ゲームエンジンへのシェーダ統合

Bryan Dudash
NVIDIA
Developer Technology

議題

- なぜシェーダを使うか?
- シェーダとは何?
  - グラフィクスの進化
- シェーダを使う
  - 高級シェーディング言語
  - C++側のAPIとセマンティクス
  - 代替処理
  - シェーダについてのアドバイス
- ツール

なぜシェーダを使うか?

- ピクセル・シェーダは次世代ゲームを視覚的に差別化する最も重要な要素
- 類のない質感
  - メッシュ以外でディテールを作る
  - マットやプラスチック以外
  - **Blinn/Phong**からの離脱
  - カスタマイズした光源タイプ
    - 体積光源
    - **OpenGL**の固定パイプラインに制限されない

シェーダなしとシェーダあり

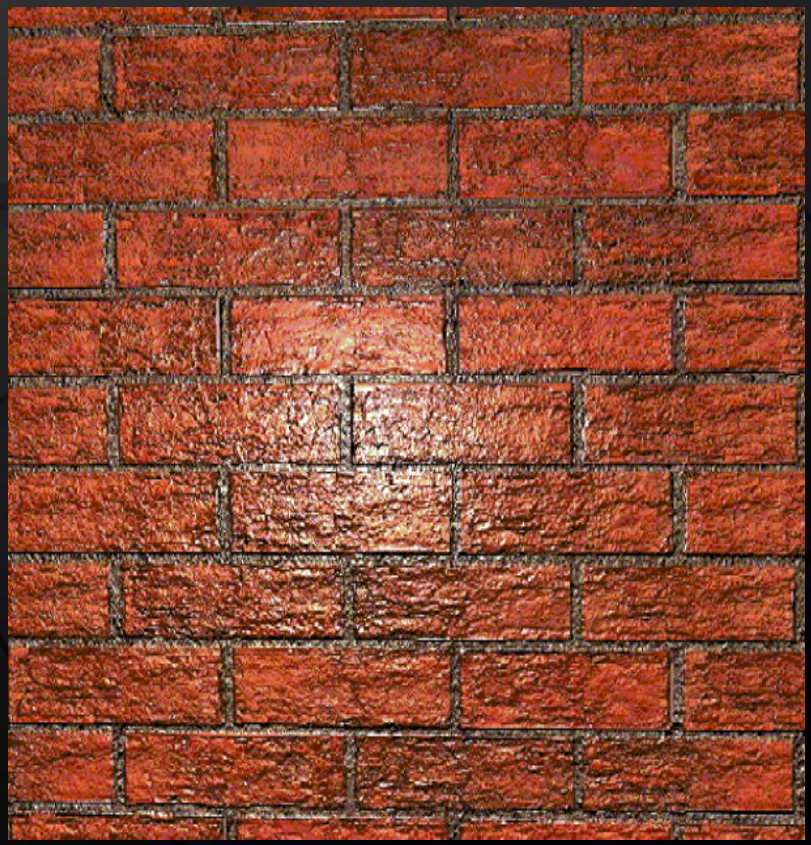
