

ゲームグラフィックス特論

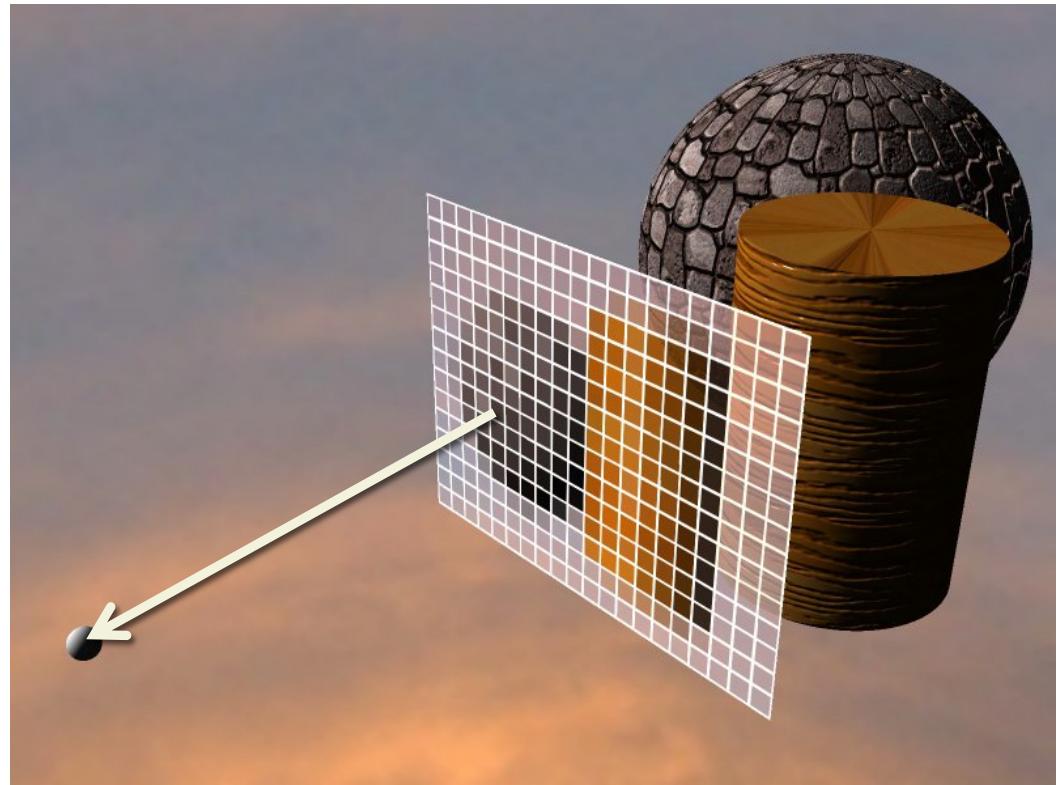
第11回 影

レンダリング方程式

レンダリングの完全なモデル化

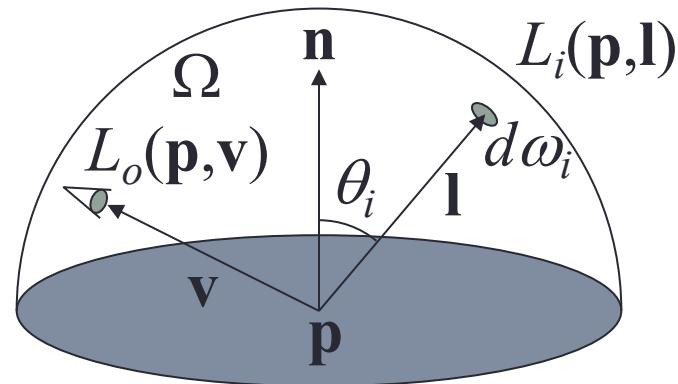
レンダリング

- ・スクリーン上の1点を通して視点に届く光の強さを求める
 - ・陰影付け
 - ・隠面消去処理
 - ・影付け処理
 - ・映り込み
 - ・透過・屈折
 - ・...
- ・隠面消去処理
 - ・不透明の物体に対して光の反射位置を求める
 - ・物体が半透明なら
 - ・ボリュームレンダリング



