DirectX10のエフェクトとパフォーマンス

ブライアン・デュダーシュ（Bryan Dudash）

CEDEC 2006

本日のセッション

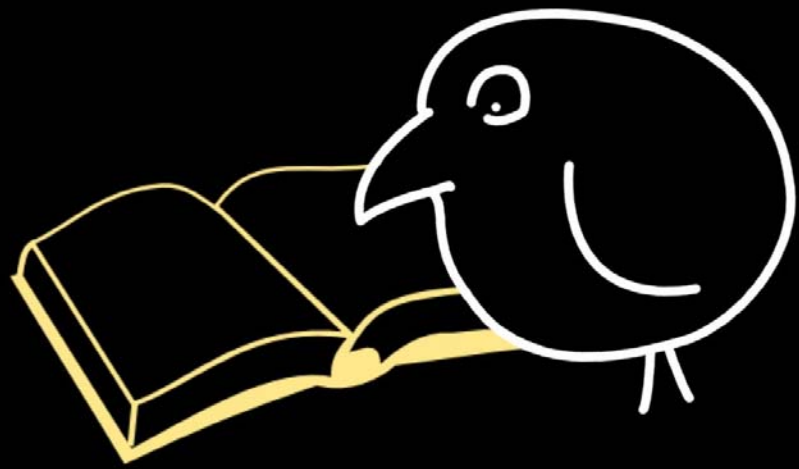
今の時間	***DX10のエフェクトとパフォーマンスならびに使用法***
	*ブライアン・デゥダーシュ（**Bryan Dudash**）NVIDIA*
16:50 – 17:00	**休憩**
17:00 – 18:30	**NVIDIA　GPUでの物理演算**
	*サイモン・グリーン（**Simon Green**）NVIDIA*

モチベーション

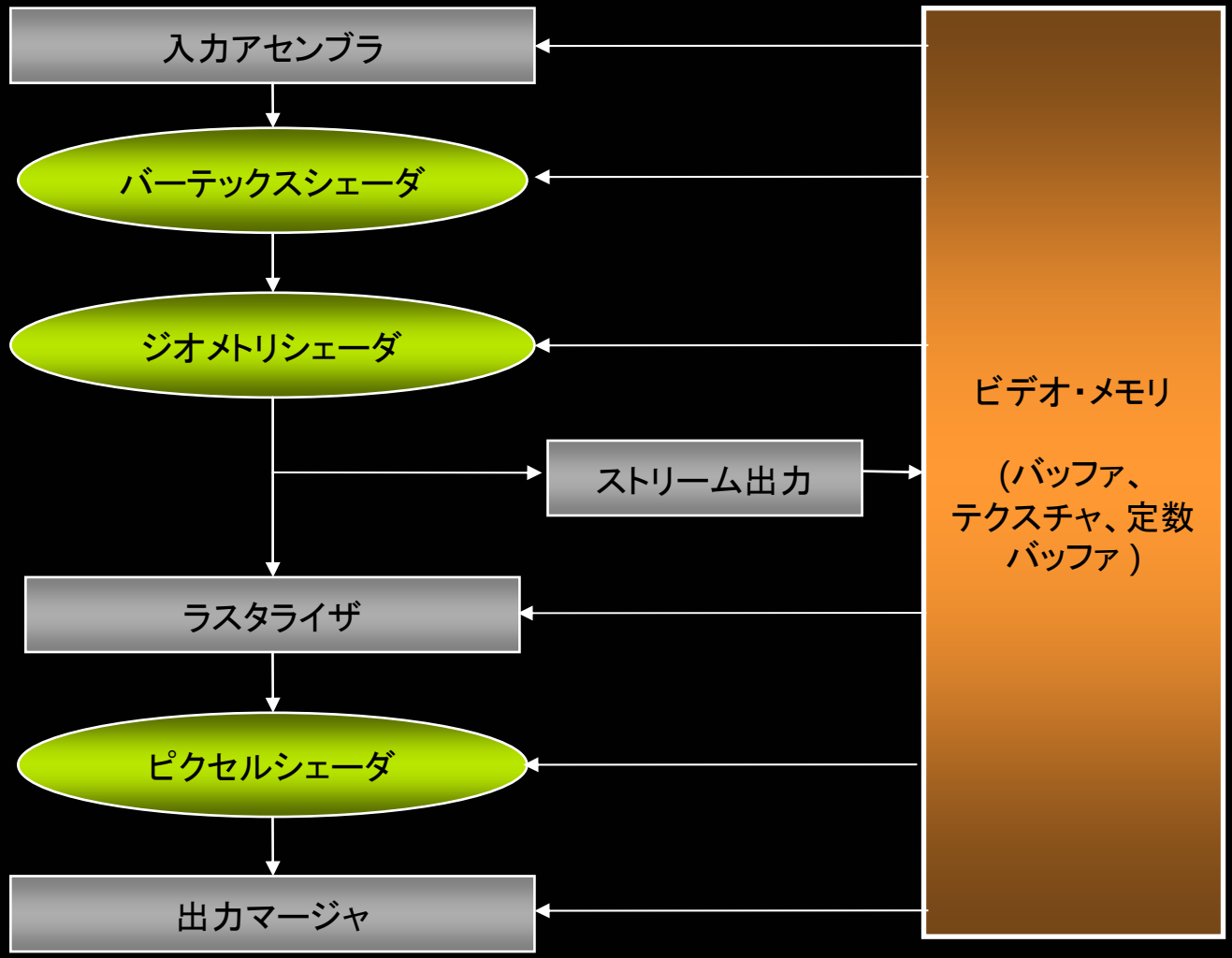
- Direct3D 10 はマイクロソフトの次期グラフィックスAPI
  - 次世代GPUの機能セットを推進
- 多数の新機能
  - 新しいプログラム可能性、汎用性
- 新しいドライバ・モデル
  - パフォーマンスを改善
- APIをクリーンアップ
  - 状態処理を改善。事実上capsビットなし!

議題

- DX10 のパイプラインと機能の概要
- エフェクトのケース・スタディ
  - 毛皮向けフィン – GPU シルエット検出
  - 布– 少ないパスでのGPUシミュレーション
  - メタボール – GPU等値面抽出
- 結論



Direct3D 10 パイプライン



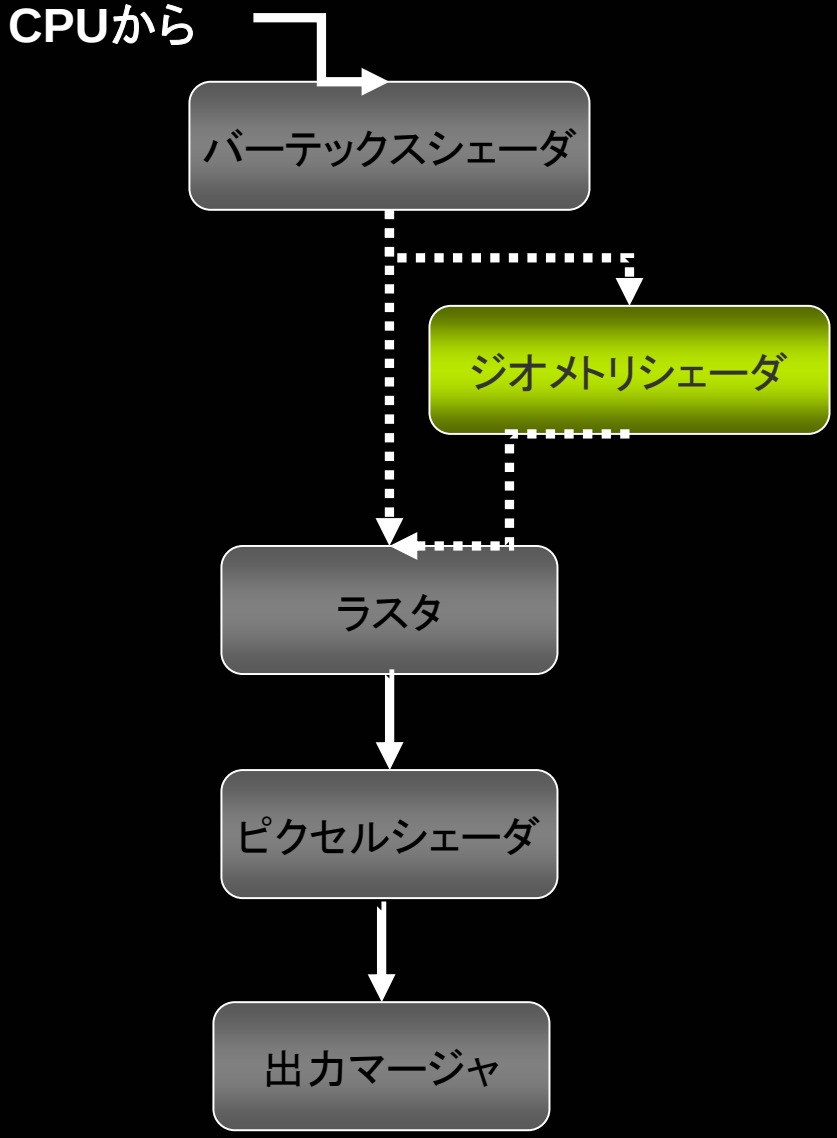
Direct3D 10 の新機能

- 共通シェーダ命令セット
  - シェーダによる整数演算
  - すべてをインデックス化可能
- ジオメトリシェーダ
- ストリーム出力
- レンダターゲットアレイ
- テクスチャアレイ
- MRT × 8
- 入力アセンブラ生成ID; InstanceID、VertexID、PrimitiveID
- Alpha to Coverage

これからの内容...

