次世代グラフィクス特殊効果

Bryan Dudash

NVIDIA

セッション概要

- 3.0シェーダーモデルの概観
  - ps.3.0とps.2.0
  - vs.3.0とvs.2.0
- 次世代描画の例
  - 動的な水
    - 頂点テクスチャ読み出し
    - 浮動小数点フィルター/ブレンド
    - GPUによる物理シミュレーション
  - ボリューム・フォグ
    - MRTとブランチによる高速化
  - ディファード・レンダリング
    - MRTとブランチによる高速化
- ジオメトリ・インスタンシング
  - 視覚的な複雑さを演出
  - パフォーマンスの最適化

ピクセル・シェーダー3.0機能比較

シェーダー機能	シェーダー2.0	シェーダー3.0	解説
シェーダーの長さ	96	65535以上	より複雑なシェーディングやライティング手続き型マテリアルを可能にする
動的分岐	なし	あり	処理の必要がないピクセルをスキップしパフォーマンスを向上する
シェーダーによるアンチエイリアス	サポートなし	デリバティブ命令	任意の関数の画面に対する偏微分(変化量)を求めることにより、シェーディングの頻度を調整して描画問題を克服する
最低の精度	fp24	fp32	描画問題を減らし、広いダイナミックレンジを確保
背面レジスタ	なし	あり	1パスで両面の描画が可能

ピクセル・シェーダー3.0機能比較

シェーダー機能	シェーダー2.0	シェーダー3.0	解説
背面レジスタ	なし	あり	1パスで両面の描画が可能
色補間の形式	最低8ビット整数	最低32ビット浮動小数点	より広域で高い精度の色指定により、頂点でのハイダイナミックレンジ・ライティングを可能にする
複数描画対象(MRT)	任意	最低４つ	高度なライティングのアルゴリズムによりフィルタリングや頂点の仕事量を減らし、結果的により多くのライトを使用可能にする
フォグとスペキュラ	最低8-bit定型処理	Fp16からfp32でのシェーダー処理	以前は定型処理だったフォグやスペキュラの処理をシェーダーモデル3.0では制御できるようになった
テクスチャ座標数	8	10	特に皮膚などで、より多くのピクセル毎の入力を使うことができ、現実感が増す

バーテクス・シェーダー3.0機能比較

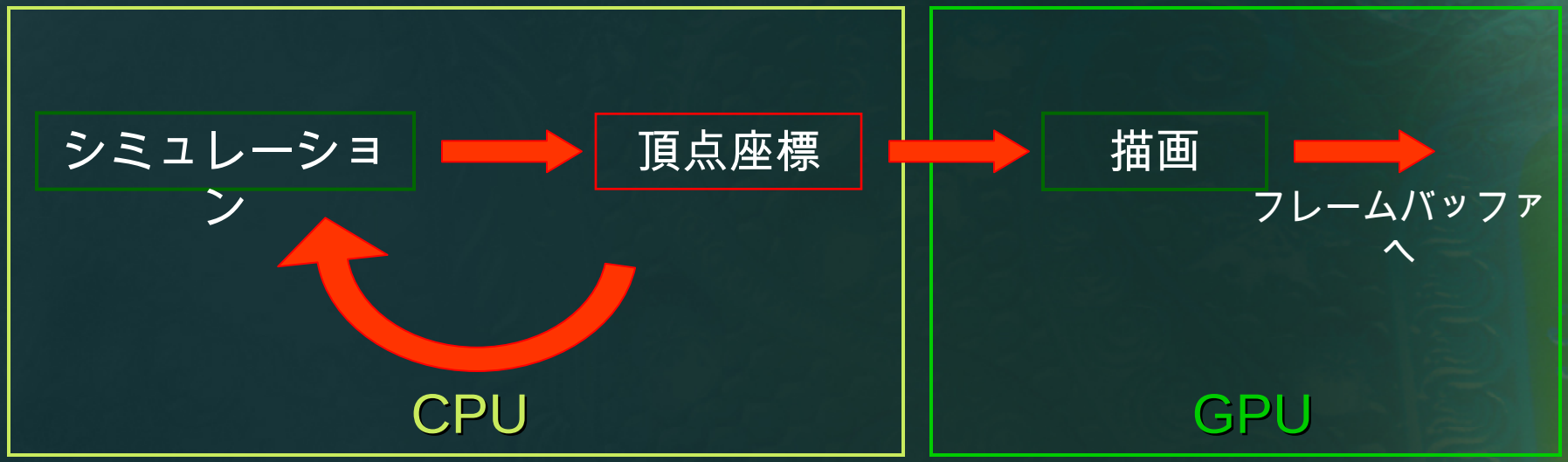
シェーダー機能	シェーダー2.0	シェーダー3.0	解説
シェーダーの長さ	256命令	65535命令	より多くの命令により、より精密なキャラクタのライティングやアニメーションが可能
動的分岐	なし	あり	関係ない頂点のアニメーションや計算をスキップすることにより、パフォーマンスを向上
頂点テクスチャ	なし	4つのテクスチャから任意の回数の読み出し	ディスプレイスメント・マップやパーティクル効果を可能にする
インスタンシングサポート	なし	必要	ひとつのコマンドにより、多くの変化のあるオブジェクトを描画する

ではこれで何ができるの？

- 動的な水の描画
  - VS3.0頂点テクスチャ読み出し
  - 浮動小数点フィルター/ブレンド
  - GPUによる物理シミュレーション
- アニメーションするボリューム・フォグ
  - ポリゴンによるフォグの境界設定
  - MRTや分岐による高速化
- ジオメトリ・インスタンシング
  - 一回の描画呼び出しでたくさんのインスタンスを表示

