GPUプログラム:

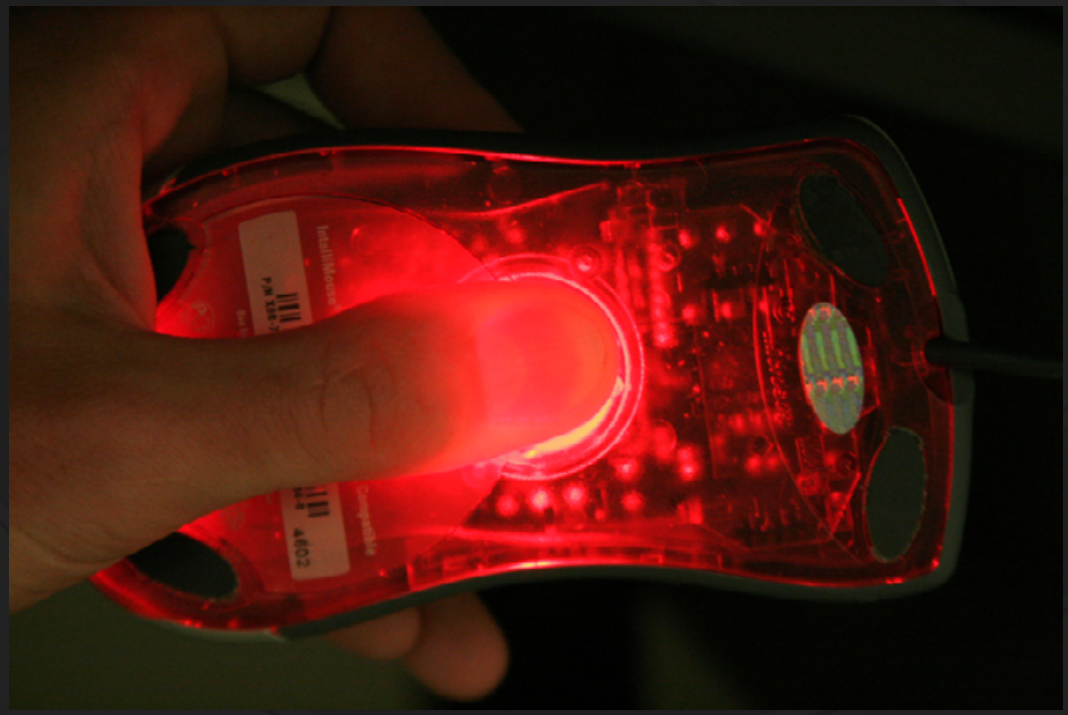
NVIDIAデモのまたさらなる秘密と
次世代特殊効果

Lunaデモ

- 透過光
- ディスプレイスメント・マップ
- 眼球のレイトレース
- スーツのシェーダ

透過光の有効性

- より肉的で有機的な表現
- 強烈な光の表現

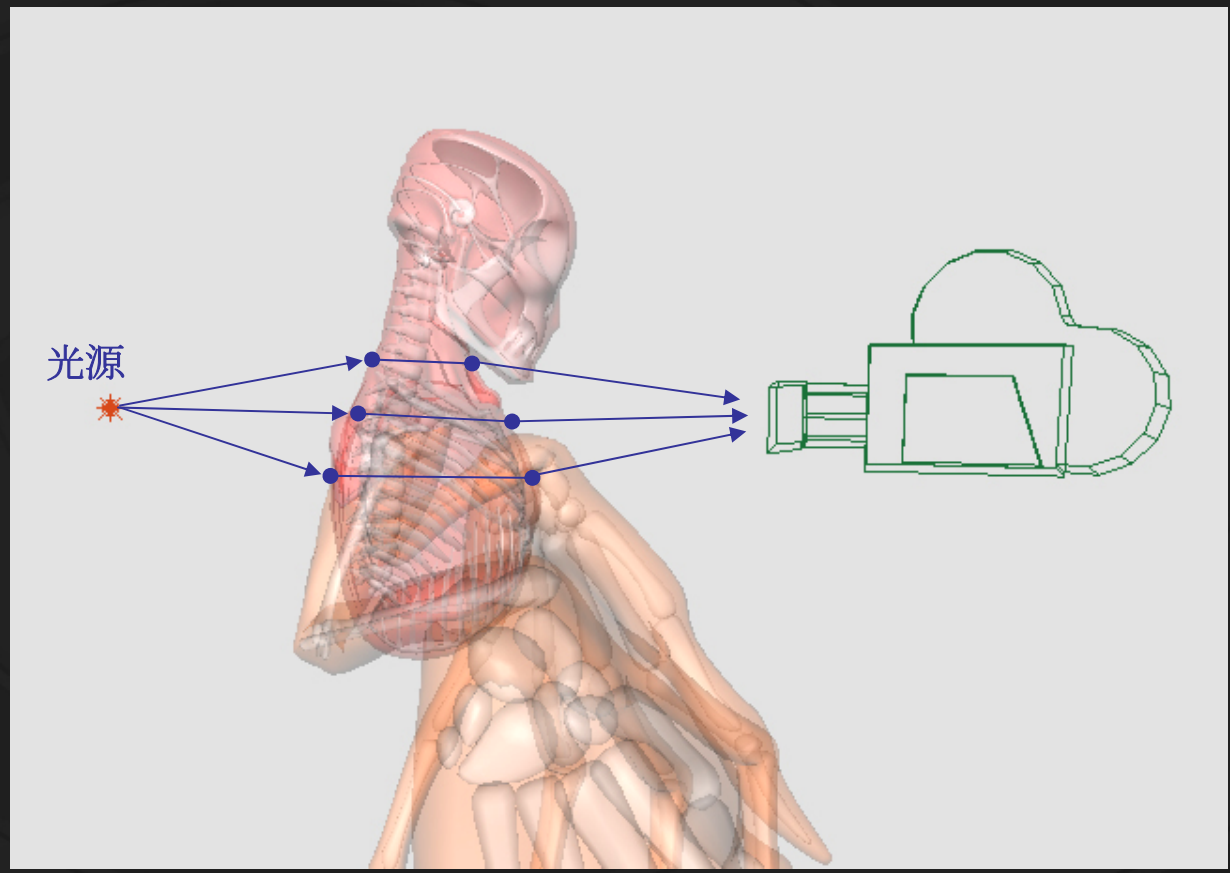


- 純粋にクールである ☺

基本的なアイデア

- キャラクタが明るい光源とカメラの間に存在
- カメラの視点から、キャラクタの厚さを測定
- 厚さによってテクスチャから皮膚に適当な色を選ぶ
- キャラクタとカメラがどれだけ光の影響を受けるか計算
- これで通常色と透過光色を合成

面を通過する光量を概算



厚さの計算

- 背面のピクセル位置を**fp16**バッファに描画
- 前面を描画
  - 背面のピクセル位置を前面のピクセル画面座標から割り出し、取得する
  - 背面と前面の距離を計算
  - テクスチャ座標として使えるように、距離を**[0,1]**の範囲に正規化
- しかし…

厚さの計算

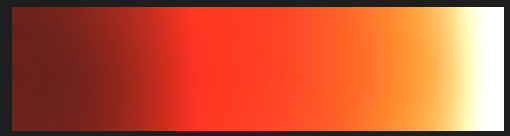
- 計算されたのは本当の厚さではない
- 結果はジャギー
- このままでは使用できない

- 他の方法:
  全ての前面を加算、背面を減算ブレンディングで描画

厚さの計算

- 概算された結果を**11x11**ブラーでなめらかに
  - さらに**2**パス必要
- 最終結果には問題が見えなくなる程度になめらか

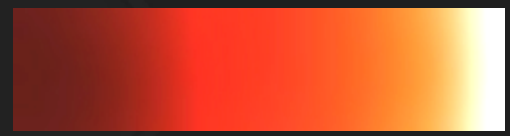
透過光による皮膚の色



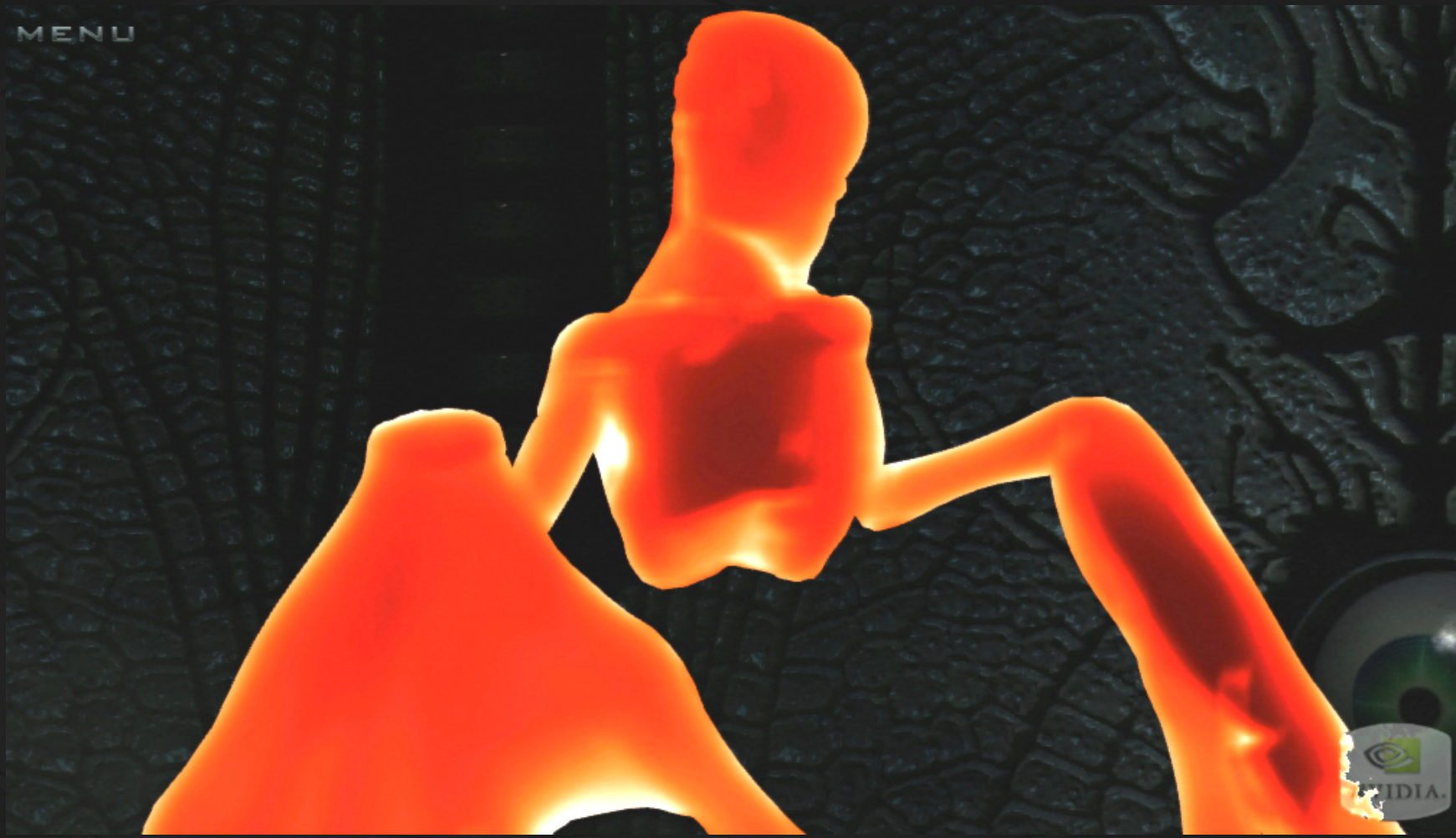
- 厚さから、そこを光が通過した際の皮膚の色を求める
- 正規化された厚さをテクスチャ座標として使用
  - 各物質の密度などを簡単に調節
  - 厚い部分は濃い赤、薄い部分は淡いオレンジに

参考資料

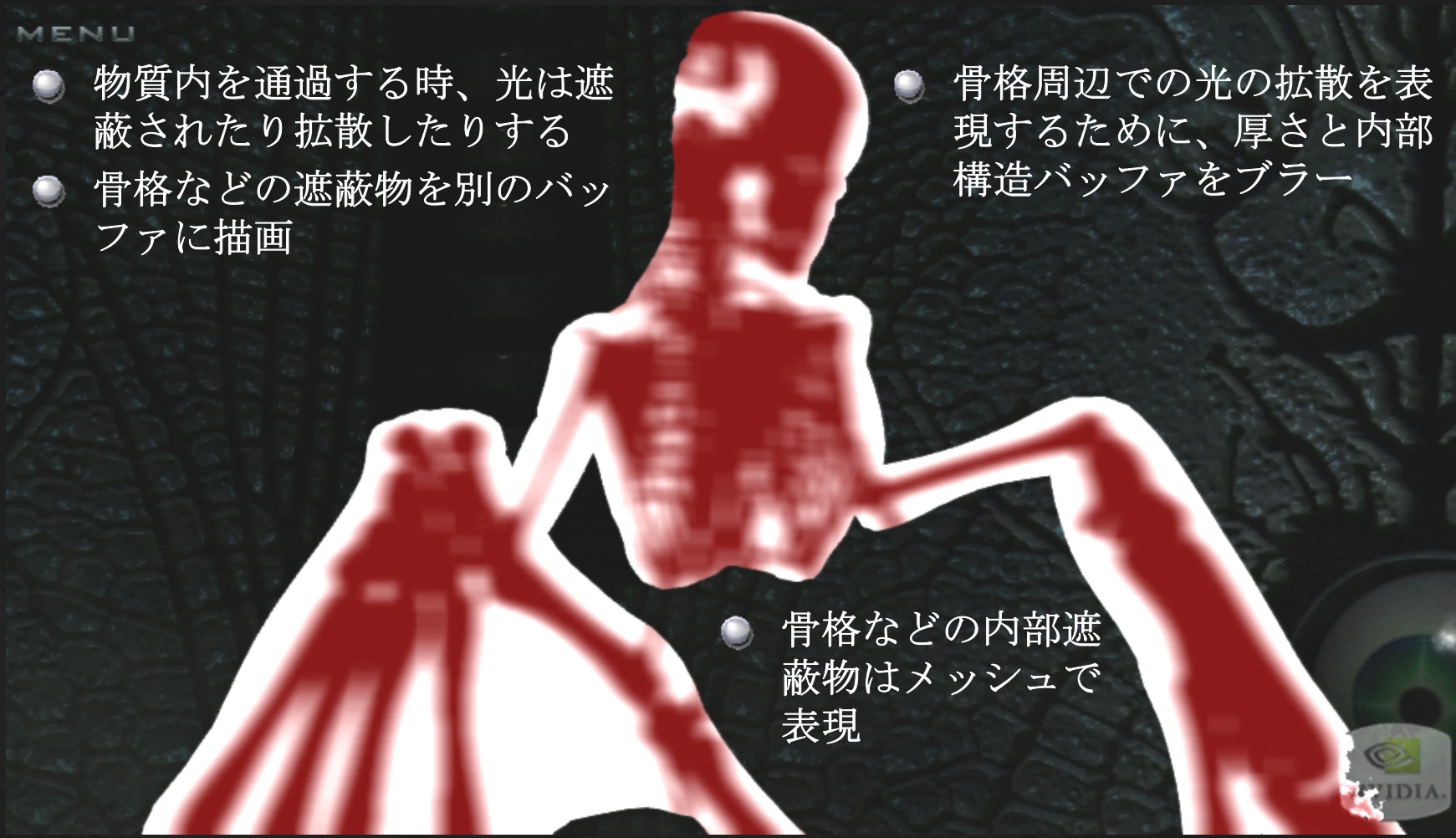
- **Simon GreenによるGPU Gemsの記事『Real-Time Approximations to Subsurface Scattering』**
- **NVIDIA Developer SDK中のテクスチャ読み出しを利用したさまざまな光源効果 (anisotropic lighting等)**
  http://developer.nvidia.com/object/sdk_home.html



最終的な各層の合成

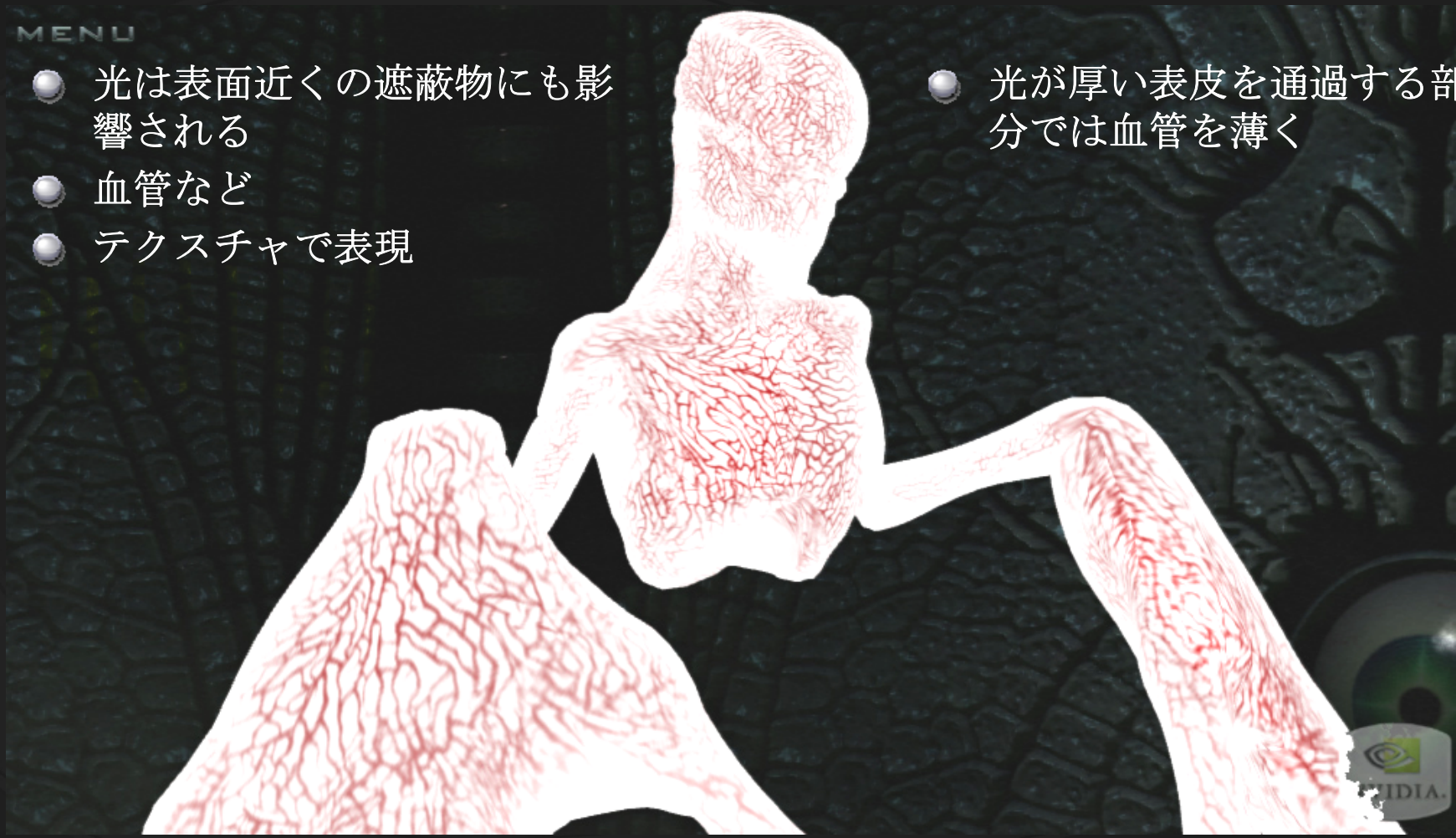


内部遮蔽物


MENU
物質内を通過する時、光は遮蔽されたり拡散したりする
骨格などの遮蔽物を別のバッファに描画
骨格周辺での光の拡散を表現するために、厚さと内部構造バッファをブラー
骨格などの内部遮蔽物はメッシュで表現

表面の遮蔽物

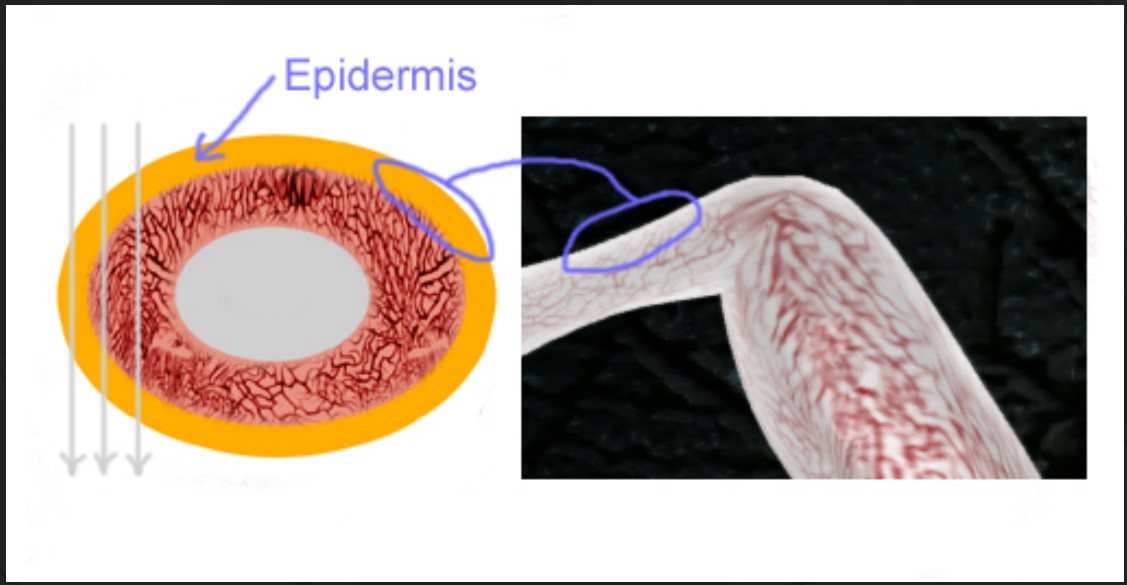
- 光は表面近くの遮蔽物にも影響される
- 血管など
- テクスチャで表現



- 光が厚い表皮を通過する部分では血管を薄く

血管の可視性

- 光が厚い表皮を通過する部分では、血管は見えない
- 血管層を調節



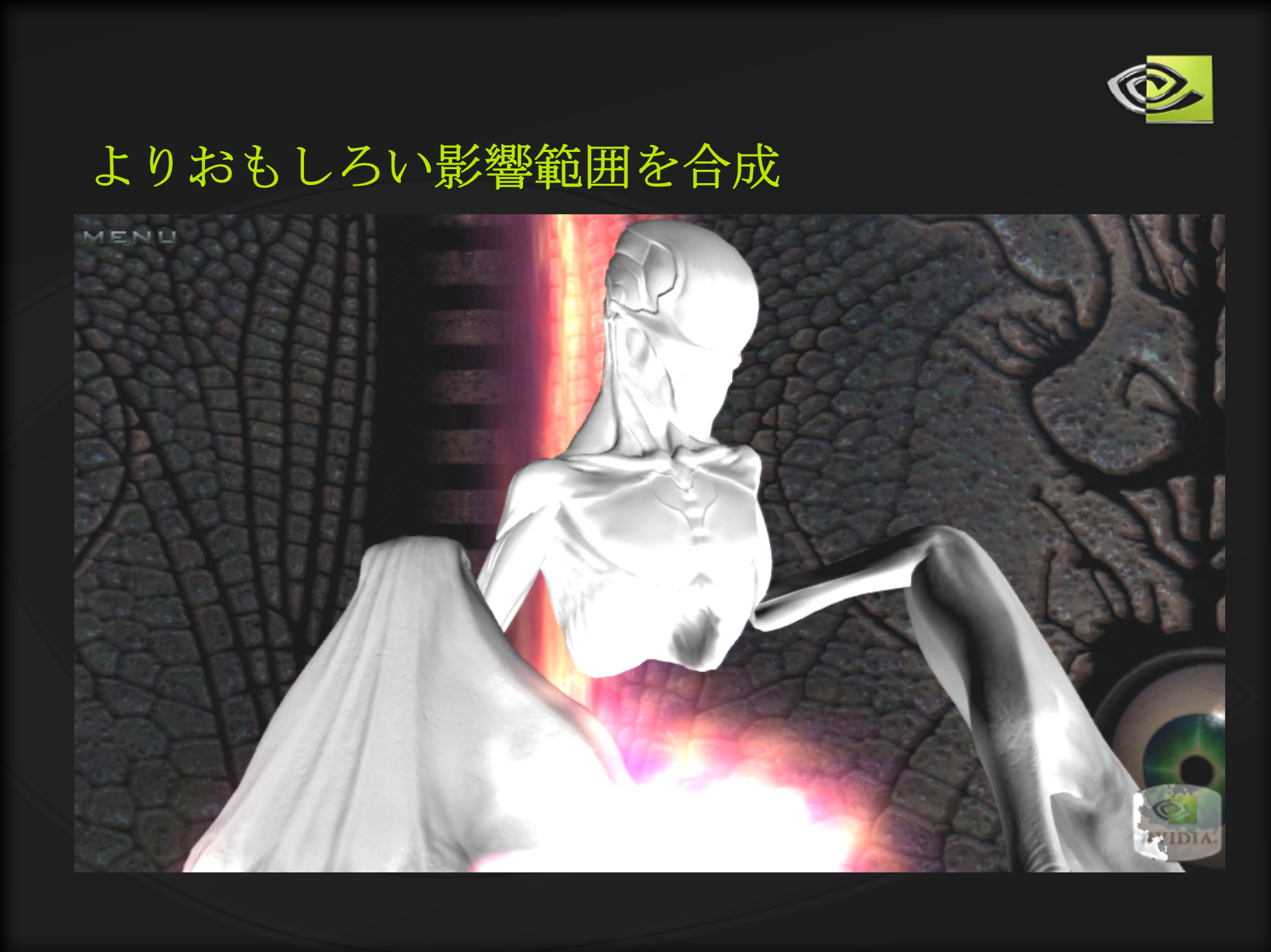
影響範囲

- 光源、物体、カメラの位置で透過光効果の影響を調整
  - **pow(dot(V, L), n)**による近似
  - **V –** フラグメントからカメラへのベクタ
  - **L –** 光源からフラグメントへのベクタ
  - **n –** 調整可能なパラメタ
- デモでの光の柱の場合、直線と平面の接点で**L**を近似
- これによって…