平坦なテクスチャ、単一テクスチャ、頂点単位の光源処理、影なし

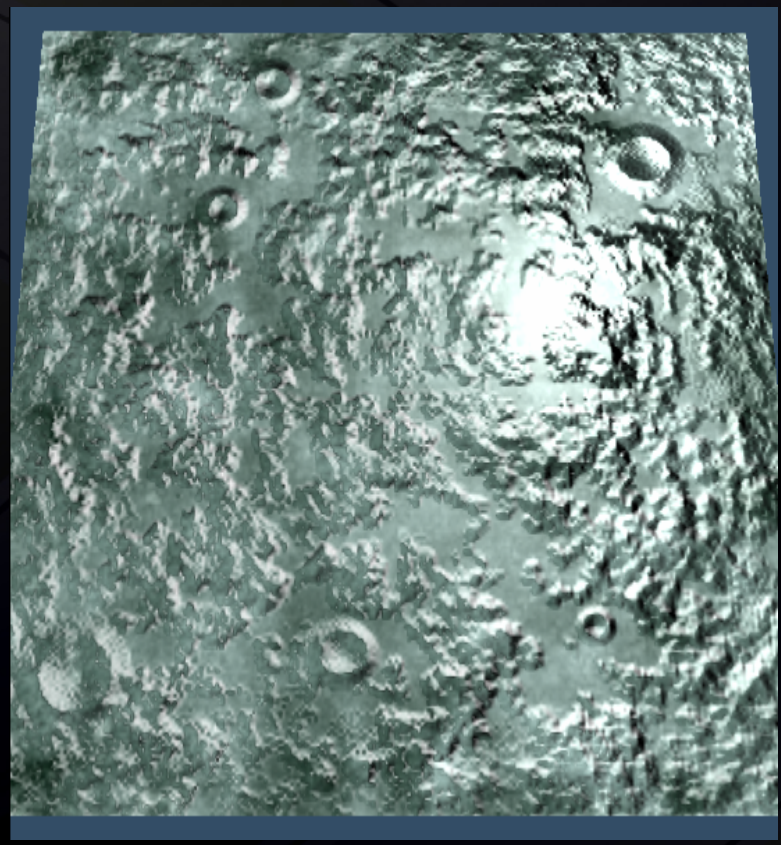
バンプ・マップ、複数テクスチャ、ピクセル単位の光源処理、やわらかい影

ピクセル単位の光源処理

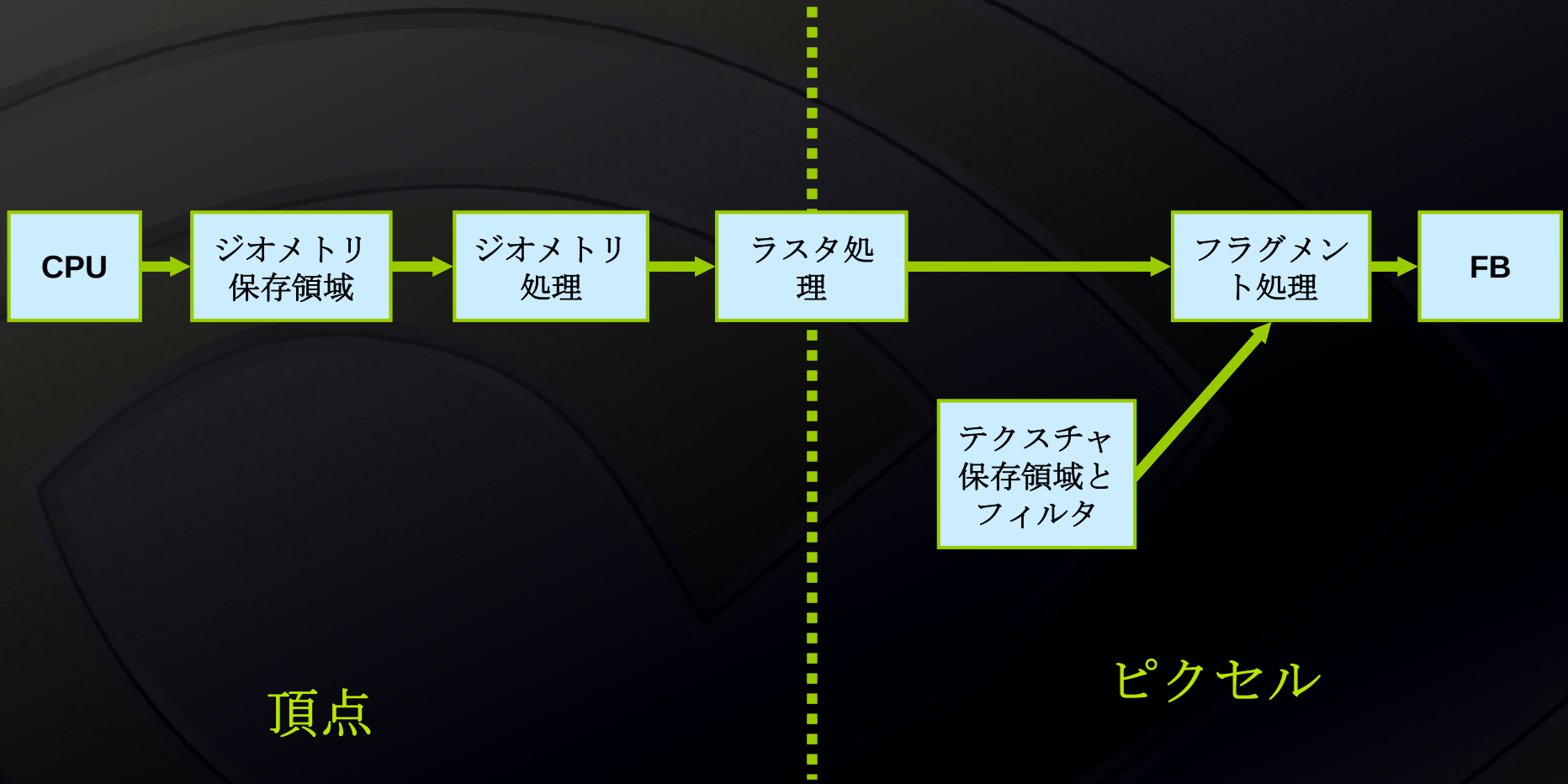
- バンプ・マップ / ノーマル・マップ / ピクセル単位の光源処理は同義語
  - **Blinn**のディフューズとスペキュラ光源処理
  - タンジェント空間のバンプ・マップ
- 頂点ごとの法線ではなく、ピクセルごとの法線で光源処理

ピクセル単位で光源処理された短形





パイプライン構成


CPU
ジオメトリ保存領域
ジオメトリ処理
ラスタ処理
フラグメント処理
FB
テクスチャ保存領域とフィルタ
頂点
ピクセル

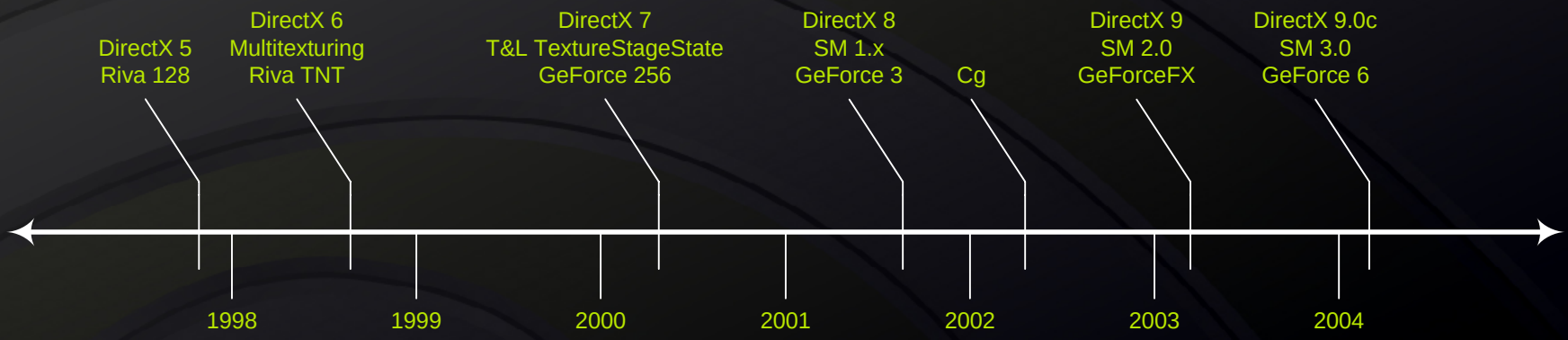
シェーダとは何?