反射方程式

- $L_i(\mathbf{p}, \mathbf{l})$: 面上の点 \mathbf{p} における \mathbf{l} 方向から入射する放射輝度
- $f(\mathbf{l}, \mathbf{v})$: \mathbf{l} 方向からの入射光が \mathbf{v} 方向に反射する際の BRDF
- $L_o(\mathbf{p}, \mathbf{v})$: 面上の点 \mathbf{p} から視点方法 \mathbf{v} に向かう放射輝度



視点に向かう
光の放射輝度

$$L_o(\mathbf{p}, \mathbf{v}) = \int_{\Omega} f(\mathbf{l}, \mathbf{v}) \otimes L_i(\mathbf{p}, \mathbf{l}) \cos \theta_i d\omega_i$$

レンダリング方程式

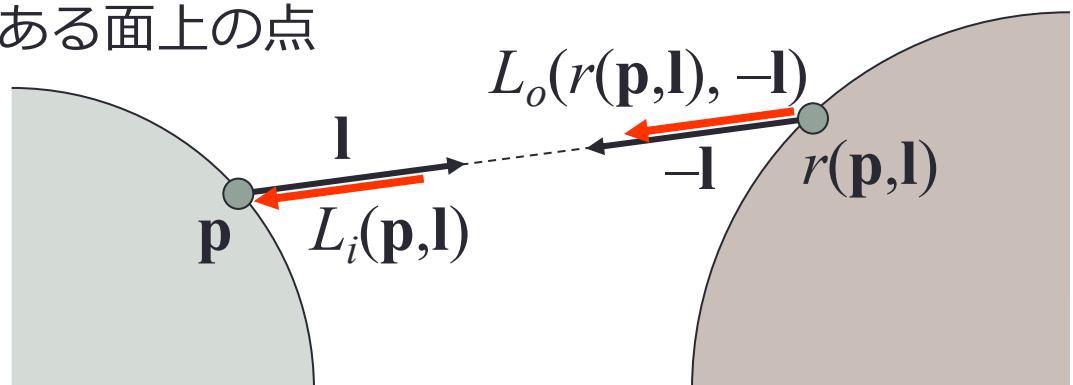
- Kajiya のレンダリング方程式の別形式

$$L_o(\mathbf{p}, \mathbf{v}) = L_e(\mathbf{p}, \mathbf{v}) + \int_{\Omega} f(\mathbf{l}, \mathbf{v}) \otimes L_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l}) \cos \theta_i d\omega_i$$

- $L_e(\mathbf{p}, \mathbf{v})$: 面上の点 \mathbf{p} から視点方法 \mathbf{v} に向かう自己放射輝度
- 面上の点 \mathbf{p} における \mathbf{l} 方向からの入射光の放射輝度

$$L_i(\mathbf{p}, \mathbf{l}) = L_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l})$$

- 他の点の \mathbf{l} の逆方向 $-\mathbf{l}$ に出発する放射輝度に等しい
- $r(\mathbf{p}, \mathbf{l})$: \mathbf{p} から \mathbf{l} 方向にある面上の点



レンダリング方程式の意味

- 点 p の陰影

- その点の自己放射輝度 L_e と反射光の放射輝度 $L_o(p, v)$ の和
- 反射光は点 p から視点方向 v に向かう
- 反射光の放射輝度 $L_o(p, v)$ は, その点に入射する別の点の放射輝度 $L_o(r(p, l), -l)$ によって決まる

- 再帰的

- ある点の放射輝度
 - そこに入射する他の点の放射輝度が決まらないと決まらない
- 他の点の放射輝度
 - 更に他の点の放射輝度が決まらないと決まらない
 - 更に他の点がめぐりめぐってもとの点だったりする