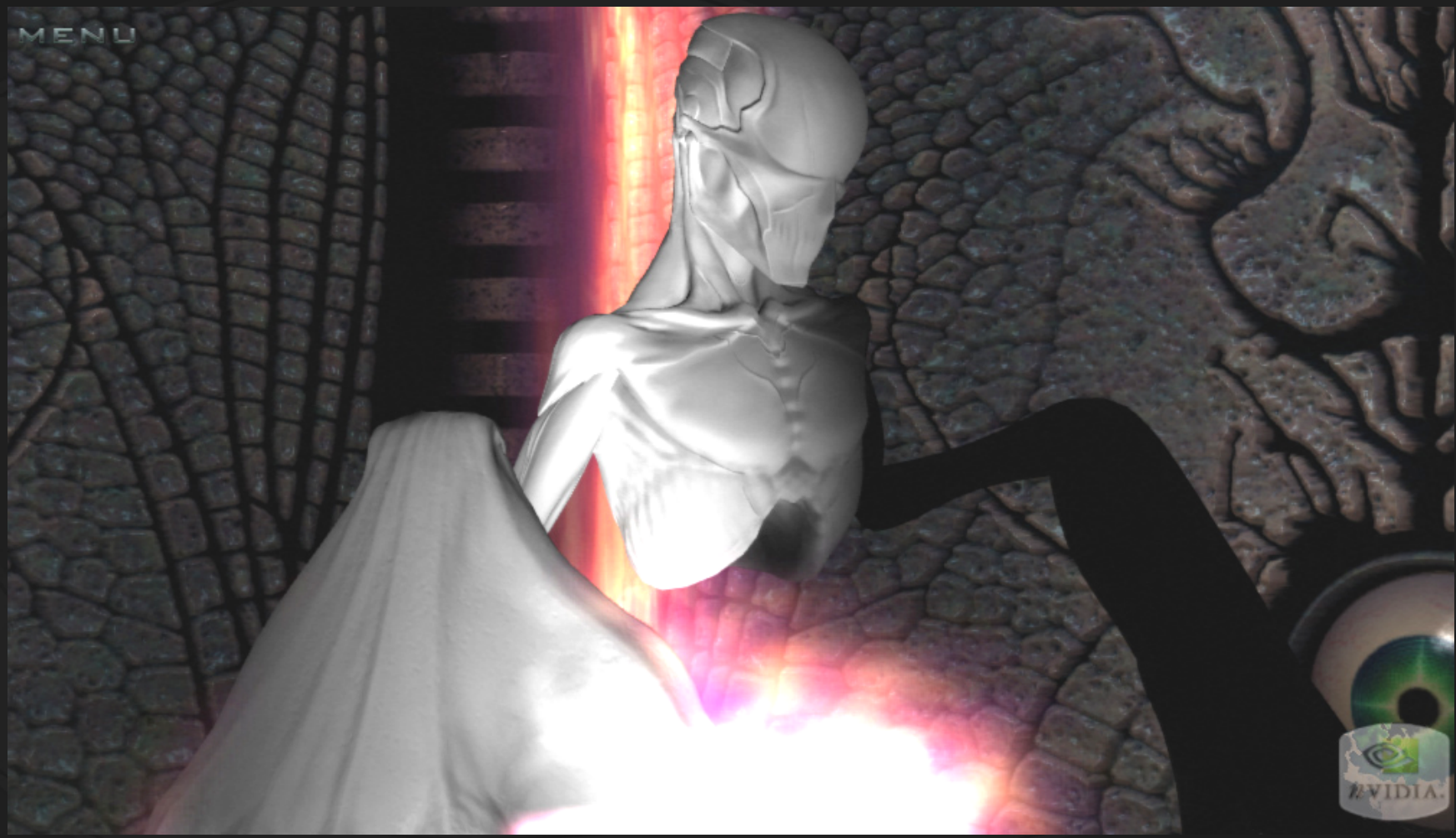
影響範囲

- これだけでは十分ではない
- 光源の向こう側でも半透明になる
- **dot (N, L)**で近似

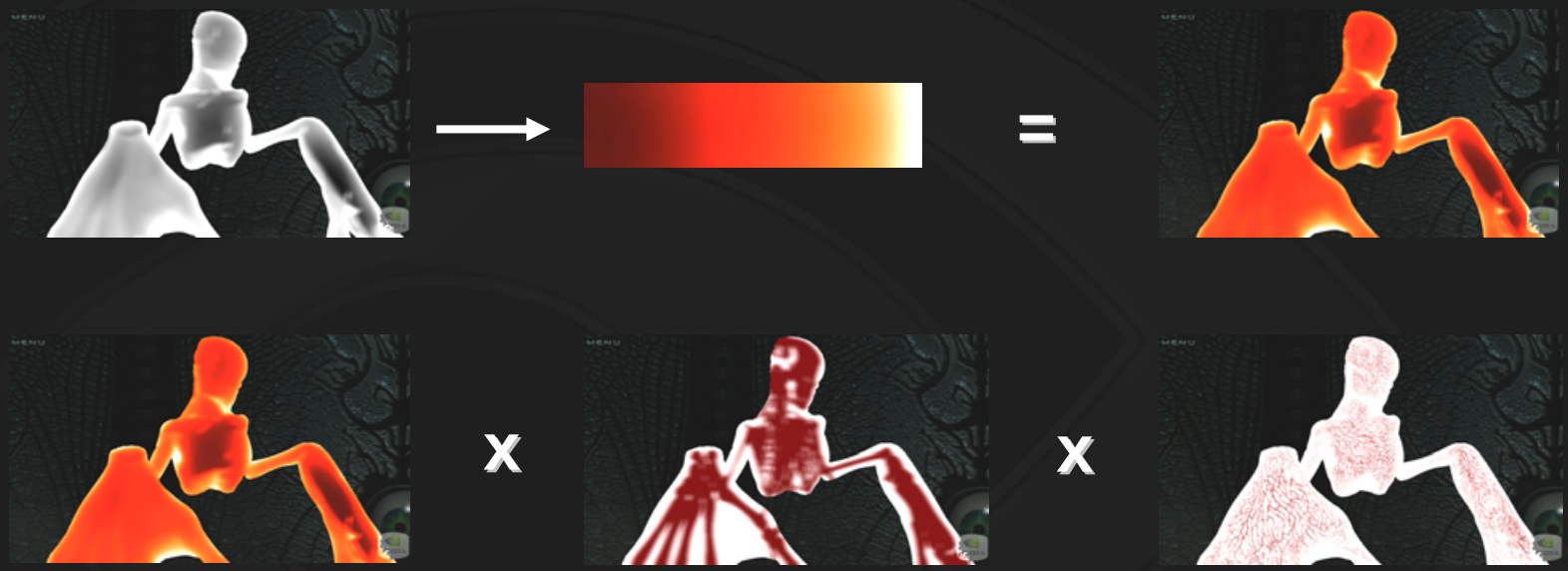
- **dot (N, L)**をわずかにずらした範囲にマップしなおして急な変化を防ぎ、部分的な透過光を表現
- これによって…


よりおもしろい影響範囲を合成
MENU

合成された透過光色と元の表面色の補間

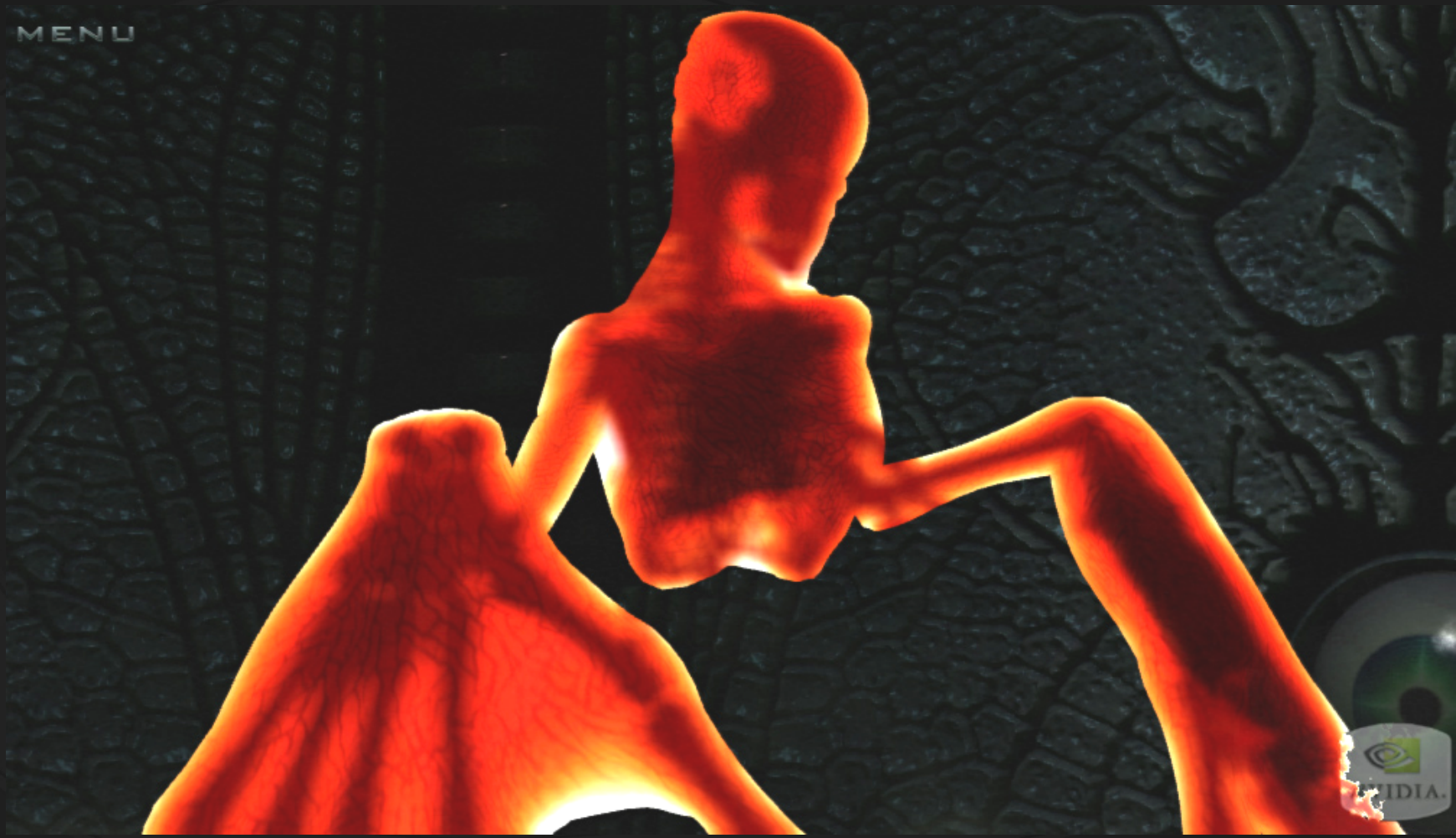


半透明色を算出

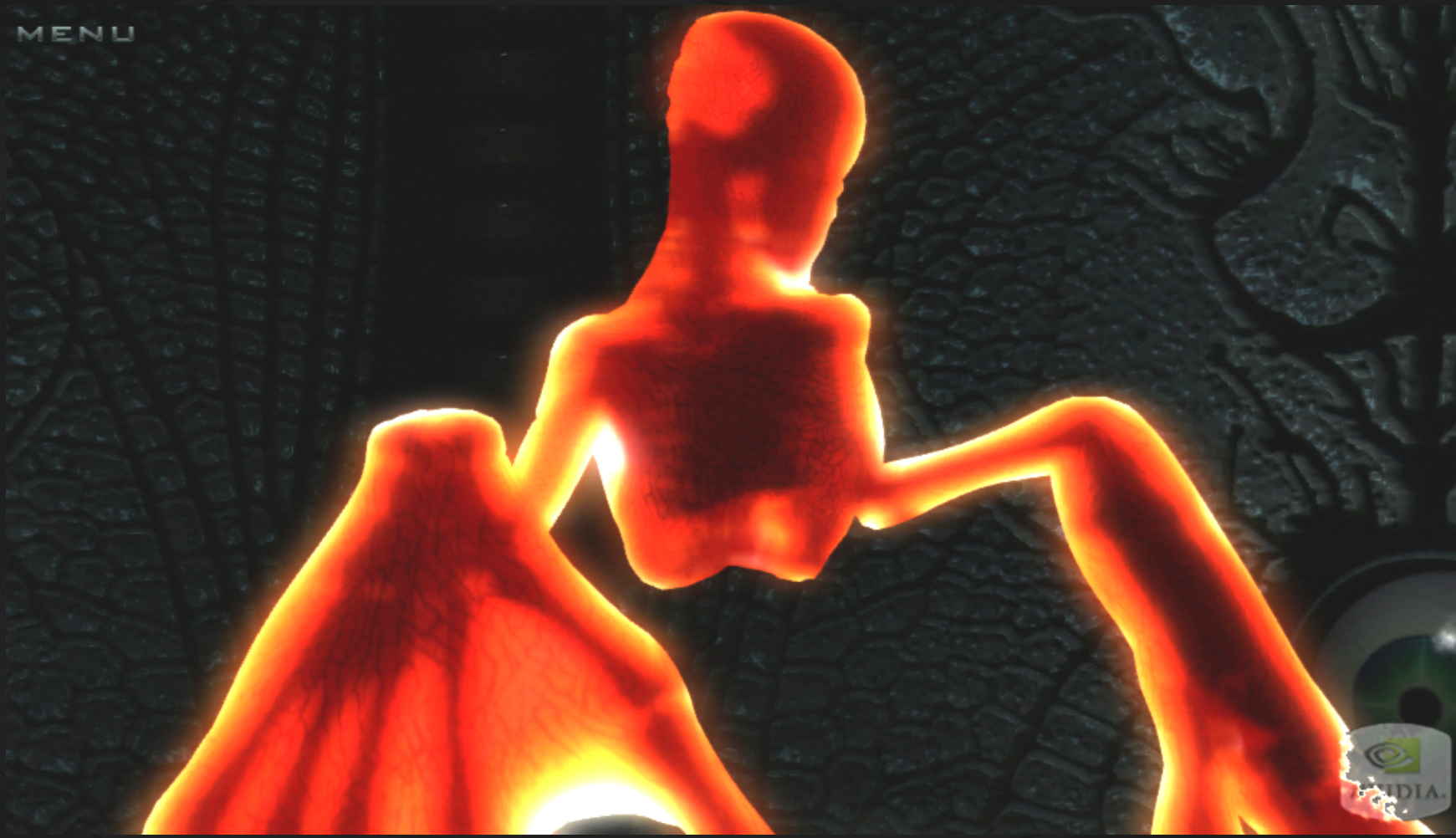


- これによって…

ディフューズには明るい部分の拡散光を加算



通常色との合成



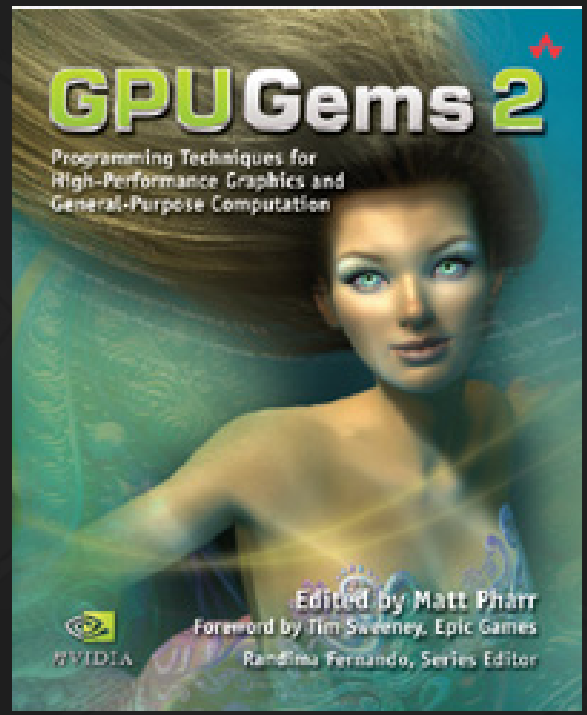
透過光無しの色

最終結果

遮蔽を考慮したディスプレイスメント・マップ

ディスタンス・マップの効率的な計算方法

- セクション 8.4
- GPU Gems 2 CDのSource/tools
- Danielsson, Per-Erik. 1980. “Euclidean Distance Mapping.” *Computer Graphics and Image Processing* 14, pp. 227–248.



パララックス・マップ – 既にゲームで実用化

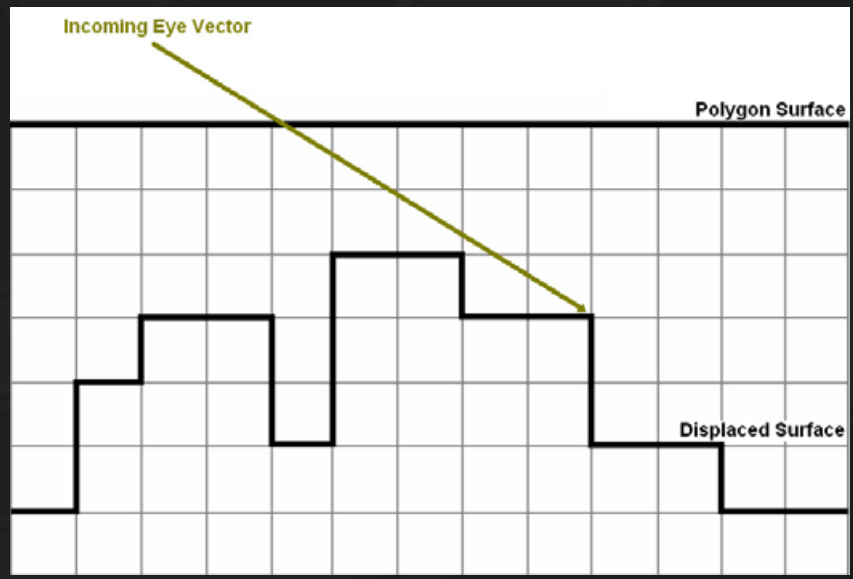
- パララックス・マップ
  - **T. Kaneko et al. “Detailed Shape Representation with Parallax Mapping.” In *Proceedings of the ICAT 2001 (The 11th International Conference on Artificial Reality and Telexistence)*, Tokyo, Dec. 2001.**
- なめらかな変化の場合に有効
  - 遮蔽なし
  - 大きな起伏なし
  - 高周波要素なし

遮蔽を考慮したディスプレイスメント・マップ

- 自分自身の遮蔽を考慮
- 起伏の激しい表面に有効
  - 深い彫りの入った壁
  - ブロックや石畳
  - 格子

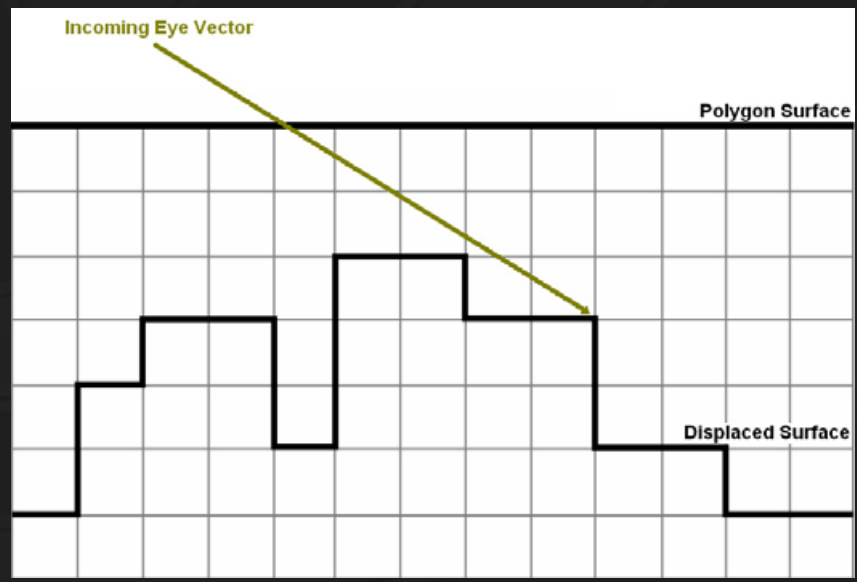
レイキャストでのリアルタイムな高低差表現:

- 物体の表面から始める
- 高低差をつけられた表面まで視線ベクタを追う



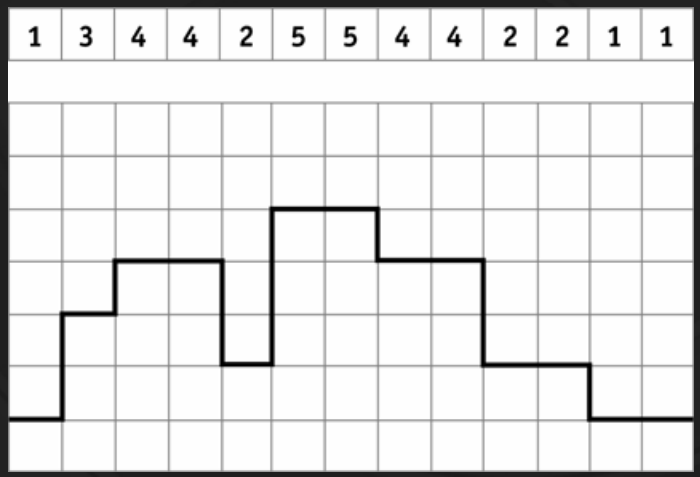
考え方: 3Dテクスチャを見ていく

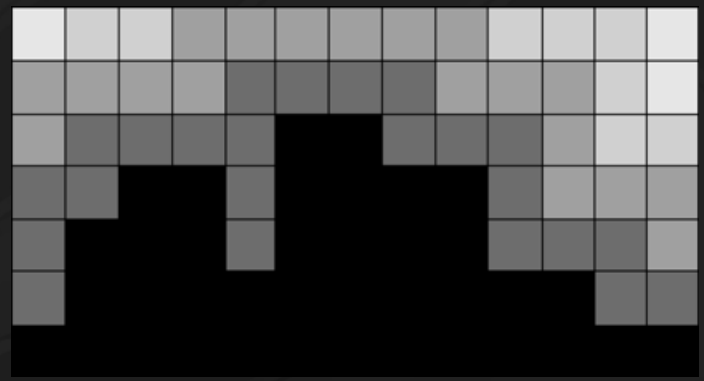
- 3Dテクスチャを作成
  - 何も無いところに『1』、面があるところに『0』
- フラグメント・シェーダ
  - TanEyeVec, TexCoordIter (U, V, 1.0)
  - TanEyeVecを伸縮した値でTexCoordIterを少しずつ増加



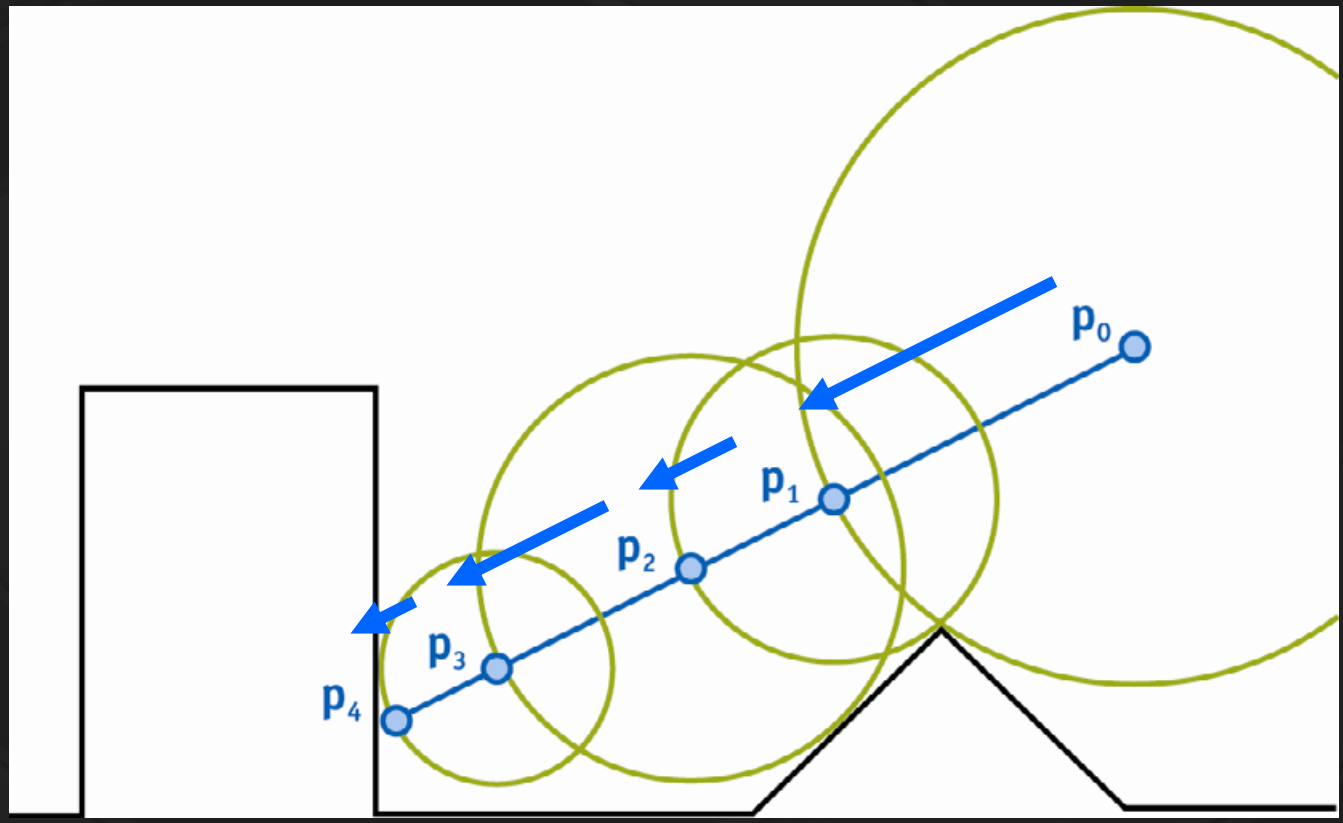
考え方: 改善は可能?

- 大きなステップ: 面を飛び越えてしまう
- 小さなステップ: 実行速度に悪影響
- 面の最も近い部分までの距離を保存しておけば?





不定期な更新: 球トレース



遮蔽物を考慮した高低差

頂点シェーダによって

- **UV**テクスチャ座標**(texCoord)**
- タンジェント空間での視線ベクタ**(tanEyeVec)**

```
tanEyeVec.x = dot(worldEyeVec, worldTangent);
tanEyeVec.y = dot(worldEyeVec, worldBinormal);
tanEyeVec.z = dot(worldEyeVec, worldNormal);
tanEyeVec   = normalize(tanEyeVec);
```

- 視線高低差ベクタ**(displaceEyeVec)**

```
displaceEyeVec = tanEyeVec *
                 float3(1.0, 1.0, 1/bumpDepth);
```

遮蔽物を考慮した高低差

フラグメント・シェーダで高低差付きの**UV**を計算

```
float3 texCoord = float3(v2f.texCoord.xy, 1.0);
float3 displaceEyeVec = normalize(v2f.displaceEyeVec);

// March the ray (NUM_ITERATIONS = 16)
for (int i = 0; i < NUM_ITERATIONS; i++){
  float distance  = f1tex3D(distanceTex, texCoord);
  texCoord += distance * displaceEyeVec;
}

// texCoord.xy is now our displaced UV
```

高低差付きの**UV**でテクスチャを取得

[色、スペキュラ、透明度、反射率、屈折率…]

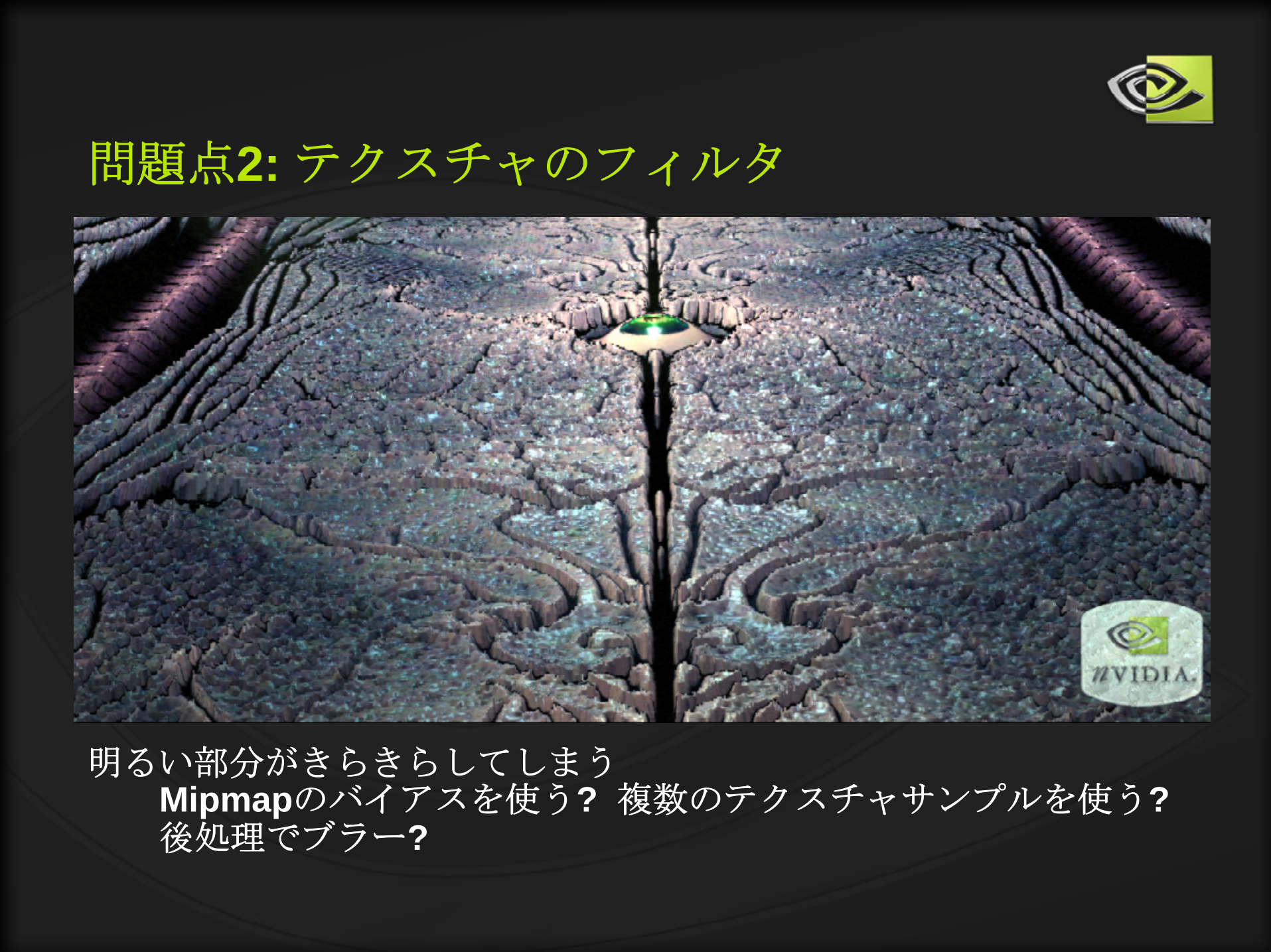
パフォーマンス

- 毎回の計算**{tex; mad;}**
  - **GeForce FX, 6と7シリーズでは1サイクル**
- 繰り返し回数は以下に依存
  - **3D**テクスチャの解像度
  - データのなめらかさ
  - テストの結果**16**回で十分だった
- 簡単な光源処理でのパフォーマンス**:**
  - **GeForce 6800 GTで90M pixels/s**
  - **GeForce 7800 GTXで180M pixels/s**
- やはり単純なパララックス・マップの方が速い
  - 遮蔽が無いときには使用すべき

問題点1: テクスチャの拡張

法線に対してUVが適用された

3Dマテリアルやノイズを使う?



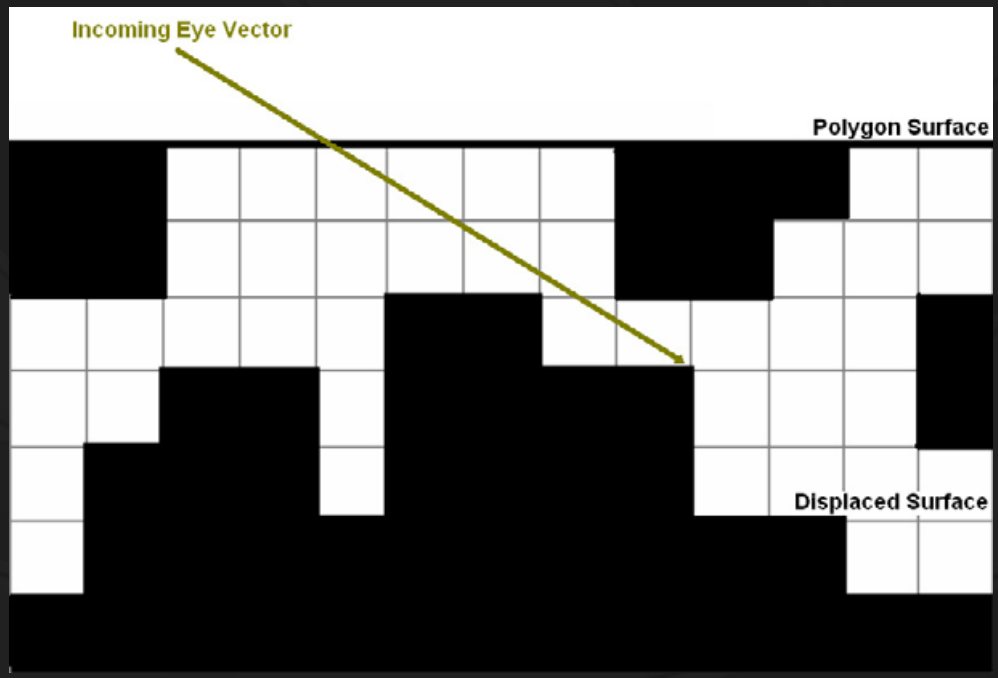
問題点2: テクスチャのフィルタ

明るい部分がきらきらしてしまう

Mipmapのバイアスを使う**?** 複数のテクスチャサンプルを使う**?**
後処理でブラー**?**

将来の作業

- 問題点1: 横方向への拡張を改善
- 問題点2: テクスチャのフィルタを改善
- 曲率とピクセル・キルでシルエットを修正
- 高さフィールドに限らず有効!



現在のGPUによるリアルタイムなレイトレースの現実的な適用

難点

- 現実感のある3D眼球を作る
  - デモ中で、眼球が画面いっぱいになることもある
- 解決しなければならない問題:
  - 角膜を通した透過光
  - 濡れて光る眼球の表面
  - 透明度: 角膜の横を通過し、シーン内の物体に当たる光

解決法の概要

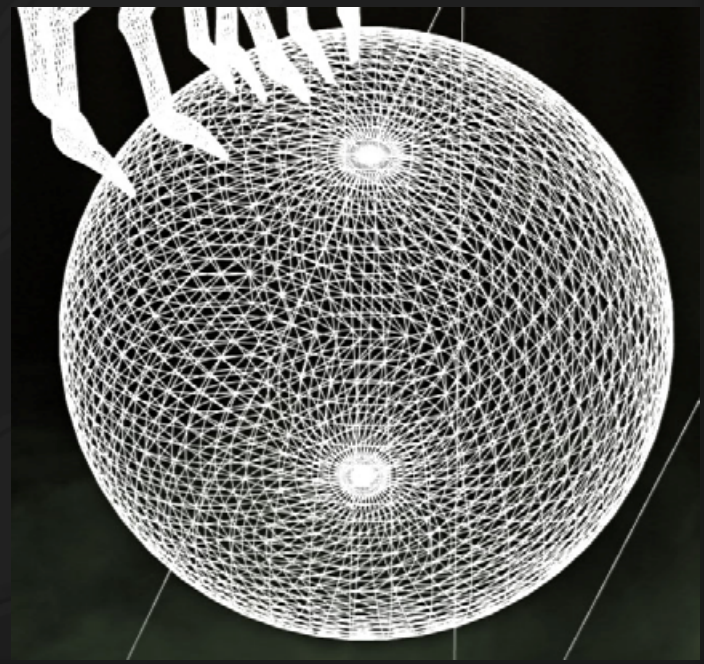
- レイトレースで透過光を再現
- 計算を単純化するためにオブジェクト空間で光源処理
- 虹彩/瞳孔は計算処理で決定
- シェーダ・モデル3.0の分岐を使い部分によって違うシェーダを適用し、さらに境界ではふたつをブレンド
- 仮定:
  眼球は球体、コースティクスは無視

レイトレースされるメッシュの処理

- 計算処理の色彩メッシュをレイトレースで描画するには、新しい技術が必要:
  - 影マップを正しく行うために、新しい影座標を計算
- レイトレースと白目の部分には境界がある
  - 境界をなめらかにできる?

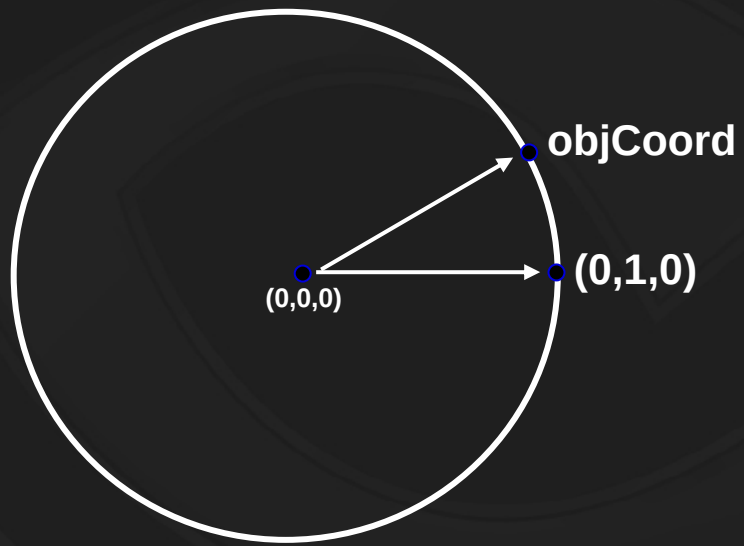
なめらかな球体メッシュから

- 単純な球体メッシュ(50x50)を使用、眼球が実際は歪むことは無視
- 他にはメッシュは使用しない
- 虹彩はレイトレースで計算的に定義

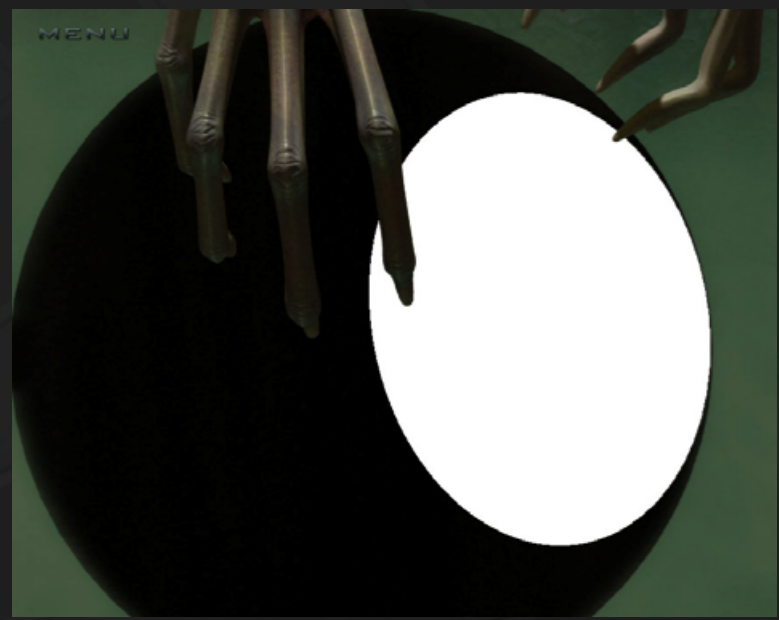


計算的に虹彩領域を決定

- 頂点シェーダはオブジェクト空間の座標をわたす
- フラグメント・シェーダでの内積により境界のなめらかな虹彩を作る

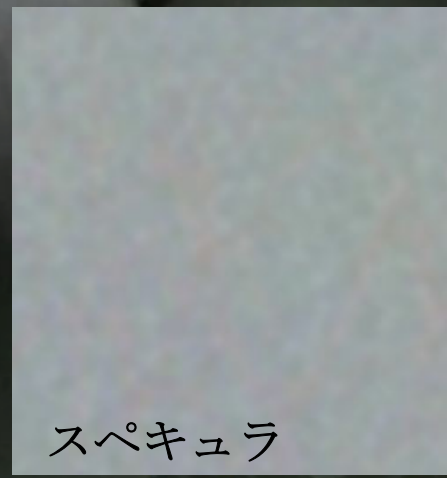


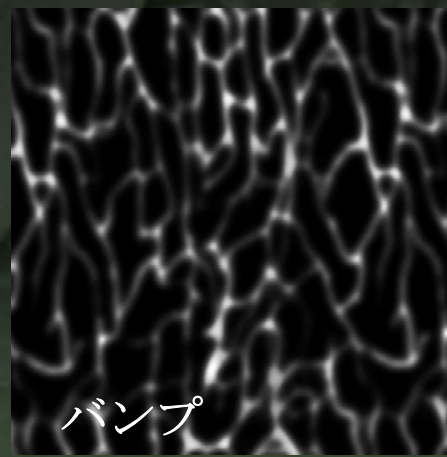
dot(objCoord, (0, 1, 0)) > 0.805?



