **計算力の**50%**をグラフィックス、**50%**をゲームプレイに使う場合**

将来のグラフィックス:
レイトレーシング

- 各ピクセルに対して
  - シーン内へのレイをキャスト
  - どのオブジェクトにヒットするかを検査
  - 反射、屈折等に続く
- 考察
  - 純粋なレンダリングより効率が悪い
  - 古典的なレンダラー内で、反射に利用可能

将来のグラフィックス:
REYES レンダリングモデル

- シーン内の全てのオブジェクトを、サブピクセルサイズの三角形まで“さいの目”に切る



- 以下のように描画する
  - フラットシェイディング (!)
  - 解析的アンチエイリアシング
- 利点
  - ディプレースメントマッピングの負荷が無い
  - 解析的アンチエイリアシング
  - 高度なフィルタリング (ガウシアン)
  - テクスチャサンプリング不要

将来のグラフィックス:
REYES レンダリングモデル

現在のパイプライン

- 400万ポリゴンの "高精細" キャラクタを作成
- 高精細ジオメトリから法線マップを生成
- ゲーム内では2万ポリゴンの "低精細" キャラクタを描画

2012年に可能なﾊﾟｲﾌﾟﾗｲﾝ

- 400万ポリゴンの "高精細" キャラクタを作成
- ゲーム内で、それを描画!
- 高度なLODスキームにより、正しいサブピクセルサイズの三角形が得られる

将来のグラフィックス:
ボリュームレンダリング

- ボクセルレンダリングを直接行う
  - レイキャスティング
  - 樹木や、枝葉に有効
- 三角化したボリュームレンダリング
  - マーチングキューブ
  - マーチチングテトラヘドロン (四面体)
- ポイントクラウド
- 信号空間ボリュームレンダリング
  - フーリエ投影断層定理
  - 雲などの、半透明なボリュームデータに有用

将来のグラフィックス:
ソフトウェアタイルレンダリング

- **フレームバッファを領域** (bin) **に分割**
  - **例**: 1 bin = 8x8 **ピクセル**
- bin**を順番に処理**
  - bin**内の全てのオブジェクトを変換、ラスタライズ**

- **考察**
  - **キャッシュ効率が良い**
  - **深いフレームバッファ、アンチエイリアシング**

複合グラフィックスアルゴリズム

- **解析的アンチエイリアシング**
  - 解析的方法は、1024x MSAAより良い
- **ソート不要の半透明**
  - フラグメントのピクセル毎にソートされたリンクリストは、ピクセル毎にメモリを確保し、ポインタを追跡し、条件分岐を必要とする (Aバッファ)
- **高度なシャドウテクニック**
  - 物理的に正しい、ピクセル毎の半影ボリューム
  - 良く知られたステンシルバッファアルゴリズムの拡張
  - メモリ確保とポインタに関連付けられた、ピクセル毎のとてもシンプルなBSPツリーを保持し、トラバースし、更新する必要がある
- **非常に大量のオブジェクトを含むシーン**
  - 固定機能GPUとAPI は、10〜100倍のステートチェンジを行う欠点がある

2012-2020年のグラフィックス産業のゴールとしてあり得る事

ムービー品質を得る:

- アンチエイリアシング
- 直接照明
- シャドウ
- パーティクル効果
- 反射

劇的な改善:

- キャラクタアニメーション
- オブジェクト数
- 間接照明

LESSONS LEARNED

学んだこと：
生産性は必須！

ハードウェアは20倍速くなるが:

- **ゲーム開発の予算は2倍以下しか増えないだろう**

なので…

- **開発は、生産性を得るために、性能を犠牲にするしかない**
- **ハイレベルなプログラミングが、低レベルプログラミングに勝つ**
- **簡単に使えるハードウェアが、より速いハードウェアに勝つ**
- **凄いツールが必要: コンパイラ、エンジン、ミドルウェアライブラリ…**

学んだこと：
現在のハードウェアは難し過ぎる！

- **効率的なシングルスレッドのアルゴリズム開発コストを X (時間、お金、苦労) とすると…**
  - **マルチスレッド版のコストは 2倍**
  - PlayStation 3 Cell **版のコストは 5倍**
  - **現在の** "GPGPU" **版のコストは: 10倍かそれ以上**
- **2倍以上は、大抵のソフトウェア会社にとって経済的ではない!**
- **これは、以下に反する:**
  - **難しいプログラミングテクニックが必要なハードウェア**
  - **制限のある** "GPGPU" **プログラミングモデル**

学んだこと：
今後の計画

以前の世代:

- **エンジン開発にかかる時間は3年だった**
- Unreal Engine 3:
  - 2003: **開発開始**
  - 2006: **最初のゲーム出荷**

次世代:

- **エンジン開発には5年かかる**
- **2008年に始めたら、出荷は2013年!**

だから、今始めよう!

CONCLUSION

END