レンダリング方程式の解法

- レンダリング方式を厳密に解くことは困難
 - 計算が終わらない
- 何らかの近似を行う
 - 局所照明モデル
 - 直接光しか取り扱わない
 - 光源と受光面だけが陰影計算（照明計算）に関与する
 - 間接光は環境光として定数で表現する
 - リアリティの低下を招く
 - 大域照明モデル
 - 間接光も取り扱う
 - 陰影計算に光源と受光面以外の存在も考慮する
 - リアリティが向上する
 - 計算時間が長くなる
 - リアリティとのトレードオフは他に何を考慮するかで決まる

近似のレベル

影

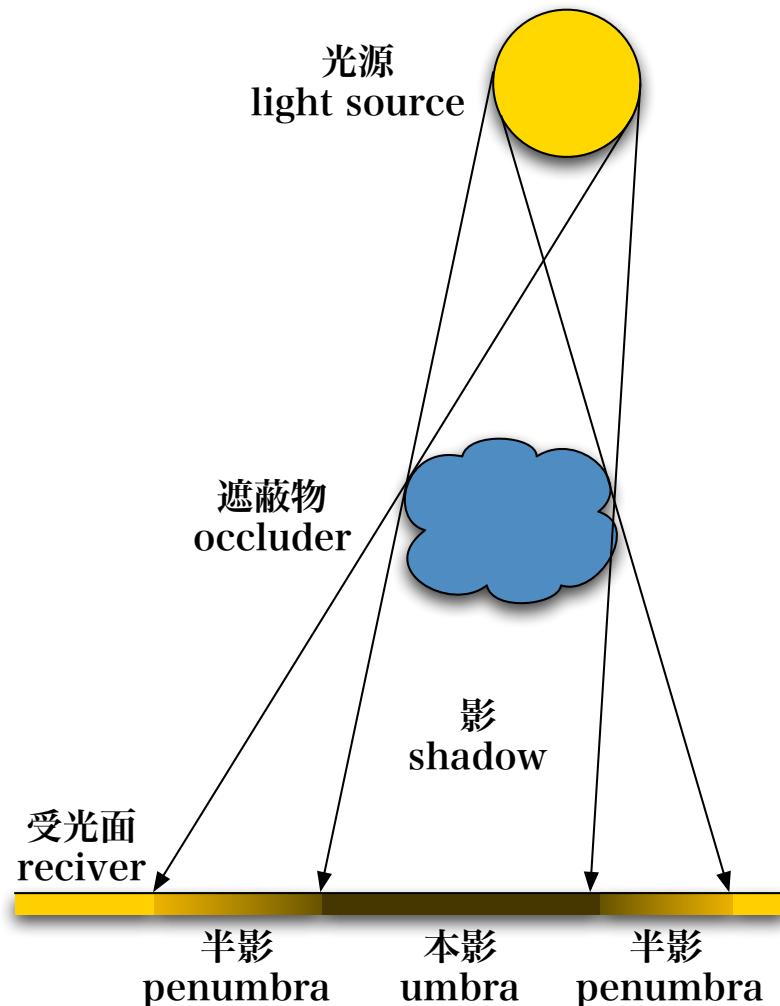
第三の物体による影響

影

- 影の意味
 - 光源と受光面の他に影を落とす遮蔽物 (occluder) の存在を考慮する
 - 影の効果
 - シーンのリアリティの向上
 - 観測者が物体の配置を知覚するための手がかり
 - 影の生成手法
 - 非常に多くの手法が存在する
- ↓
- 決定的な手法が存在しない