白目部分

- スポットライトひとつと点光源ふたつ、さらにアンビエント
- **16**サンプルの均質**4x4**拡散シャドウ読み出しで、上からのスポットライトの落とす影を作る
- わずかなバンプと血管マップ
- **Soft-wrap**のディフューズ…



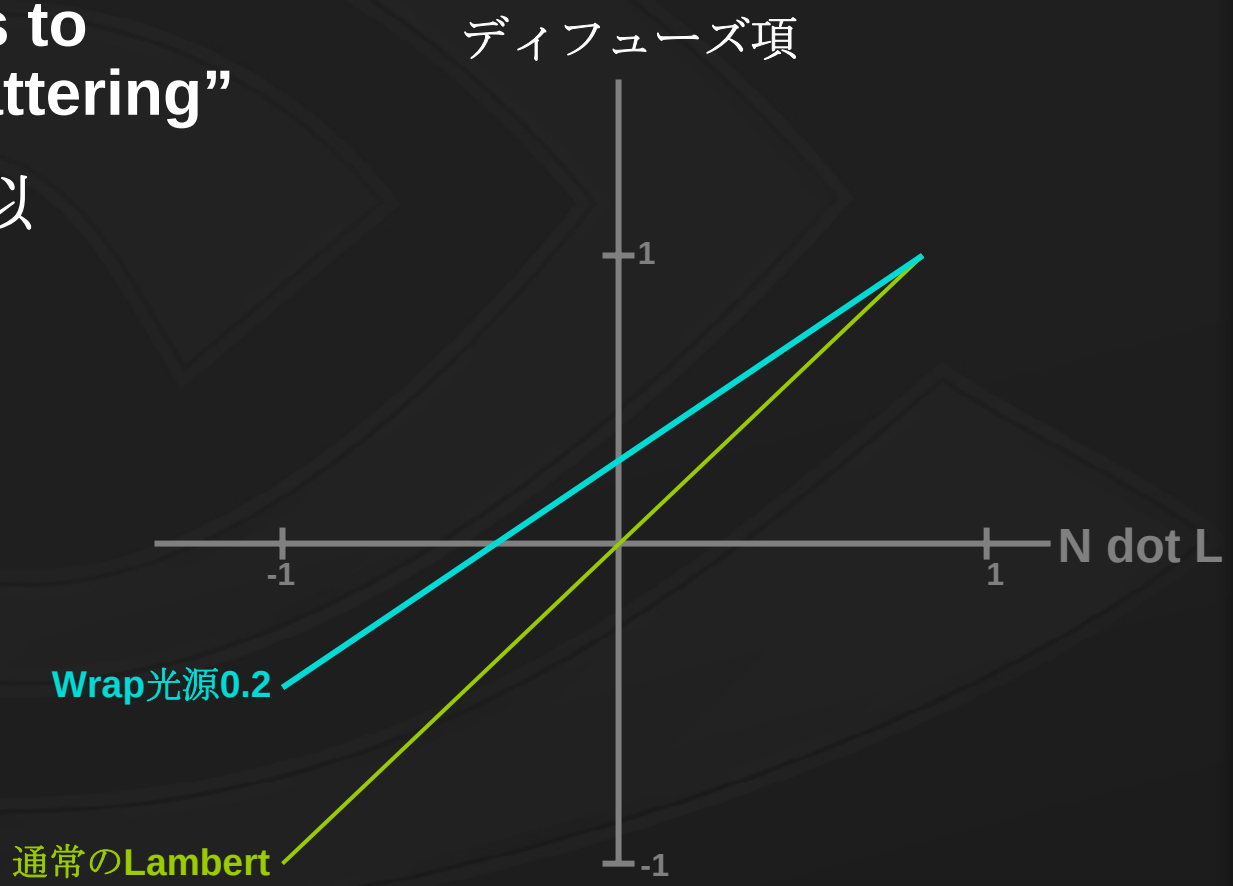


Soft wrapディフューズ光源処理

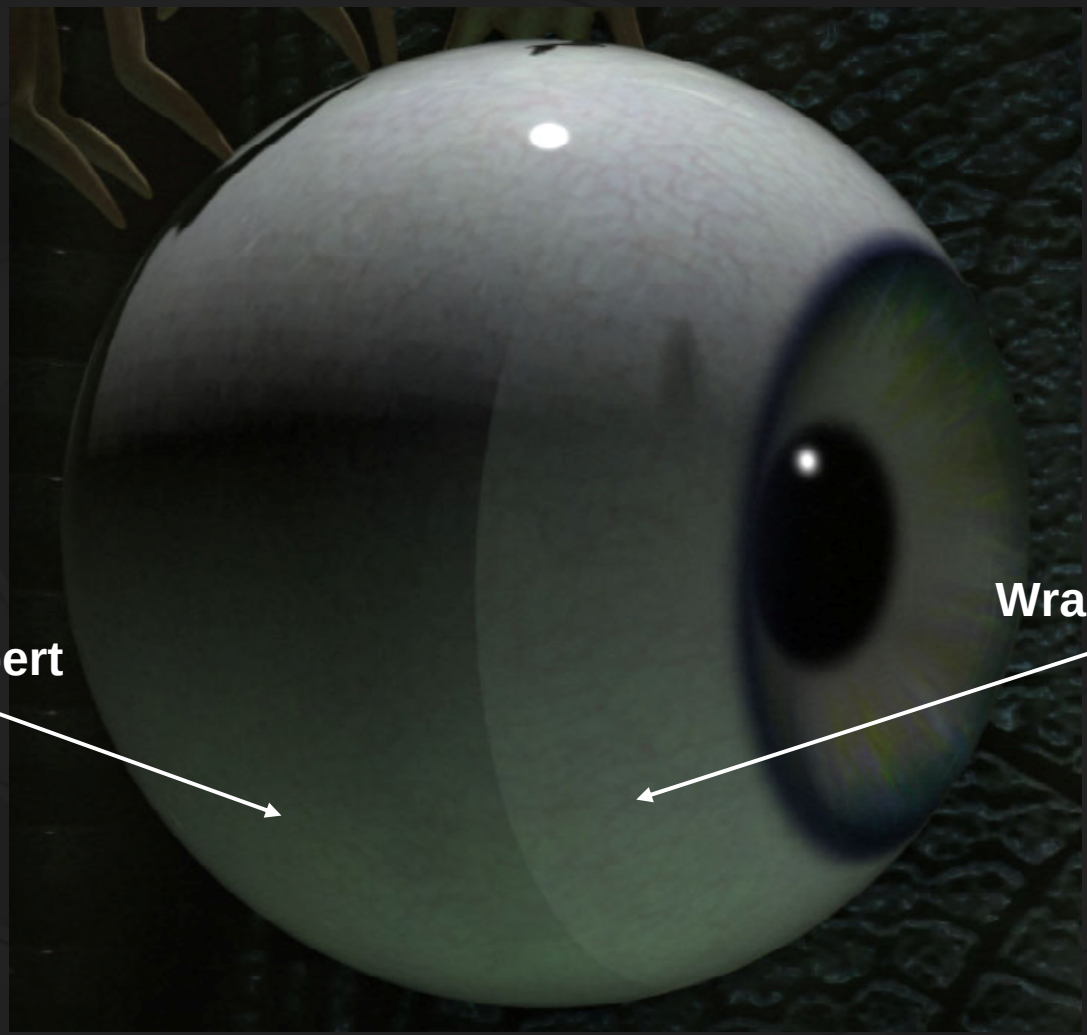
- GPU Gems (Simon Green):

  “Real-Time Approximations to Subsurface Scattering”

- 表面下拡散の近似



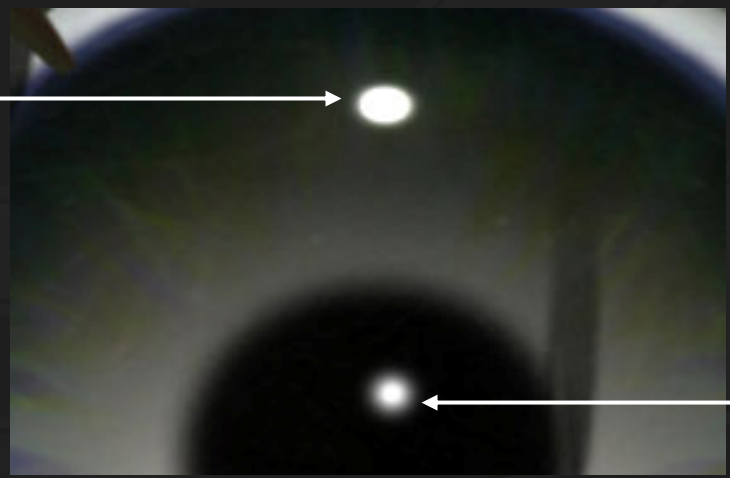
Wrapディフューズの比較



環境反射光と角膜層

- 眼球全体の表面に透明でなめらか、反射性のある角膜層を表現
- ふたつの項目:
  - **1)** フレネル項により減衰する環境キューブマップからの反射
  - **2)** 高い階乗指数を用いた、とても明るいスペキュラ

Exp = 4000、
スペキュラ色を**6.0**倍

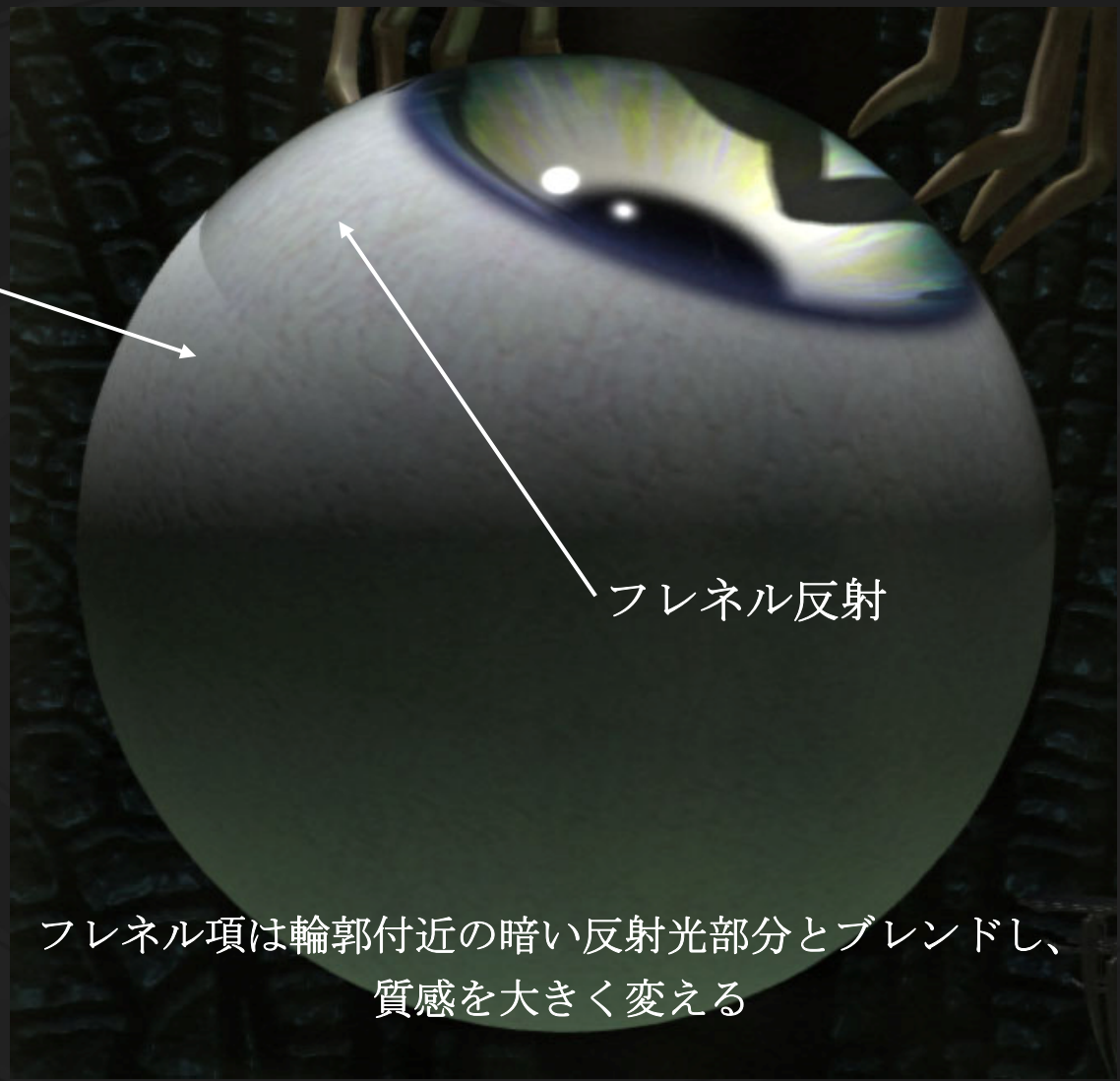


Exp = 4000、
スペキュラ色 **= (1,1,1).**

フレネル反射と角膜ハイライトのみ

加算される反射光はわずか、しかし…

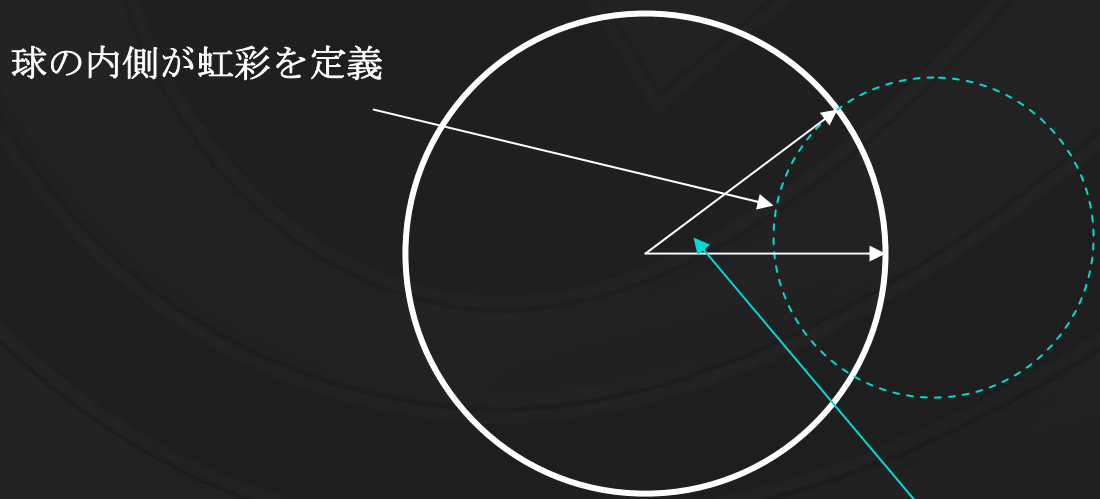
フレネルの有無



フレネル項は輪郭付近の暗い反射光部分とブレンドし、
質感を大きく変える

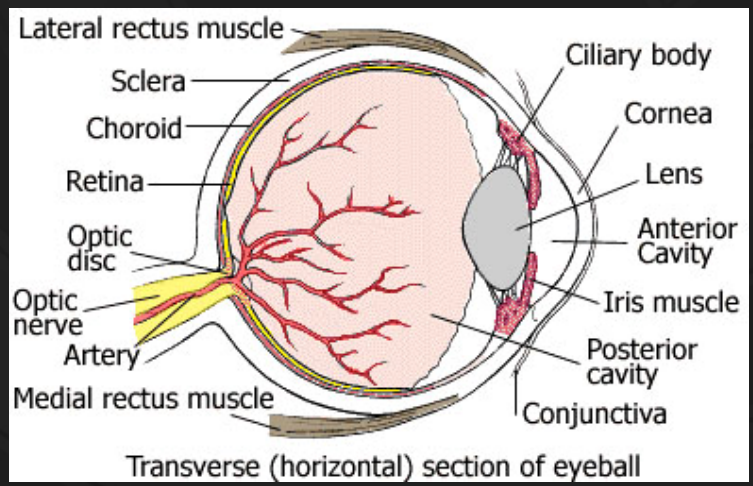
虹彩を作る

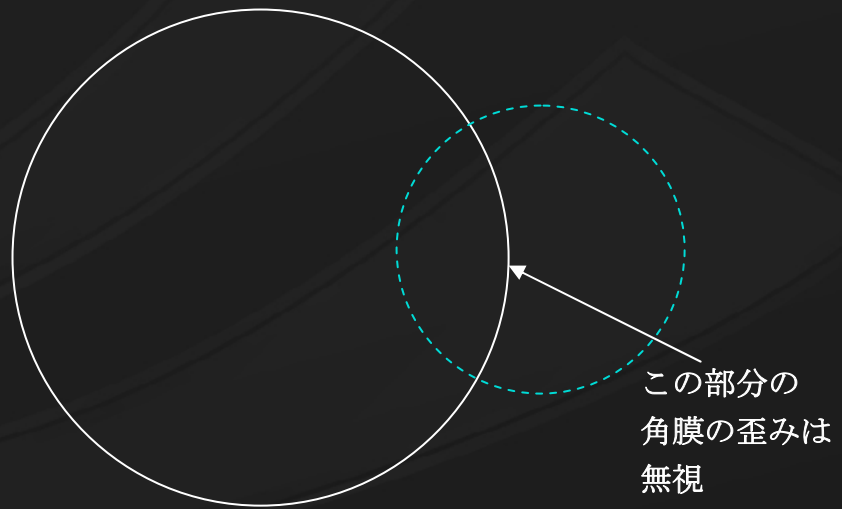
- 虹彩メッシュは球:
  - レイトレースが単純
  - 位置から法線を算出
  - ふたつめの球は内積の結果が正しくなるように配置
  - 正しい形ではないが、光源処理のために球体が必要...



しかし角膜と虹彩は球ではない

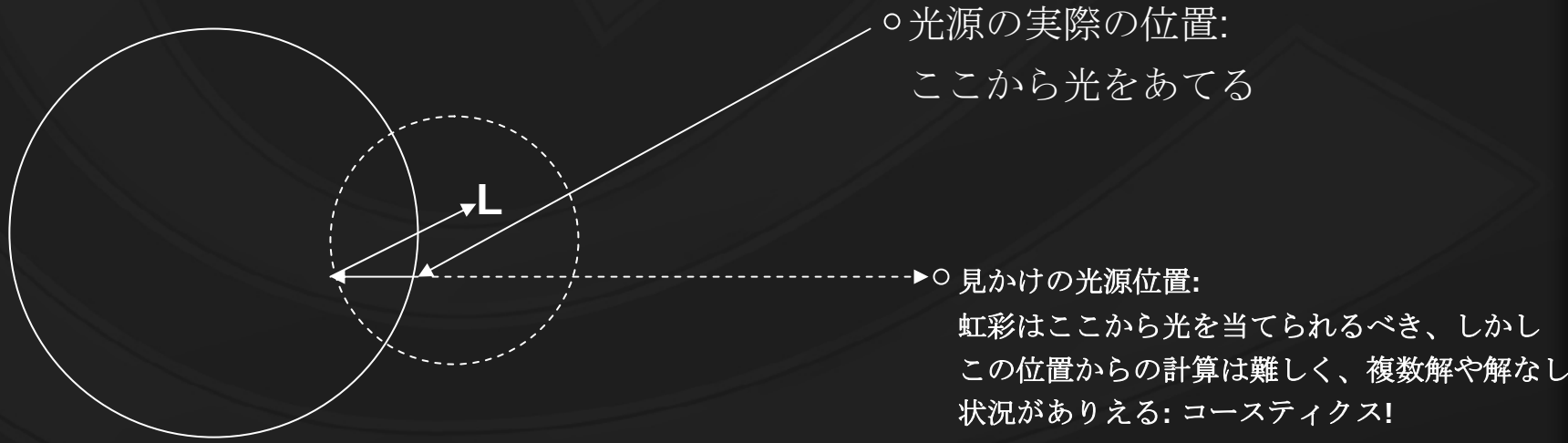
- 角膜はわずかに歪み、虹彩は基本的に平面
- この図では歪みが強調されており、実際は球体の眼球でも変には見えない
- 光源処理のために少しでも球面な虹彩が必要…

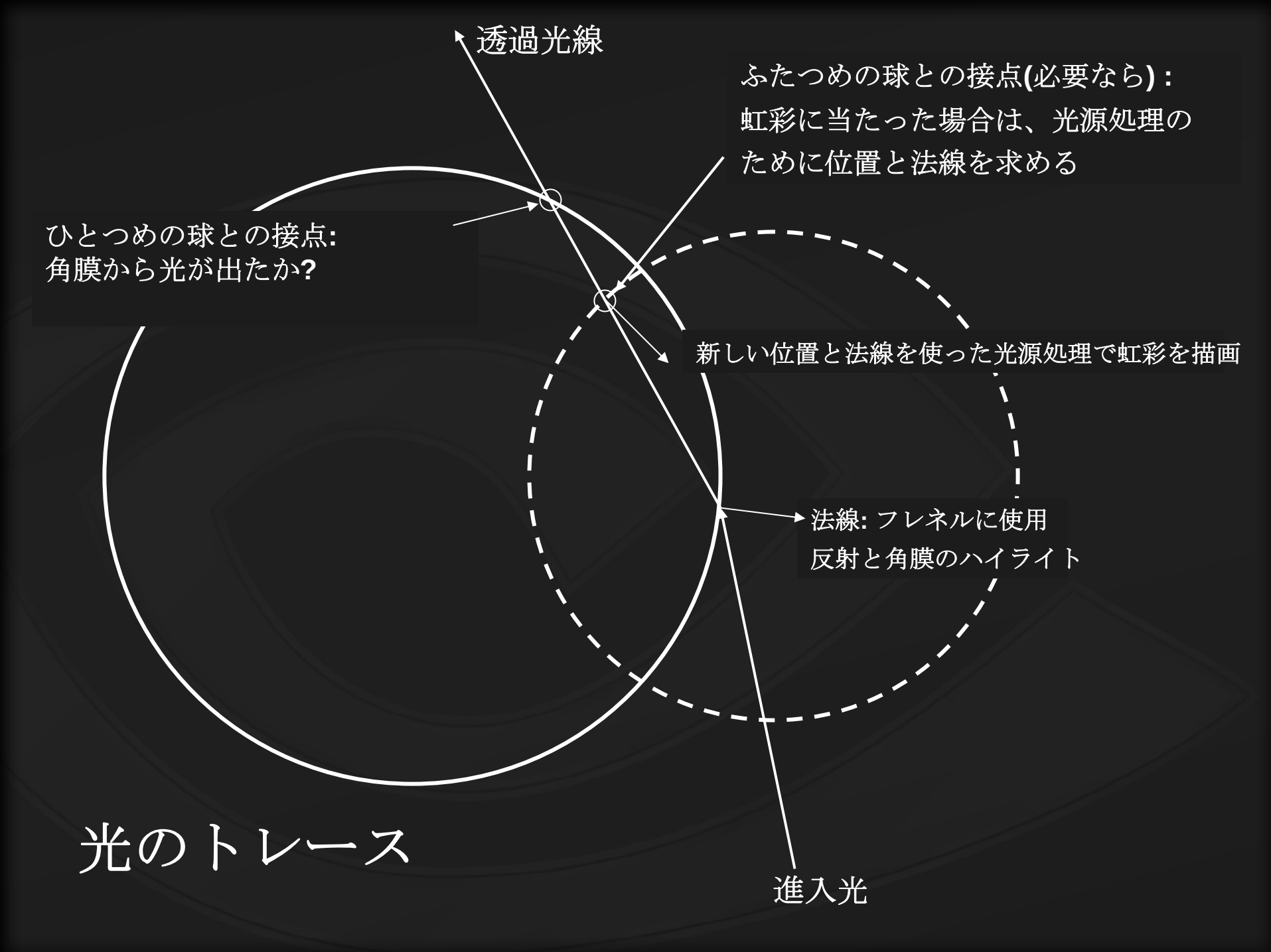




虹彩の光源処理

- 現実では虹彩は平面、しかし透過光の屈折により結果的には衛星放送アンテナのような形になる
- コースティクスは難しいので正しく光源処理されるわけではない
- しかし、球体で表現しコースティクスを無視することにより、似たような表現が可能



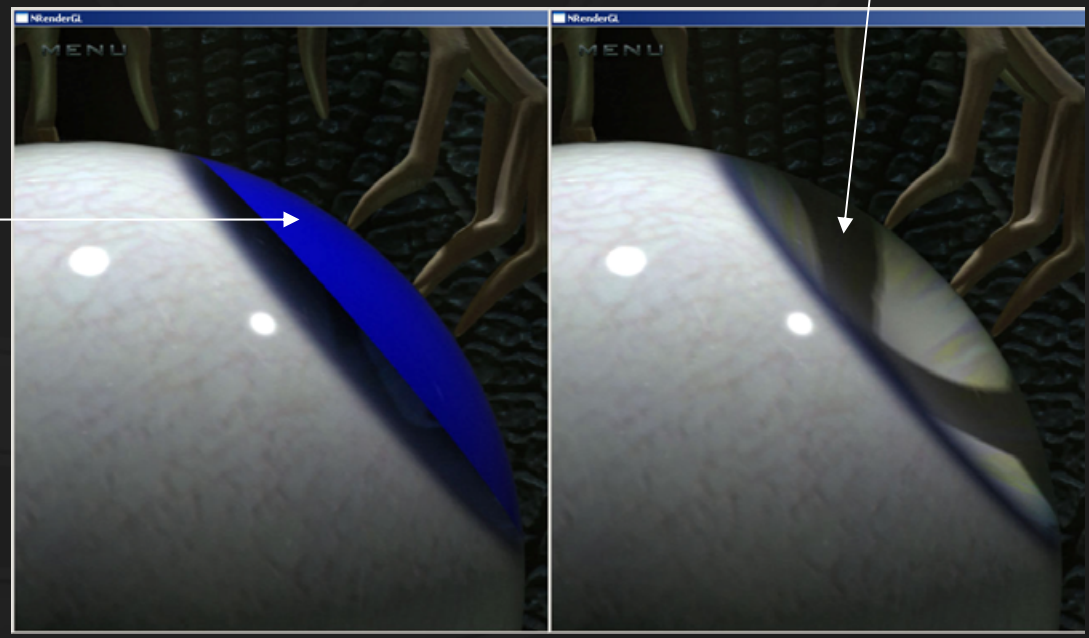

透過光線
ふたつめの球との接点(必要なら) :
虹彩に当たった場合は、光源処理の
ために位置と法線を求める
ひとつめの球との接点:
角膜から光が出たか?
新しい位置と法線を使った光源処理で虹彩を描画
法線: フレネルに使用
反射と角膜のハイライト
光のトレース
進入光

光が角膜に戻る場合は?