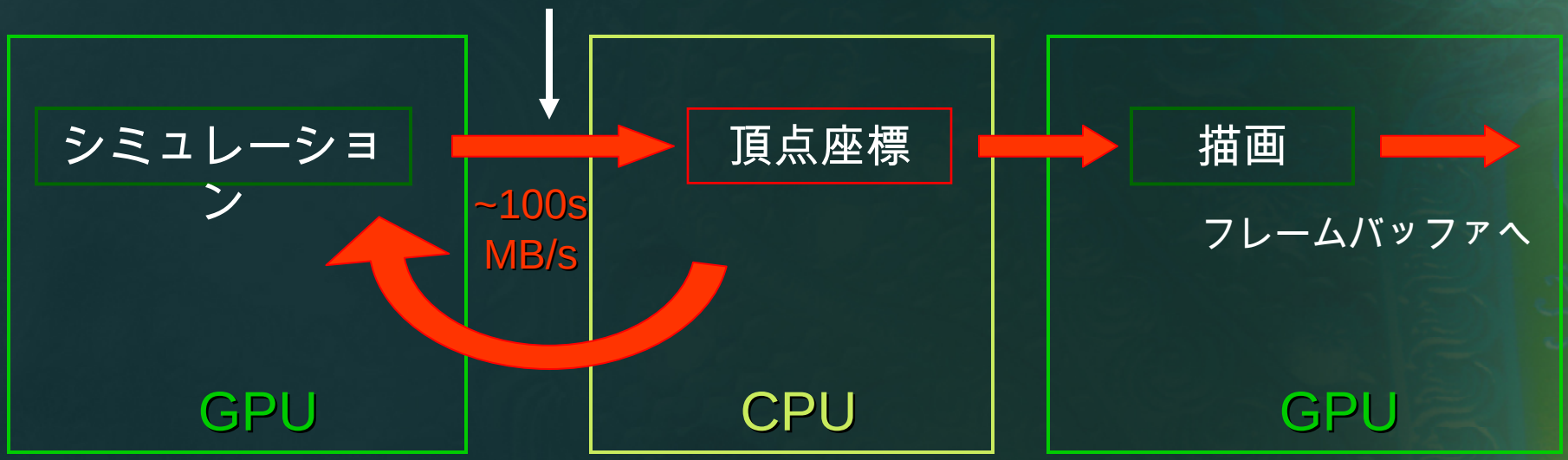
一般的な作業の流れ

一般的な作業分担



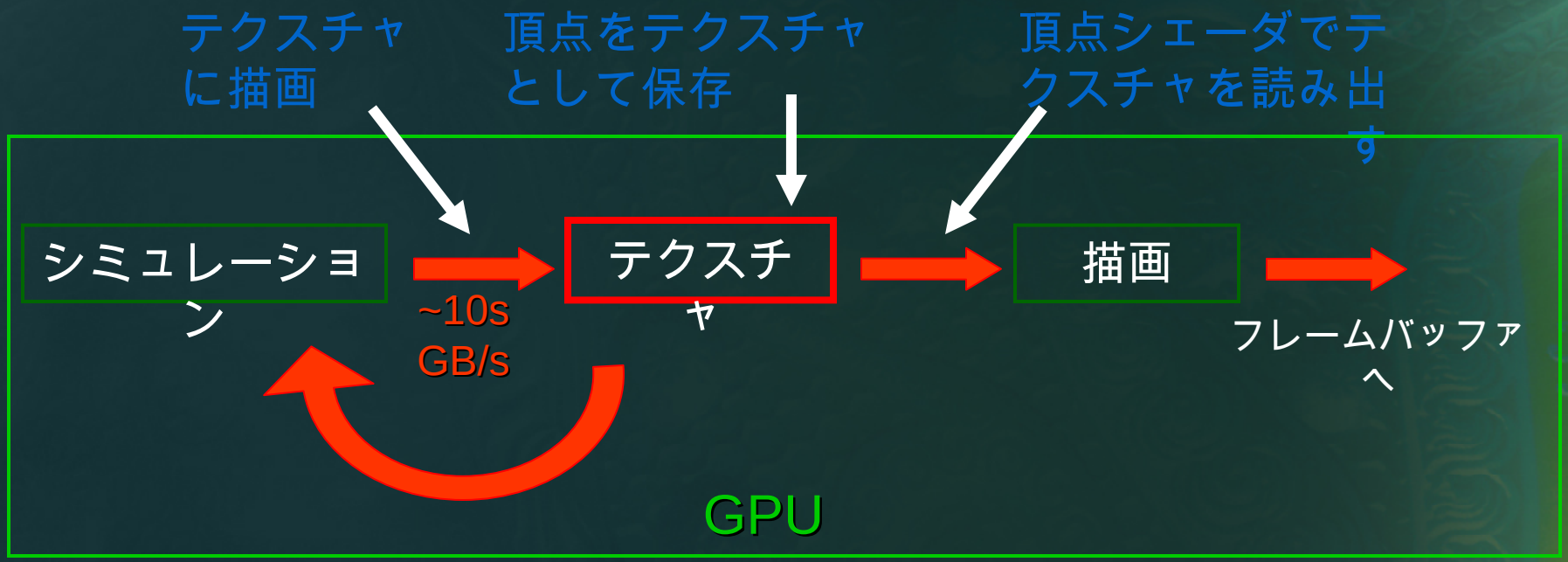
GPUでシミュレーション？

- シェーダーを使う
  - 読み出しが問題
  - PCIでの場合。PCI Expressなら改善される



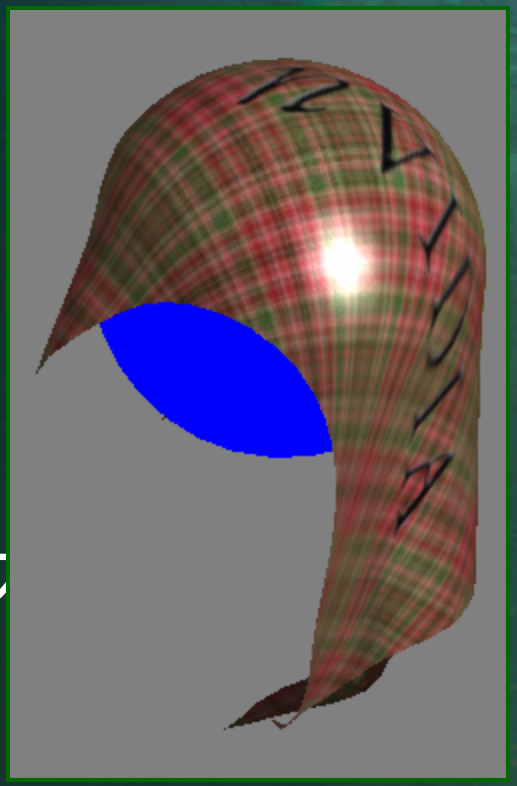
頂点バッファへの描画

- GPUからCPUへの読み出しをなくす



例

- 布
  - 布とシーンの衝突判定
  - 布のシミュレーション:
    ダンパー入りのスプリング
- ディスプレイスメント・マッ
  - 頂点の位置移動

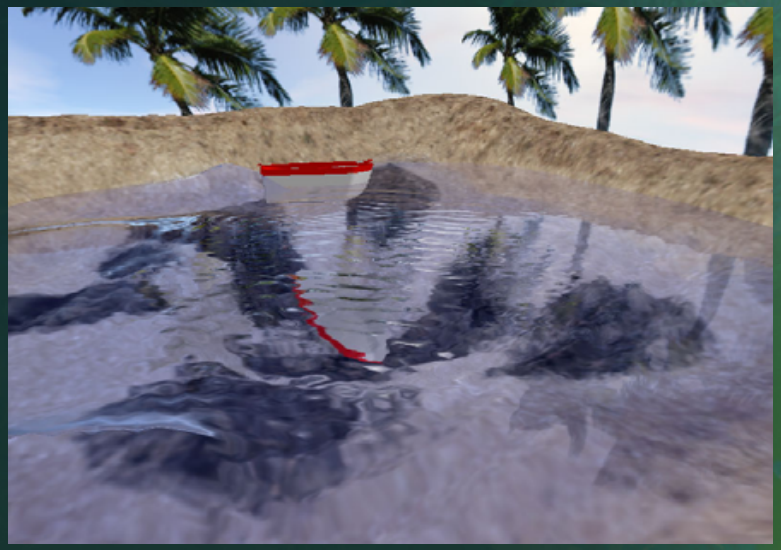


他の例

- 雪/砂が積もる
  - 摩擦や滑り落ちのシミュレーション
- 風によってなびく草のシミュレーション
- パーティクル・システム
- 波や波頭

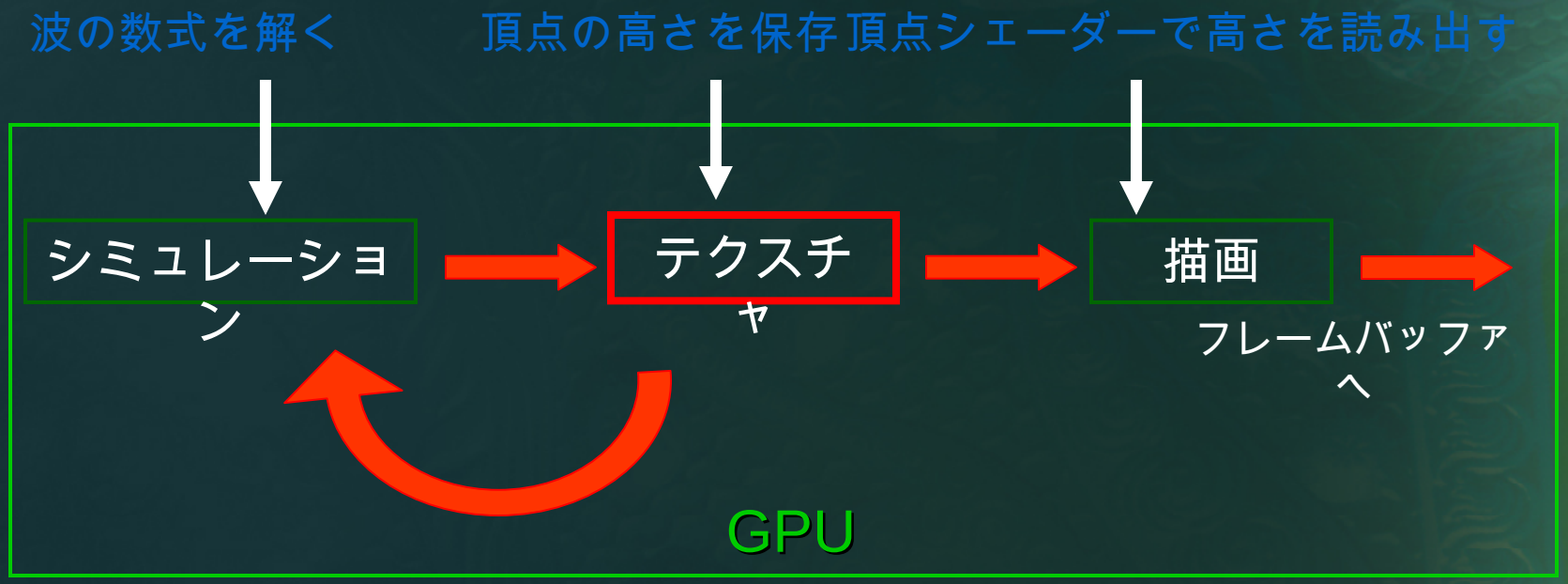
水の描画 – アルゴリズムの概略

- ピクセルシェーダーでの水のシミュレーション
  - テクスチャへの描画 (D3DFMT_A16B16G16R16F)
- 反射と透過マップの描画
- 水面の描画
  - VS3.0の頂点テクスチャ読み出しを使って、シミュレーション結果を読み出す
  - 移動したテクスチャ座標の計算
  - フレネルで透過と反射の合成

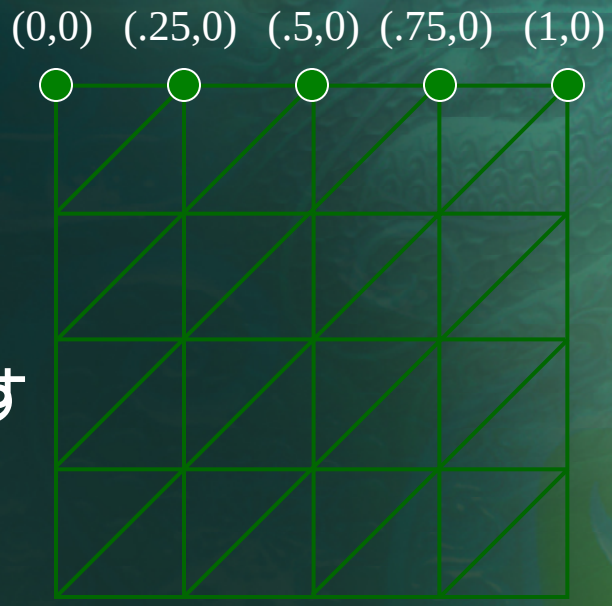


水の描画

- 細分化された波のない水面メッシュ



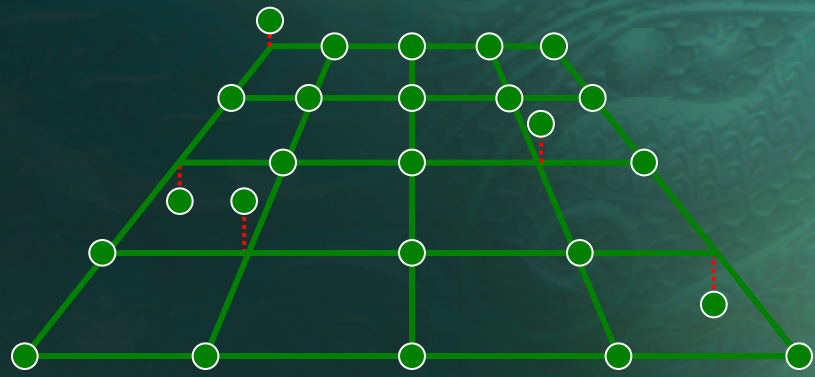
どうやって動く？



- 水面の頂点メッシュを作成
  - 例：128x128の頂点
  - 頂点のメッシュ位置をUV座標とする

頂点シェーダー処理

- 高さマップの読み出し
  - 浮動小数点テクスチャ
  - 頂点のUVに対応する位置から読み出し
- 頂点のYに結果を足す
- 変換、投影マトリクス



vertex.y += tex(u, v)

Out.pos = WorldViewProj * vertex

高さマップは動的

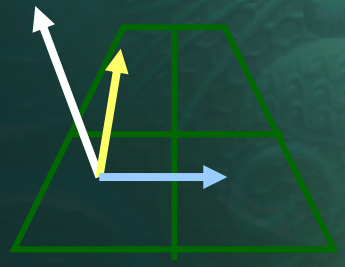
- フレームごとに更新
  - GPUのテクスチャ描画によって
- Verlet積分