- ジオメトリシェーダ (GS) の概要
- GSの使用例 : フィン
  - GSの使用法を示すHLSLコード
- ストリーム出力 (So)の概要
- Soの使用例 :布
  - SOの使用例を示すHLSLコード
- メタボール :マーチングキューブ

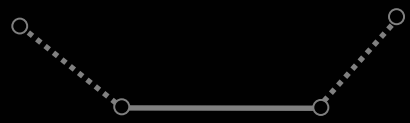
ジオメトリシェーダ



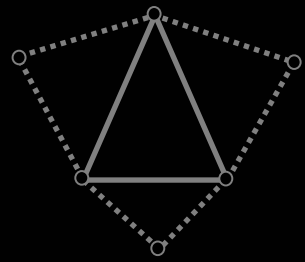
入力

点
線
三角形

線の結合



三角形の結合



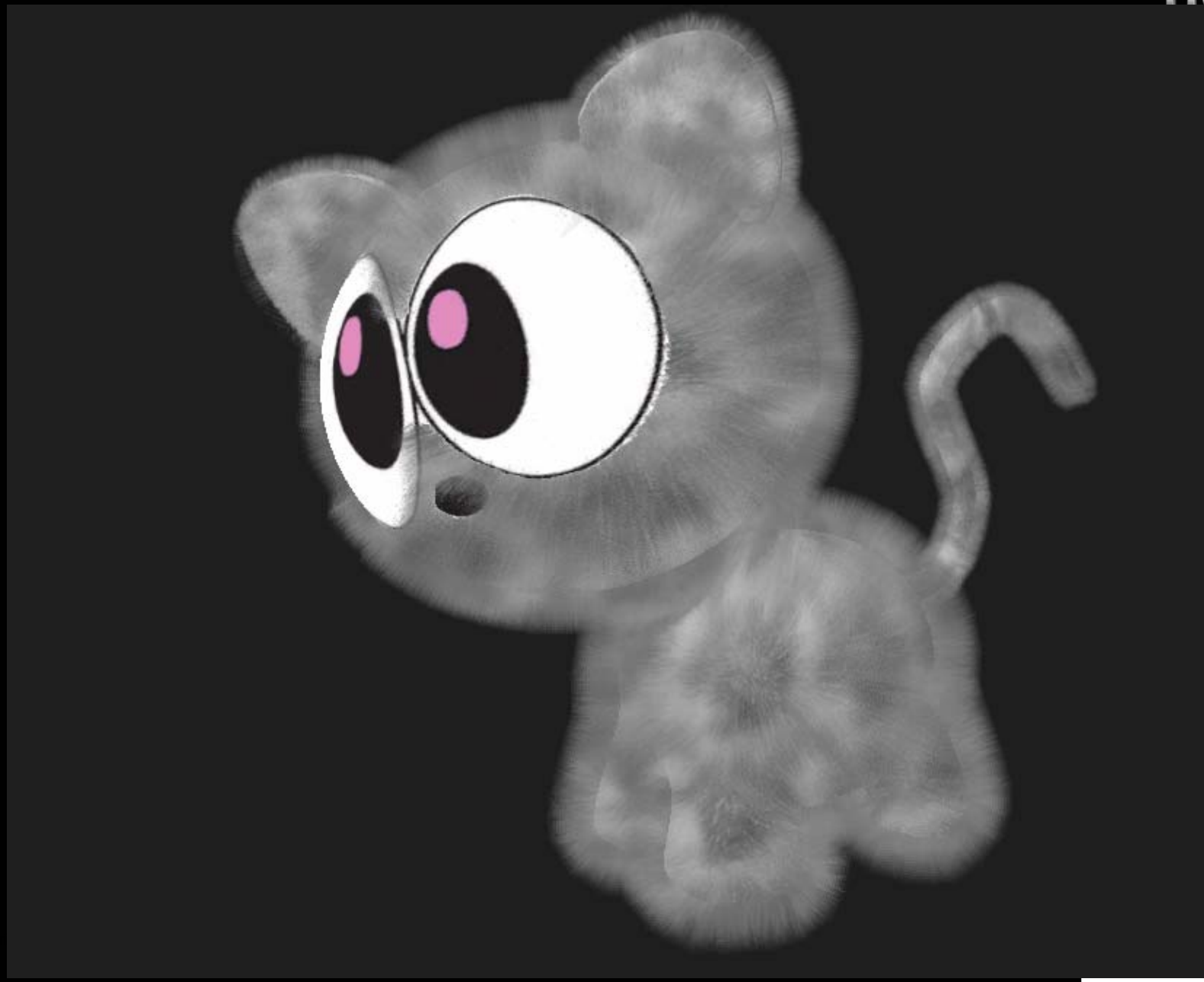
出力

点リスト
線ストリップ
三角ストリップ

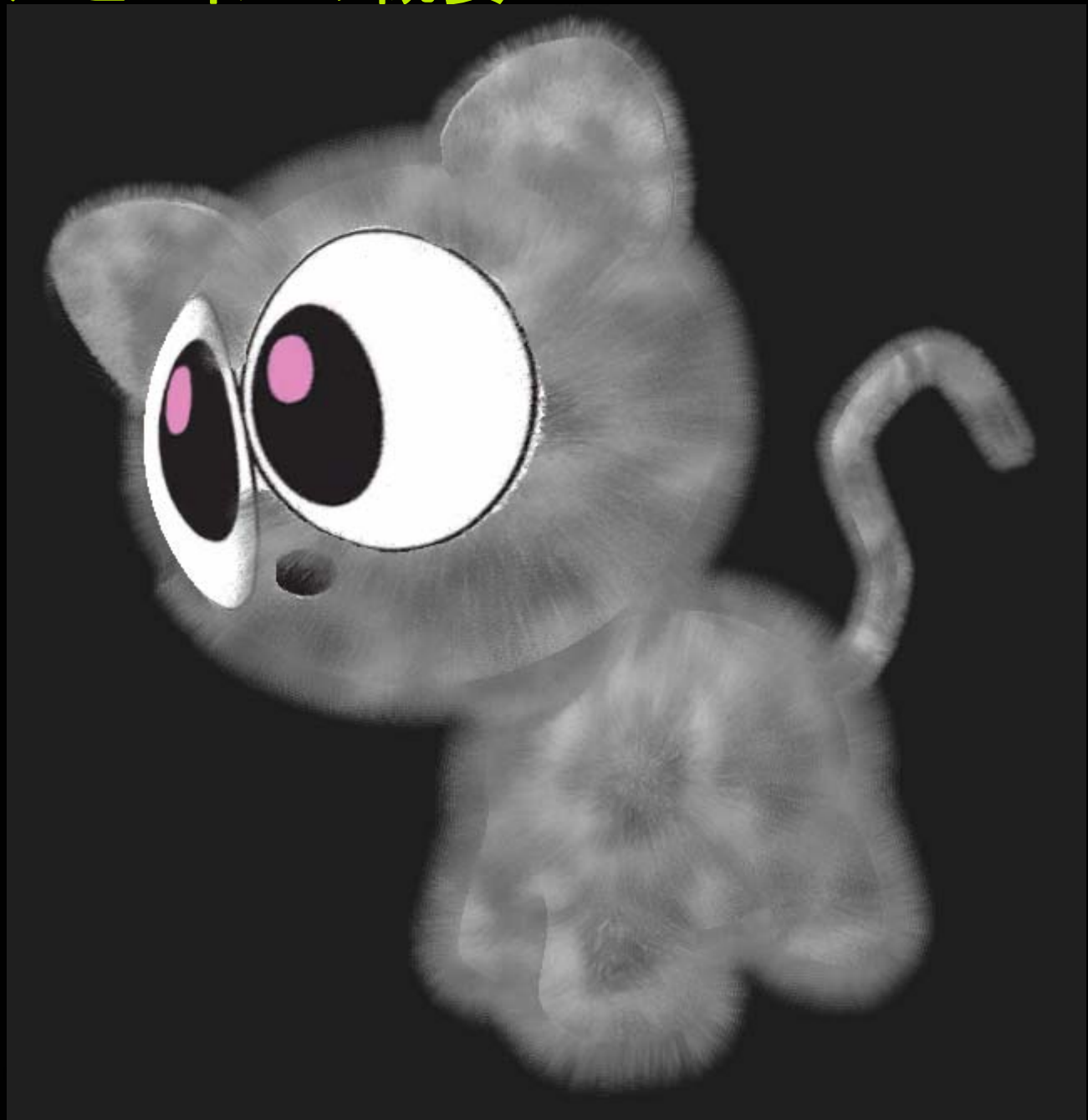
GSの注意メモ

- ストリーム出力 が有効な場合
  - GS は*リストを出力*
- ストリーム出力が無効な場合
  - GS は*ストリップのリストを出力*

毛皮ー GPUでフィンを描画

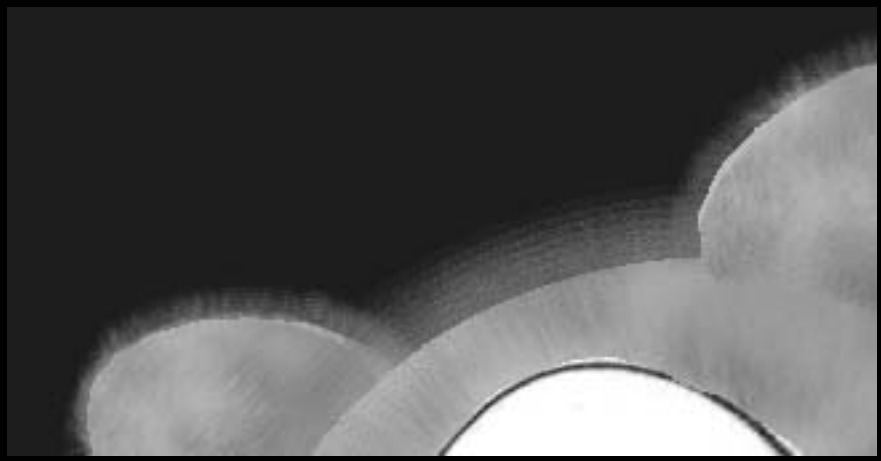


シェルとフィンの概要



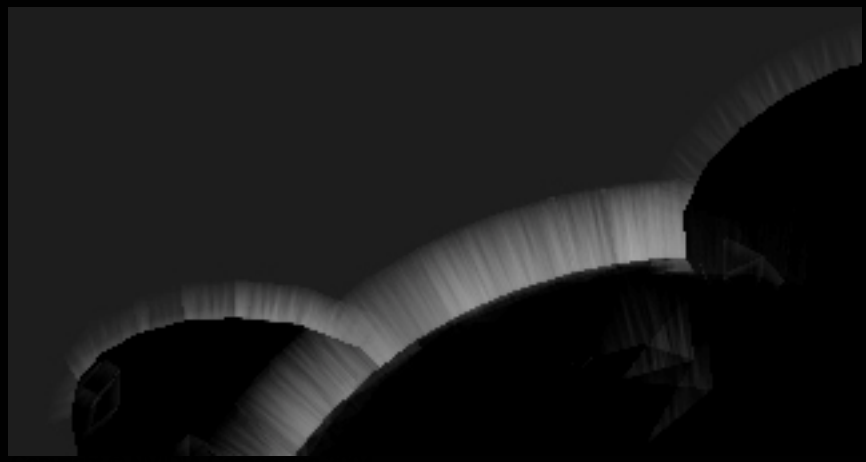
NVIDIA.