影処理に関する用語



- 本影 (umbra)
 - 光源からの光がまったく届かない領域
- 半影 (penumbra)
 - 面積を持った光源の一部が遮られた領域

ハードシャドウとソフトシャドウ

ハードシャドウ

点光源によるシャープな影



ソフトシャドウ

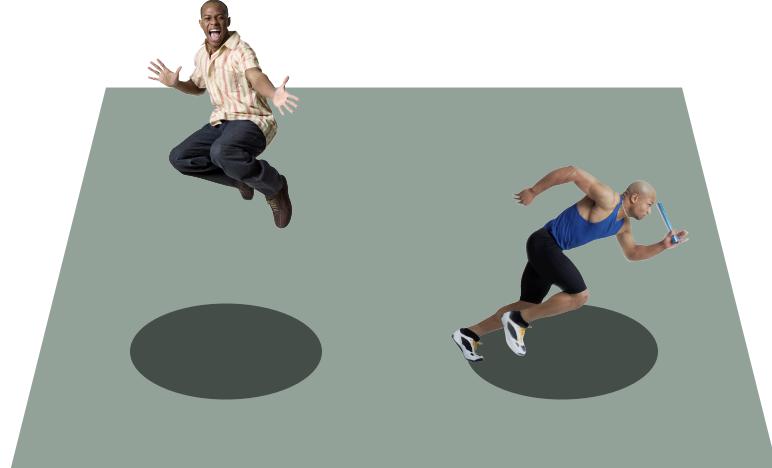
面積を持つ光源によるソフトな影



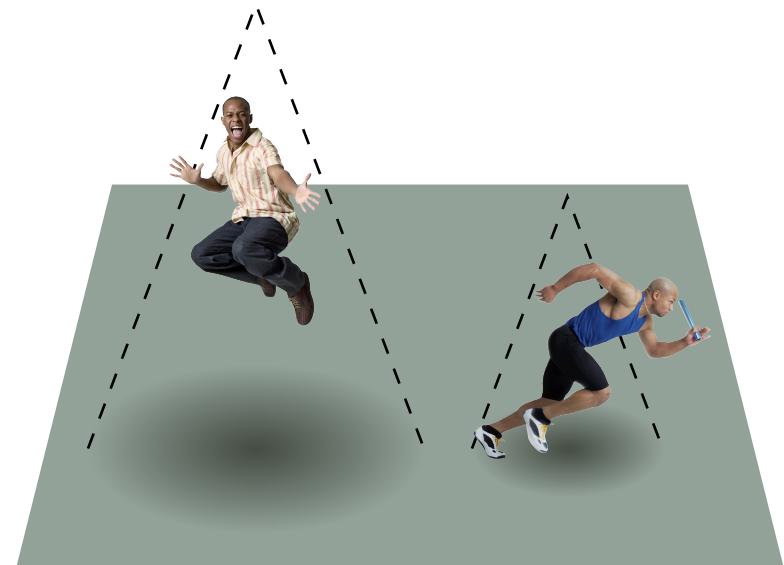
(実は点光源をいっぱい置いた)