- 反射係数が**0.723**の場合、光が眼球から出ることはない
- **0.723**は人間の眼球の角膜に対する(実証はされていない)反射係数

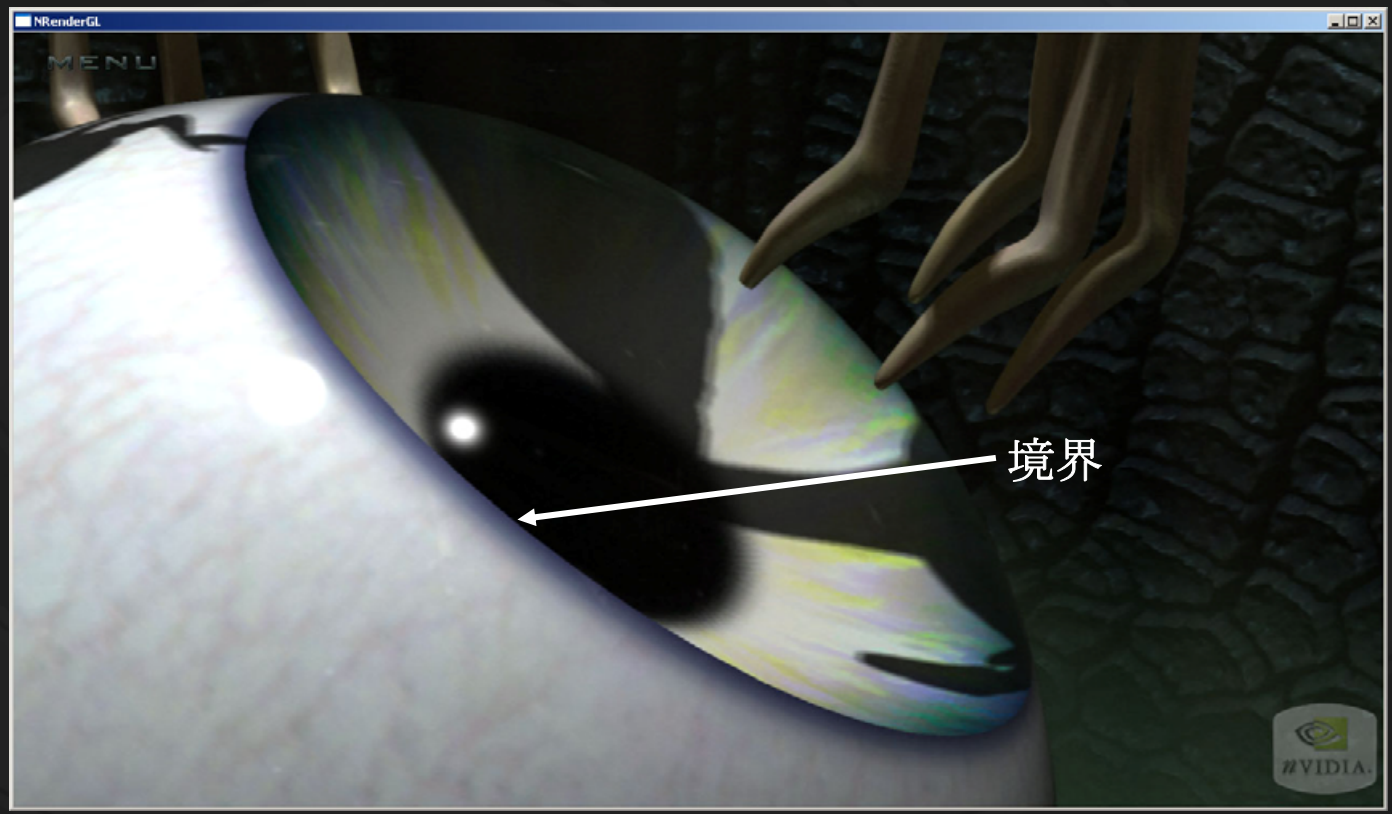
反射係数 **0.723**

ゼロ (または非常に小さい)透過光:
この場合、光が眼球から出て行く
ケースを考慮する必要がある
この描画では単純に青を使用



境界の解決

- ここまでの解説どおりに描画すると、白目と虹彩の間に境界ができる

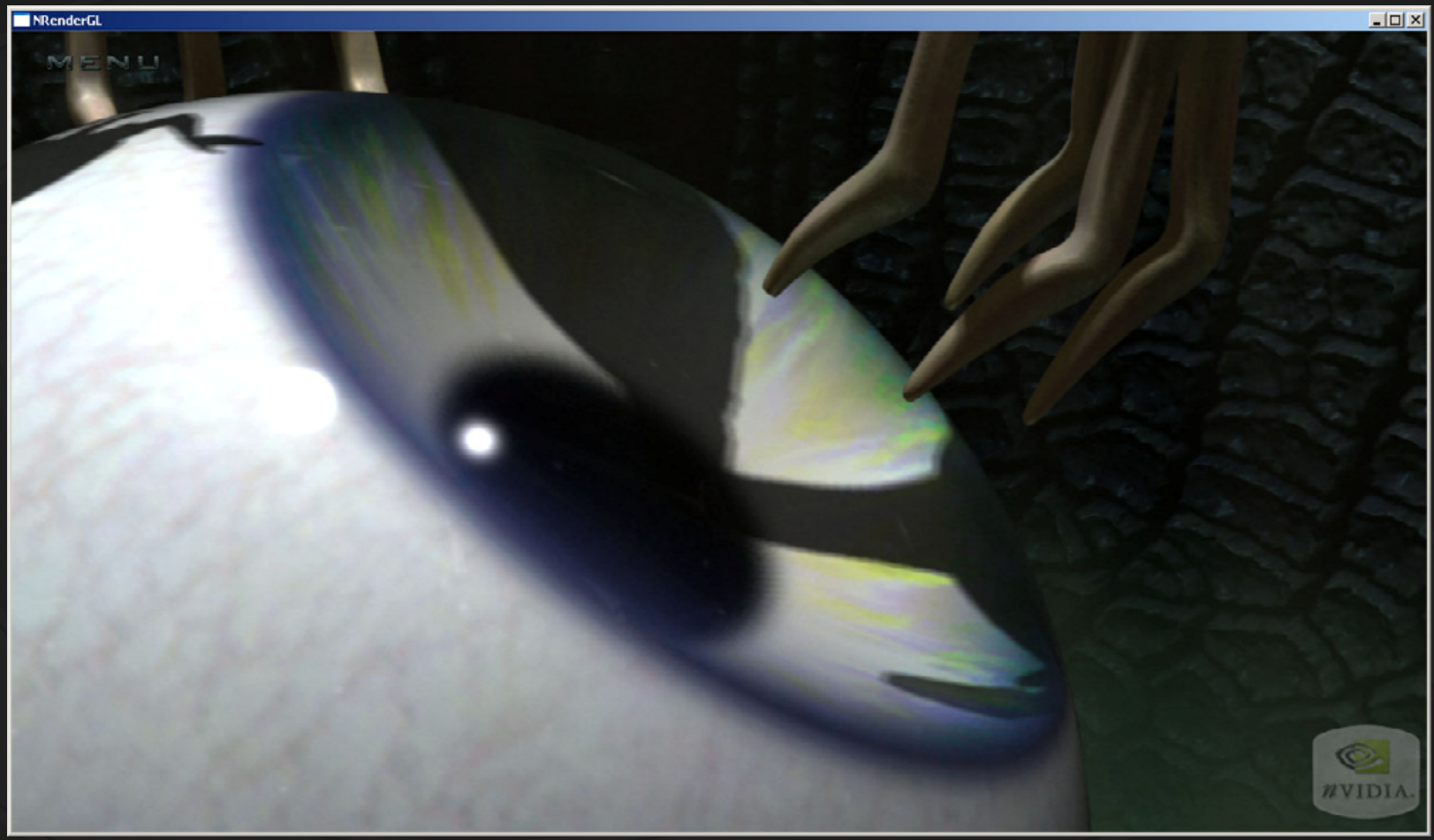


解決策: 境界付近をブレンド

- 白目部分と虹彩部分は同じ光源モデルを使用
- 以下の四項の光源パラメタのみに違いがある:
  - 法線
  - ディフューズ色
  - スペキュラ色
  - スペキュラ指数

- これらよっつの項を境界付近では線形補間
- シェーダの最後にひとつの光源処理計算だけが行われる

境界なし



眼球内の影

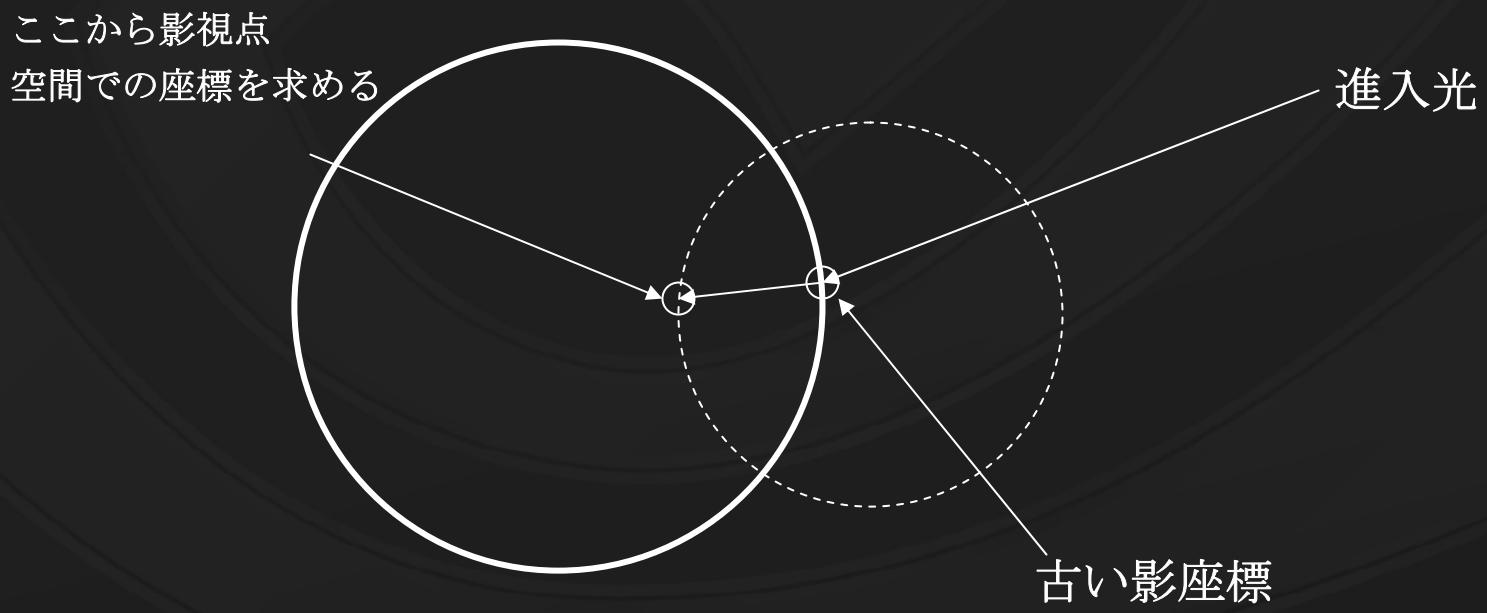
- どうやって正しい影が計算できるか?

正しい虹彩の影座標

ひとつめの球からの影座標

解決法:

- 眼球のオブジェクト空間から影視点のビュー・プロジェクション空間への変形行列を使用
- レイトレースでオブジェクト空間での眼球のどこに本当の接点があるかを決定、この位置を行列で変換し、影マップ上の対応する位置とZ値を算出



GeForce 7800 GTX以前では不可能

- **GeForce 7800 GTX**のふたつの機能によりシェーダが可能
  - シェーダ・モデル**3.0**の分岐
  - ピクセル処理速度
- 最終的なシェーダはフラグメント・プログラムで**289**命令
  - 分岐により、実際に実行される命令数は制限される**(**特に白目部分**)**
  - **GeForce 6800**でも**SM3.0**分岐は可能、しかしこの眼球を全画面でリアルタイムに描画するにはピクセル処理速度が足らない

速度比較

- **Luna**デモ**: 1280 x 1024**ピクセル**(**全画面の眼球、虹彩のアップ**)**
  - **GeForce 6800 GT: 8.5 fps**
  - **GeForce 7800 GTX: 17 fps**
  - ちょうど**2**倍のパフォーマンス**!**
- 分岐の有無**: (**全画面の眼球、白目と虹彩**)**
  - **GeForce 7800 GTX, PS2.0: 19 fps**
  - **GeForce 7800 GTX, PS3.0: 24 fps**

適用範囲

- この技法のパフォーマンスは、描画ピクセル数に比例:
  - **100**キャラクタの目に適用することも簡単に可能
  - 目が画面上で大きく描画されなければ透過光はわずかな差しか作らないが、わからなくもない
  - 頭の中に埋め込まれた目の場合、影座標の再計算はおそらく必要ではないだろう

美しいヒーロー・スーツ

難点

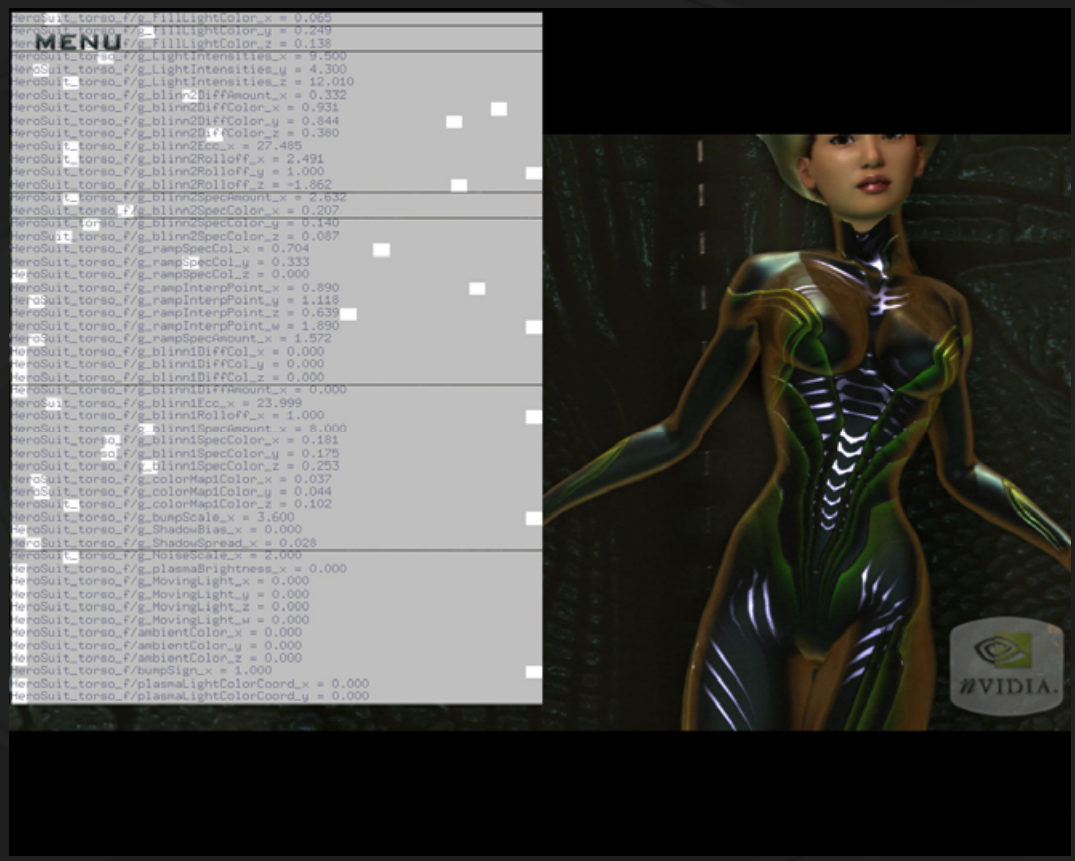
- デザイナのスケッチでは金属的なものが要求された:
  - 属性の違うい くつもの金属層
  - 非常に細かいつくり

初期概念図

全体の流れ

- **GeForce 7800 GTX**の膨大なピクセル処理量を使用
  - 汎用性や自由度の非常に高いシェーダを作る
  - 合成: 違った設定で何度も光源処理をし、それらをリアルタイムに合成
  - 全パラメタのスライダーを作る
  - デザイナの創造性に任せる
  - 最終的な表現が決まった時点で最適化作業を始める

スライダー



- シェーダ内の全ての定数パラメタに対して、プログラム起動時に自動的にスライダーを作成

解決法の概要

- 3層のシェーダがフラグメント・シェーダで合成される
- 第1層: ディヒューズと、微かで広範囲でノイズのあるスペキュラ
- 第2層: 1層目の下にあるように見える(しかし実際は1層目の後に合成される)切れのいい金属
- 第3層: 輪郭の光源処理で金属的な反射を表現


第1層: ディヒューズ


第1層: スペキュラ
総量テクスチャ

第1層: スペキュラ光源処理

- みっつの光源からのみっつのスペキュラ
  (blinn-phongを固定の指数で使用)


第1層: スペキュラ
色テクスチャ


第1層: スペキュラ合成

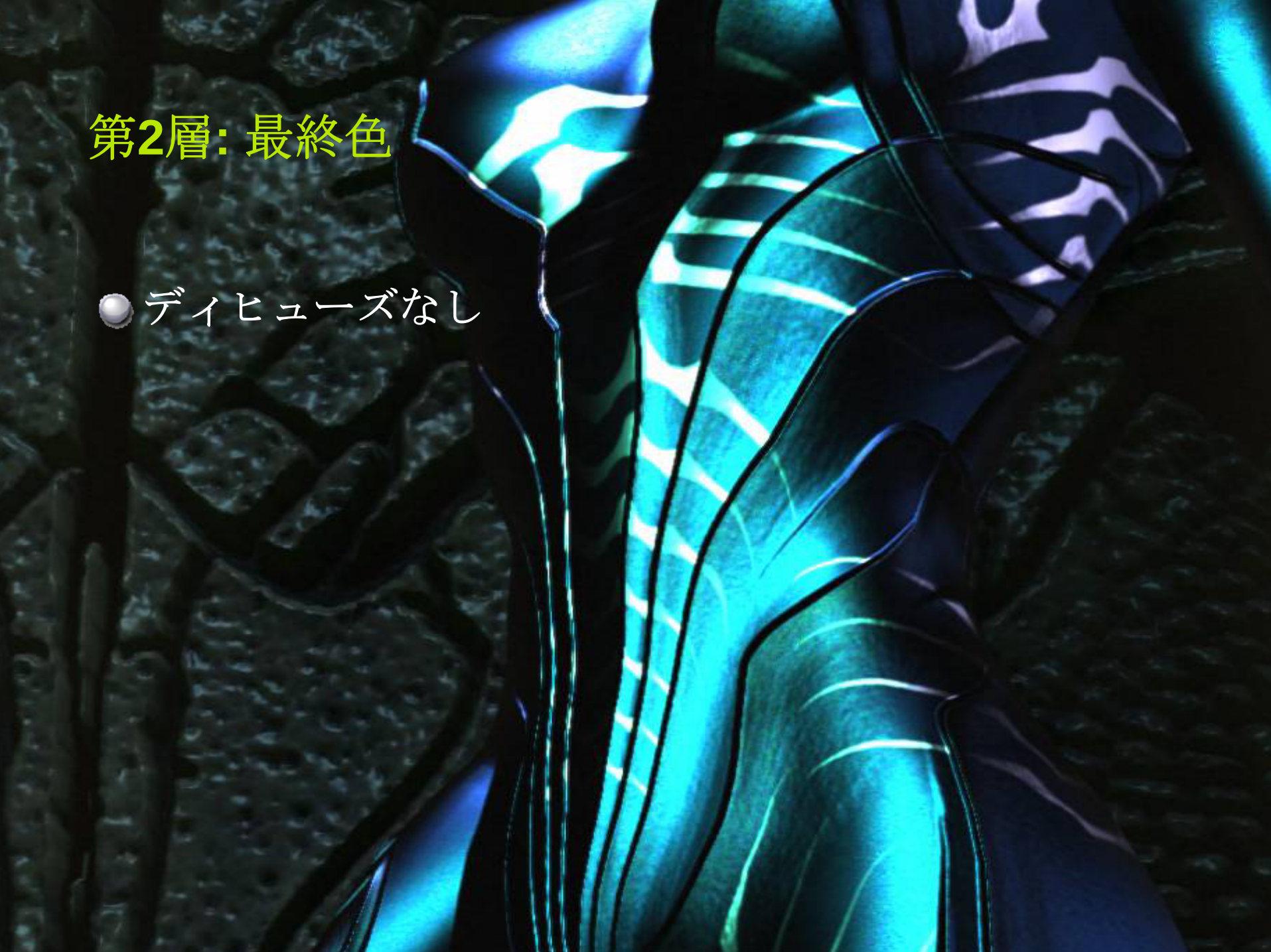

第1層: 最終色 (ディヒューズ + スペキュラ)