シェル: シルエットの問題点



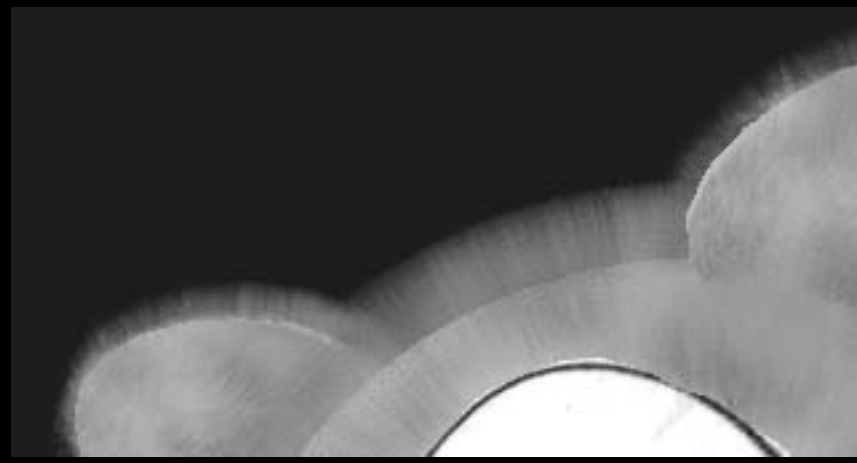
8 シェル

+



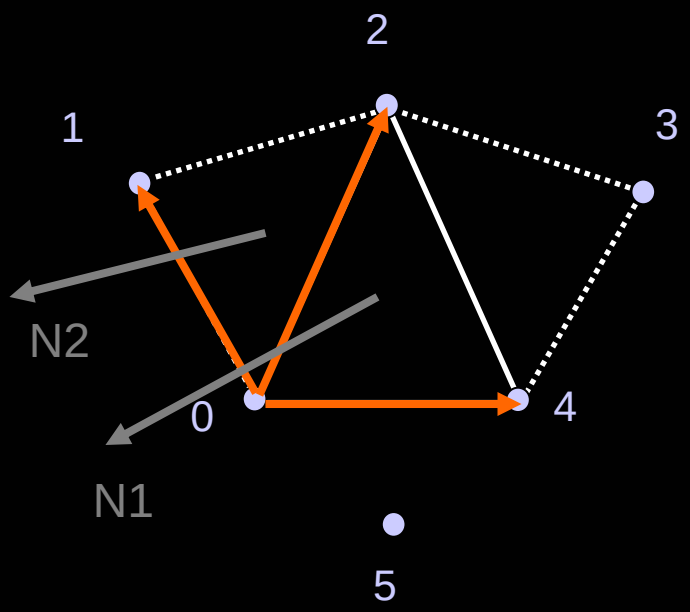
フィン

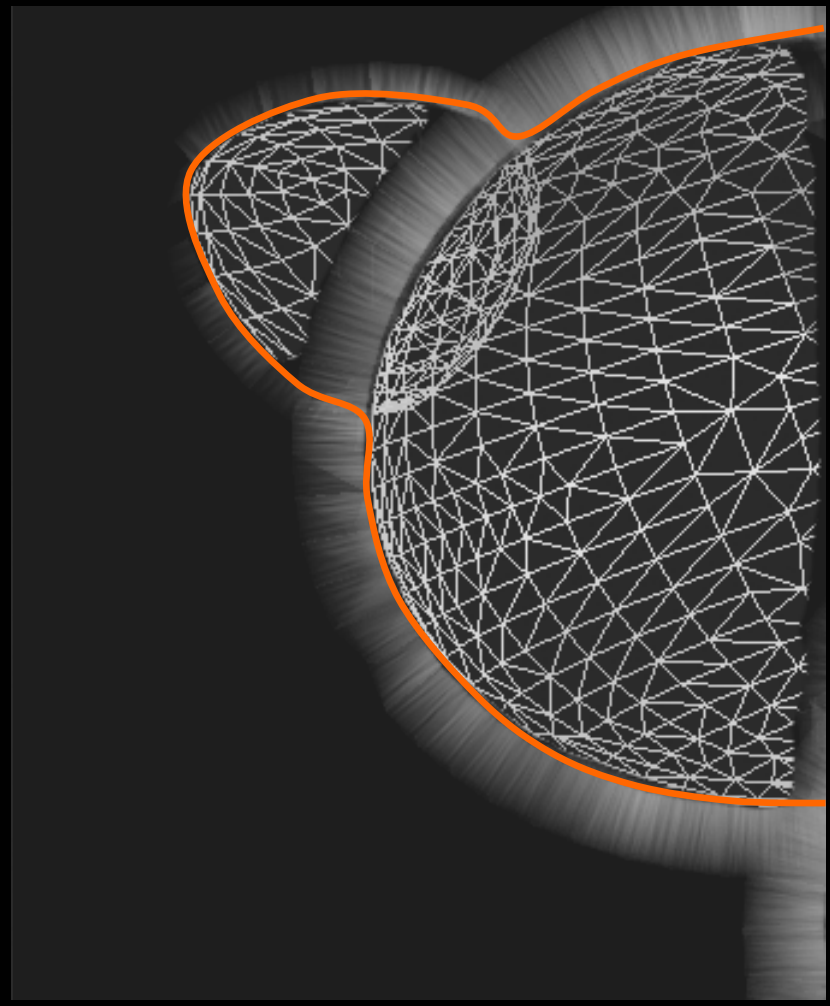
→



フィン

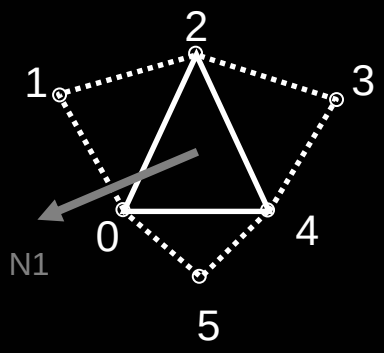
GSでのシルエット検出





If(dot(eyeVec,N1) * dot(eyeVec,N2) < 0)

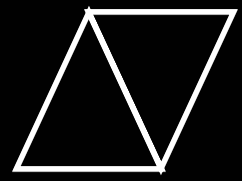
入力



triangleadj

ジオメトリシェーダ

出力

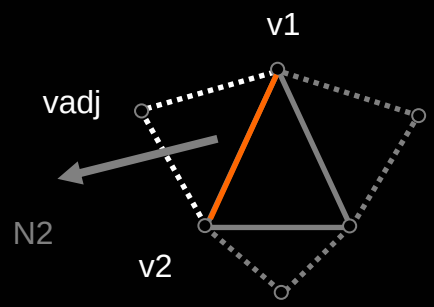


TriangleStream

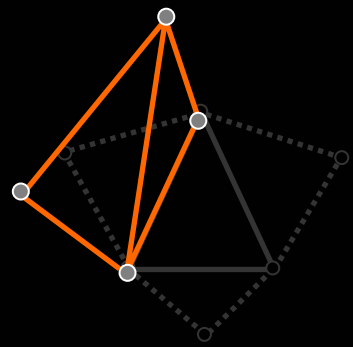
```
//GS shader for the fins
[maxvertexcount(12)]
void GS( triangleadj VS_OUTPUT input[6],
inout TriangleStream<GS_OUTPUT_FINS>TriStream)
{
    //compute the triangle's normal
    float3 N1 = normalize(cross( input[0].Position - input[2].Position ,
     input[4].Position -input[2].Position ));
    float3 eyeVec = normalize( Eye – input[0].Position);

    //if the central triangle is front facing, check the other triangles
    if( dot(N1, eyeVec)  > 0.0f )
    {
        makeFin(input[2],input[0],input[1], TriStream);
        makeFin(input[4],input[2],input[3], TriStream);
        makeFin(input[0],input[4],input[5], TriStream);
    }
}
```

シルエットのエクストルージョン

入力



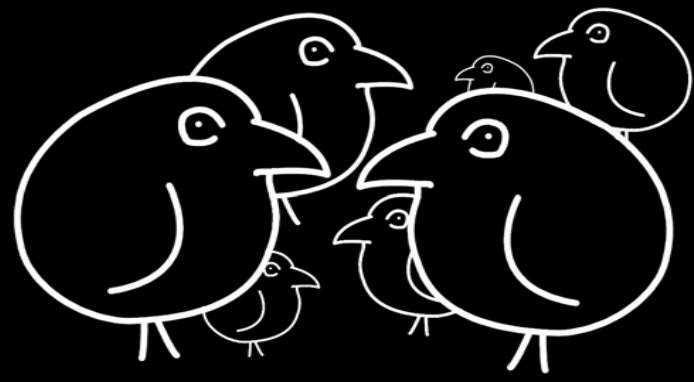
出力



```
void makeFin( VS_OUTPUT v1, VS_OUTPUT v2, VS_OUTPUT vAdj inout TriangleStream <GS_OUTPUT_FINS>
TriStream
{  float3 N2 = normalize(cross( v1.Position - v2.Position, vAdj.Position - v2.Position ));
   float3 eyeVec = normalize( Eye - v1.Position );

   if( dot(eyeVec,N2) < 0 )
   {   //this is a silhouette edge, therefore extrude it into 2 triangles
       GS_OUTPUT_FINS Out;

       for(int v=0; v<2; v++)
       {
     Out.Position = mul(v2.Position + v*float4(v2.Normal,0)*length, WorldViewProj );
     Out.Normal = mul( v2.Normal, World );
     Out.TextureMesh = v2.Texture;
     Out.TextureFin = float2(1,1-v);
     Out.Opacity = opacity;
     TriStream.Append(Out);
       }
   TriStream.RestartStrip();
   }
 }
```