- ユーザが定義する頂点とフラグメントでの処理
  - アニメーション、光源処理、画像処理などのカスタマイズ
- 偏在的なプラットフォームとAPIのサポート
  - PC、次世代家庭用機、携帯
  - Direct3D、OpenGL、OpenGL-ES
- Cに似た高級言語で記述
  - HLSL (Direct3D)
  - GLSL (OpenGL)
  - GLSL-ES (OpenGL-ES)
  - Cg (OpenGL、OpenGL-ES)

シェーダの分類

- 普通、ハードウェア機能はDirect3Dのシェーダ・モデル1から3にあてはめられる
- 新しいシェーダ・モデルはプログラム性が高い
  - **SM 1:** 固定小数点の色合成、静的な依存テクスチャ、 16命令以下
  - **SM 2:** 浮動小数点の演算、プログラム可能な依存テクスチャ、64命令以下
  - **SM 3:** 分岐とサブルーチン、数千命令の使用

PC / DirectXシェーダ・モデルの時系列



Quake 3

Giants

Halo

Far Cry

UE3

DirectX 5 / OpenGL 1.0とそれ以前

- ハードウェアのパイプライン
  - 入力は**DIFFUSE**、**FOG**、**TEXTURE**
  - 命令は**SELECT**、**MUL**、**ADD**、**BLEND**
  - **FOG**との合成

    RESULT = (1.0-FOG)*COLOR + FOG*FOGCOLOR

- ハードウェア例
  - **RIVA 128**、**Voodoo 1**、**Reality Engine**、**Infinite Reality**

- 『命令』、『ステージ』、繰り返しなどの概念はない

DirectX 6、OpenGL 1.2

- 複数テクスチャ
- 制御可能な『**A op B op C**』形のパイプライン
  - 命令は**SELECT**、**ADD**、**MUL**、**MAD**、**SUBTRACT**、**BLEND**
  - 入力は**CURRENT**、**DIFFUSE**、**TEXTURE**、または**CONST**
  - **RGB**とアルファに別々の設定
- ハードウェア例
  - **Riva TNT**、**Rage 128**、**Voodoo 2**、**Matrox G500**

DirectX 6時代のゲーム: Quake 3

- 複数テクスチャで効率的なライト・マップが実現でき、現実的な環境光源処理が可能に
- **DX5**時代のディヒューズ光源処理から劇的な改善

ディヒューズ光源処理

複数テクスチャによるライト・マップ

DirectX 7 / OpenGL 1.3

- 複数テクスチャの改善
  - 新しいハードウェアでは**DOT3**と**EMBM**が可能になった
  - 新しい入力: **SPECULAR**
  - キューブマップと投射テクスチャ
- ハードウェアでの変形、光源処理サポート
  - 方向光源、点光源、スポットライト
  - 頂点でのアニメーション補間、スキニング
  - テクスチャ座標変形、作成
- ハードウェア例
  - **GeForce 256、ATI Radeon、Intel Extreme Graphics 2**

DirectX 7時代のゲーム: Giants

- T&Lによって、シーンの複雑さが大きく増した
- 頂点での補間、スキニングがアニメーションを改善
- DOT3で全ての物体を動的にピクセル単位の光源処理
- 環境マップで物体にスペキュラ
- 投射テクスチャで地面に影を落とす



DirectX 6とDirectX 7

DirectX 8、SM 1.x / OpenGL 1.4

- プログラム可能な頂点シェーダ
  - 最大128浮動小数点命令
- プログラム可能なピクセル・シェーダ
  - 最大16固定小数点ベクタ命令と4テクスチャ
  - 3Dテクスチャのサポート
  - 最大1段階の依存テクスチャ
- 進化したテクスチャ描画のサポート
- ハードウェア例
  - GeForce 3、ATI Radeon 8500、XGI Volari V3、Matrox Parhelia

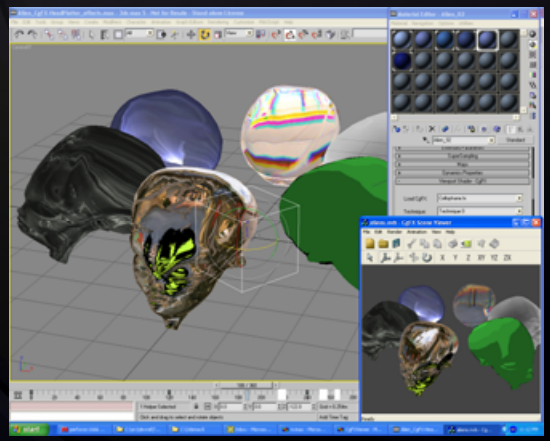
SM 1.x時代のゲーム: Halo

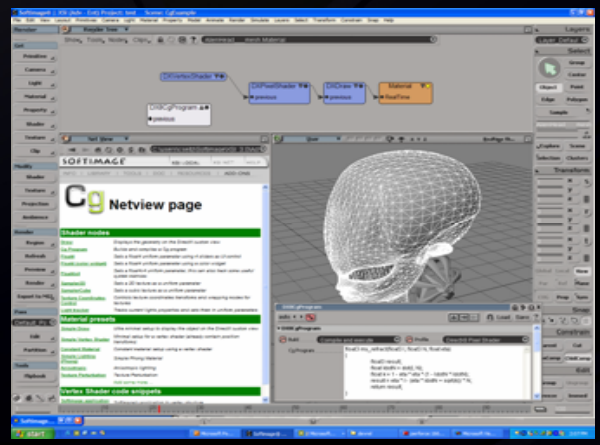
- 頂点シェーダで氷にフレネル反射
- ピクセル・シェーダで太陽に輝き
- テクスチャ描画でピストルのスコープに歪み
- 依存テクスチャで水のアニメーションと光源処理