Verletって？

$A = \sum (\text{周辺}) - 4 H_n$

$H_{n+1} = H_n + (H_n - H_{n-1}) + A$

$H_{n+1} = (2*H_n - H_{n-1}) + A$



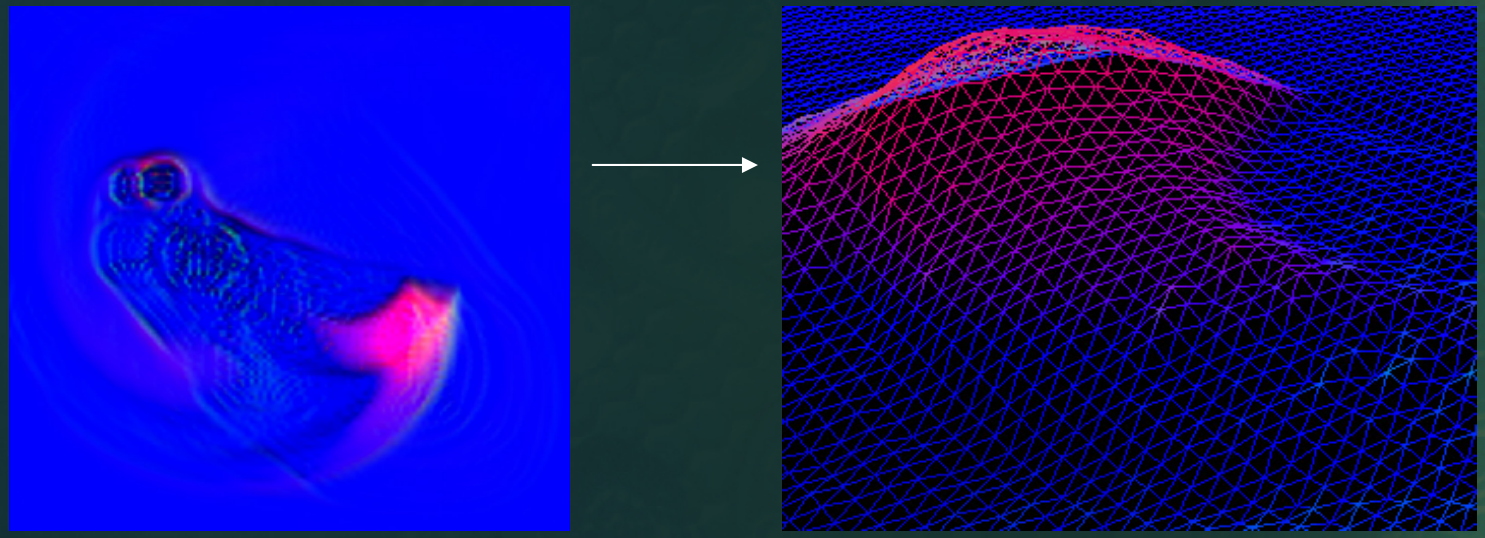
- 位置情報でのみ可能
  - 速度や加速度は要らない
- 位置から法線を算出:

  N = Normalize(S x T)

頂点テクスチャ読み出し(VS3.0)

- 頂点シェーダーはシミュレーション結果を頂点テクスチャ読み出しで取得

シミュレーション・テクスチャ　　高さマップを使用



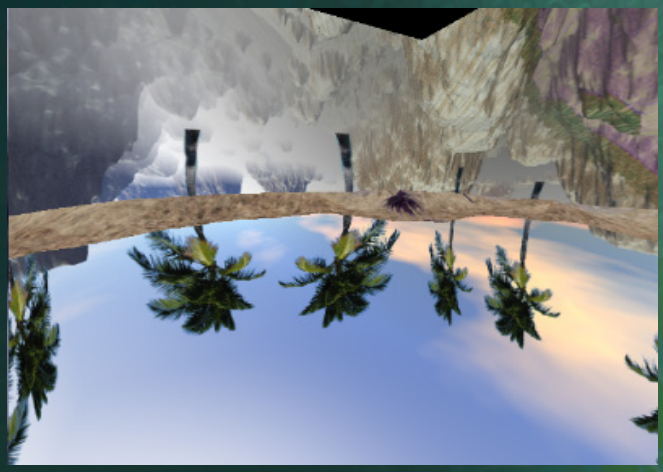
高さマップにかく乱成分を足す

- 変化量を水にブレンドする
  - 例: ボート、岩、岸
- Verlet積分で次のフレームに影響
- 浮動小数点ターゲットのブレンドを使用

透過マップ

- カメラ位置からシーンを描画
- 水面の向こう側にあるものを描画する
  - 水面でジオメトリをクリップ
  - カメラが水面の上にある場合
    - 水中のジオメトリを描画
  - カメラが水面の下にある場合
    - 水上のジオメトリを描画

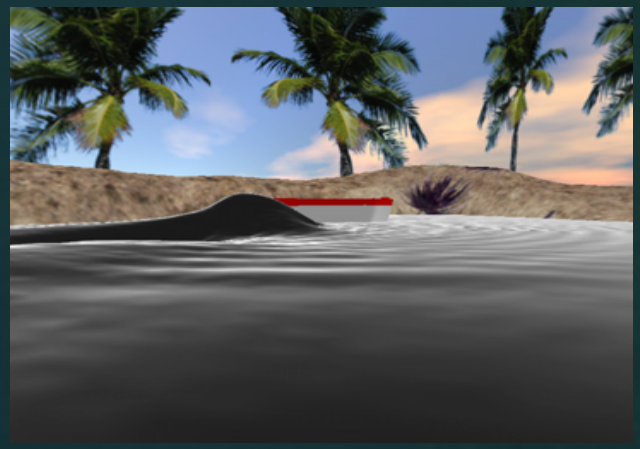
反射マップ

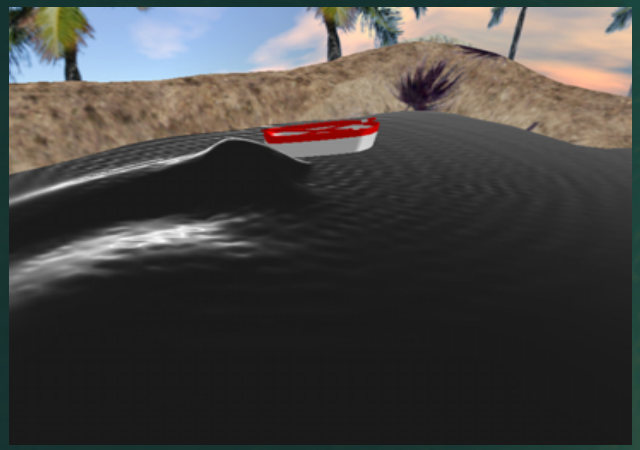


- カメラを反転した位置から描画
  - 水面に対して反転
- 水面を使ってジオメトリをクリップ
- 水に対して、カメラと同じ側にあるものを描画
  - カメラが水面の下にある場合
    - 水中のジオメトリを描画
  - カメラが水面の上にある場合
    - 水上のジオメトリを描画

フレネル反射項

- 反射と透過の割合を決める
- 基本的にpow((1 - dot(eye, normal)), p)
  - フレネル項 = 0 => 透過のみ
  - フレネル項 = 1 => 反射のみ

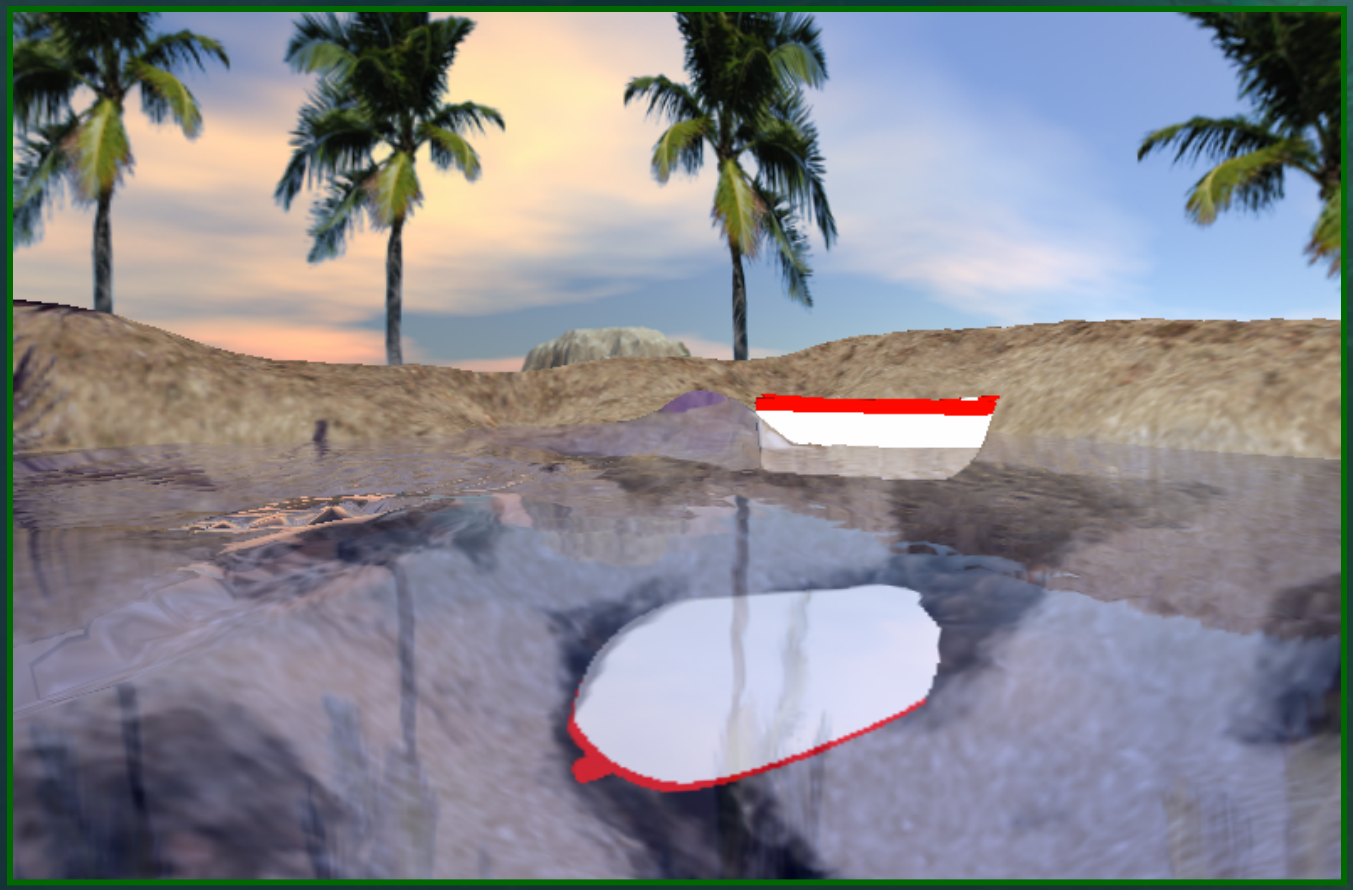




利点

- 高速度!
  - 128x128テクスチャに対してのシミュレーション
  - GPUにとっては大した大きさではない
  - シミュレーションによるフレーム・レートへの影響は見られない
- ほどよいジオメトリの複雑さ
  - 128x128で1万6千頂点