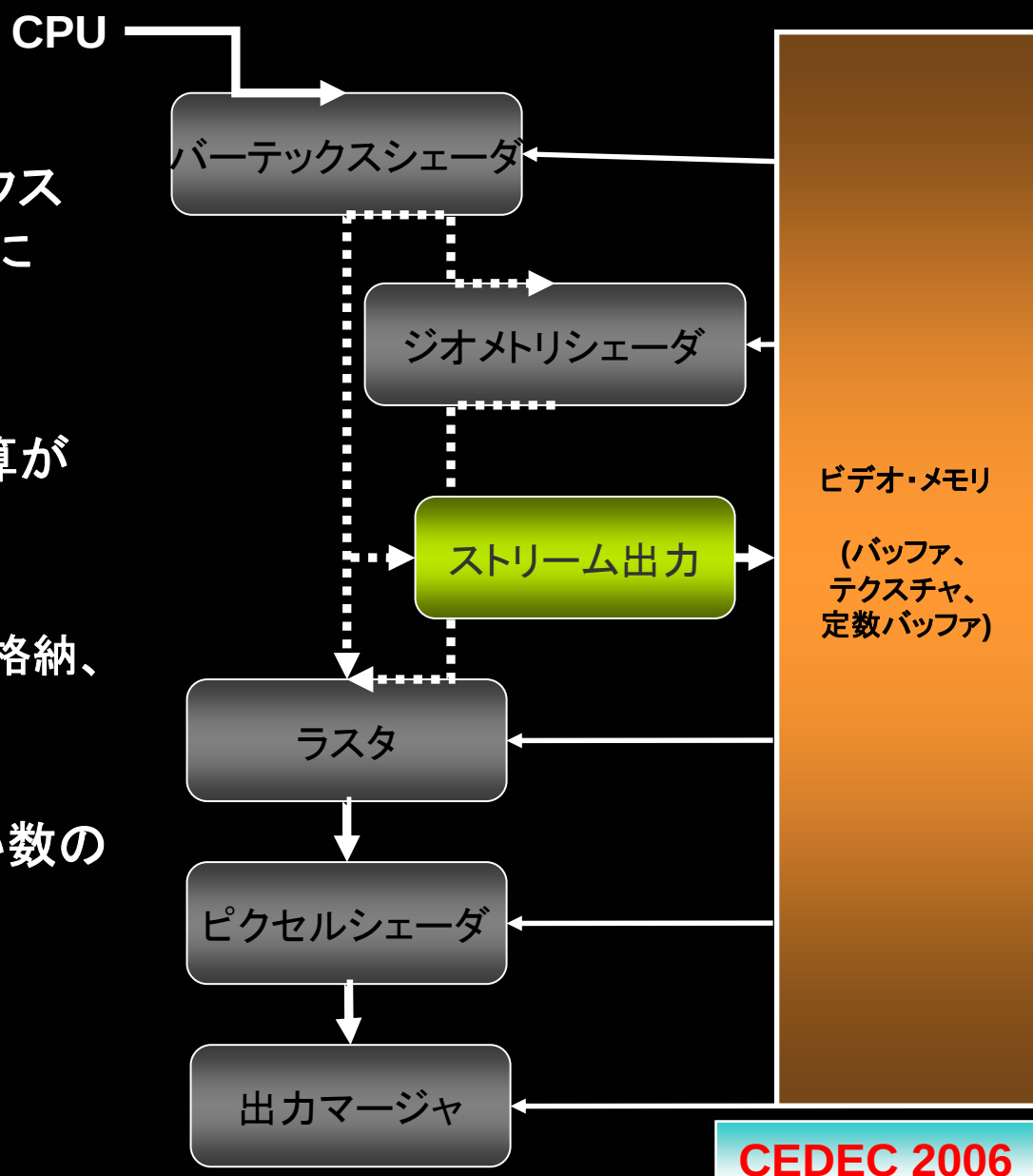
ジオメトリシェーダ・アプリケーションの例

- シルエット検出とエクストルージョン:
  - シャドウ・ボリューム生成
  - NPR
- キューブマップへのレンダ
  - シングル・パス
  - レンダターゲットアレイとの関連づけ
- GPGPU
  - シェーダからの出力数を変更可能

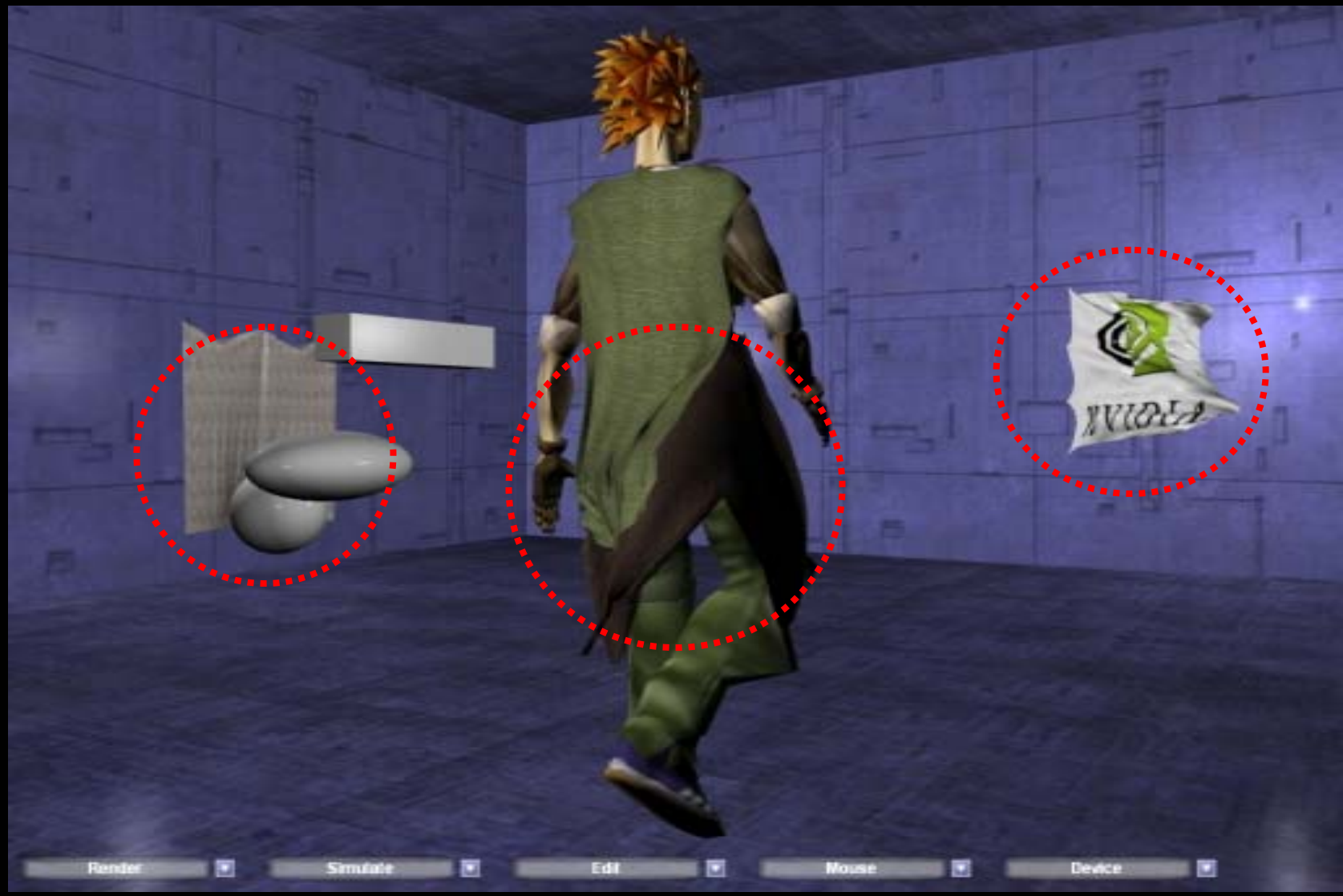


ストリーム出力

- ジオメトリシェーダ やバーテックスシェーダからの出力をバッファに格納可能
- 図形その他のマルチ・パス演算が可能。例:
  - 再帰的細分割
  - スキニング結果をバッファに格納、複数のライト処理で再使用
- DrawAuto() 機能により正しい数のプリミティブを自動描画
  - CPUの割り込み処理不要



GPUでの布処理

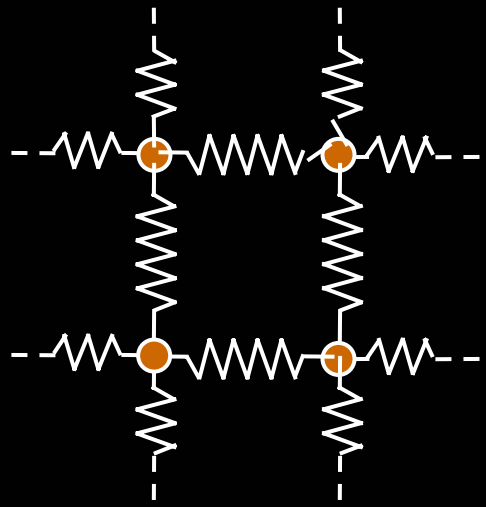


パーティクルの集合としての布

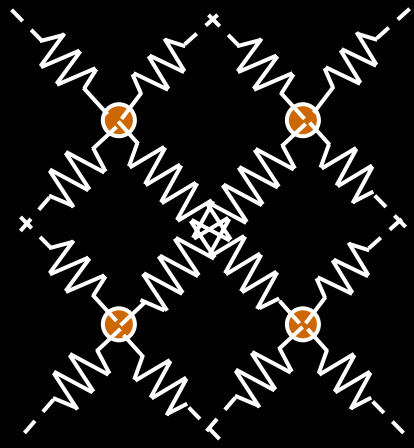
- パーティクルに影響するもの:
  - 力 (重力、風、引っ張り)
  - 各種 コンストレイン:
    - 全体的形状の維持 (スプリング)
    - 周囲との相互浸透の防止 (コリジョン)
- 各時間ステップで解決すべき方程式をコンストレインにより生成
- 陽的積分の使用: コンストレインはリラクセーションにより解決、つまり、一定回数にわたり反復実行する

スプリングのコンストレイン

- パーティクルをスプリングで結合:



構造用スプリング

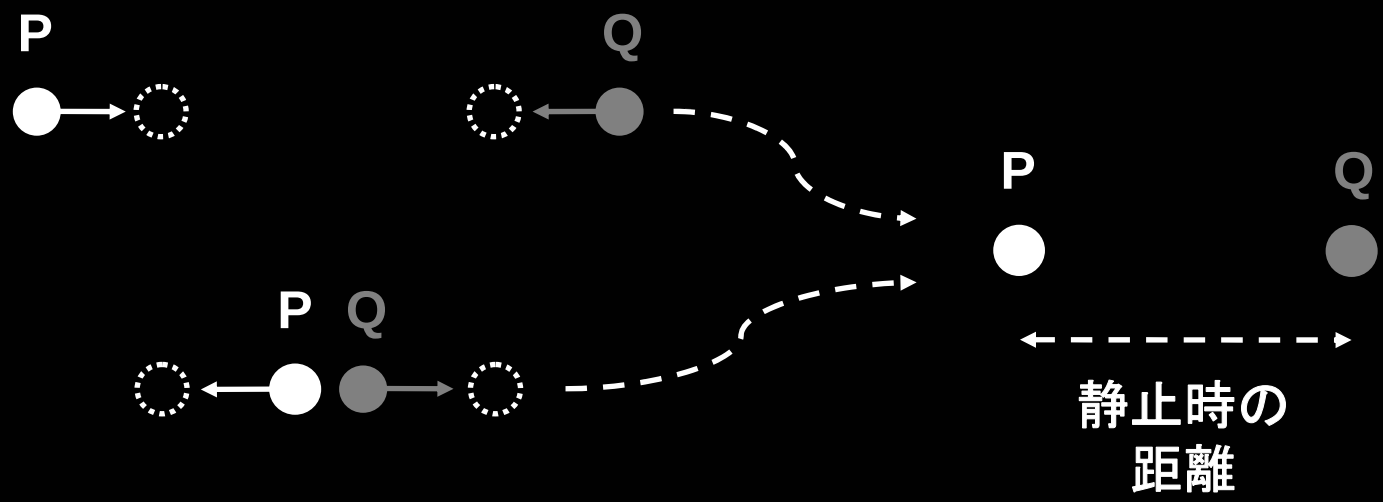


歪み補正プリング

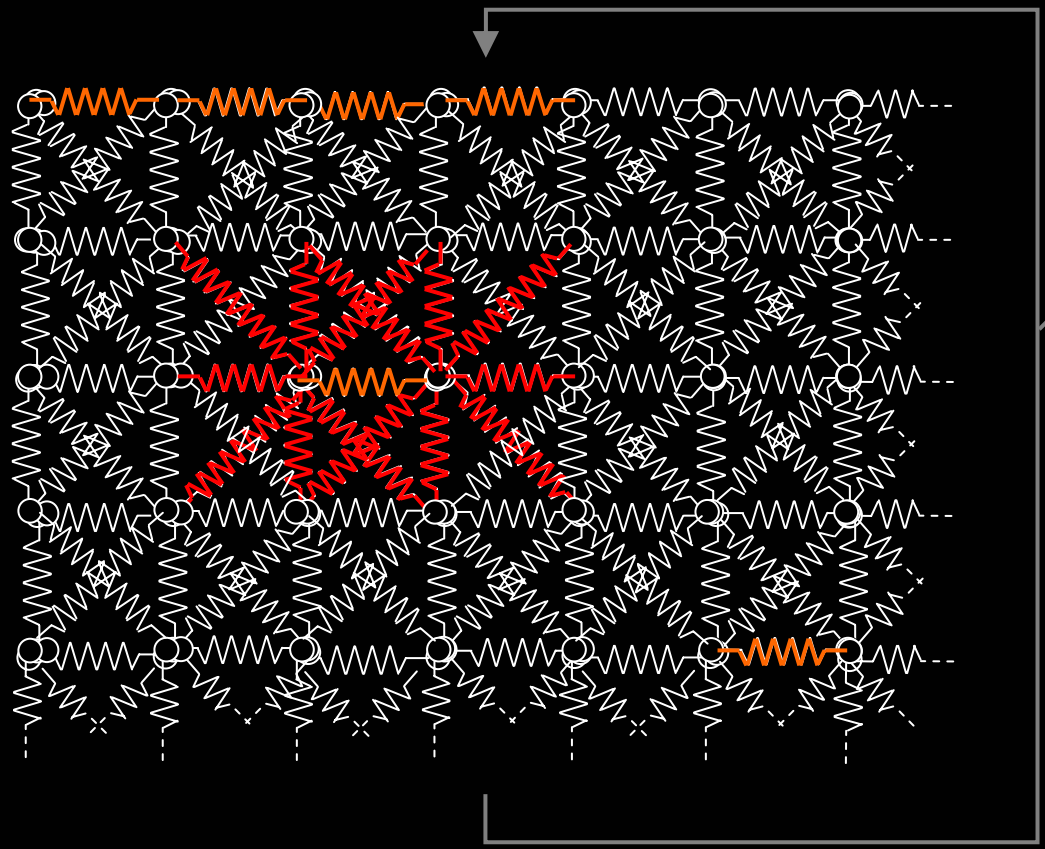
- スプリングを2つのパーティクルの間の距離コンストレインとしてシミュレート

距離コンストレイン

- 2つのパーティクル、PとQの間の距離コンストレイン DC(P, Q) は互いに接近させまたは遠ざけることにより設定される。

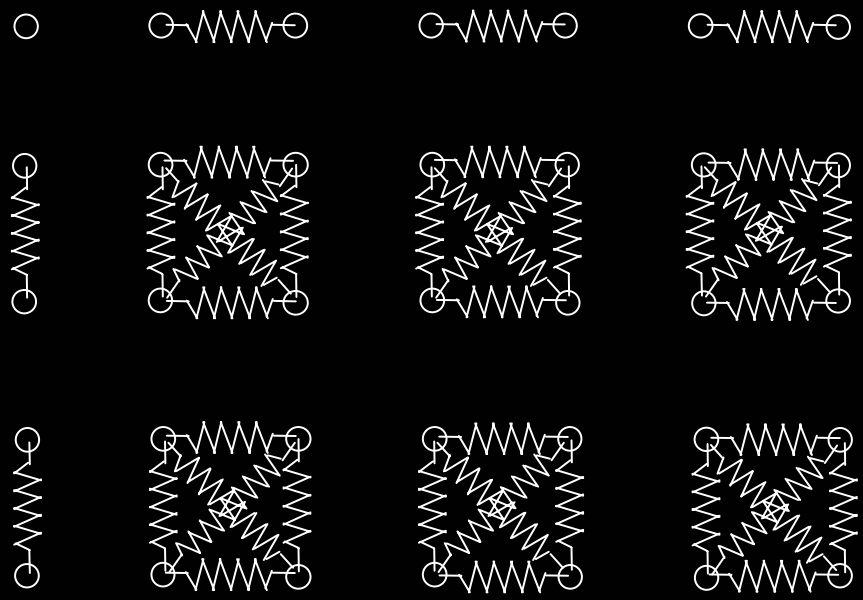


順次更新



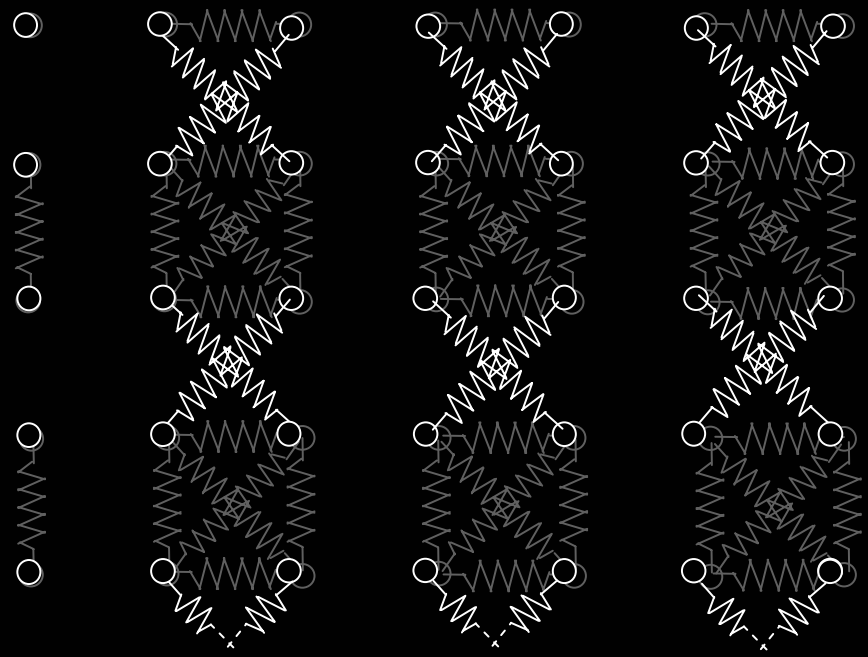
並列更新

バッチ 1



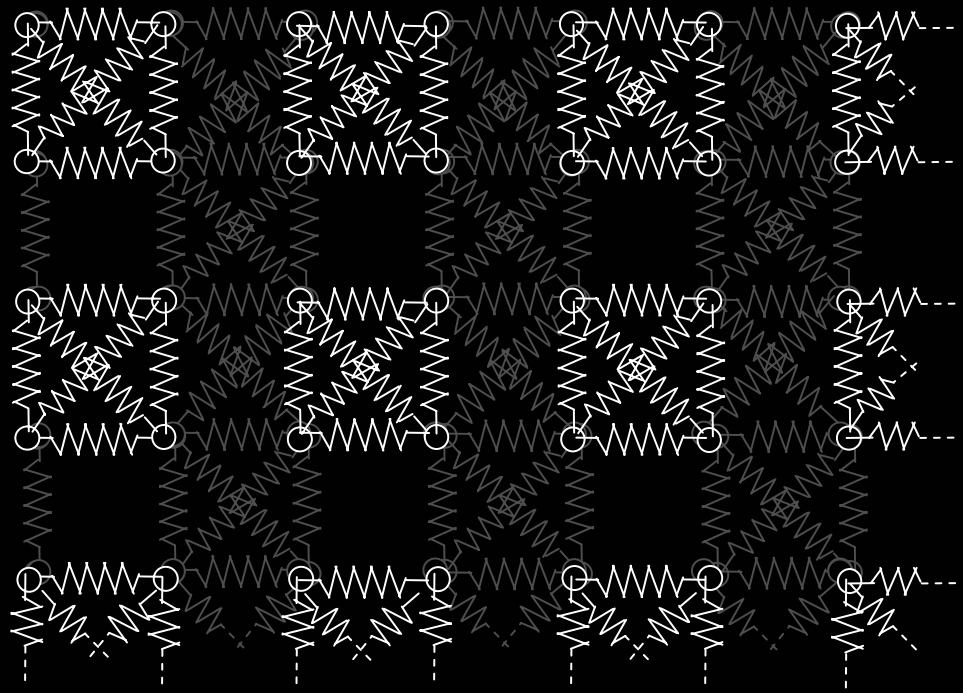
並列更新

バッチ 2



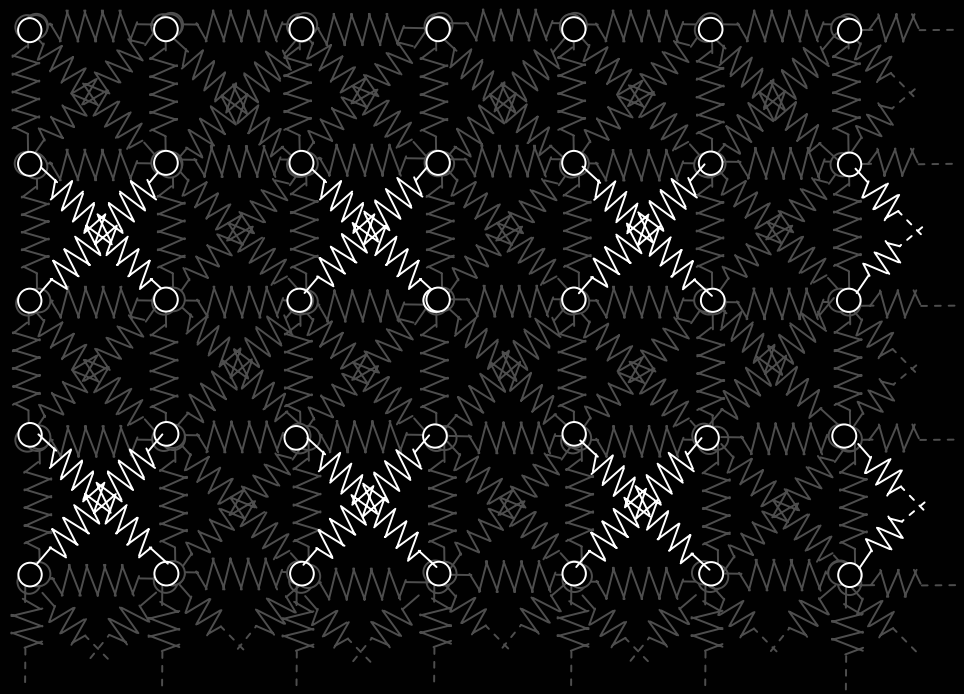
並列更新

バッチ 3



並列更新

バッチ 4



疑似コード: シミュレーション・ループ

- すべてのパーティクルに力を加える
- 必要な回数にわたり:
  - それぞれ別個のスプリングの各バッチにつき:
    - すべてのスプリングに距離コンストレインを適用する
  - コリジョン・コンストレインを適用
- メッシュをレンダリング

DirectX10 の実装

- パーティクルを頂点バッファに保存
  - DirectX9: パーティクルはテクスチャに保存される
- ジオメトリシェーダで計算
- ストリーム出力によるパス間の同期

疑似コード: 初期化

- 2つの頂点バッファを作成してパーティクルを保存:
  - 頂点の形式は現在位置、旧位置、法線、その他。
  - 1つの頂点バッファをバーテックスシェーダへの入力として使用
  - 1つの頂点バッファをジオメトリシェーダへの出力 (ストリーム出力) として使用
  - 各レンダリングの終了後、上記2つのバッファをスワップ
- 別個スプリングのバッチ数 (4) だけインデックス・バッファを作成
  - 各インデックス・バッファがジオメトリシェーダに4-Tupleのパーティクルを提供
- レンダリング用インデックス・バッファを作成