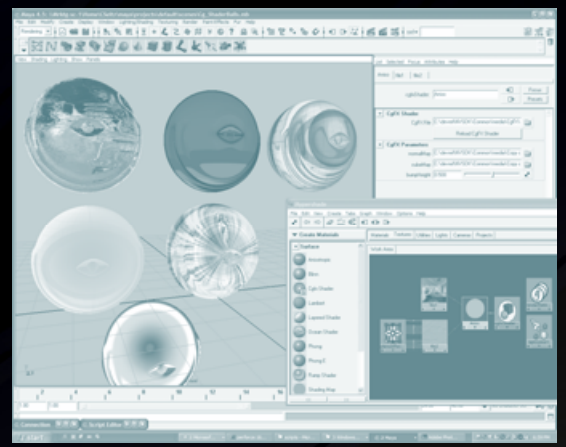
DirectX 7とDirectX 8

Cg – C for Graphics

- リアルタイムのシェーダのための高級言語
- 主なDCCアプリケーション(Maya、Max、XSI)でサポート
  - デザイナがツール上で見たものがゲームの結果と符合





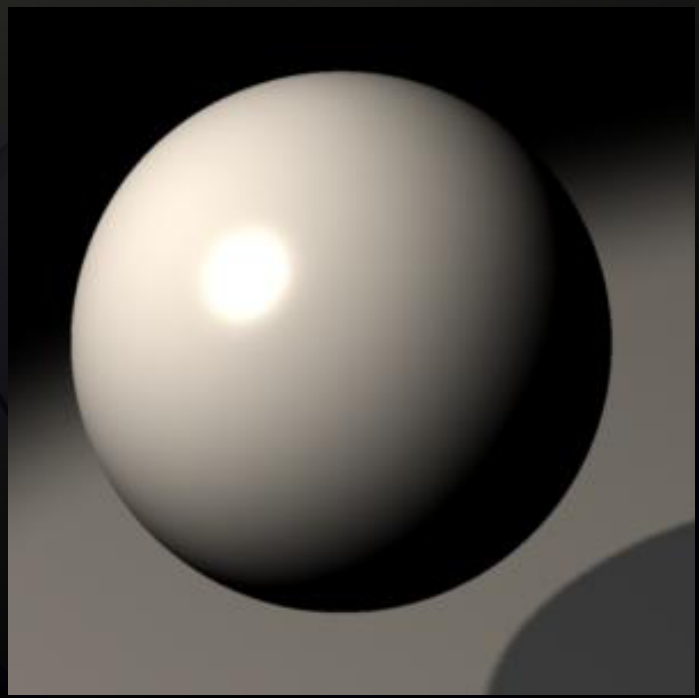


高級言語とアセンブリ

高級言語のソースコード

```
float3 L = normalize(lightPosition – position.xyz);
float3 H = normalize(L + normalize(eyePosition –
                                   position.xyz));

color.xyz = Ke + (Ka * globalAmbient) +
   Kd * lightColor * max(dot(L, N), 0) +
   Ks * lightColor * pow(max(dot(H, N), 0), shininess);
color.w = 1;
```

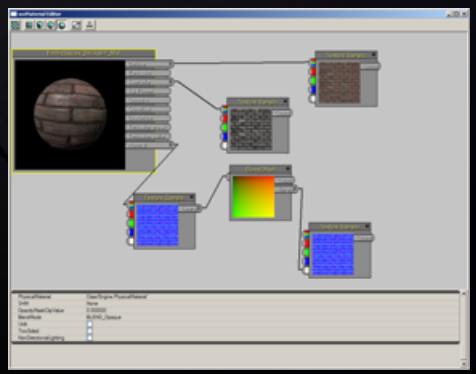


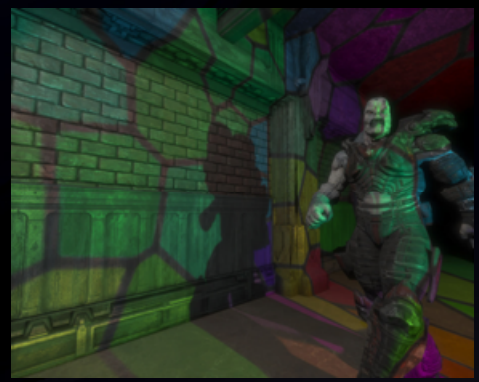
アセンブリ

```
ADDR R0.xyz, eyePosition.xyzx, -f[TEX0].xyzx;
DP3R R0.w, R0.xyzx, R0.xyzx;
RSQR R0.w, R0.w;
MULR R0.xyz, R0.w, R0.xyzx;
ADDR R1.xyz, lightPosition.xyzx, -f[TEX0].xyzx;
DP3R R0.w, R1.xyzx, R1.xyzx;
RSQR R0.w, R0.w;
MADR R0.xyz, R0.w, R1.xyzx, R0.xyzx;
MULR R1.xyz, R0.w, R1.xyzx;
DP3R R0.w, R1.xyzx, f[TEX1].xyzx;
MAXR R0.w, R0.w, {0}.x;
SLER H0.x, R0.w, {0}.x;
DP3R R1.x, R0.xyzx, R0.xyzx;
RSQR R1.x, R1.x;
MULR R0.xyz, R1.x, R0.xyzx;
DP3R R0.x, R0.xyzx, f[TEX1].xyzx;
MAXR R0.x, R0.x, {0}.x;
POWR R0.x, R0.x, shininess.x;
MOVXC HC.x, H0.x;
MOVR R0.x(GT.x), {0}.x;
MOVR R1.xyz, lightColor.xyzx;
MULR R1.xyz, Kd.xyzx, R1.xyzx;
MOVR R2.xyz, globalAmbient.xyzx;
MOVR R3.xyz, Ke.xyzx;
MADR R3.xyz, Ka.xyzx, R2.xyzx, R3.xyzx;
MADR R3.xyz, R1.xyzx, R0.w, R3.xyzx;
MOVR R1.xyz, lightColor.xyzx;
MULR R1.xyz, Ks.xyzx, R1.xyzx;
MADR R3.xyz, R1.xyzx, R0.x, R3.xyzx;
MOVR o[COLR].xyz, R3.xyzx;
MOVR o[COLR].w, {1}.x;
```