第1層: バンプ・マップと高解像度スペキュラのノイズを組み合わせた最終色


第2層: スペキュラ色

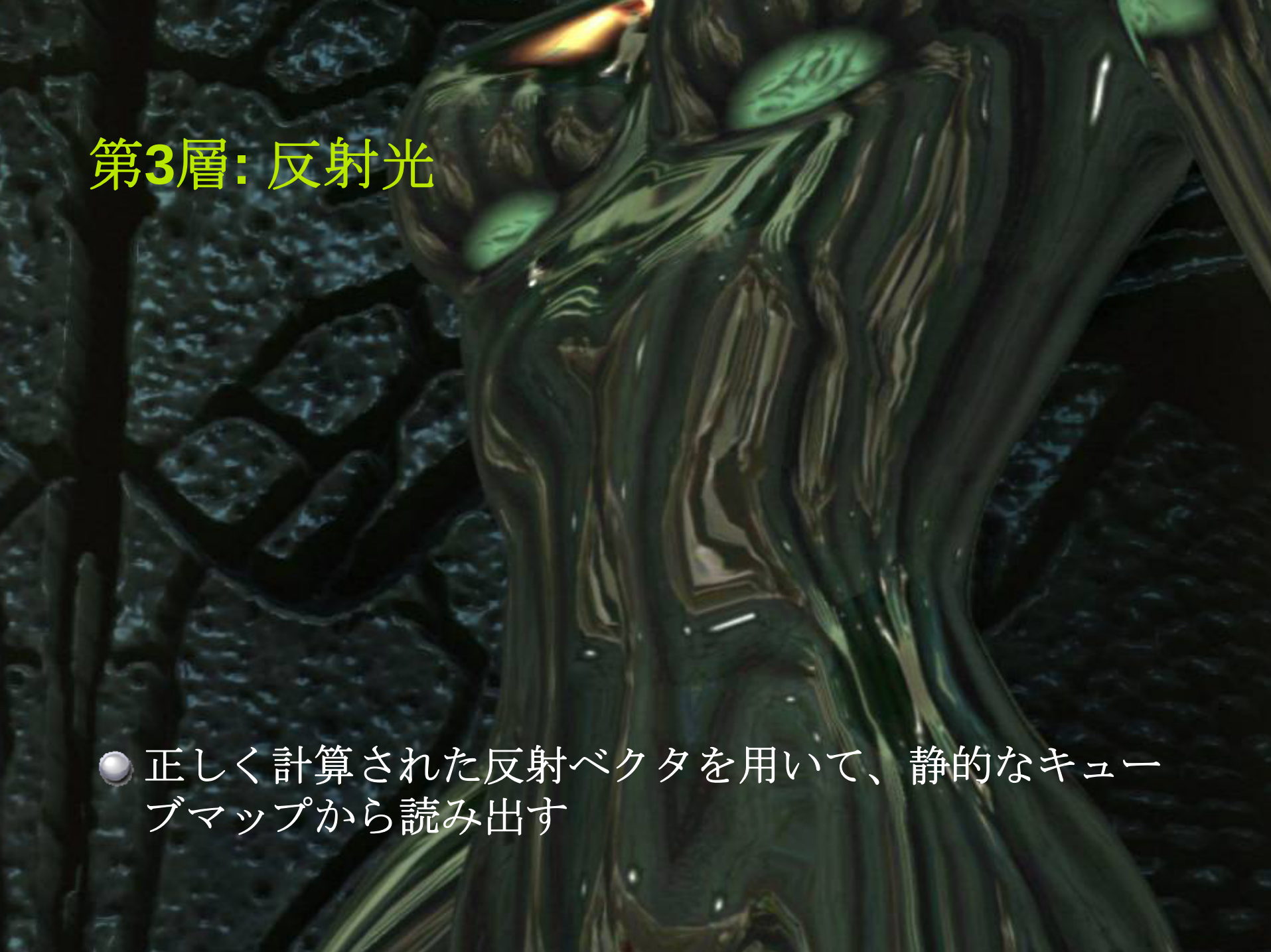
第2層: スペキュラ光源処理

- さらにみっつの**blinn-phong**光源


第2層: 最終色
ディヒューズなし


第1層と第2層の合成比率

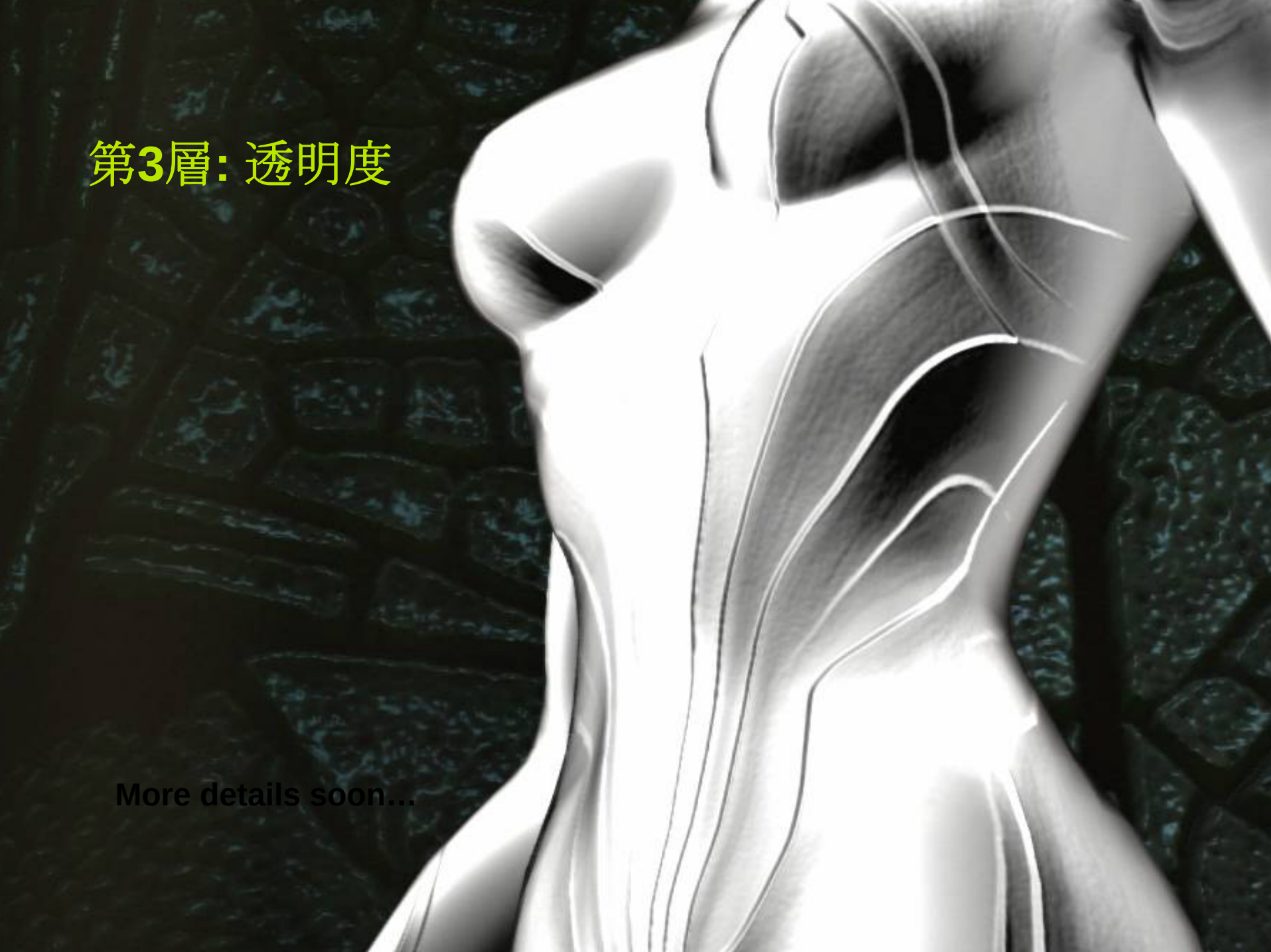

第1層と第2層の合成

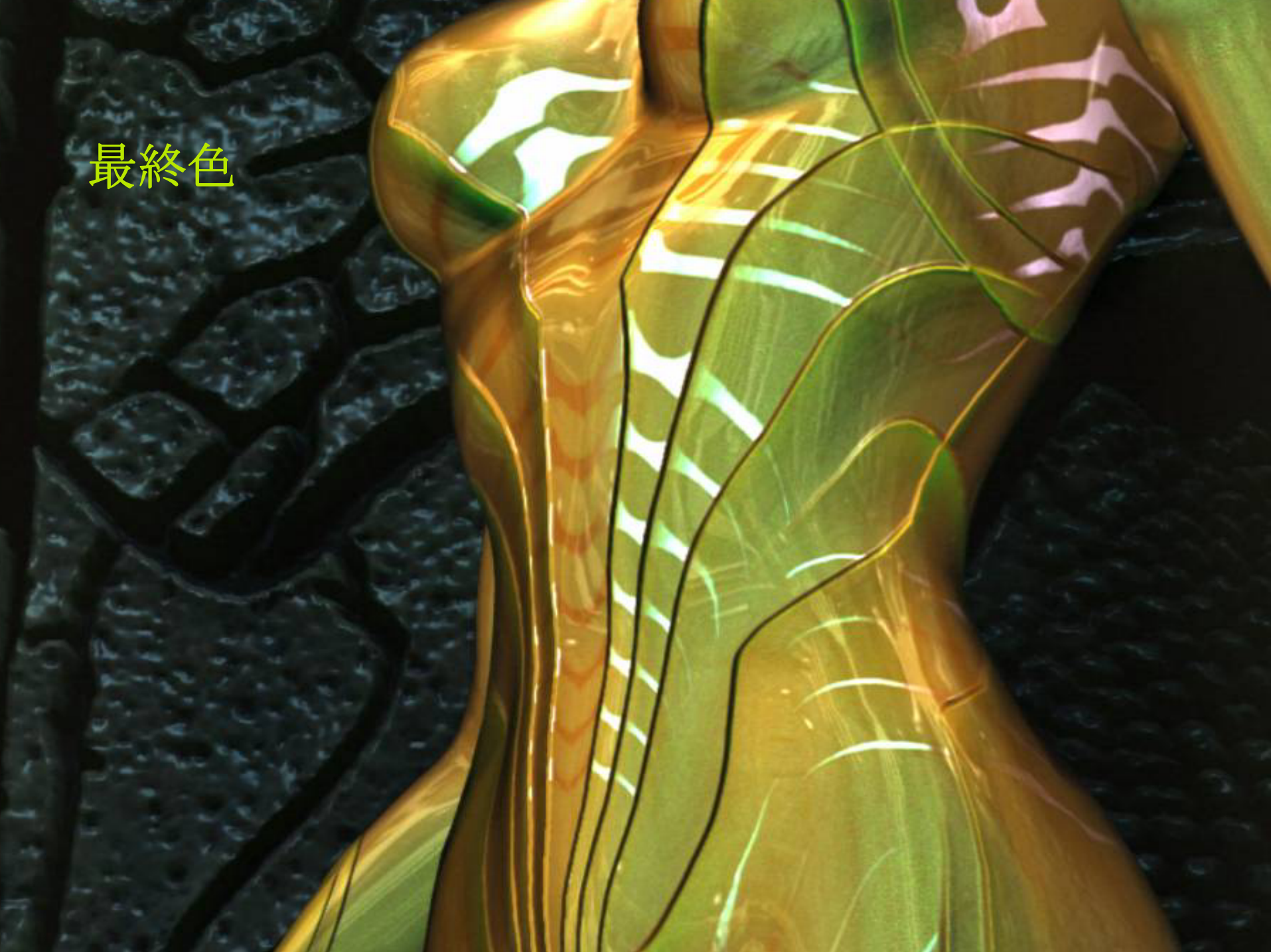


第3層: 反射光

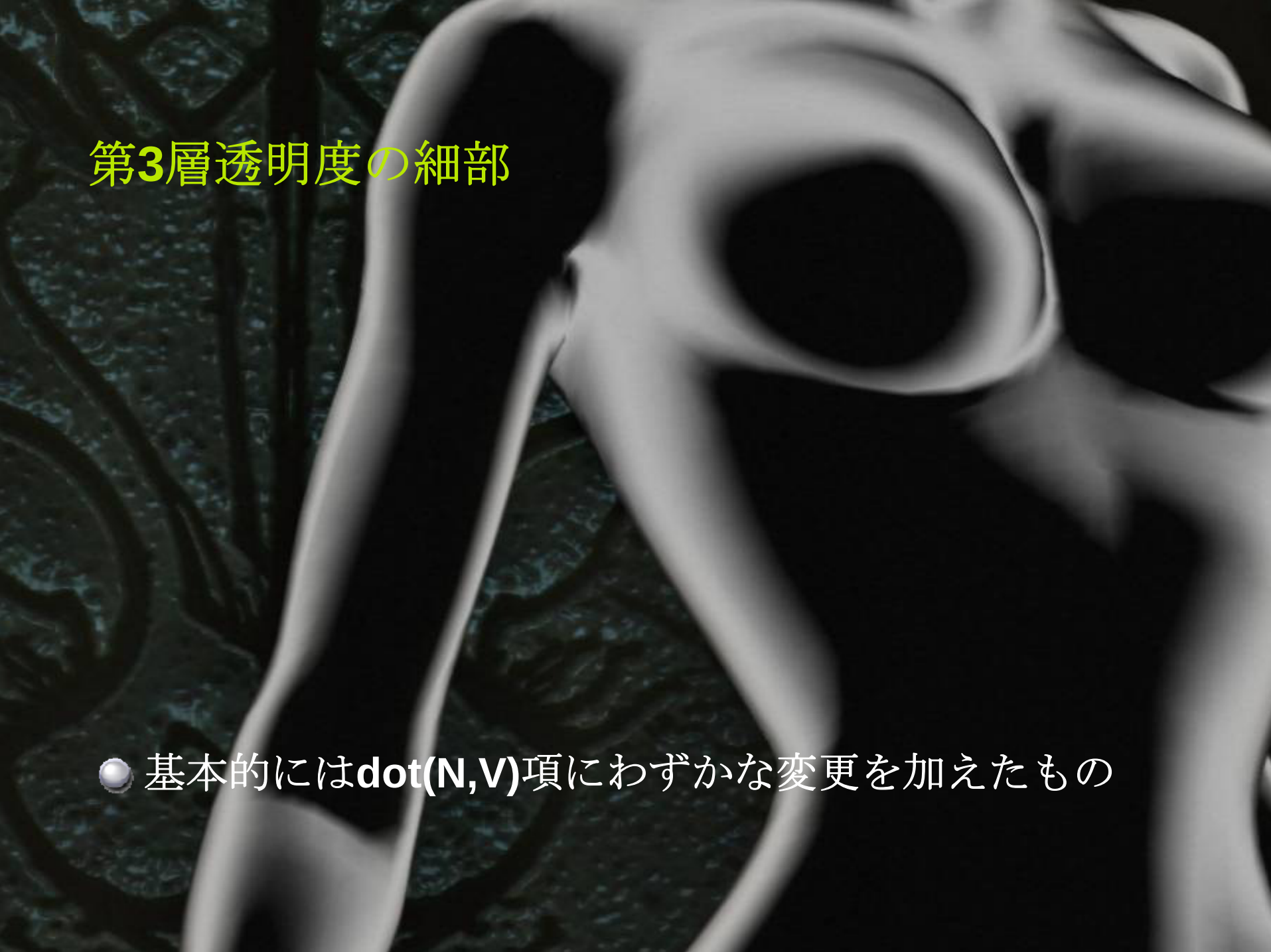
- 正しく計算された反射ベクタを用いて、静的なキューブマップから読み出す


第3層：色


第3層: 透明度
More details soon…


最終色

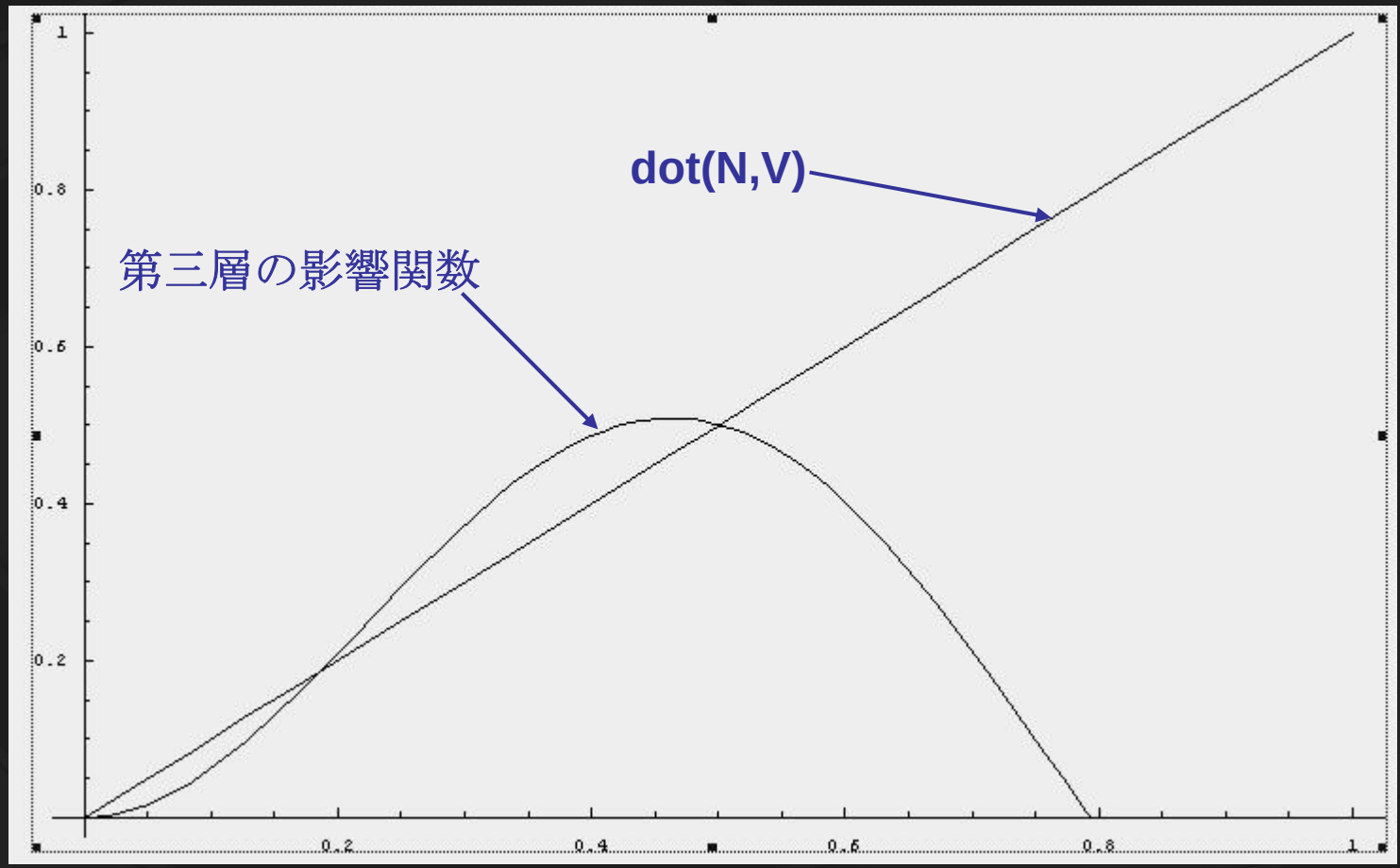

さらに通常の光源処理



第3層透明度の細部

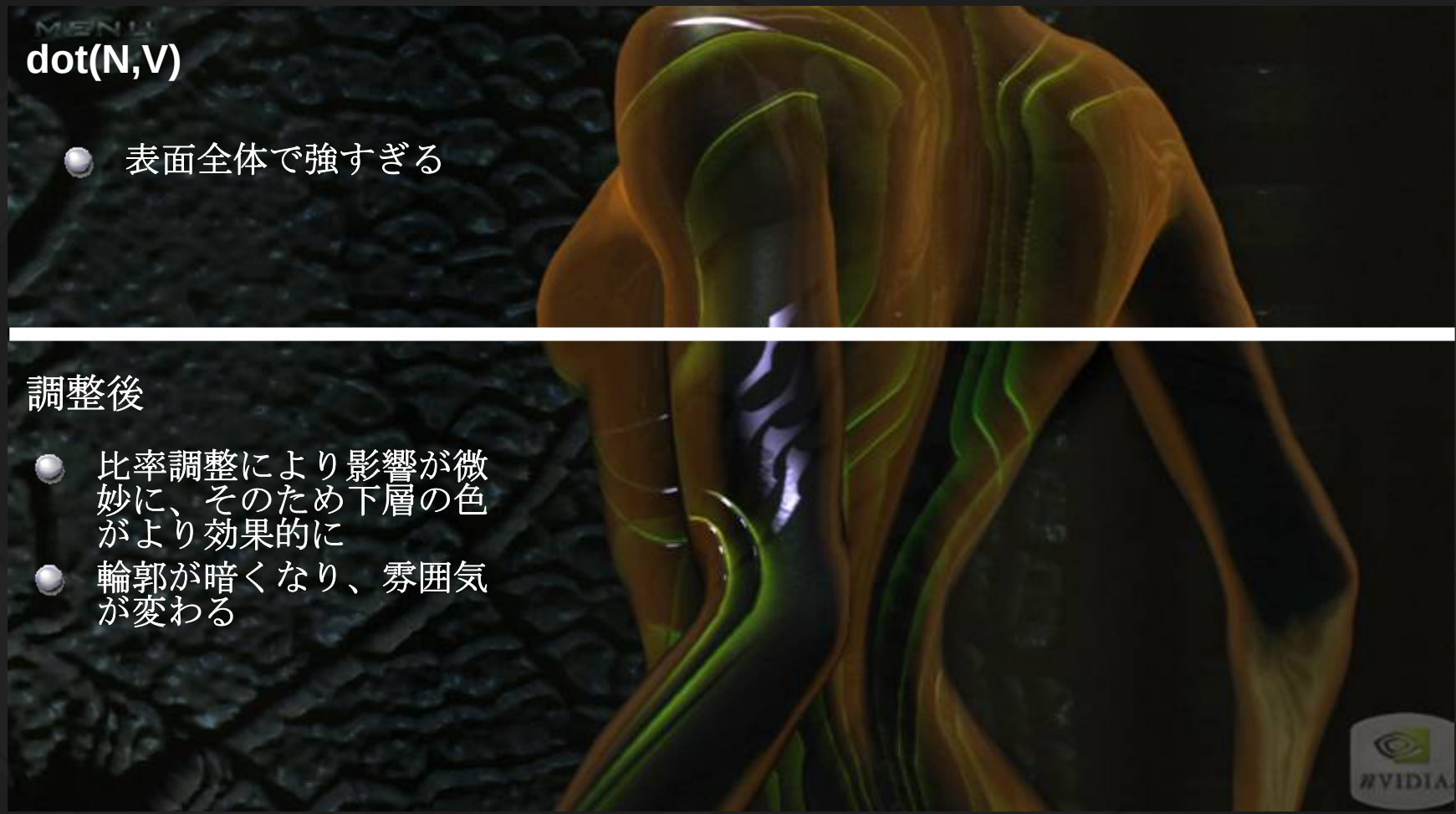
- 基本的には**dot(N,V)**項にわずかな変更を加えたもの

dot(N,V)と減衰部分調整の比較



視覚的な違い

視覚的な違い


dot(N,V)
表面全体で強すぎる
調整後
比率調整により影響が微妙に、そのため下層の色がより効果的に
輪郭が暗くなり、雰囲気が変わる

最終的なシェーダ

- ピクセル・シェーダの命令数: **238**
- ピクセルごとの使用テクスチャ数: **8**
- ピクセルごとのテクスチャ参照回数: **24**
- ピクセルごとの**Blinn-Phong**スペキュラ計算回数: **6**

Mad Mod Mikeデモ

- 間接光処理
- 被写界深度効果
- 全方向影処理

- **25**パスでの描画
- 非常に複雑な彩色処理:
  - **Mike: 85 – 150**命令
  - 環境: **74 – 144**命令

直接/間接光処理

直接光:

面に直接あたり、照らす光源

間接光:

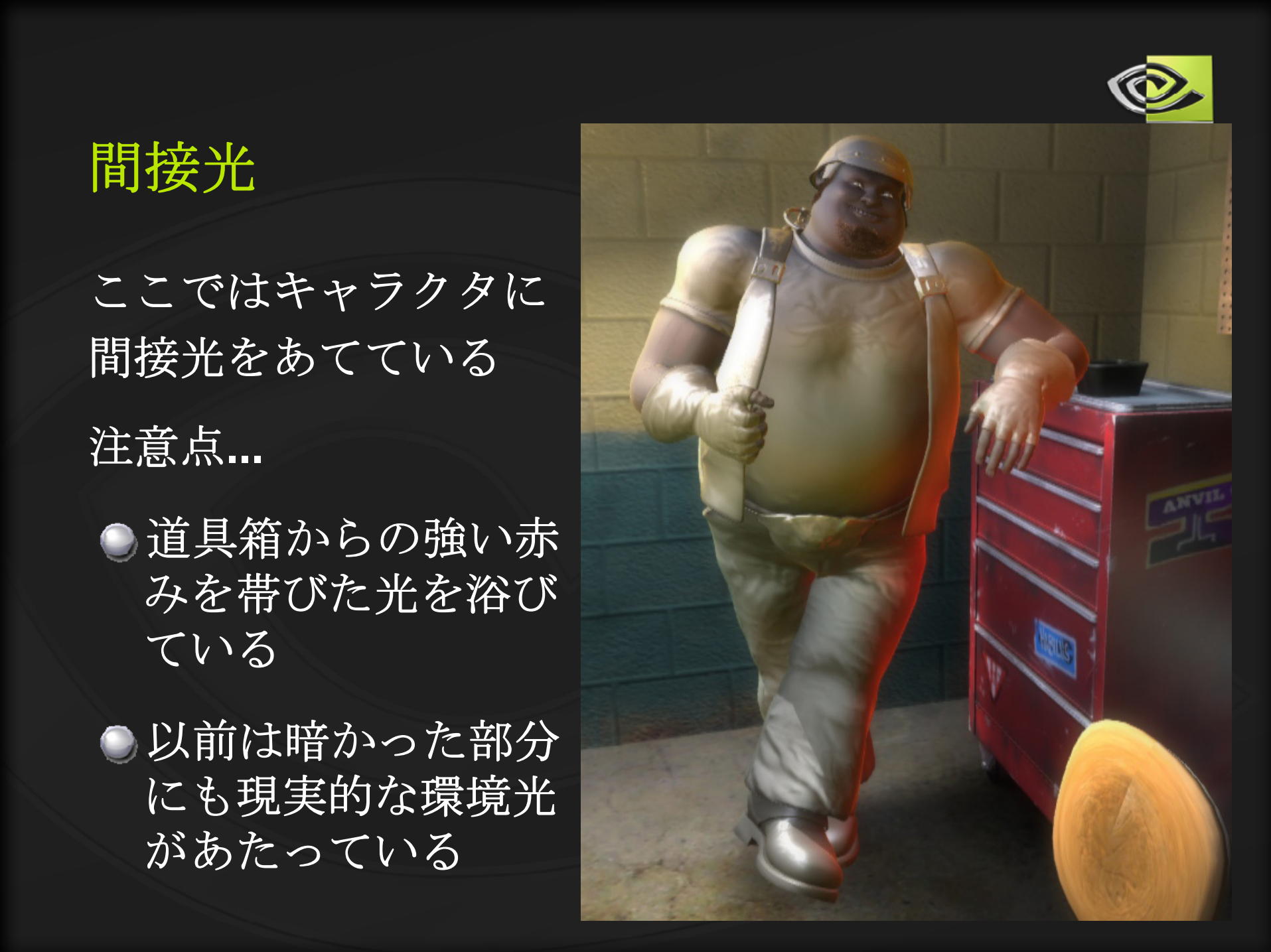
面で跳ね返り(その際に面の色が混じり)、別の面を照らす光

直接光

画像中で、**Mike**はふたつの点光源に照らされている

注意点...

- 体の大部分は非常に暗い(照らされていない)
- 周辺にある物体の色が反映されていない



間接光

ここではキャラクタに間接光をあてている

注意点...

- 道具箱からの強い赤みを帯びた光を浴びている
- 以前は暗かった部分にも現実的な環境光があたっている

他の例 [直接光のみ]

他の例 [間接光つき]

間接光

- たくさんの暗い点光源を使って表現?
  - 非常にたくさんの光源が必要
    - 少なくとも目立つ家具ひとつにつきひとつの光源
    - 壁や床に数個ずつ
  - 光源の管理も必要...
    - ...位置
    - ...色 (物体色の平均から)
    - ...*強さ* (いろいろな仮定と近似が必要!)
  - 遅く、美しくなく、手間がかかる...

間接光

もっと簡単な手法:

毎フレーム...

1. 小さなキューブマップにシーンを描画(キャラクタ以外)
2. キューブマップをブラー
3. キャラクタの光源処理の際、直接光以外にもキューブマップからの色を使用

手順1: シーンを小さなキューブマップに描画

- 面ごとに**96x96; BGRA**形式
  - **(**さらに**16-bit**の深度バッファ**)**
- フレームごとに、キューブの中心はキャラクタの腹部に**(...**もしくは骨格の接続部**)**
- キャラクタ以外の環境を描画
- **LOD**などのために低解像度のメッシュがあるのならそちらを使用
- 簡略化したフラグメント・シェーダで描画
  - 通常のシェーダは**100-150**命令
  - 簡略化された『放射光』シェーダは**44**命令
  - **(**それでもピクセルが少ないので非常に高速**)**

簡略化されたシェーダ

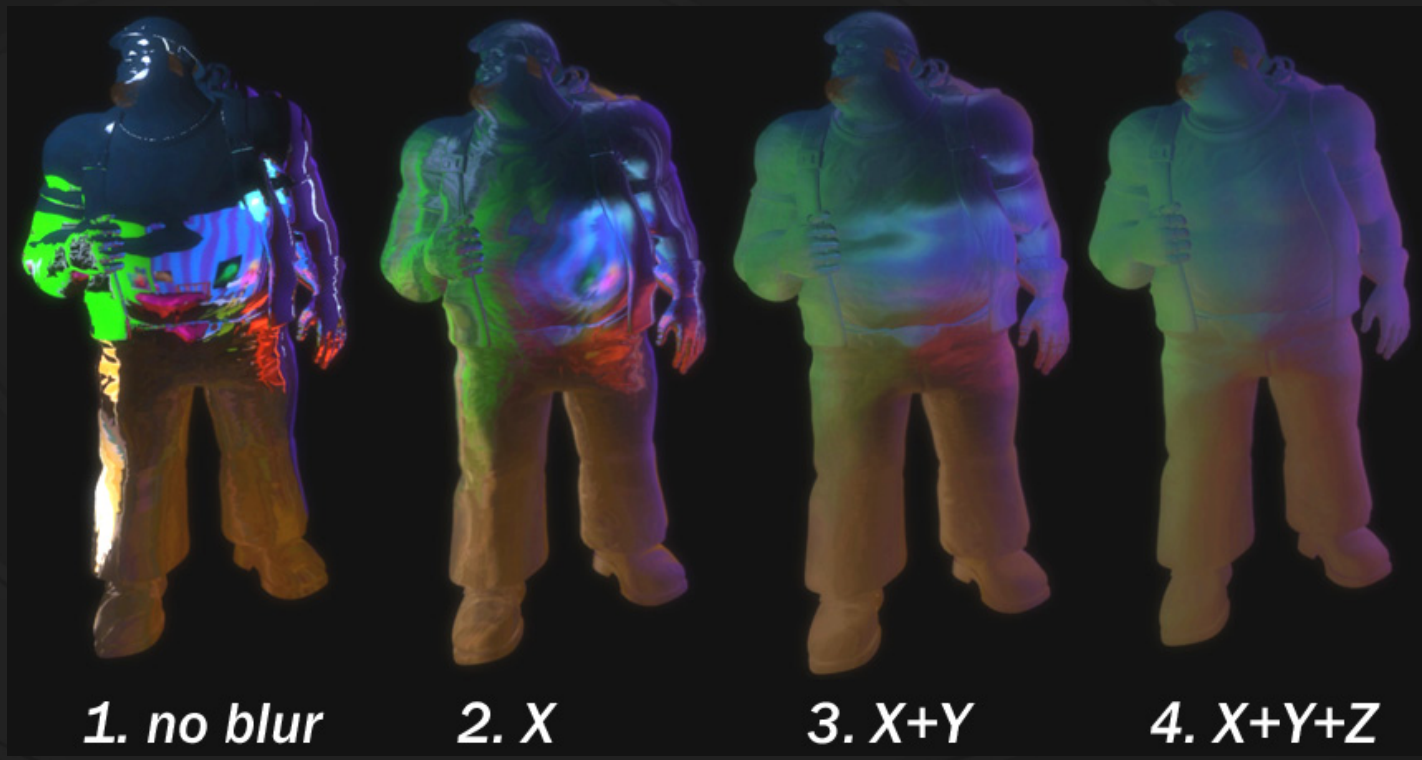
- ふたつの点光源に対し、簡単なディヒューズ光源処理(N • L)のみ行う; スペキュラはなし
- メインのディヒューズ色のみテクスチャから取得(バンプ・マップ、スペキュラ・マップなどは使用しない)
  - 最終的にブラーされるので、1以上のミップマップ・バイアスを使用
- キャラクタ (もしくはキューブ)の中心からの距離によって、最終色の輝度を減衰させる!
  - 遠い物体の影響が少なくなる
  - 近い物体は100%; 遠い物体は20%程度
- 効果を強調するために最終色を強くすることもできる
- 影は考慮に入れていないが、あったほうが良かっただろう!

手順2: キューブマップをブラー

- 3パス(実際は3 x 6 = 18、しかし非常に高速)
  - X軸沿いに円柱状にブラー; さらにYとZ
- 単純にキューブマップをブラーしたのでは、面の境界に線ができる
- 最初のキューブマップを動的な反射光処理に使用可

間接光処理

- 鉛色のMikeが4段階のブラー処理中のキューブマップを全反射
- 画像で分かるように、みっつの円柱状ブラーを全て行うことが大切!



手順3: キャラクタの光源処理

- 光源処理の際、直接光以外にもブラーされたキューブマップからの色を使用
- 問題: キューブマップ座標を決めるベクタは?
- 注意: キューブマップは3Dベクタひとつで参照される!
  - (本当は6Dテクスチャから表面の位置と法線を使って色を取得するのが理想、しかしそれは不可能...)

間接光処理

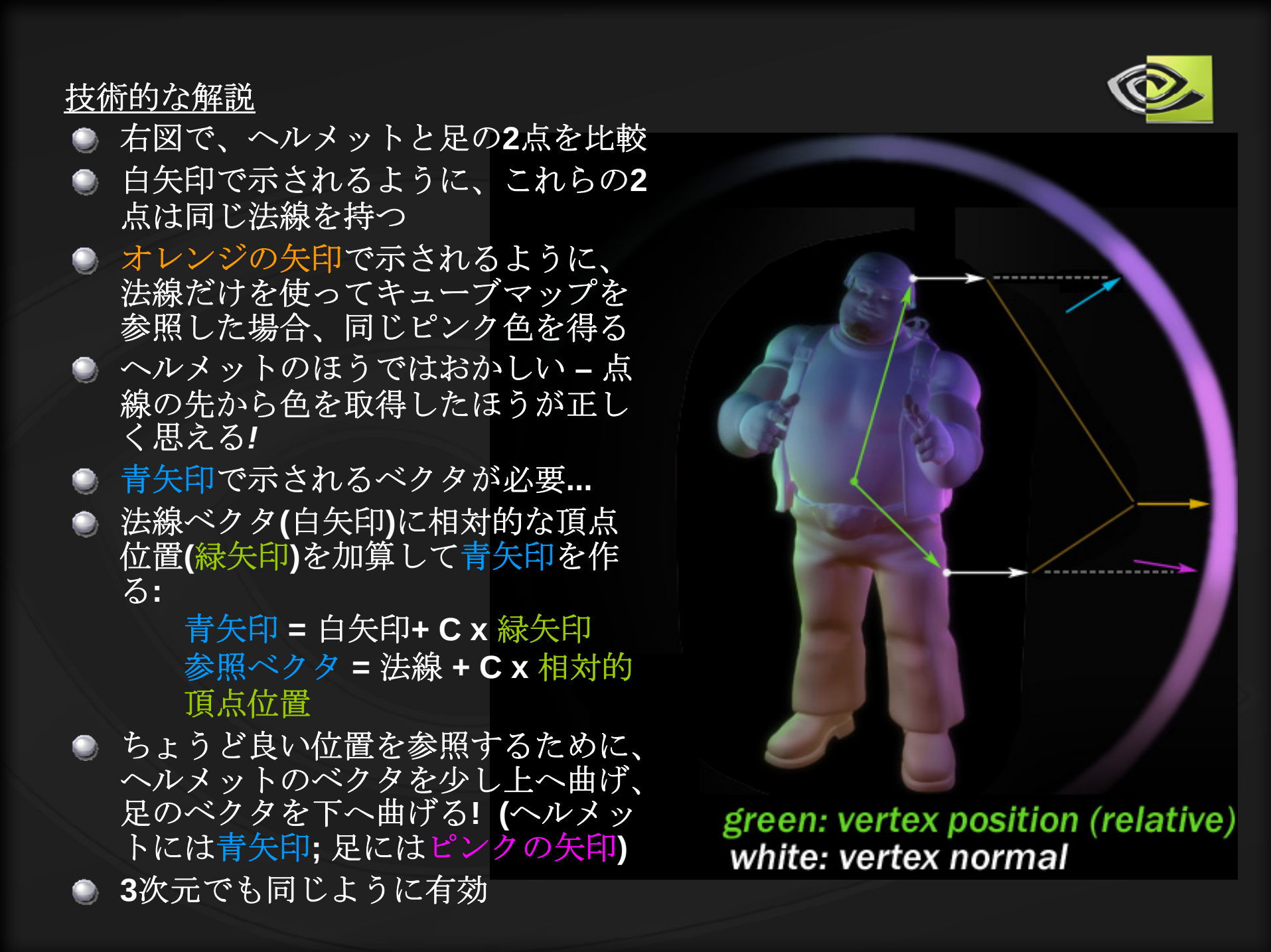
- *頂点の法線* を用いてチューブマップを参照
- 環境が非常に遠くに見え、隣接した環境には不向き
  - 左側の緑のハイライトが多すぎる
  - 下からの茶色や紫が足にあまり投影されない

間接光処理

- キャラクタ(もしくはキューブ)の中心を基準とした頂点位置を使ってキューブマップを参照
- 改善が見られるが、法線も使ったほうが良い …

間接光処理

- ふたつのベクタを合成
- ここでは、主に法線を使っているが、頂点位置も影響
- **Bjorkeの*localized reflections* technique (GPU Gems)**に相当
- 現実的な描画!



技術的な解説

- 右図で、ヘルメットと足の2点を比較
- 白矢印で示されるように、これらの2点は同じ法線を持つ
- オレンジの矢印で示されるように、法線だけを使ってキューブマップを参照した場合、同じピンク色を得る
- ヘルメットのほうではおかしい – 点線の先から色を取得したほうが正しく思える!
- 青矢印で示されるベクタが必要...
- 法線ベクタ(白矢印)に相対的な頂点位置(緑矢印)を加算して青矢印を作る:

  青矢印 = 白矢印+ C x 緑矢印

  参照ベクタ = 法線 + C x 相対的頂点位置
- ちょうど良い位置を参照するために、ヘルメットのベクタを少し上へ曲げ、足のベクタを下へ曲げる! (ヘルメットには青矢印; 足にはピンクの矢印)
- 3次元でも同じように有効

green: vertex position (relative)
white: vertex normal

間接光処理

シェーダのコード

```
float3 rel_vtx_pos = wsCoord – g_wsIndirLightCubePos;
float3 sample_vec = wsNormal  +  C * rel_vtx_pos;
half3 env_diffuse = h3texCUBE(blurred_XYZ, sample_vec);
```

- rel_vtx_posもsample_vecも正規化しないことに注意!!

間接光処理

相対的頂点位置の割合**C**は環境までの距離と、シーン内でのキャラクタの大きさに依存...

1. 環境が無限大に遠い: **C == 0**
2. 非常に大きな部屋: **C ~= 2.5 / wsCharHeight**
3. 小さな部屋: **C ~= 7.0 / wsCharHeight**

- **wsCharHeight**はキャラクタのシーン内での単位による身長
- 『小さい部屋』は**Mad Mod Mike**デモで使用されている程度の部屋 **(Mike**の大きさに対して**)**
- これらは経験的に決められた値でしかないので、調整可**!**

その他の注意点

- **75%** 程度の直接光と**25%**程度の間接光でちょうど良い
  - 直接光 **(N • L)**が大部分の光源を提供するべきで、影やバンプマップも顕著になる
  - 間接光は壁などの物体から跳ね返った光を表現し、キャラクタの環境内での存在感を出す
- 複数のキャラクタに間接光をあてる場合、それぞれのキューブマップを作ってブラーしなければならない

間接光処理

- 最後に、シェーダが影処理をするのなら、影部分で**env_diffuse**項をわずかに暗くする
  - **env_diffuse *= (0.8 + 0.2*shadow_mult);**
  - あまり正確ではないが、見栄えがする

- 最終的な光源処理式:
  - diffuse_light = env_diffuse
    + Σ ((N • L_i) * $LightColor_i$ * $shadow\_mult_i$);
  - final_color = diffuse_light * diffuse_color
    + spec_light * spec_color;

間接光処理

パフォーマンス:

Mad Mod Mikeを**GeForce 7800 GTX**で実行:

キューブマップ描画	**1.0 ms / frame ***
キューブマップのブラー(合計)	**< 0.9 ms / frame**
その他の最終的彩色処理	**+ 1.0 – 1.5 ms / frame ****
合計:	**2.9 – 3.4 ms / frame**
FPS:	**30fps中約2.5fps**

* 物体数、頂点数などに依存、これは**Mad Mod Mike**デモでの数字
** キャラクタの画面でのピクセル数に依存

間接光処理

他の情報源

球面ハーモニクスを使用した高速化(...スペキュラのコンボリューションにも依存!)は以下を参照:

- **King, Gary. "Real-time Computation of Dynamic Irradiance Environment Maps" in *GPU Gems 2*. pp 167-176. Ed. Matt Pharr. Addison-Wesley, 2005.**

被写界深度効果

- 現実の写真には被写界深度がある – この距離でちょうどフォーカスし、これより近いか遠いものはぼやける
- コンピュータでは全てのピクセルにフォーカス – *意識的に回避しない限り*

被写界深度効果: 多層方式

多層方式: ふたつの画像(層)を描画、背景をブラーして合成

(+) 全ての物体が前景と背景に分かれる場合は有効

(–) 物体が動く場合、突然前景と背景の間を行き来する

(–) ひとつの物体が前景と背景の両方に存在する場合はうまくいかない(...例えば線路や長テーブルなど)

被写界深度効果: 深度依存ブラー方式

深度依存ブラー方式: Z値によって、ブラーの半径を拡大

- (+) 管理が楽(物体をフレームごとに前景と背景に分ける必要はない)
- (+) ひとつの物体が前景と背景に属する場合にも良い
- (–) 前景のピクセルが大きくブラーされた背景に写る問題が起こる!

被写界深度効果: 混合方式

われわれの解法: ふたつの方式をあわせる

- シーンを2回描画: 『遠景』パスと 『近景』パス
- カメラのNEAR/FARクリップ面以外は同じ
  - 両方のパスで全ての物体を描画(*... しかしほとんどの物体は少なくともひとつのパスからビュー・フラスタムで消去される*)
- それぞれを深度依存ブラー方式で独立してブラー

処理

1. 遠景パスを描画
2. 近景パスに画面大短形を描画、この際に遠景パスの結果から参照(色と深度)し、なおかつ深度依存ブラー、さらにZ=1.0を描く
3. 近景パスをその『背景』画像のうえに描画
4. 近景に深度依存ブラーをかける

被写界深度処理: 利点と制限事項

利点:

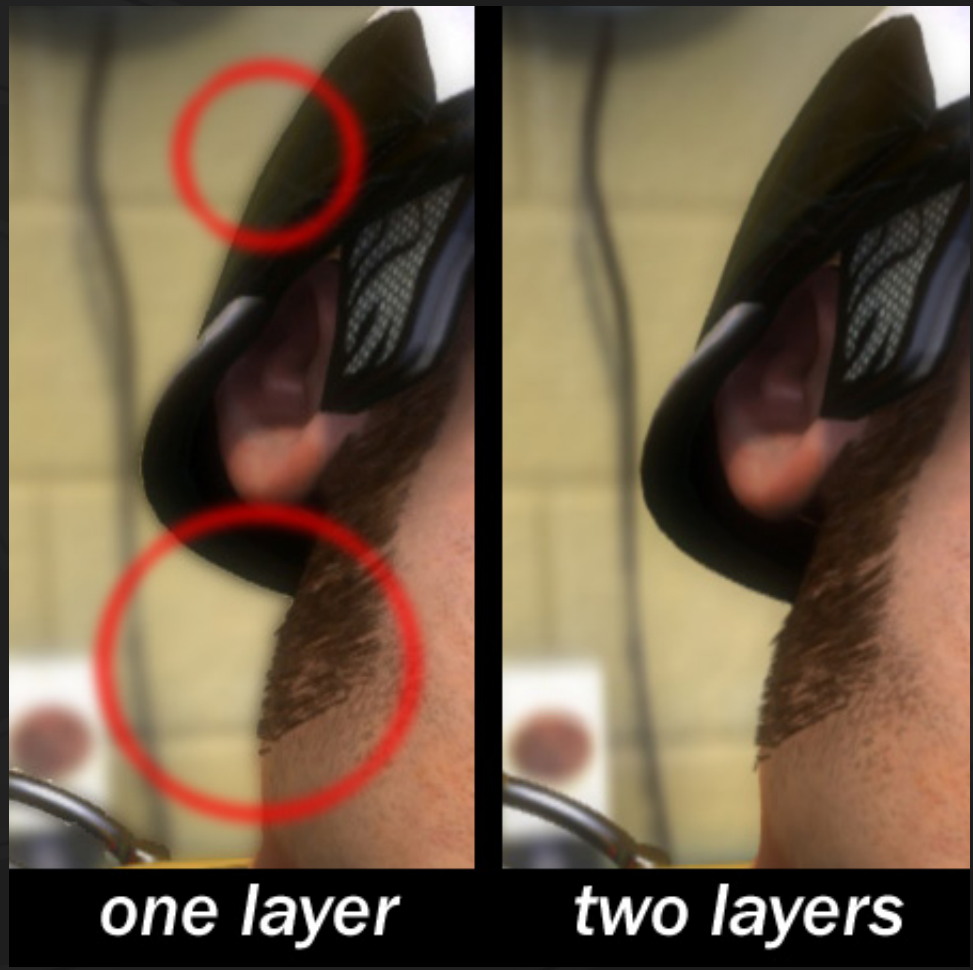
- **(+)** 被写界深度効果が**Z**軸沿いに連続的
  - カメラや物体が動く際の突然の変化がない
  - ぼやけたピクセルと明瞭なピクセルの境界がない
- **(+)** 明瞭な前景ピクセルが背景ピクセルに映りこまない
  - ブラーが各層で独自に行われるため
- **(+)** 非常に高速**!!!**

制限事項:

- ぼやけた背景のうえに明瞭な前景がのる場合のみ可能(明瞭なピクセルの前にあるぼやけた物体は処理できない)

前景をフォーカス、背景をぼかす

- 1層のみの場合、『近景』のピクセルが『遠景』の色を拾ってくる (十分な情報がないため!)
- 2層の場合、背景は近景に映りこまない



被写界深度処理: パフォーマンス

- 頂点処理:
  - 一般的に1.3倍; 最大2倍 (...つまり1.3倍の負荷、30%遅くなる)
    - ビュー・フラスタムでのカリングに依存
- フラグメント処理:
  - 一般的に1.1倍; 最大約1.6倍
    - 遠景パスは半分の大きさ (71% x 71%)で処理できるので約0.5倍
    - 近景パスは普通半分程度のピクセルしかないので約0.5倍
    - さらに深度依存ブラーを各層にかけるための負荷が約0.1倍
    - 0.5倍 + 0.5倍 + 0.1倍 = 1.1倍
- 全体負荷:
  - 頂点制限の場合(メッシュが複雑)、約30%
  - フラグメント制限の場合(複雑なシェーダ)、約10%
  - (CPU制限の場合、ほぼ0%)

被写界深度処理とマルチサンプル(AA)

- *問題:*
  マルチサンプル(AA)使用の場合、そのバッファからはピクセル・シェーダでZ値を読み出せない(...深度依存ブラーに必要)

- *解決法1:* 近景と遠景を、AAのない深度バッファにシェーダなしで余分に一回ずつ描画
  - 頂点とフィル処理量が倍だが、深度のみの描画はGeForce 6と7では倍速

- *解決法2:* glCopyTexImage2Dを使用し、マルチサンプルの深度バッファから通常のバッファへ複写/変換

被写界深度処理: もうひとつの問題点!