疑似コード: シミュレーション・ループ

- 力を加えるバーテックスシェーダを設定
- ポイント・リストとしてSOに描画
- 頂点バッファ同士をスワップ
- 必要な回数にわたり:
  - それぞれ別個の各スプリングのバッチにつき:
    - 距離コンストレイン を適応するジオメトリシェーダ を設定
    - 隣接データ付きインデックス設定三角形リストとしてSOに描画
    - 頂点バッファ同士をスワップ
  - コリジョン・コンストレインを適用するバーテックスシェーダを設定
  - SOをポイント・リストとして表示
  - 頂点バッファ同士をスワップ
- インデックス付き三角形リストとしてカラー・バッファにレンダリング

疑似コード: シミュレーション・ループ

- 力を加えるバーテックスシェーダを設定
- ポイント・リストとしてSOに描画
- 頂点バッファ同士をスワップ
- 必要な回数にわたり:
  - それぞれ別個のスプリングの各バッチにつき:
    - 距離コンストレインを設定するジオメトリシェーダを設定
    - 隣接データ付きインでックス設定三角形リストとしてSOに描画
    - 頂点バッファ同士をスワップ
  - コリジョン・コンストレインを適用するバーテックスシェーダを設定
  - SOをポイント・リストとして表示
  - 頂点バッファ同士をスワップ
- インデックス付き三角形リストとしてカラー・バッファにレンダリング

疑似コード: シミュレーション・ループ

- 力を加えるバーテックスシェーダを設定
- ポイント・リストとしてSOに描画
- 頂点バッファ同士をスワップ
- 必要な回数にわたり:
  - それぞれ別個のスプリングの各バッチにつき:
    - 距離コンストレインを設定するジオメトリシェーダを設定
    - 隣接データ付きインでックス設定三角形リストとしてSOに描画
    - 頂点バッファ同士をスワップ
  - コリジョン・コンストレインを適用するバーテックスシェーダを設定
  - SOをポイント・リストとして表示
  - 頂点バッファ同士をスワップ
- インデックス付き三角形リストとしてカラー・バッファにレンダリング

疑似コード: シミュレーション・ループ

- 力を加えるバーテックスシェーダを設定
- ポイント・リストとしてSOに描画
- 頂点バッファ同士をスワップ
- 必要な回数にわたり:
  - それぞれ別個のスプリングの各バッチにつき:
    - 距離コンストレインを設定するジオメトリシェーダを設定
    - 隣接データ付きインでックス設定三角形リストとしてSOに描画
    - 頂点バッファ同士をスワップ
  - コリジョン・コンストレインを適用するバーテックスシェーダを設定
  - SOをポイント・リストとして表示
  - 頂点バッファ同士をスワップ

- インデックス付き三角形リストとしてカラー・バッファにレンダリング

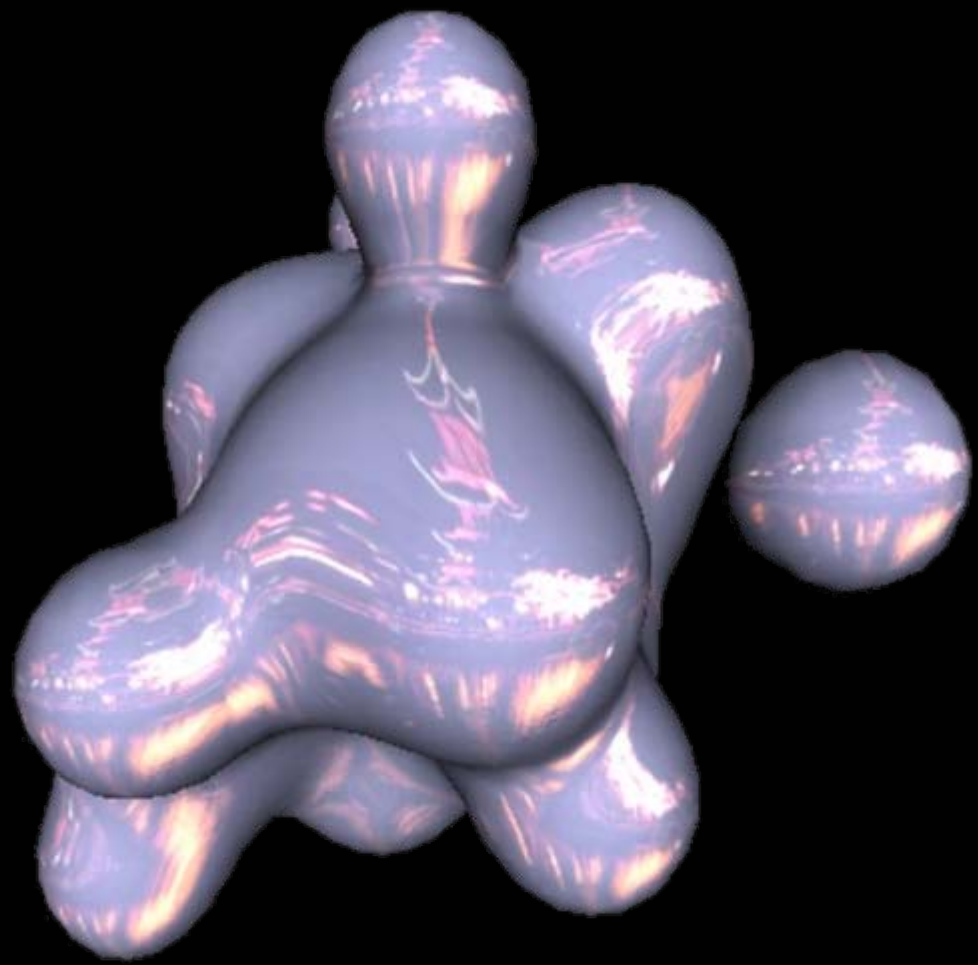
疑似コード: シミュレーション・ループ

- 力を加えるバーテックスシェーダを設定
- ポイント・リストとしてSOに描画
- 頂点バッファ同士をスワップ
- 必要な回数にわたり:
  - それぞれ別個のスプリングの各バッチにつき:
    - 距離コンストレインを設定するジオメトリシェーダを設定
    - 隣接データ付きインデックス設定三角形リストとしてSOに描画
    - 頂点バッファ同士をスワップ
  - コリジョン・コンストレインを適用する バーテックスシェーダを設定
  - SOをポイント・リストとして表示
  - 頂点バッファ同士をスワップ
- インデックス付き三角形リストとしてカラー・バッファにレンダリング

力を加える

```
 pass ApplyForces
    {
        SetVertexShader(CompileShader(vs_4_0, VS_ApplyForces()));
        SetGeometryShader(ConstructGSWithSO(CompileShader(vs_4_0, VS_ApplyForces()),
                                            State.x; Position.xyz '));
        SetPixelShader(0);
    }


void VS_ApplyFor inout Particle :le particle, OldParticle oldParticle)
{
    // Forces
    float3 force = 0;

    // Gravity
    force += float3(0, - GravityStrength, 0);

    // Damping
    float speedCoeff = 1 - TimeStep * 0.3;

    // Ignore position discontinuity (usually due to collision)
    float3 diffPosition = particle.Position - oldParticle.Position;
    if (dot(diffPosition, diffPosition) > 1 * TimeStep * TimeStep)
        speedCoeff = 0;

    // Integration step
    if (IsFree(particle))
        particle.Position += speedCoeff * diffPosition + force * TimeStep * TimeStep;
}
```

GPUでのメタボール



等値面とは?

- 関数を検討する $f(x, y, z)$
  - 3D空間でスカラー場を定義する
  - 手続型関数、または3Dシミュレーションに基づく
- *等値面* Sは下記の陰方程式を満たす点の集合

$$f(x, y, z) = const$$

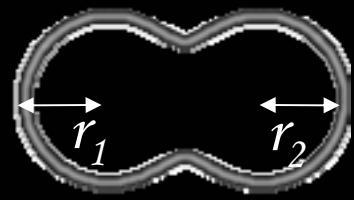
- 陰関数サーフェスとも呼ばれる

メタボール

- シンプルかつ興味深い事例
- ソフト/ぐにゃぐにゃの複数のボディが1つに融合
  - ゲームでの流体や爆発のモデル化に最適

- 下記の形式の陰方程式を使用

$$\sum_{i=1}^{N} \frac{r_i^2}{\left\| \mathbf{x} - \mathbf{p}_i \right\|^2} = 1$$



- グラジエントは下記の式で直接計算

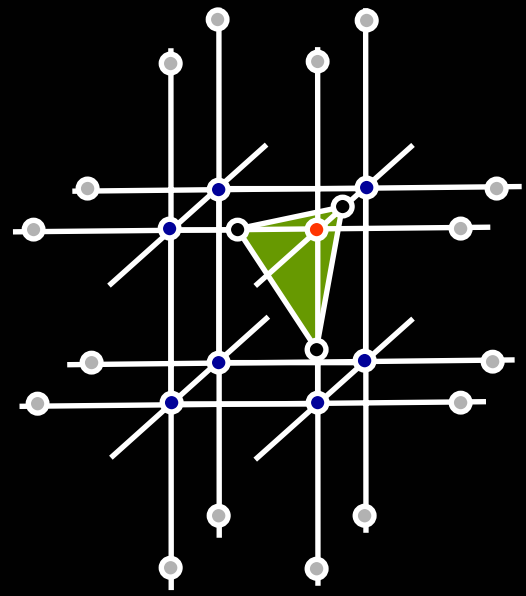
$$\mathbf{grad}(f) = -\sum_{i=1}^{N} \frac{2 \cdot r_i^2}{\left\| \mathbf{x} - \mathbf{p}_i \right\|^4} \cdot (\mathbf{x} - \mathbf{p}_i)$$

マーチングキューブのアルゴリズム

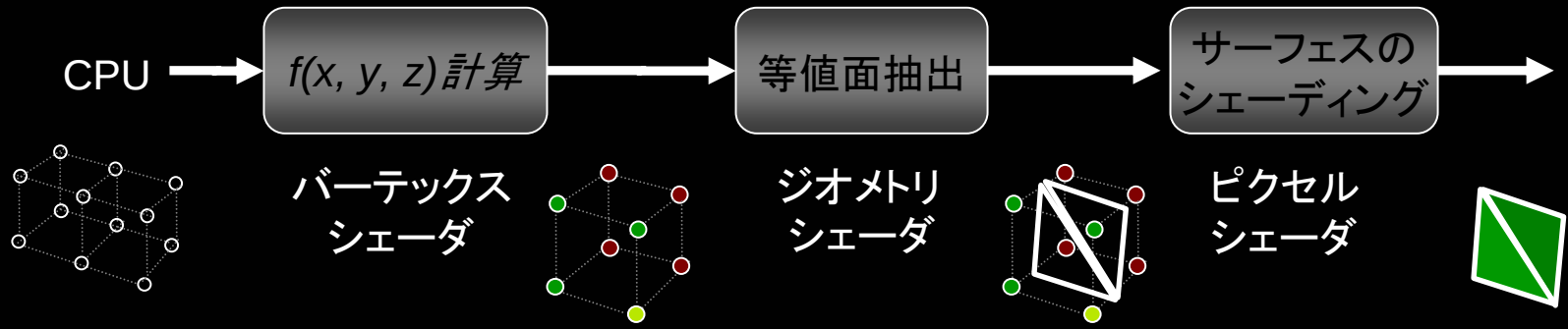
- 等値面のレンダ方法はレイ・トレーシングかポリゴン化
- マーチングキューブ: 等値面のポリゴン化のよく知られた例
- 立方格子でのサンプル $f(x, y, z)$
- 各頂点は「内側」か「外側」のどちらか
- ポリゴンの集合で各立方体サーフェスを近似表示