高級言語の影響

- シェーダの採用を促進
  - 年間数十のゲームから数百へ
- ゲーム開発の移行
  - シェーダがただの技術要素ではなく、デザインの必須事項に
  - 『何ができるか』ではなく『何が必要か』へ
  - デザイナがゲームの外観を制御





DirectX 9、SM 2.0 / OpenGL 1.5

- 浮動小数点ピクセル処理
  - **16/32**ビットの浮動小数点シェーダ、描画ターゲットとテクスチャ
  - 最大**64**ベクタ命令と**16**テクスチャ
  - 無制限の依存テクスチャ
- より長い頂点処理 – **256**命令
- 複数描画ターゲット – ピクセルごとに最大**16**の出力
- ハードウェア例
  - **GeForce FX 5900、ATI Radeon 9700、S3 DeltaChrome**

DirectX 9.0c、 SM 3.0 / OpenGL 2.0

- 統合されたシェーダのプログラム・モデル
  - ピクセルと頂点シェーダの流れ制御
  - 無限長の頂点、ピクセル・シェーダ
  - 頂点シェーダのテクスチャ読み出し
- 浮動小数点フィルタとブレンド
- ジオメトリ・インスタンシング
- ハードウェア例
  - **GeForce 6800、GeForce 7800 GTX**

SM 3.0時代のゲーム: Unreal Engine 3

- 16ビット浮動小数点ブレンドでHDR光源処理
- 16ビット浮動小数点フィルタで輝きと露出効果を高速化
- 長いシェーダと流れ制御で仮想ディスプレイスメント・マップ、やわらかい影、虹光沢、フォグなどを実装

GPGPU?

- General Purpose computation on GPUs
  - GPUでグラフィクス以外の処理を
- GPUは非常に柔軟で高性能な演算装置
  - プログラム性
  - 精度
  - パフォーマンス
- 応用例 (GPGPU.orgより)
  - 高等な描画: 広域照明、画像ベースのモデリング
  - Computational Geometry
  - Monte Carlo法
- NVIDIA SDKにも多くのGPGPU例
  - http://developer.nvidia.com/object/sdk_home.html
- GPU Gems 2
  - http://developer.nvidia.com/GPUGems2/

例: 流体シミュレーション

- **Navier-Stokes**流体シミュレーションを **GPU**で
- 内部の障害物
  - 分岐無しで
  - **Zcull**での最適化
- 最新の**GPU**では高速
  - **GeForce 6800 Ultra**を用い **256x256**で約**8ms**
- **NVIDIA SDK 8.5**に収録

...d Dynamics Simulation on the ...PU", Mark Harris. In *GPU Gems*.

シェーダを使う

『エフェクト』

- **Direct3D FX and CgFX**
  - ***ID3DXEffect*または*CGeffect***
- ピクセルと頂点シェーダのラッパ
- 設定可能
  - 対象シェーダのバージョン
  - 一般的な変数
- ライブラリやシェーダの関数を参照可能
- 複数パスを定義

セマンティクス

- どんな変数名も定義可能
  - セマンティクスで定数が設定される

```
float4x4 wvp      : WorldViewProjection;
```

- **D3D SAS**が標準化
  - 多くのアプリケーションでのサポート
    - **FX Composer**
    - **3D Studio Max**