VTFデモ



半透明と拡散

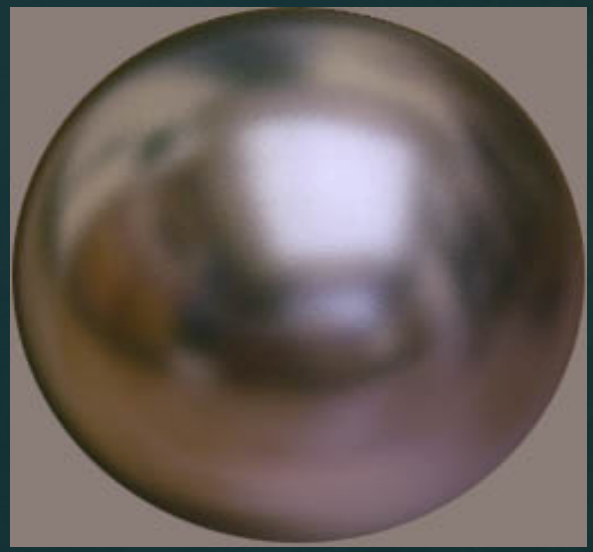
- 全ての物質は光を通す
  - 光の波長による
- 光は全ての表面をいくらかの度合いで透過する
  - 違った波長で
    - 透過の深度が違う
    - 深度による減退が違う
- 吸収か反射されなければ、光は拡散して他の面へ

透過光の基本

- 光学的属性
  - 吸収 (距離に対する確率)
  - 拡散 (距離と角度に対する確率)
  - インピーダンスの変化 (反射と透過)
    - 光学的インピーダンスで屈折率が決まる
- 全ての物質は吸収と拡散を行う
  - 液体、固体、気体や清浄な空気さえ
- 不透明、透明、半透明
  - 吸収と拡散の確率が違う

不透明

- 高い確率での吸収と拡散
- 光は短い経路をたどる
- 光は内部からではなく表面から



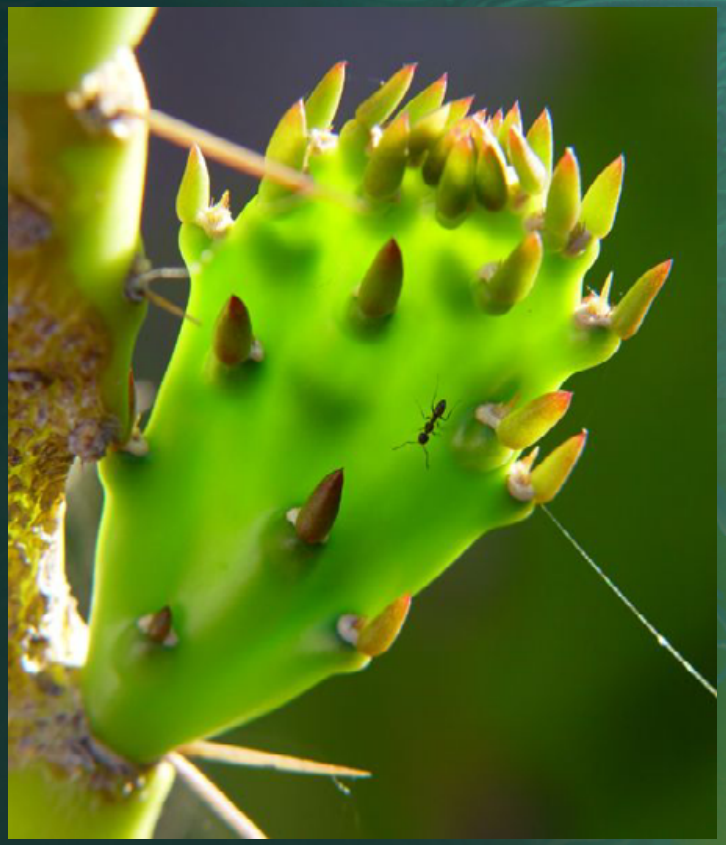
透明

- 低い確率での吸収と拡散

Images courtesy of Leigh Van Der Byl

半透明

- 低い確率での吸収
- 高い確率での拡散



Leigh Van Der Byl

リアルタイムの態度

- みかけが大切。数式は後回し
  - 拡散の計算式についてはHoffman & Preethamを参照
- 各種のテクニック
  - 厚さや拡散のための深度マップ描画
  - テクスチャ空間での拡散
- 要求
  - デザイナーに優しくデータに優しい
  - 高速
  - 代替描画方法
  - ライティングやセルフ・シャドウでのアニメーションも可

深度マップ

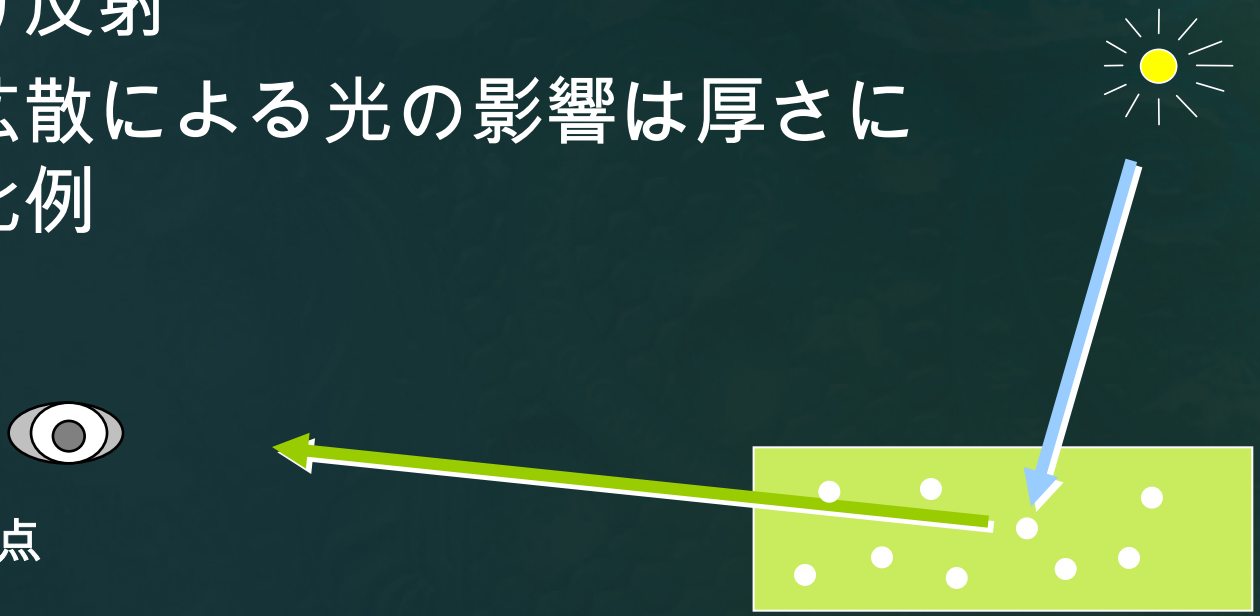
- フォグは普通のポリゴン
- テクスチャへの描画を用いてフォグの深さを測る
- ps.1.3
- ps.2.0はより高速
- ps.3.0は更に高速

ボリューム・フォグ

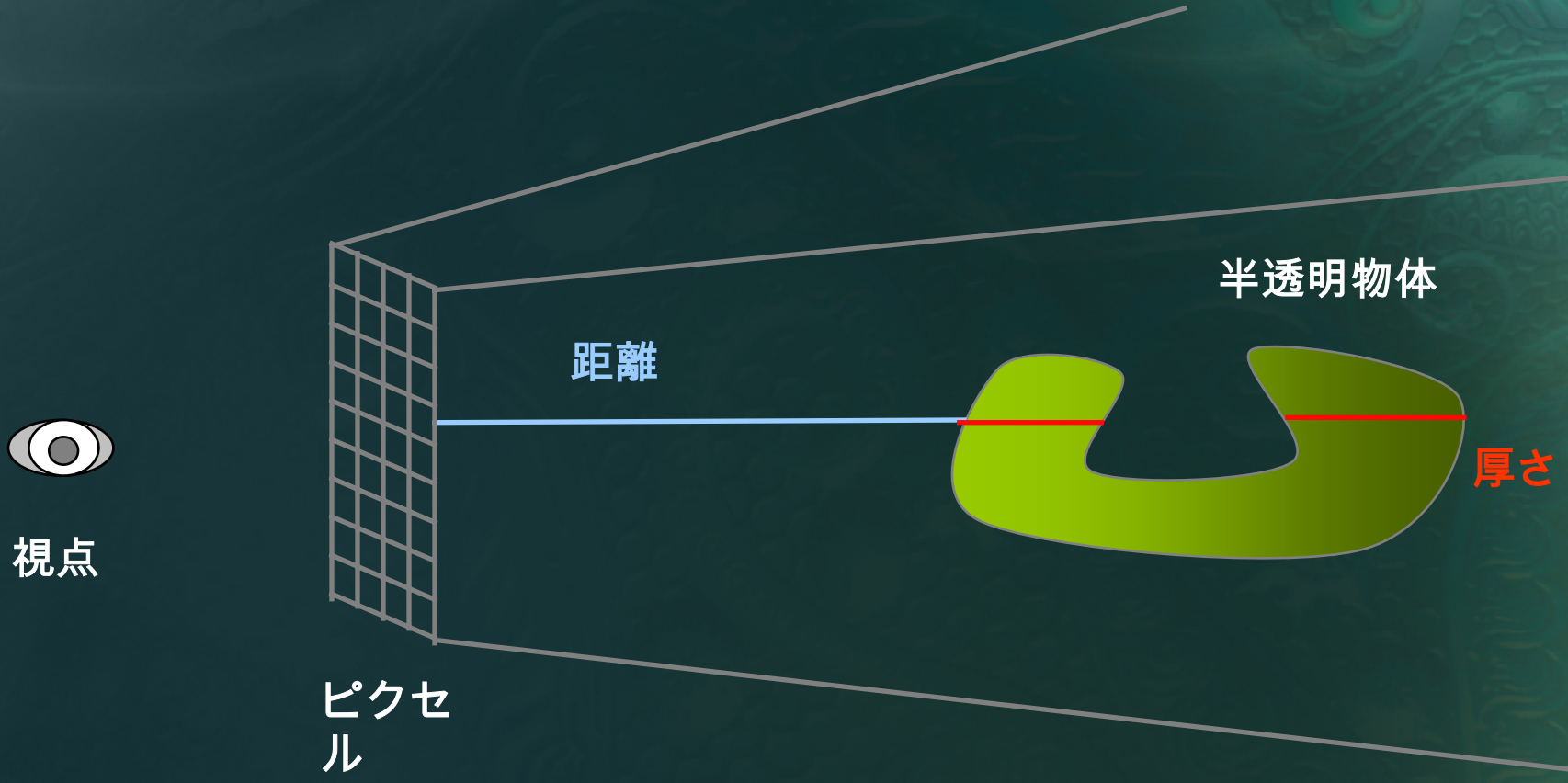
- マイクロソフトによる『ボリューム・フォグ』DXSDK デモ(Dan Baker)からヒントを得た
- [Mech01]からもヒントを得た
- 普通のポリゴンモデルを使用してカメラ位置からのフォグの厚さを求める
  - 物体の前面と背面の距離を計算
- 厚さから色を決定
- 一次拡散の計算方法として有効