マーチングキューブのアルゴリズム

- 各立方体セルについて:
  - どれかの辺が内側の頂点と外側の頂点を結んでいる場合、等値面はその辺と交差する
  - 線形補間により、等値面と辺が交差する場所を推定する
  - 当該面が交差する辺の数に基づき、描出する三角形の数を変える

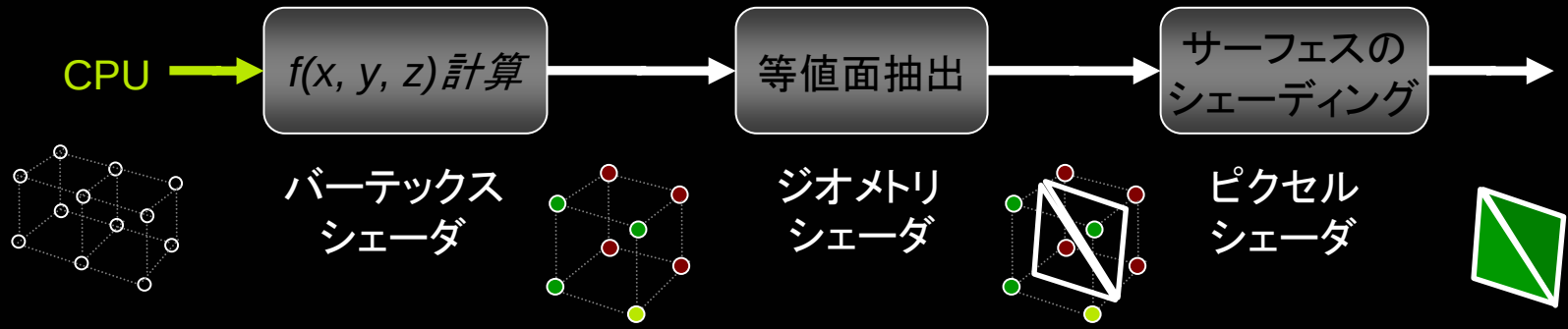


実装 - 疑似コード



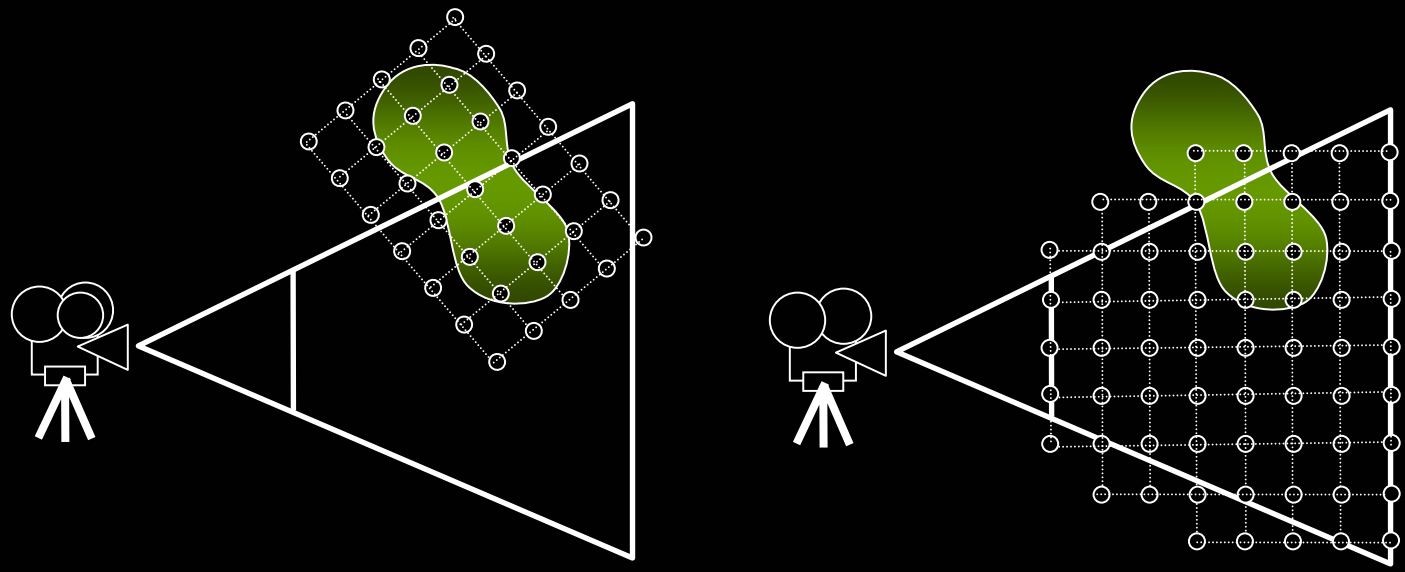
- バーテックスシェーダがグリッドの頂点を変換して
  f(x, y, z)を計算し、立方体をジオメトリシェーダに供給
- アプリケーションがGPUに頂点グリッドを提供
- ジオメトリシェーダが各立方体を順に処理して三角形を描出

実装 - 疑似コード



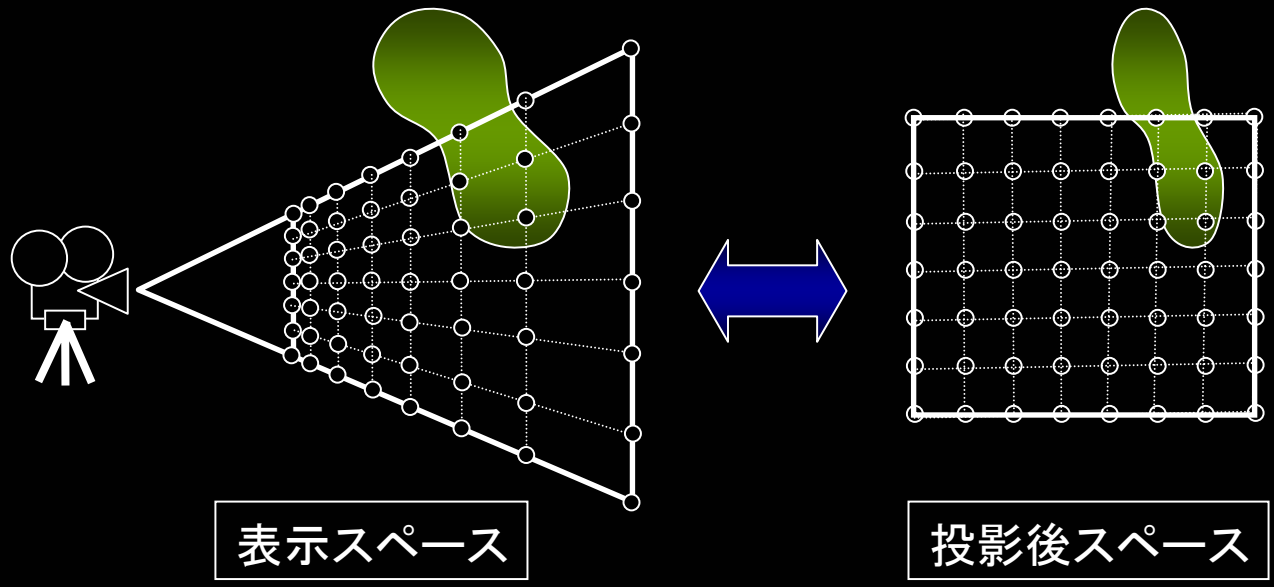
- アプリケーションがGPUに頂点グリッドを提供
- バーテックスシェーダがグリッドの頂点を変換して f(x, y, z)を計算し、立方体をジオメトリシェーダに供給
- ジオメトリシェーダが各立方体を順に処理して三角形を描出

モザイク処理スペース



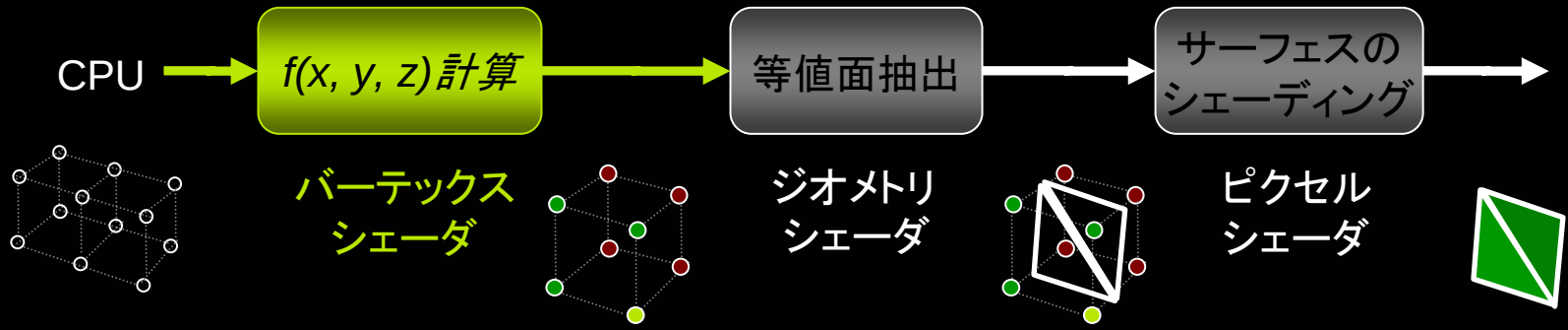
- オブジェクト・スペース
  - メタボール周囲のBBを計算するときに使用
- 表示スペース
  - 効果はすぐれているが、サンプリング・レートの分布が不適切

投影後スペースのモザイク処理



- 投影後スペース
  - 最適な選択肢と考えられる
  - 無料のLODも入手可能!