- **OpenGL**のセマンティクスは**CgFX**の**1.4**で標準化

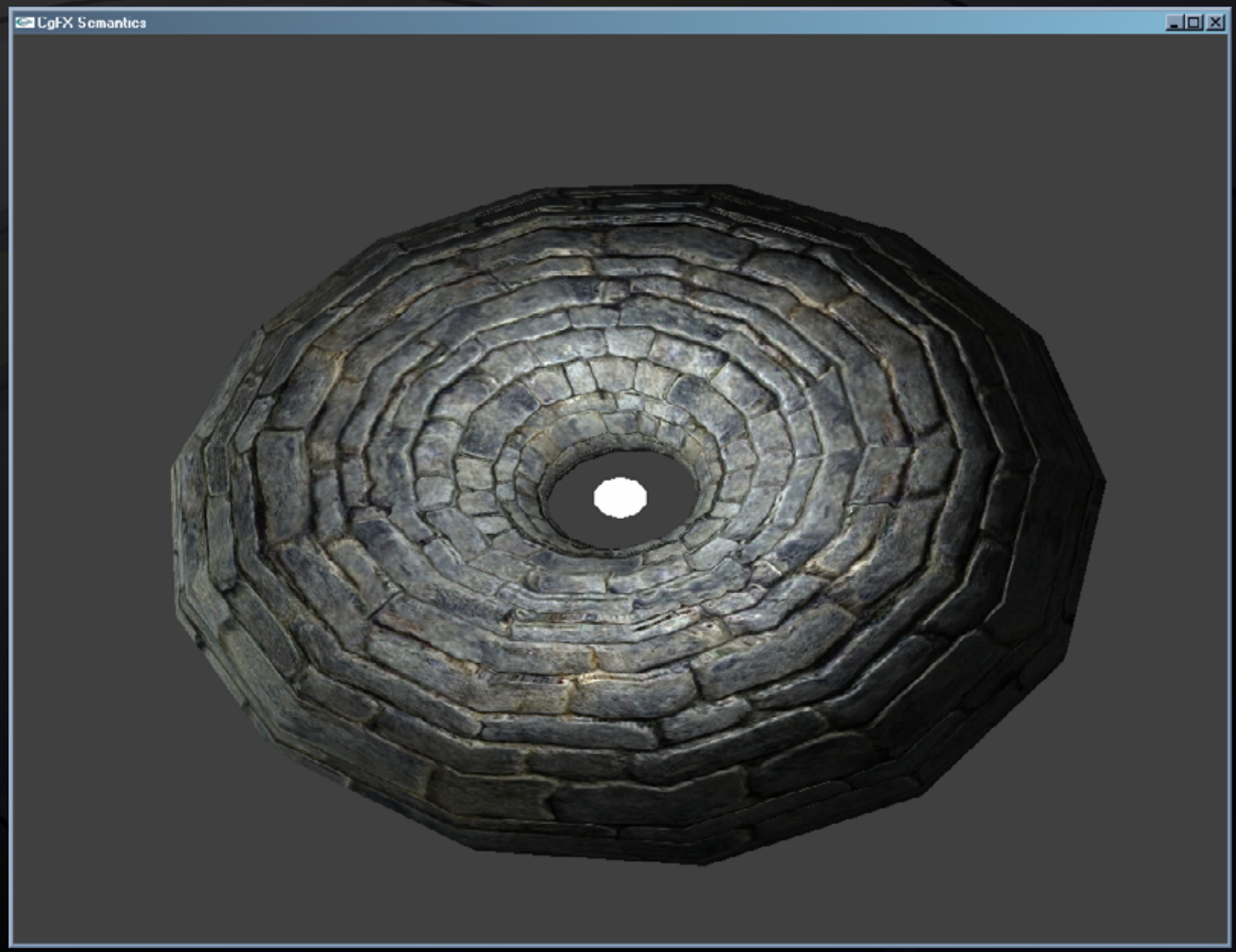
アノテーション

- **HLSL**や**CgFX**エフェクトの任意の要素に関連付けられたデータ

```
sampler2D anisoTextureSampler <
    string file = "Art/stone-color.png";
> = sampler_state {
        generateMipMap = true;
        minFilter = LinearMipMapLinear;
        magFilter = Linear;
        WrapS = Repeat;
        WrapT = Repeat;
        MaxAnisotropy = 8;
};
```

- オブジェクトごとのデータへの関連付けが可能
  - 例**:** シェーダのツールや**UI**コントロールで多く使用される

CgFXセマンティクスのデモ



セマンティクス例の重要点

- 頂点シェーダからタンジェント空間を補間
- 単一のフラグメント・シェーダで光源処理
- C++から大きさを変更できる不定長配列の光源構造体
- 光源インターフェイスを実装する数種類の光源タイプ
  - 点光源
  - スポット・ライト
  - その他...
- 定数値により、オプションでバンプ・マップ

光源インターフェイス

- 光源を表現するために**CgFX**のインターフェイスを定義

```
interface Light
{
    LIGHTRES compute(const LIGHTINFOS infos, LIGHTRES prevres);
};
```

- **C++**側でいくつでも、どんな光源も設定可能
  - **CgFX**は自動的に再コンパイル
  - もしくは、すべて事前にコンパイルしておき、直接使用

単一の光源関数

- 単純な表面色マップを基礎色に
- 補完されたベクタを正規化
  - タンジェント空間基底ベクタ
- オプションで法線を法線マップ補正
- 各光源を処理し、ディヒューズとスペキュラ色を加算
- 色と光源値を構成し、最終結果を作る

光源シェーダ

```
float4 color = tex2D(AlbedoMapSampler,inUV);

// Up front normalize to correct interpolated error, skip tangent basis, as the bumped normal gets
normalized
float3 toView = normalize(inToView);
float3 lightDir = normalize(inLightDir.xyz);

// get our bumped normal, or jsut use the original based on the configuration (compiled out)
float3 normal = bumpNormal(inNormal,inTangent,inBiNormal,NormalMapSampler,inUV.xy);;

LIGHTINFOS lightInfo;
lightInfo.Pw = inWorldPos; lightInfo.Vn = toView;
lightInfo.Nb = normal; lightInfo.SpecExpon = sExp;

// Go through our lights
LIGHTRES blinnValues;
for(int i=0;i<aLights.length;i++)
{
        blinnValues = aLights[i].compute(lightInfo,blinnValues);
}

return float4(blinnCombine(color.xyz,ambient.xyz,blinnValues),1);
```

C++側

```
CGparameter lArray;
lArray = cgGetNamedEffectParameter(effect,"aLights");
assert(lArray);
cgSetArraySize(lArray,numLights);
for(int i=0;i<numLights;i++)
{
 CGparameter p = cgGetArrayParameter(lArray,i);
 if(p)
 {
        cgDisconnectParameter(p);
        cgConnectParameter(lights[i].handle,p);
 }
}
```

C++側

- 変数のハンドルを用い、エフェクトから光源位置を設定
- プログラムによって、光源数と光源情報を動的に設定
- 法線マップの使用を決定
  - 使用しない場合、シェーダは高速化
- すべての動的な設定はユニフォームのパラメタか広域定数で表される