一次拡散

- 光源から目まで、光は一回だけ反射
- 拡散による光の影響は厚さに比例



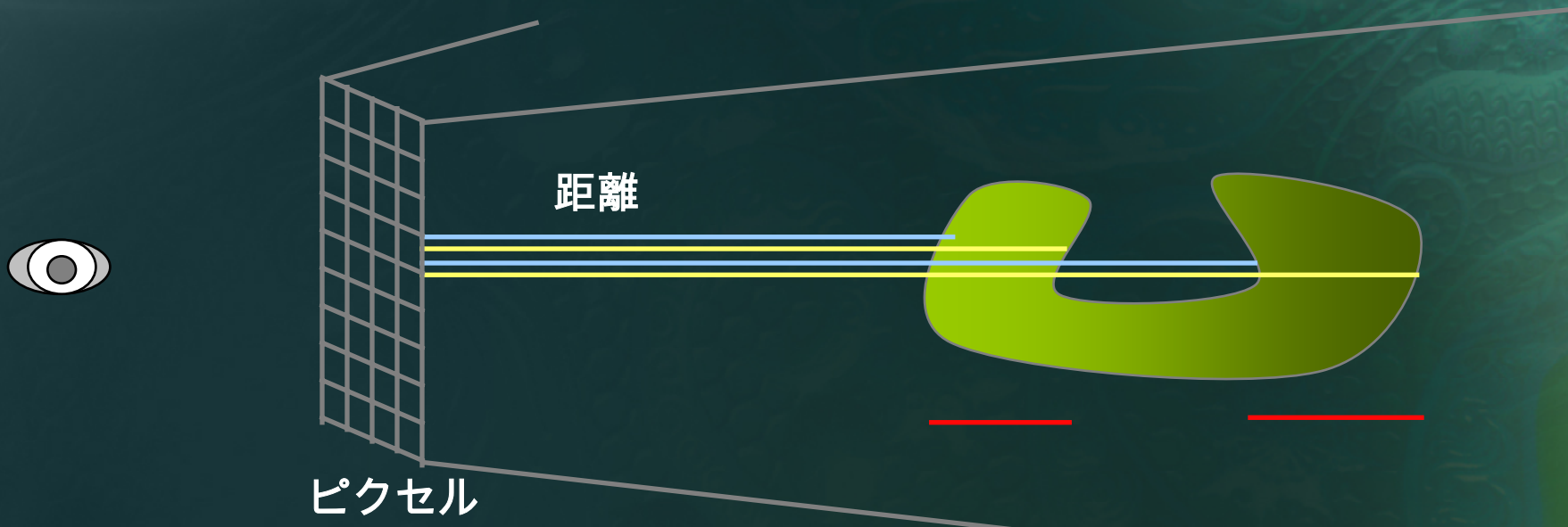
ピクセルごとに厚さを描画



距離から厚さを求める

厚さ = 背面 – 前面

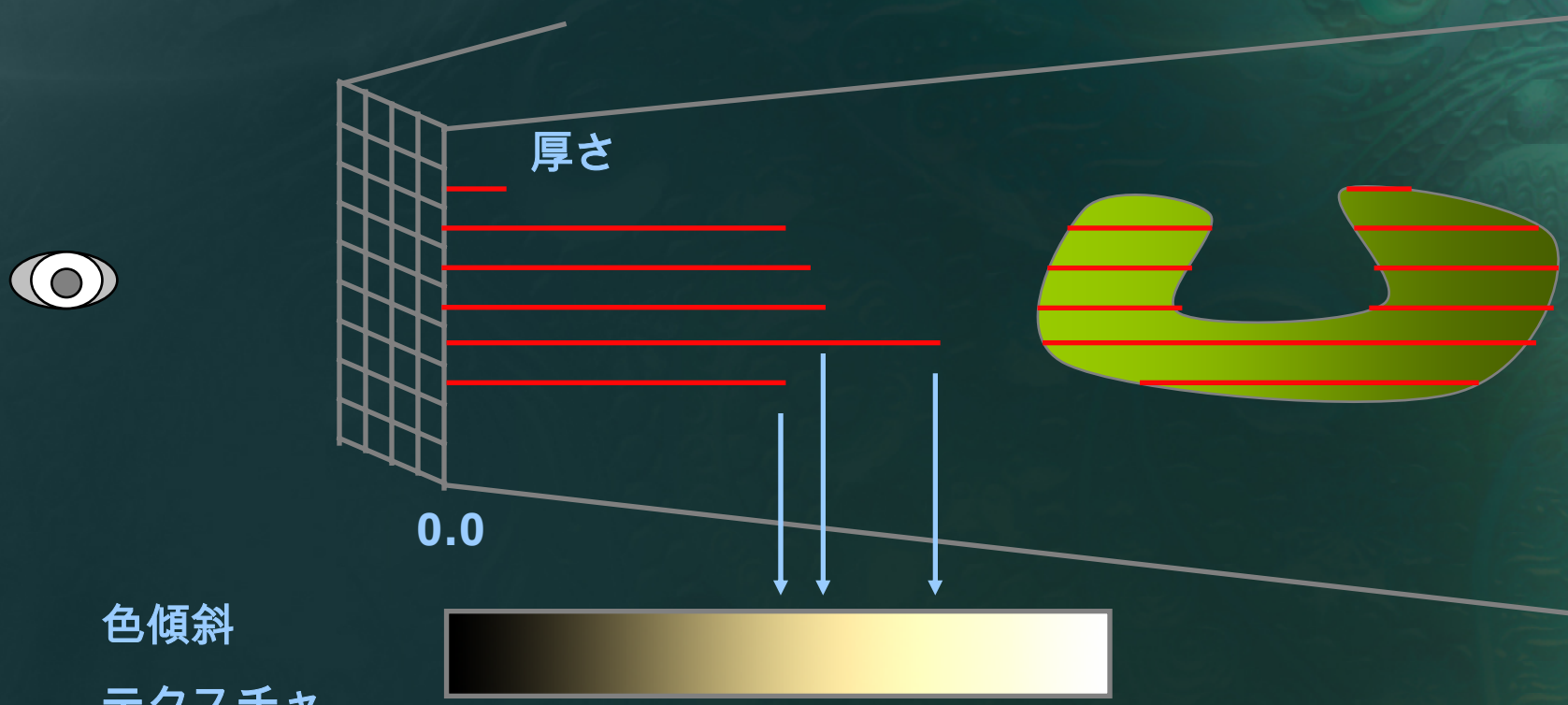
ピクセルごとに厚さを描画



$$Thickness = \sum Back - \sum Front$$

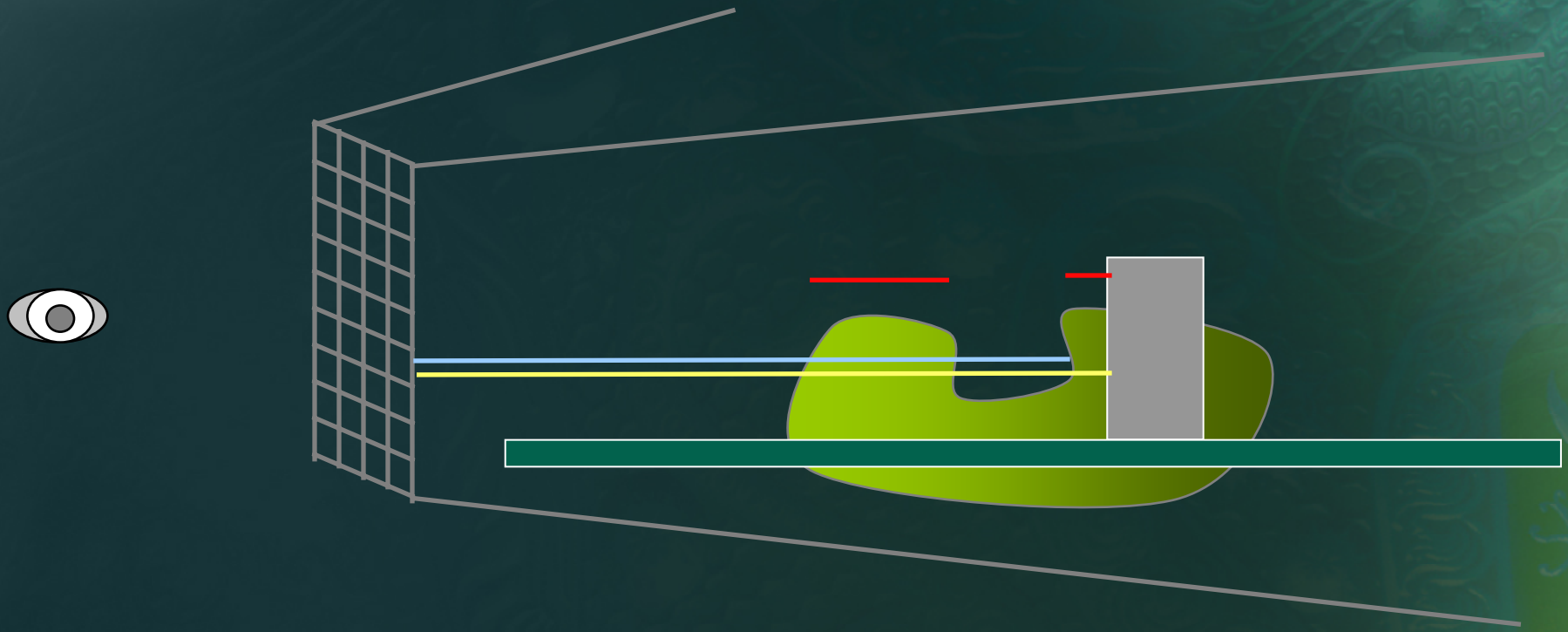
- 一定密度の物体の厚さは簡単
- Zを使用せず、加算ブレンドを使用

厚さを色に変換



- 厚さ * 係数 ➔ TexCoord.x
- 色傾斜テクスチャ: デザイナが作るか、計算で求める
- 見栄えの調整が容易

ポリゴンの交差については？

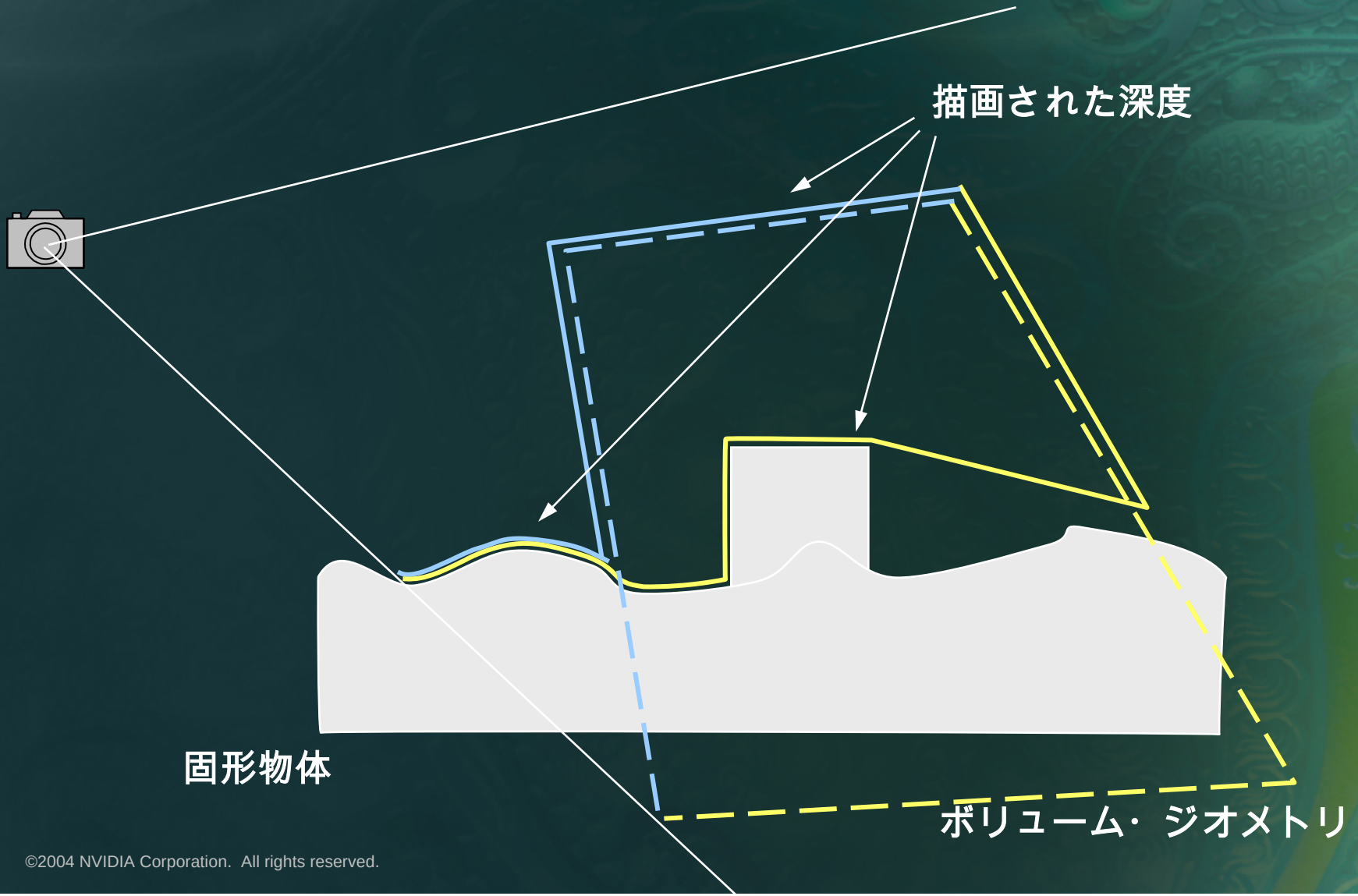


- 固体までの深さが必要
- ボリューム・オブジェクトの面までの深さではない

交差の解決方法

- 一番近い固形の物体までの距離が必要
  - テクスチャへの描画
  - ピクセル・シェーダーにテクスチャを読み込む
- ボリューム・オブジェクトの面を描画する際
  - ピクセル・シェーダーでは小さい方の値を選択
    - ボリューム・オブジェクトの面の深さ
    - ピクセル位置に対応するテクスチャにある固形オブジェクトの深さ
  - Zテストは無効にしておく
  - フレームバッファに結果を加算ブレンド

交差の解決方法



交差解決方法の利点

利点

- ステンシルを必要としない