実装 - 疑似コード



- アプリケーションがGPUに頂点グリッドを提供
- バーテックスシェーダがグリッドの頂点を変換して
  f(x, y, z)を計算し、立方体をジオメトリシェーダに供給
- ジオメトリシェーダが各立方体を順に処理して三角形を
  描出

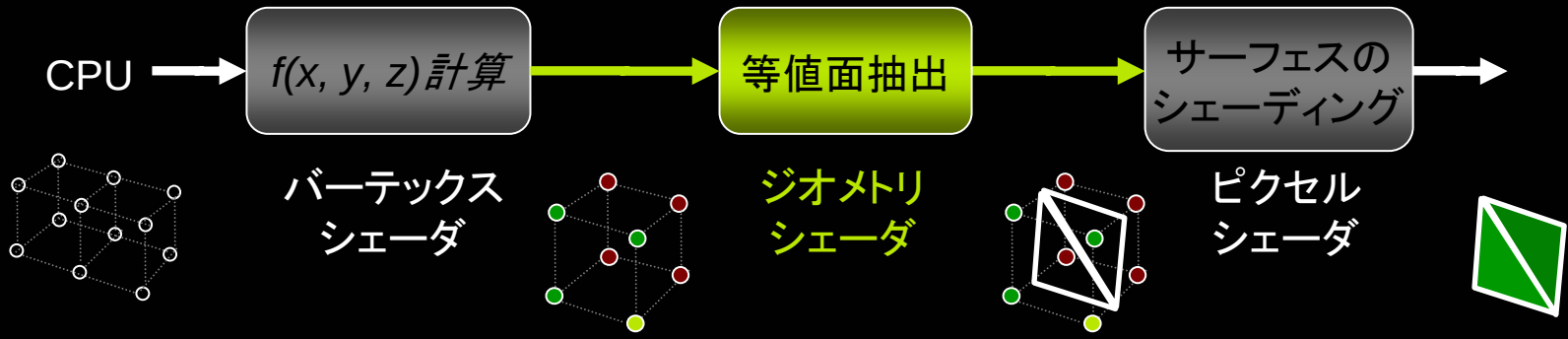
バーテックスシェーダ

- 各頂点vについて下記の値を計算:
  - スカラー場の値 $$f(v) = \sum_{i=1}^{N} \frac{r_i^2}{\left\| v - \mathbf{p}_i \right\|^2}$$
  - 頂点が場の中にあるかどうかを示すフラグ

$$Field = f(v) > 1 ? 1 : 0$$

  - スカラー場の法線
  - 頂点の投影位置

実装 - 疑似コード



- アプリケーションがGPUに頂点グリッドを提供
- バーテックスシェーダがグリッドの頂点を変換して f(x, y, z)を計算し、立方体をジオメトリシェーダに供給
- ジオメトリシェーダが各立方体を順に処理して三角形を描出

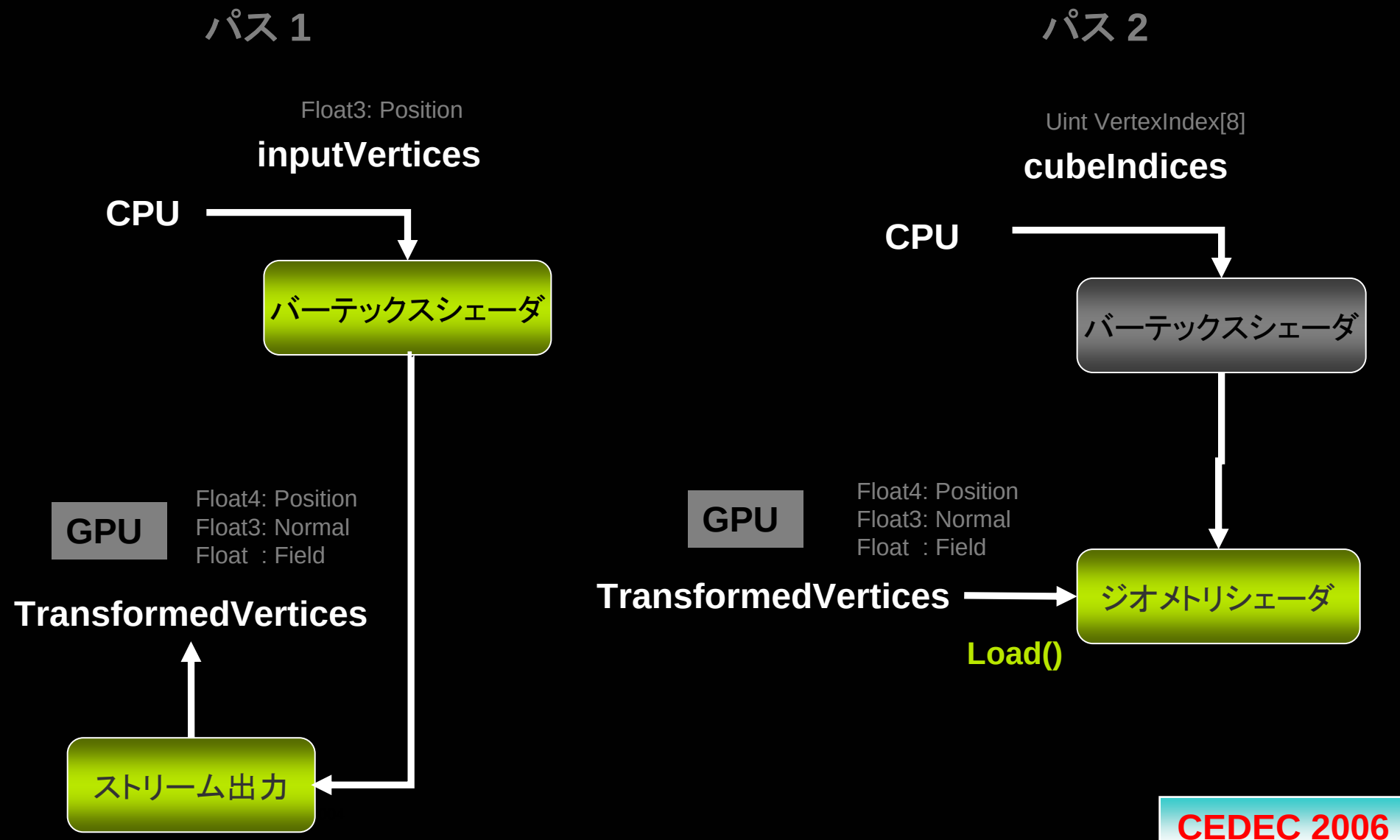
ジオメトリシェーダ(GS)で8個の頂点を取得する方法

- ジオメトリシェーダから直接に頂点バッファ内の特定のインデックス値を読み取る:

```
vertexValue = VertexBuffer.Load(index);
```

- 8個のステートメントを発行して特定の立方体のすべての頂点をフェッチする

GSで8個の頂点を取得する方法

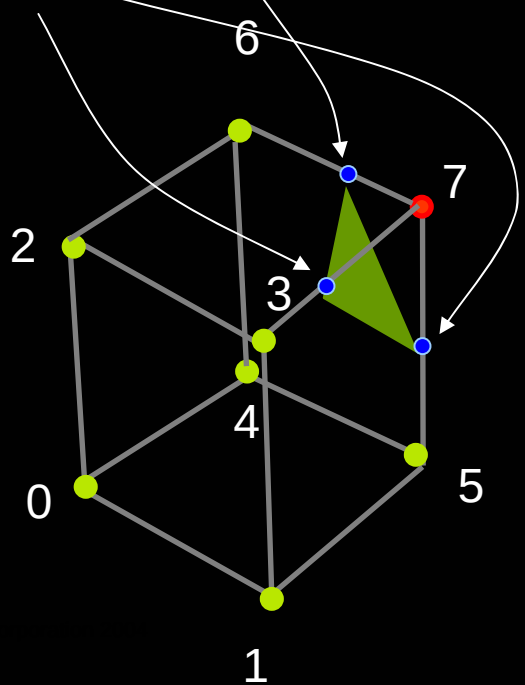


ジオメトリシェーダ

```
[MaxVertexCount(16)]
void GS_TesselateCube(point CubePrimitive In[1],inout
    TriangleStream<SurfaceVertex> Stream)
{
    //1. Construct index and load field data into temporaries
    uint index = 0;

    for (uint i = 0; i<8; i++)
    {
        //construct bit field with a bit set for every vertex inside surface
        index |= SampleDataBuffer.Load( In[0].VertexIndex[i] ).Field > 1 ? 1 : 0;
        index <<= 1;
    }
```

エッジテーブルの構築

```
// StripCount contains number of triangle strips to generate for particular index value
const uint2 StripCount[256] = {
          { 0,                                                    //index = 0
            1,                                                    //index = 1
            // …
     };
// EdgeTable stores precomputed vertex indices for each cube edge which needs to be
     interpolated
const uint2 EdgeTable[256][16] = {
          { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }, //index = 0
          { 7, 3, 7, 5, 7, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }, //index = 1
          // …
     };
```



Index = 00000001,
すなわち頂点7は「内側」

ジオメトリシェーダ

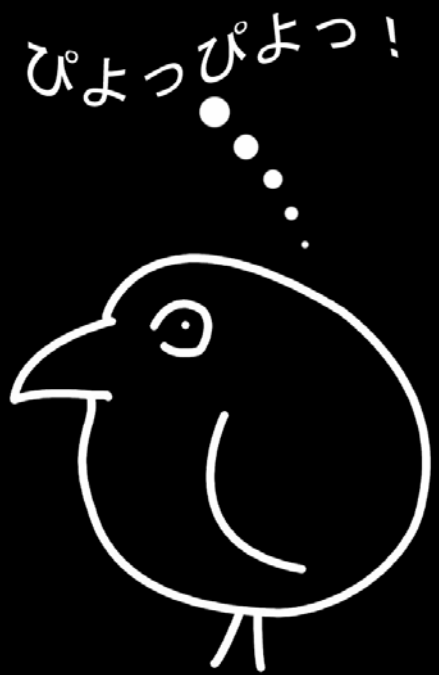
```
    // 2. Generate triangle strips according to "index" value

    // Get number of triangle strips for this index
    uint NumStrips = StripsCount[index];

    // Emit that many triangle strips...
    uint j = 0;
    for (uint i = 0; i<NumStrips; i++)
    {
        while (1)
        {
            uint2 edge = EdgeTable[index][j++];
            if (edge.x == edge.y) {    // edge.x == edge.y indicates a restart
                Stream.RestartStrip();
                break;
            }

            Stream.Append( CalcIntersection(
                  SampleDataBuffer.Load( In[0].VertexIndex[edge.x] ),
                  SampleDataBuffer.Load( In[0].VertexIndex[edge.y] )
                              ) );
        }
    }
}
```

メタボール

- ジオメトリシェーダ を等値面抽出に活用可能
- 医療データの視覚化にマーチングキューブも利用可能
- まったく新しい効果の作出ができる
  - 動くバルジによる有機化合物
  - 流体のゲーム中の動作などのモデル化(モデル流体によるパーティクル系)
  - GPGPU によるメタボールのアニメ化
  - ノイズを追加して乱流フィールドを作成

結論

- 新しい機能を提供 … CPUでのみ稼働するアルゴリズムからGPUで稼働するアルゴリズムへ:
  - メタボール
  - フィン
- フレキシビリティの増大 … GPGPUのような他のアプリケーションへの実装がより容易、より効率的に
  - 布



ブライアン・デゥダーシュ（Bryan Dudash）
bdudash@nvidia.com

CEDEC 2006