シェーダ代替処理

- 代替処理はアプリケーションで速度と品質を調整する方法
- 移植を非常に簡単に!
- シーン内の最大光源数は非常に良い代替処理パラメタ
- 複雑な画面全体の効果を抜き差しできる
- 通常の影とやわらかい影
- シェーダ・モデルの差異
  - 同じ効果でも、**SM3.0**では分岐により高速に

CPU側.. 代替処理を扱う

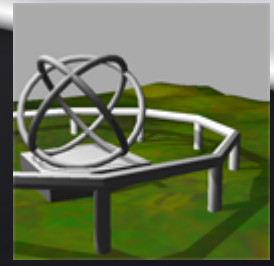
- 多くの効果に対して複数のハードウェアを扱わなければならない
  - 効果を有効、無効にする
- ゲーム販売開始間際に、見栄えの良い効果を追加するのに便利
- カードに対して正しい方法を選択
  - カードが**SM2.0**をサポートするからといって、**SM2.0**の機能全てを使えばよいというわけではない
    - 低位の**SM2**カードでは**SM1**を使用したほうが良いかも
  - **QA**テストで例外を定義
- **NVIDIA**は**CSR**テスト・ラボで協力**!**

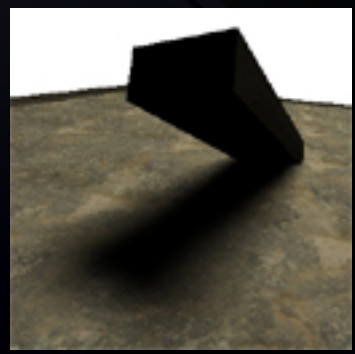
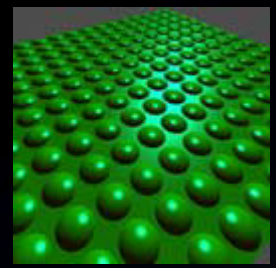
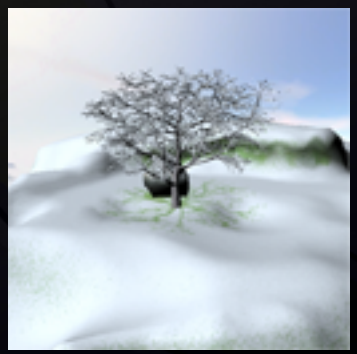
代替処理について最後に

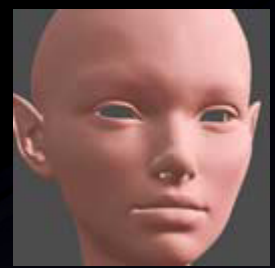
- 初期段階で実装すると非常に良い
  - 新しい代替処理を簡単に追加可能
    - カードを別のグループに変更
  - 各シェーダのレベルで、満足できるフレーム・レートを維持し、最高の画質を
- **Device Caps**を使う!
  - ドライバやハードウェアのバグ、もしくはユーザの設定によりのみ上書き
- いくつかの一般的なパスを作成
  - もっとも大きなレベルではシェーダ・モデルで
  - 高位と互換性パスだけでも
  - いくつかの微調整パラメタ
    - 光源数、サンプル数など

シェーダ・ライブラリ

- 各シェーダを独立して作るより
- コードの再利用!!
- 共通の補間値を確立
  - 頂点からフラグメント / ピクセル・プログラムへ
  - 例: 少なくともCOLOR0やTEXCOORD0は使わない
- 便利な関数のライブラリを作る
  - 部分を分離
  - コンパイル時間がかかるのみ (前処理できる!)