- マルチパスを必要としない

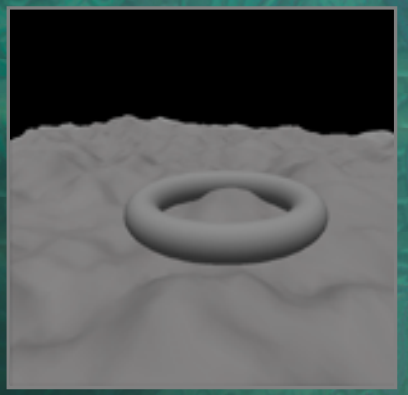
問題点

- 深度の描画が必要
  - ボリュームに交差する全てのもの
  - ボリュームを隠す可能性のある全てのもの
    - シーンによっては問題にならないかもしれない

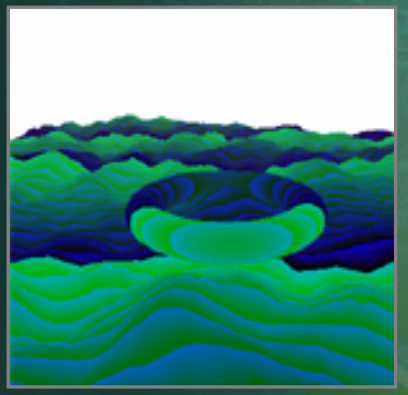
工程: ピクセル・シェーダー2.0

1. 固形の物体をバックバッファに描画
   - 普通の描画
2. フォグに交差するかもしれない固形物体の深度を描画
   - ARGB8に描画 - “S”
   - 深さをRGBにエンコード – 高解像度!
3. フォグの背面を描画
   - ARGB8に描画 - “B”
   - 加算ブレンドで深さを足していく
   - テクスチャ“S”で交差を確認

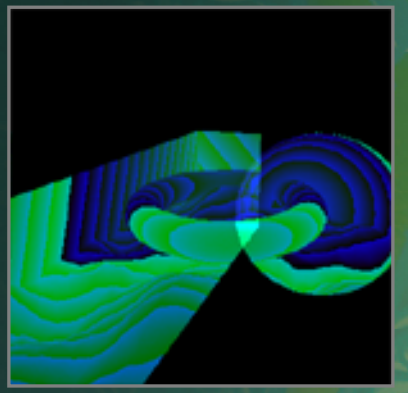
O



S



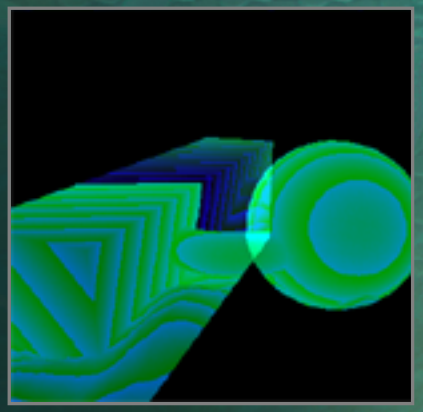
B

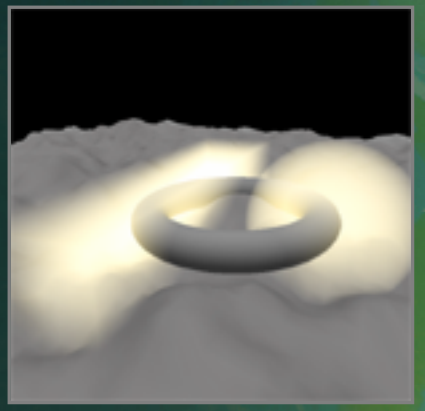


工程: PS.2.0の続き

4. フォグの前面を描画
   - ARGB8へ描画 - “F”
   - 加算ブレンドで深さを足していく
   - テクスチャ“S”で交差を確認
5. バックバッファに対して四角形を描画
   - “B”と“F”を読む
   - ピクセル毎の厚さを計算
   - 色傾斜テクスチャを使って、厚さを色に変換
   - シーンに色をブレンド
   - ps.2.0シェーダーで5命令