- 近景描画の際、深度比較関数は**LESS_EQUAL**ではなく**LESS**を使用
- これで遠景の深度**1.0000**を持つピクセルが使用される
- さらに、近景の深度依存ブラーの際、深度がちょうど**1.0000**のものはブラーしない**(**これらは遠景から来ているので**)**
  - *そうしなければ、遠景のピクセルは二重にブラーされ、近景の明瞭な物体の色が遠景に漏れる*
- 提案**:** 近景には**24**ビットの深度を使用

  (もしくは**AA**の場合、余分な深度バッファのみ**);**
  - 精度が足らないと、遠くにある近景ピクセルがまったくブラーされず、明瞭に表示されてしまう

被写界深度処理: Mad Mod Mikeの場合

- 遠景: 912 x 512 2xAA
- 近景: 1280 x 720 4xAA

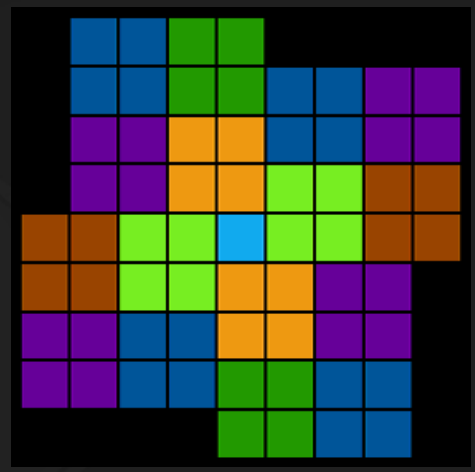
- 近景と遠景の境界深度は以下のように決定...
  - キャラクタが表示されている場合、彼に焦点をあてるため、動的に調整
  - それ以外では一般的な環境に合うように静的に設定
  - 中間のなめらかな移行は簡単
    (画面に出ているかどうかをファジーに決定し、線形補間)

被写界深度処理: 半径可変高速ブラー処理

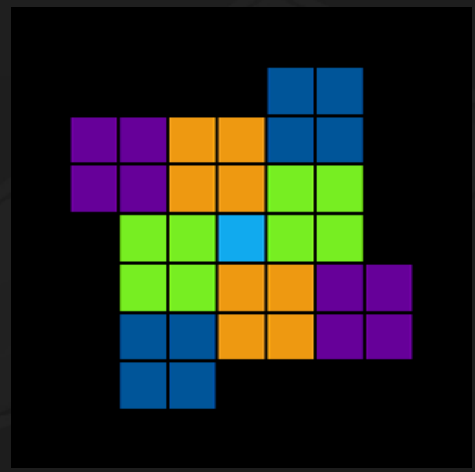
- 双線形補間(bilinear)を最大限に使用
  - 0.5テクセルずらすことにより、4テクセルを一度に取得
- 深度バッファからの取得:
  - 深度テクスチャを色テクスチャのようにバインドし、そこから取得
  - 注意: 深度値は双曲線関数で保存されている
- 取得された値はどちらかの総和値に加算: 『内』または『外』
  - 『内総和』: 常に重み付き
  - 『外総和』: 重みは深度値の関数
  - (カーネルの大きさに対する線形補間のように機能)

被写界深度処理:半径可変高速ブラー処理

- 遠景パス: 17サンプル(上図)
  - 内総和 = 5サンプル; 重みは17
  - 外総和 = 12サンプル; 重みは0から48 (深度が0から1なので)



- 遠景パス: 9サンプル(下図)
  - 内総和 = 5サンプル; 重みは17
  - 外総和 = 4サンプル; 重みは0から16 (深度値が0から1なので)
  - 近景は比較的明瞭なので小さなカーネルを使用



被写界深度処理: これからの改良点

改良点: 近景の前にもう一層いれて、Z沿いに『ぼやけ-明瞭-ぼやけ』の効果を出す。

処理:

- 第三層のアルファを0にクリア
- アルファを1にして物体を描画
- 単純な深度依存ブラーでアルファをブラー
- この層のRGBは違った方法でブラー: アルファが0のサンプルを無視(色の総和に影響を与えないよう黒にする)
  - 双線形は使えない → 4倍のサンプル数が必要
- 合成:
  - フレームバッファへは、ブラーされたアルファを不透明度とし、この層を最後に描画

全方位影処理

強い影とやわらかい影の基準:

光源に近い物体は柔らかい影を落とす;

遠い物体は強い影を落とす

Mikeのジェットパックには**3**種類の影:

1. 彼の体から部屋全体に対するやわらかい全方位影処理
2. 部屋の物体からお互いへの強い全方位影処理
3. (彼自信への強い平面的影処理)

やわらかい全方位影処理

- やわらかい全方位影処理は間接光源処理のキューブマップに;
  アルファを使用
- 深度を使用せず投影するだけ...
- つまり深度に関係なく、シールのように貼り付けられるだけ
- 影を受けるもの(部屋の物体)が光源と影を落とすもの(Mike)の間に来ないことが分かっている場合には非常に有効

やわらかい全方位影処理: 全処理

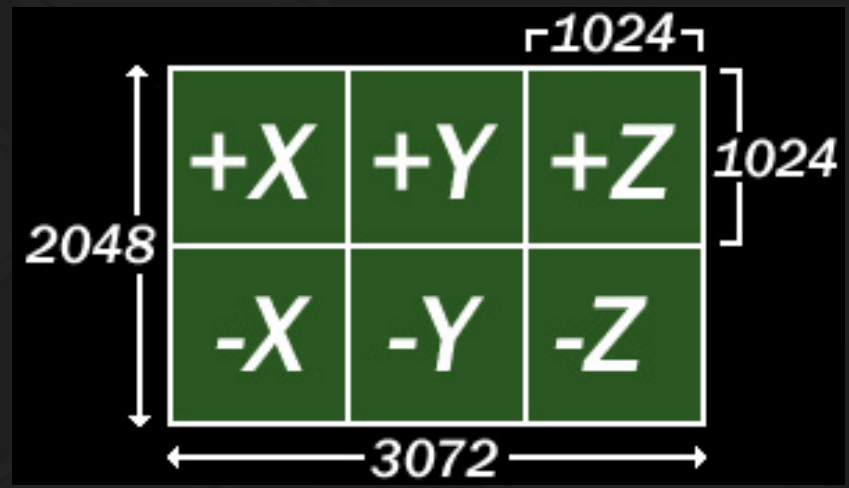
1. (キューブマップのRGBAを0,0,0,1にクリア)
2. (間接光源処理をキューブマップのRGBに描画)
3. RGB書き込みマスクを無効に; アルファ書き込みを有効に
4. 光源にカメラを配置(...キャラクタの中心から移動)
5. 低解像度のキャラクタをキューブマップのアルファに描画、アルファに0を描く
6. キューブマップが間接光源処理でブラーされる際、RGBだけでなくRGBAをすべてブラー
7. 部屋を最終的に描画する際:
   a)ジェットパックの光源へのベクタを用いて間接光源処理のキューブマップを参照し、アルファ値を得る
   b)その値をジェットパックからの光量に掛ける

強い全方位影処理

- 強い全方位影処理は理想的には**1024x1024**の面を持つキューブマップを使用すべきだが、ハードウェアに深度キューブマップのサポートはない
- かわりに『仮想』キューブマップ**(VCM)**を用い、この**3072 x 2048**の**2D**深度テクスチャ を**1024 x 1024**ごとの区画に分けて使用
  - キューブマップを開いたものと考える
- 実装は部分的に以下に基づく:
  - **"Efficient Omnidirectional Shadow Maps" Gary King & William Newhall. In ShaderX3, pp 435-448. Charles River Media, 2004.**

深度をVCMに描画 [影の投射]

- 部屋の物体を深さのみ、彩色なしで、最大6回、**3072 x 2048**テクスチャのむっつの**1024x1024**区画(ビューポート)へ描画
- ビューポートのカメラは各軸沿い:
  { +X +Y +Z –X –Y –Z }
- *(パフォーマンスについて: ほとんどの物体はみっつからいつつのパスでビュー・フラスタム消去され、頂点とフィルの負荷を下げる)*



VCMの参照: [影を受ける]

- キューブマップは普通法線(単位長さの方向ベクタ)で参照される...
- ...しかし仮想キューブマップは2D(u,v)で参照される
- 解決法: 『参照先キューブマップ』を作って、変換
  - *法線で、『参照先キューブマップ』を参照し、そこにある座標で仮想キューブマップの(u,v)座標を得る*

参照先キューブマップ

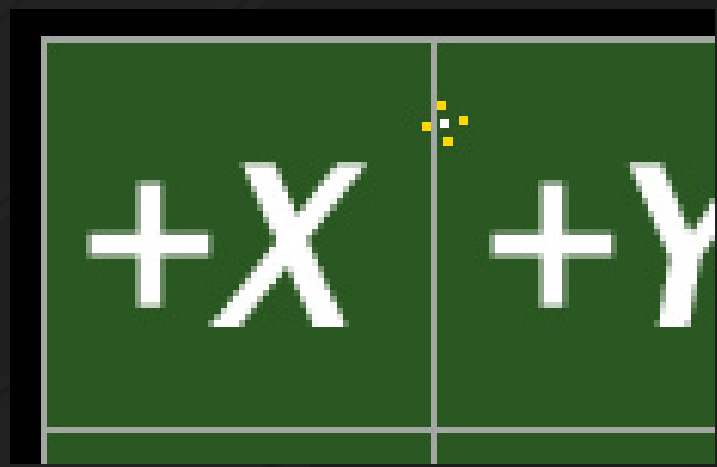
- 大きさ: **64x64**の面で充分
- チャンネル: **u**と**v**のふたつだけでよい
- フォーマット: **float4_16**を使用
  - **float4_16**キューブマップへの描画は**6**と**7**シリーズのすべてのカードで可能**!**
  - 各値**(u,v)**をふたつのチャンネルへ格納**(.rg, .ba)**
- 起動時に動的に参照先キューブマップを作成
  - ディスク読み書きの手間を省く

エイリアスの減少

- 影マップと同じように、***Percentage-Closer Filtering*** **(VCM 影の双線形補間)**を使用
- 複数値を参照して、平均することもできる
  - 参照先キューブマップは一度だけ参照し、**2D**面でその値をずらすほうが速い**(*画像を参照*)**

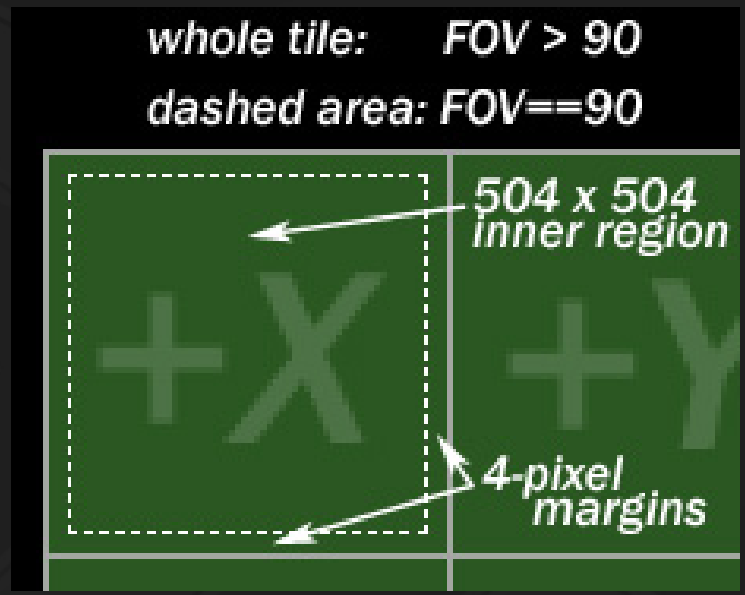
注意: 隣の区画に入ってしまうことがある**!**

- 参照一回でもおこりうる**... (**双線形補間のため**)**
- 複数回参照 **(**座標をずらすため**)**



解法: 各区画に余白をいれる

1. 90度よりわずかに大きいカメラの角度で、むっつのビューポートに描画
2. 参照先キューブマップも調整



- このプレゼンテーションの最後のスライドを参照:
  - 影の投射に使うカメラの**FOV (field of view)**を決定する方法
  - 参照先キューブマップを動的に構築する方法

強い全方向影処理

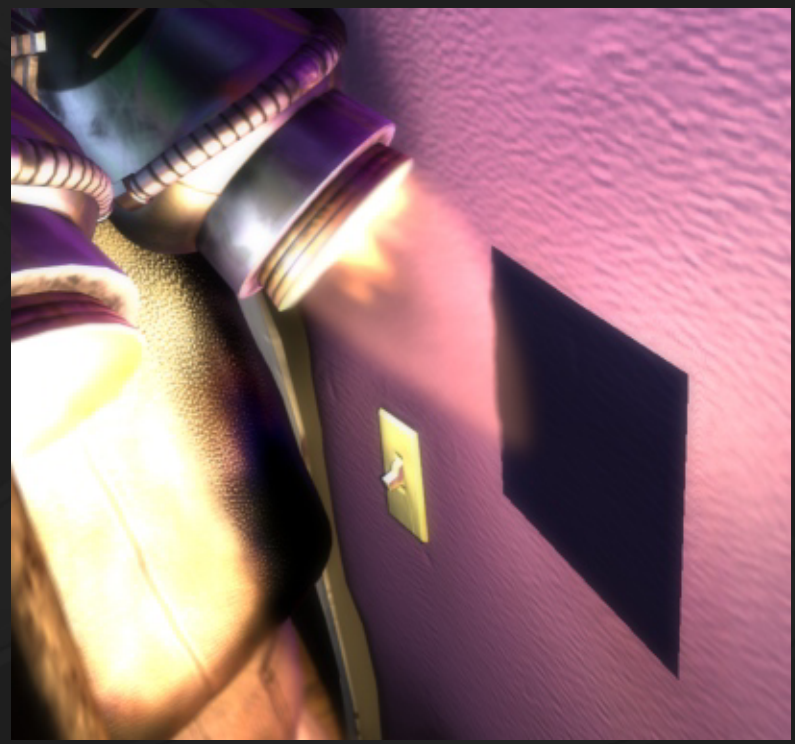
- フラグメント・シェーダで影を受ける処理をする:
  - **1**回参照に付き**6**インストラクション
  - **4**回参照に付き約**16**インストラクション
    - **[GeForce 7**シリーズなら簡単**]**

その他の問題:

- 影の中にないはずの点で大きな四角形がみえる
- 影投射の際のFOVをわずかに減少
  - 我々の場合FOV *= 0.989
- FOVを小さくしすぎると、仮想キューブマップ面の境界で、非連続性が見られる *(下図)*
- 影投射の際のポリゴン・オフセット調整も効果がある

近クリップ面

- 注意: 影を落とす物体を近クリップ面より光源に近づけてはいけない
- ...間違った結果 →
- 解決法: 近クリップ面までの距離を縮める



Mad Mod Mike

以前のデモ: Nalu

- 髪はメッシュからアルゴリズムで作成
- パーティクル・システムで髪を制御
- ワールド空間での物理演算
- 物理演算と静的な髪型の合成
  - 静的な髪型はローカルな空間で保持
  - 基本的な形の制御が可能

Naluの髪を改善

- Naluの髪は主に物理演算で形が決まった
  - ほとんど制御できず、髪型を作ることができない

新しいデモでの要求

- **Mad Mod Mike**は短い髭がある
  - もしくは肉的で弾力のある顔
- 髪型のある髪
  - アルゴリズムで作ったものでは駄目
  - **Luna:** 学生でヒーロー
  - **Mad Mod Mike**も髪型がある
- 両方で改善が必要
  - 髪型、柔軟性、制御
  - あまりパフォーマンスに悪影響なく

新しいデモでの要求

- **Mike**のバイク乗り的外見には髭が必要

新しいデモでの要求

Mikeの髭: 変化ある面上の毛

- **Mad Mod Mike**はブレンド・シェイプとスキニングを使用
  - ブレンド・シェイプ = モーフィング
  - スキニング = 骨格でメッシュを制御
  - 順序 = モーフィングの後、スキニング
- 皮膚表面の動きは単一の行列では表現できない
  - 問題: 皮膚にそって毛を動かさなければならない

Mikeの髭: 変化ある面上の毛

- 問題: 誘導毛をワールド空間で
- 毛のブレンド・シェイプやスキニングの情報なし
  - スキニングの重みなし
  - モーフィング対象なし
- 解決法: 誘導毛を固定された空間に
  - テクスチャ空間!

Mikeの髭: 変化ある面上の毛

- テクスチャ空間は一般的にバンプ・マップに使われる
  - 我々の場合、『表面空間』と呼んでも良い
- 表面の各点にひとつの座標系行列
  - 頂点ごとには既にあるかもしれない
- 各誘導毛には対応する座標系が存在
- 起動時に各誘導毛をテクスチャ座標系に変換

Mikeの髭: 変化ある面上の毛

- 各フレームですべての座標系を再計算
  - モーフィングの対象に座標系を埋め込んで、モーフィング、スキニングをするのが簡単
- 座標系を行列として使い、誘導毛をワールド空間で変形
- 変化する面上の毛を表現!

髪型のある髪

- 水中の髪には理由があった
  - 美しい
  - 髪型がない ☺
  - しかしこれだけでは…
- 髪型を作るにはツールが必要
  - **Dawn**と**Dusk**では専用のツールを使った
    - デザイナの作業に統合されていない
  - 『**Shave and Haircut**』は**Maya**のプラグイン
    - デザイナにも使用できる
    - ツールで髪をいじることができる

髪型のある髪

- 制御点の段階でスタイルを作ることができる
- **Shave & Haircut**は**NURBS**曲面を作成
  - **NURBS**の情報は破棄される
  - 簡単な直線で充分
    - アプリケーションが**Bezier**曲線を作成
- 制御点はテキスト・ファイルで出力
- それ以外にも…

髪型 – その他の制御

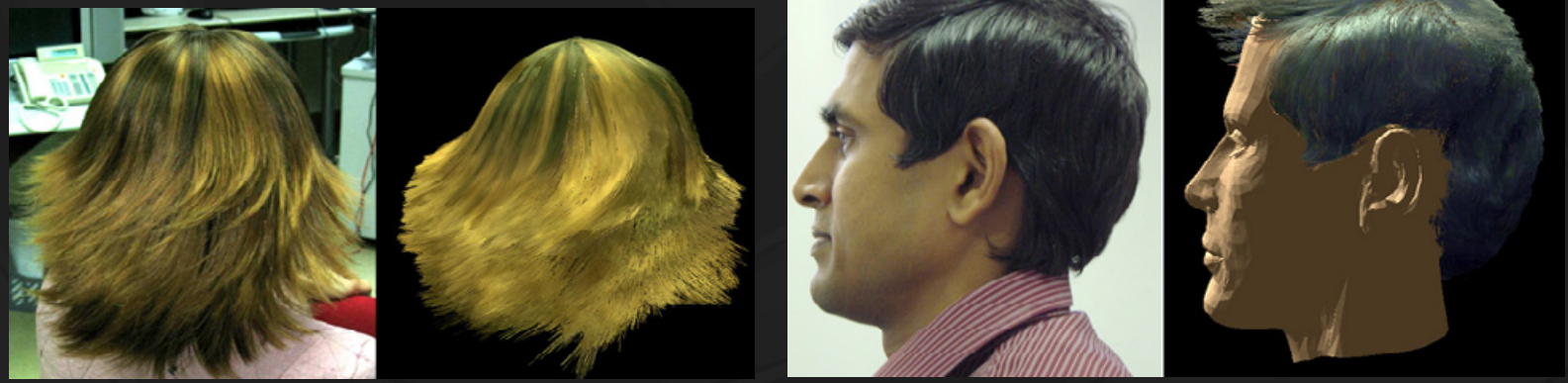
- 頭皮にテクスチャとして情報を保存
  - 密度
    - 髪の束ひとつごとの補間された髪の数
  - 色
  - 中断
    - 補間された髪が描画されないように
      - 処理はされる
  - 長さ
    - 長さの制御
    - アルゴリズムでの使用に便利(乱数など…)

髪型 – その他の制御

- これらの制御で、リソースがより効率的に使用できる: 制御髪と頂点
- 大事な部分により多くの髪
  - 生え際**(Luna)**
  - 髭など**(Mad Mod Mike)**
- 見えない部分の処理を減らす
  - 頭頂**(Luna)**
  - ヘルメットの下**(Mad Mod Mike)**

その他の髪型技術

- 自然な髪のメッシュを作成
  - **“Modeling Hair from Multiple Views”**
  - 髪のメッシュ(直線、曲線)を写真から取得



Microsoft Research Asia and the Hong Kong University of Science & Technology
Eyal Ofek, Yichen Wei, Long Quan, Heung-Yeung Sum

その他の髪型技術

- 頂点ごとの色指定!
- (+) 簡単に取得
- (+) 作業が少ない
- (+) ある程度正確
- (-) 見えない部分は作成できない
- (-) カールのあるスタイルなどは難しい

成功

- 変化のある表面上の髪
- 髪型があるほうが良い
  - しかしまだ難しい
- その他の制御を使用
  - 表面上で均一ではない髪質
  - リソースの効率的使用

- まだ充分ではない!
  - 充分にはならないだろう ☺

今後の改善点

- 髪型はまだまだ難しい
- 髪の動力学、評価、分割をより高速に
  - **CPU**では遅すぎるので、**GPU**へ作業を移す
- リソースの使用
  - より良い**LOD**の使用
    - 距離、ビューポートを使う
- 彩色の改良
- より良い物理演算**(**摩擦など**...)**
- その他**...**

参照

- 光源処理
  - **Steve Marschner**
- 影処理
  - **Opacity shadow maps**
- 髪型
  - **Shave & Haircut**
- 髪型
  - **Modeling hair from multiple views**
- 文献
  - **GPU Gems 2**
    http://developer.nvidia.com/GPUGems2/