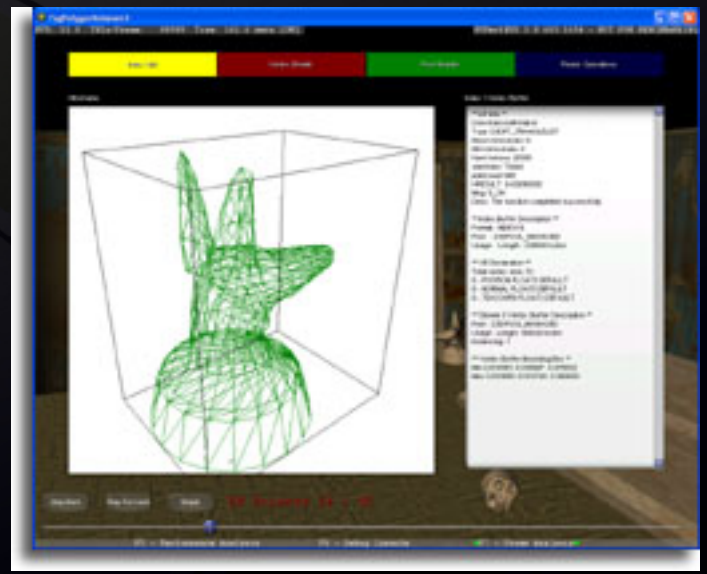



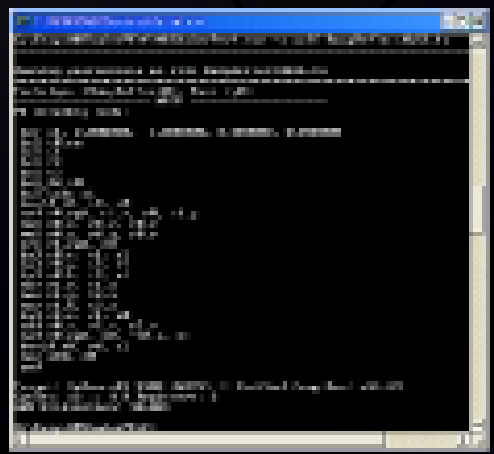

拡張性を考えて記述

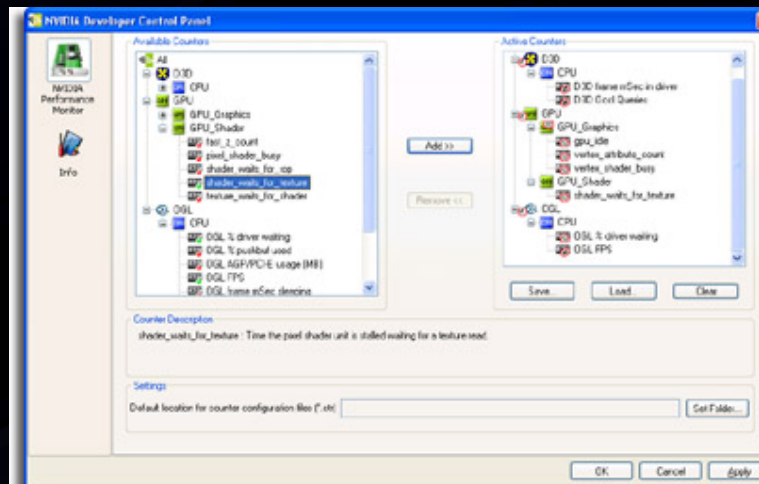
- ハックはプロトタイプのみ
- 通常のコードと同じ
- スタイルの基準を確立
- 完全なプリプロセッサのサポート
  - **#ifdef #define**など
- テクニックの名前付け基準
- アセンブリを使わない**!**

パフォーマンス

- CPU制限でなければ、おそらくフラグメント / ピクセル制限
- NVIDIAのGPU Programming Guide
- NVIDIAはいくつかの便利なパフォーマンス解析ツールを提供
  - NVShaderPerf – シェーダのスケジューリングを確認
  - NVPerfHUD – 実行時にボトルネックを確認
  - NVPerfKit – アプリケーションからカウンタにアクセス





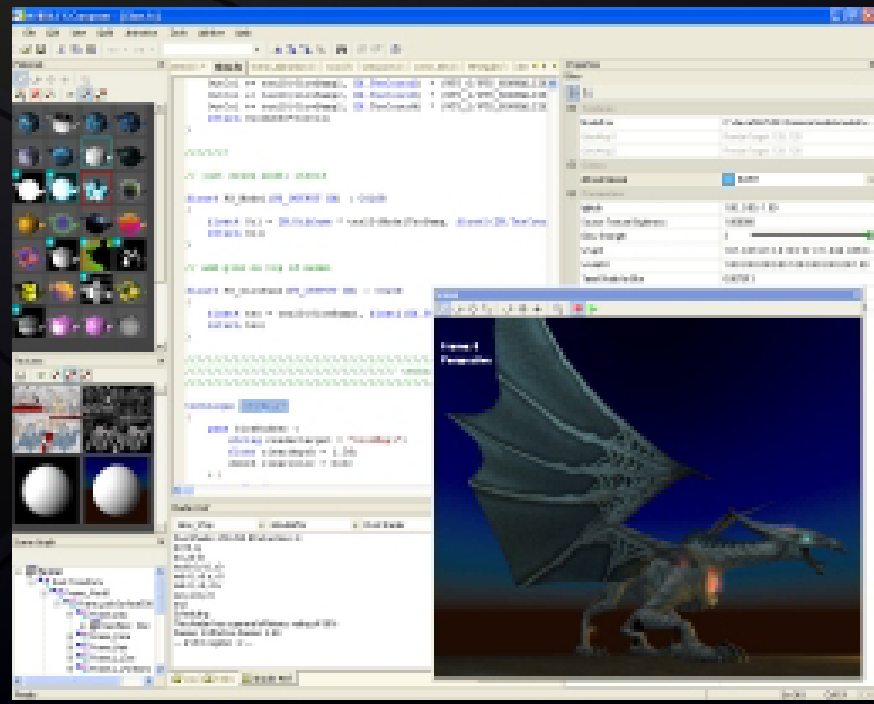


シェーダ作成パイプライン

- ゴール
  - 簡単なシェーダの作成と統合
- 要求
  - デザイナが調節可能なパラメタ?
  - 技術的なデザイナがシェーダを作成?
    - DCCツールから出力?

FX Composer

- FX Composer 1.8
  - D3Dの.fxとSAS
  - シェーダのプロトタイプ
  - シェーダ・ライブラリを開発
  - 代替処理の作成
  - カスタマイズしたパラメタ
- FX Composer 2.0
  - CgFXのサポート!
- ツールのセッションを参照!
- http://developer.nvidia.com



Collada

- XML交換様式
- シェーダとデータを読み出し、関連付ける標準的な方法
- モデルのデータも処理可能
  - 頂点など
- Sonyによってこのツールとの互換性が保たれる
- FX Composer 2.0によるサポート

DCCツール

- Discreetの3ds Max 5
  - HLSL FXとSAS
- Alias|WavefrontのMaya 4.5
  - CgFx
- XSIのSoftImage
  - HLSL FXとCgFX
- FXファイルを読み、カスタマイズ可能なパラメタをDCCアプリケーションから操作
- これらの操作された値をシェーダの標準に
- シェーダとカスタマイズされたパラメタを関連付けたモデルを出力し、ゲーム中での外観をデザイナが制御できる

まとめ

- 高級シェーディング言語を使用
- FXファイルを使用
  - CgFXまたはD3D FX
- セマンティクスを使用
  - 効果を追加するのに非常に簡単な方法
- デザインのパイプラインを事前に定義
  - デザイナがどのようにシェーダを操作し、閲覧するかを理解
- シェーダをC++コードのように扱う
  - 良いデザインで、ゲームをすばらしく見せるための時間を大きく短縮できる!

質問

- http://developer.nvidia.com
- http://developer.nvidia.com/CgTutorial
- **Email:** bdudash@nvidia.com