F





結果

PS.3.0ハードウェアでの改善

- 前面/背面レジスタ
- 複数描画対象(MRT)
- 浮動小数点フレームバッファのブレンド

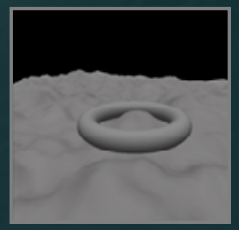
- 少ないパス
- 少ない描画対象テクスチャ

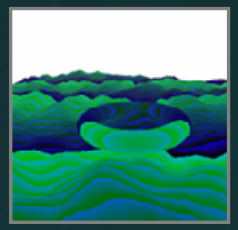
PS.3.0 vs. PS.2.0

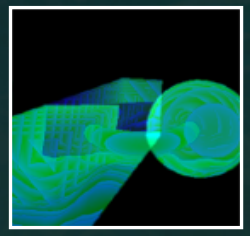
MRT

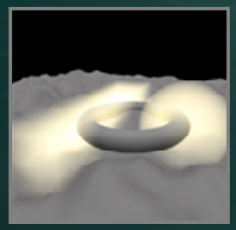
F/Bレジスタ

ps.3.0 HW


RT-Tex

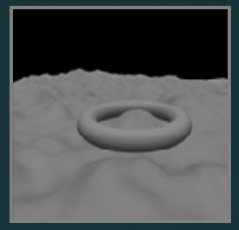

RT-Tex ‘O’

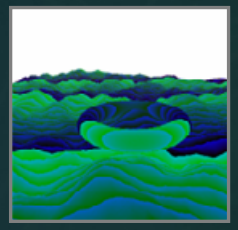

浮動小数点 RT-tex

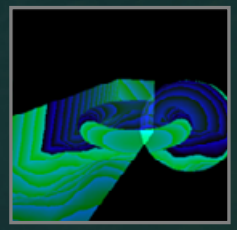

バックバッファ

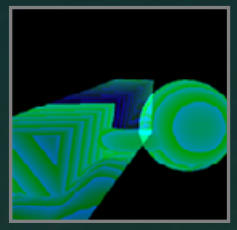
3パス

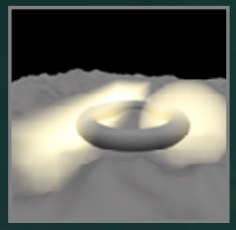
ps.2.0 HW


バックバッファ


RT-Tex ‘O’

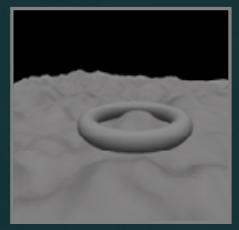

RT-Tex ‘B’

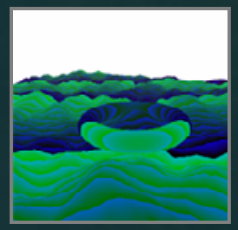

RT-Tex ‘F’

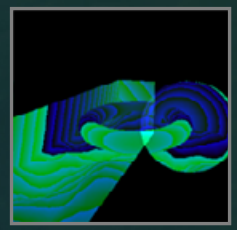

バックバッファ

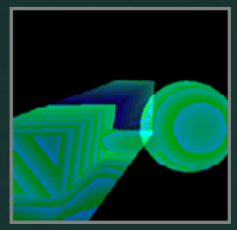
5パス

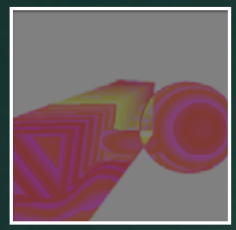
ps.1.3 HW

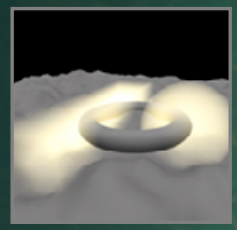






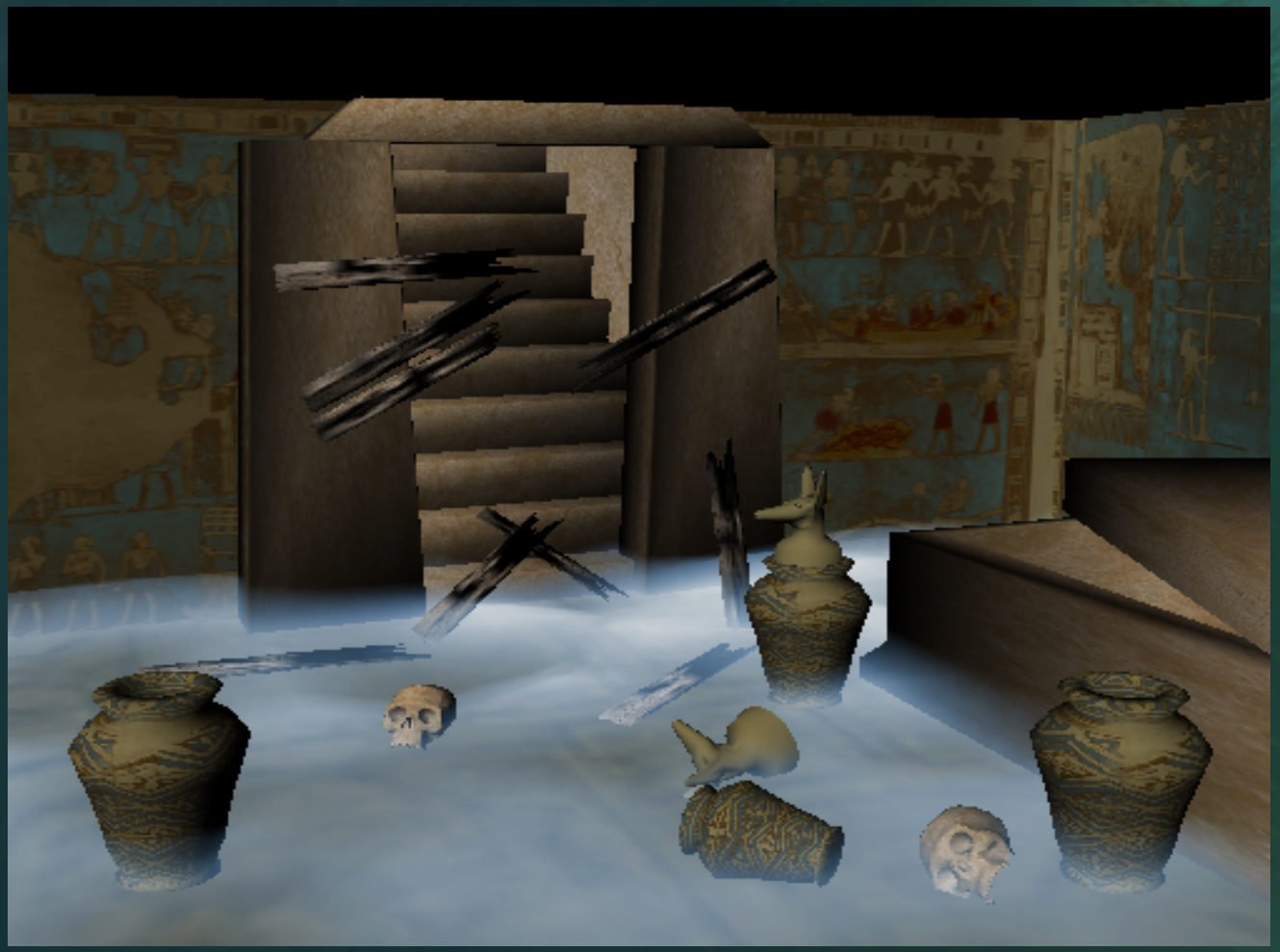






6

ボリューム・フォグのデモ



ディファード・レンダリング

- これは何?
  - オフスクリーンのバッファに簡単な計算結果を描画する
    - 法線、カメラ位置、光源ベクター等
  - 最後のパスでライティングを組み込む
- なぜ描画を後にするの?
  - 複雑なシェーダーで余分なフラグメントを描画するのはパフォーマンスに負荷が大きい
  - 最後のパスは深度の複雑さがない
    - PS3.0を使って、更にパフォーマンスの向上
    - MRTを用い、1パスで全てのライティング情報を書き出す

DX9インスタンシングAPI

- これは何?
  - 一回の描画呼び出しで、同一モデルの複数のインスタンスを描画
  - DIP呼び出しを減らし、バッチによるオーバーヘッドを少なくする
- 何が必要?
  - DX 9.0c
  - VS/PS 3.0可能なグラフィクス機器

どうして使うの?

- スピード
  - 現在のゲームで最も一般的な最大の問題は描画呼び出しの回数
- はい、描画呼び出しが悪いことは分かっています
  - でも変換マトリクスの違いで描画呼び出しを分けなければならないのです
- インスタンシングAPIはインスタンスごとの描画ロジックをドライバーとハードウェアに移す
  - D3Dとドライバーでの描画呼び出しによる負荷を減らす
  - ドライバーはインスタンスごとの最低限のステート変更であることを確かめる

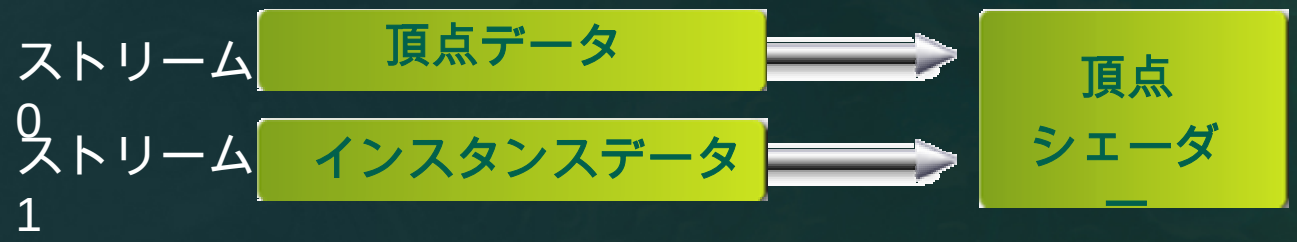
インスタンシング使用例

どんな時にインスタンシングを使うの?

- シーン内に同じモデルのたくさんのインスタンスがある
  - 森の木、パーティクル、スプライト、兵士
- インスタンスごとに移動や変更
  - 静的バッファでは処理できない
- バッチが大きいときはあまり有効ではない
  - 千ポリゴン以上の場合は、描画呼び出しがボトルネックではない
  - インスタンシングを使用する際の固定負荷
    - 余分な頂点属性
    - 余分な頂点シェーダー処理

どうやって動作するの?

- インスタンシングAPIはVertex Stream Frequency Divider (VSF) APIを用いる



- 一次ストリームはひとつのモデルデータ
- 二次ストリーム(複数可)はインスタンスごとのデータ
  このストリームのポインタは一次ストリームが描画されるたびに進む
  - **IDirect3DDevice9::SetStreamSourceFreq**を使用

どうやって動作するの? (2)

- あまりを#0へ、除算結果を#1へ
  - ストリーム0を繰り返す
    - 一回の繰り返しでひとつのインスタンス
  - ストリーム1を割る
    - これでインスタンスの後に一回ずつ進む

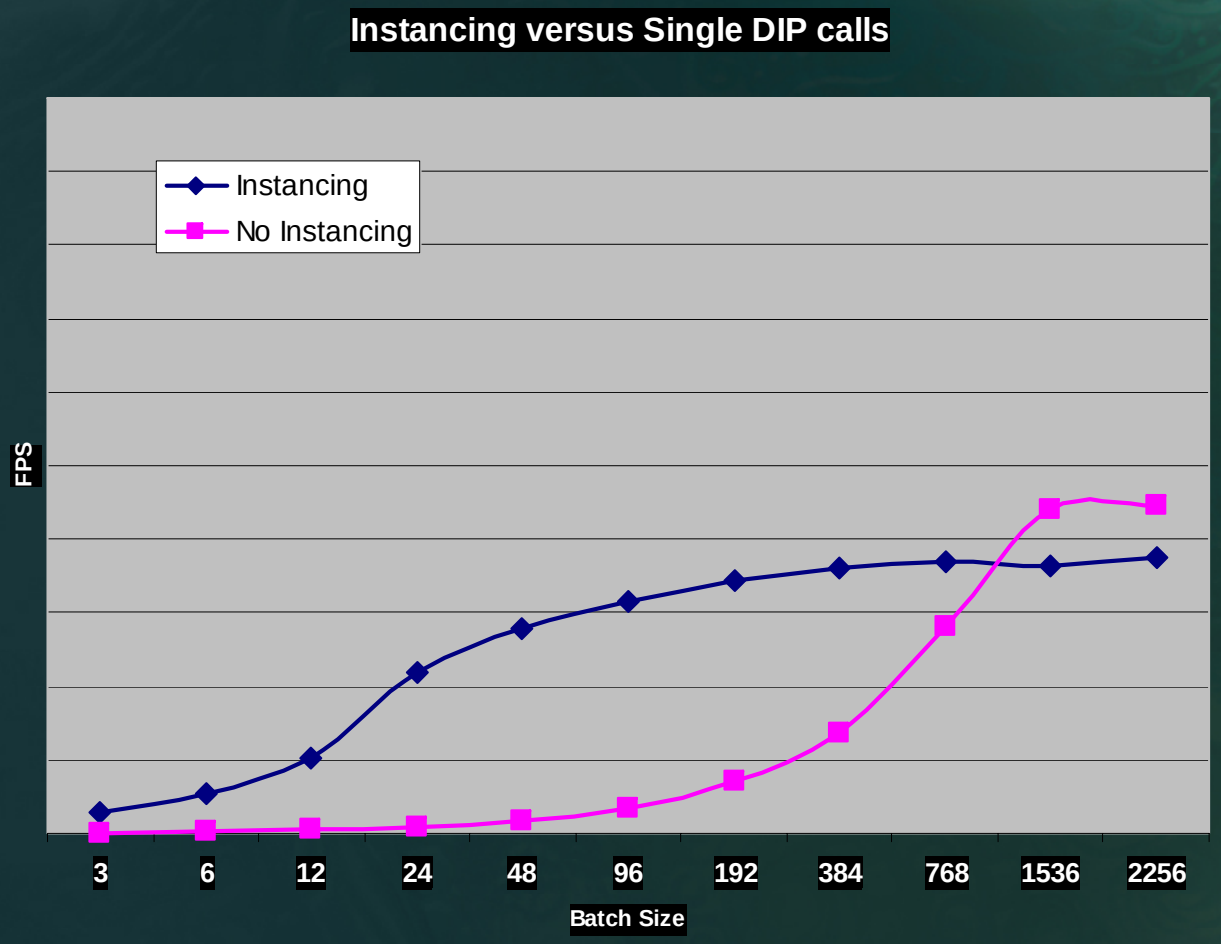
簡単なインスタンシングの例

- 100ポリゴンの木
  - ストリーム0には木のモデルがひとつだけ
  - ストリーム1にはモデルのWVP変換
    - 視界に入っているインスタンスによってはフレームごとに再計算
  - 頂点シェーダー定数ではなく、頂点ストリームから取得した変換マトリクスを使う以外は、頂点シェーダーは普通と同じ
- 一万本の木を描くなら、非常に大きなパフォーマンス向上
  - VBと変換前の頂点を操作することもできるが、簡単ではなくデータの複製が大量に必要

100万ポリゴンのテスト

- メッシュのインスタンスを描画する実際的なテスト
- 横軸はメッシュごとのポリゴン数
- シーンで描画するポリゴン数は100万に固定
  - したがって、メッシュが大きくなれば、インスタンスの数は減少
- シェーダーは簡単なもの
  - 複雑なシェーダーはシーンの挙動を変える
    - 複雑なピクセル・シェーダーはメッシュの大きさによってはボトルネックになりえ、インスタンシングでも常にひとつのDIPと同じ速度になるかもしれない

100万ポリゴンのテスト結果



100万ポリゴンのテスト結果 2

- 小さいバッチでは描画呼出しごとに節約できるので、非常に大きな改善になる
- 頂点ストリームにデータを足すことにより固定の負荷がある
  - 頂点属性読み出しがボトルネック
    - バーテクスのストライドを倍以上にした
  - システム内のほかのボトルネックによっては、これが問題にならないかもしれない
- 他の要素により、最も効率のよい点は移動する(CPUスピード、 GPUスピード、エンジンの負荷など)

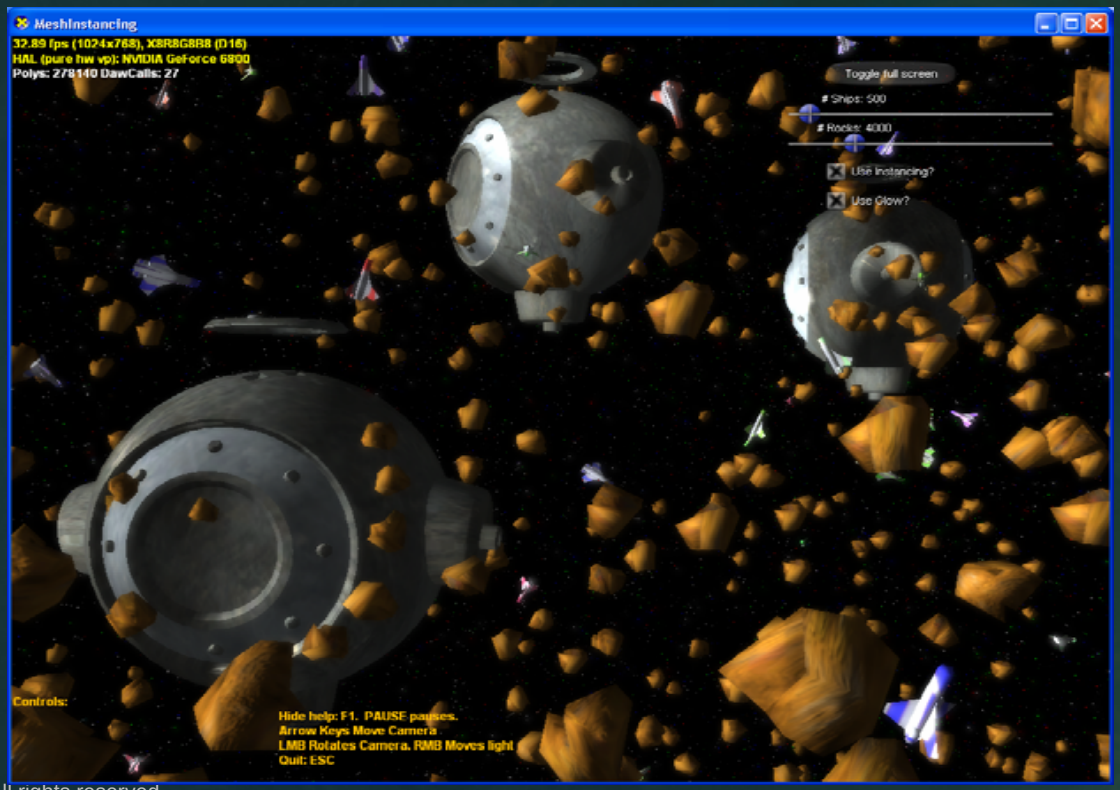
他のアイデア

- VS定数を使用する
  - VS定数メモリ領域を分割してそれぞれのインスタンスで使用
  - インスタンスのストリームにこのデータへのインデックスを入れておく
  - これで頂点シェーダー属性が多すぎる問題を緩和できる
- テクスチャの違うインスタンス
  - 複数のテクスチャを用い、インスタンスのデータでテクスチャ間を補間する
  - テクスチャ・ページを用い、UV座標のオフセットをふたつ目のストリームにいれる

他のアイデア(2)

- 色の着いたインスタンス
  - インスタンスごとの色情報を流し、インスタンスごとに頂点色を調整
- <あなたのアイデア>
  - 描画呼び出しの回数を減らすように。ただしシェーダーのmulや頂点属性などの負荷を忘れないこと
- シーンの描画回数を一回にする必要はない
  - もしよっつのインスタンスごとにひとつにまとめられるのなら、描画呼び出しは4分の１になる

インスタンシングのデモ



- 500以上の宇宙船と4000以上の岩のある宇宙のシーン
- 複雑なライティングと後処理
  - CPUでの衝突判定も行っている
    - 実際これが制限となる
- インスタンシングを使用すると、とても早くなる

さらにインスタンシングのパフォーマンス

- インスタンシングについての考慮点をまとめた文献
  - インスタンシングを使った際のパフォーマンスについてのより完全な議論
  - http://developer.nvidia.com
  - 描画回数を減らすためのその他の方法についても議論
  - たくさんのグラフ :P

質問?

更なる情報?

- http://developer.nvidia.com
- NVIDIA SDK
- bdudash@nvidia.com
  - 英語、または日本語でご質問ください

リファレンス

[Mech01] Radomir Mech, “Hardware-Accelerated Real-Time Rendering of Gaseous Phenomena.”