丸影

- 遮蔽物の真下の地面に円を描く方向
 - 影の形に遮蔽物の形は反映されない
 - 受光面は平面のみ



円形のオブジェクトを描く



投影テクスチャマッピング

Projection Shadows

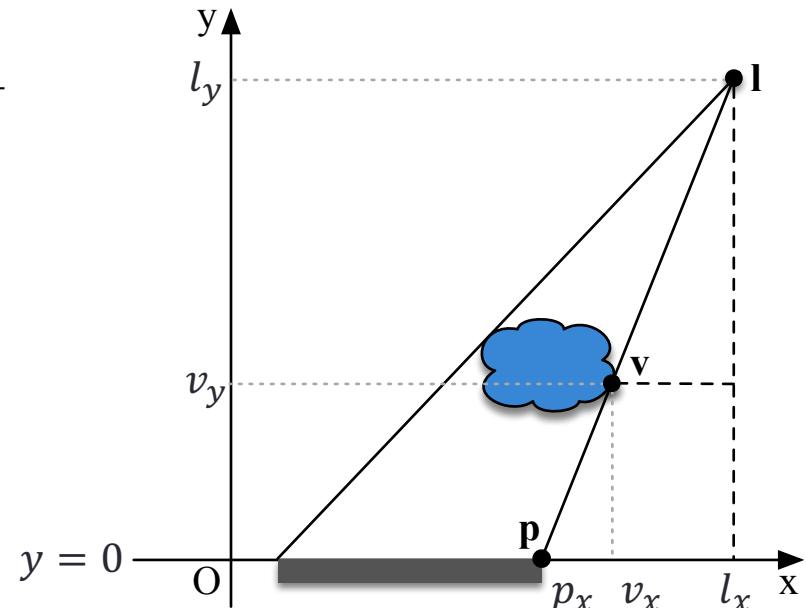
- 受光面が $y = 0$ の平面のとき

$$\frac{p_x - l_x}{v_x - l_x} = \frac{l_y}{l_y - v_y} \Leftrightarrow p_x = \frac{l_y v_x - l_x v_y}{l_y - v_y}$$

- Z軸についても同様にして求めて

$$M = \begin{pmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{pmatrix}$$

- ステップ1：
 - オブジェクトを通常の方法で投影する
- ステップ2：
 - 同じオブジェクトを M を使って投影する



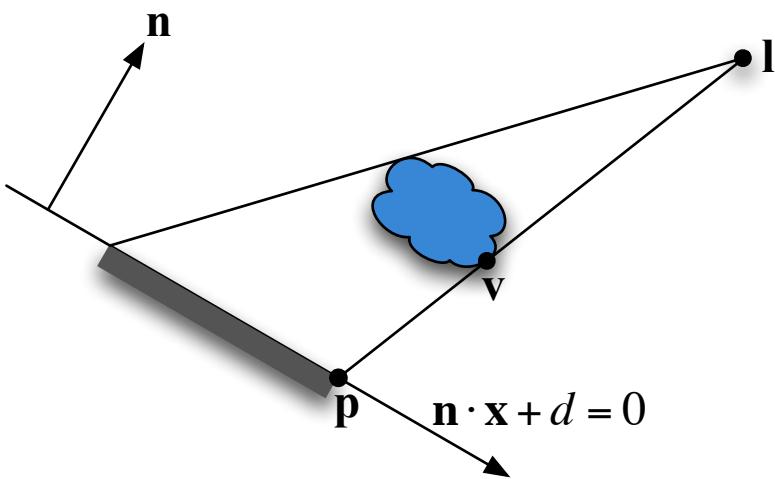
一般の平面への Projection Shadows

- 一般的な平面において

$$p = l - \frac{d + \mathbf{n} \cdot \mathbf{l}}{\mathbf{n} \cdot (\mathbf{v} - \mathbf{l})} (\mathbf{v} - \mathbf{l})$$

↓

$$\mathbf{M} = \begin{pmatrix} d + \mathbf{n} \cdot \mathbf{l} - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & d + \mathbf{n} \cdot \mathbf{l} - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & d + \mathbf{n} \cdot \mathbf{l} - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & \mathbf{n} \cdot \mathbf{l} \end{pmatrix}$$

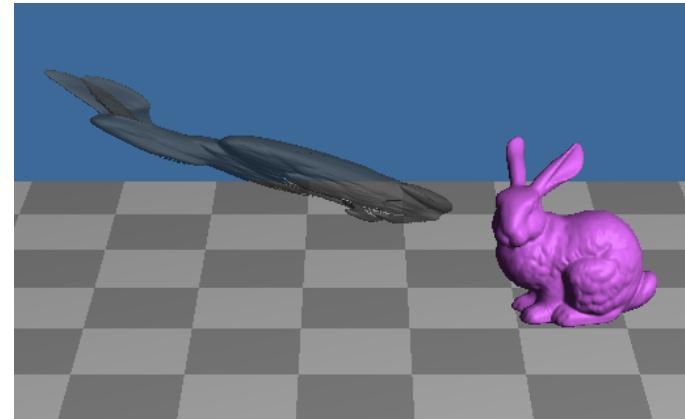


受光面のポリゴンをレンダリングしたあと影のポリゴンを（同じ位置に）重ねてレンダリングする際、影のポリゴンが受光面の下にならないよう工夫する必要がある

ポリゴンオフセット等

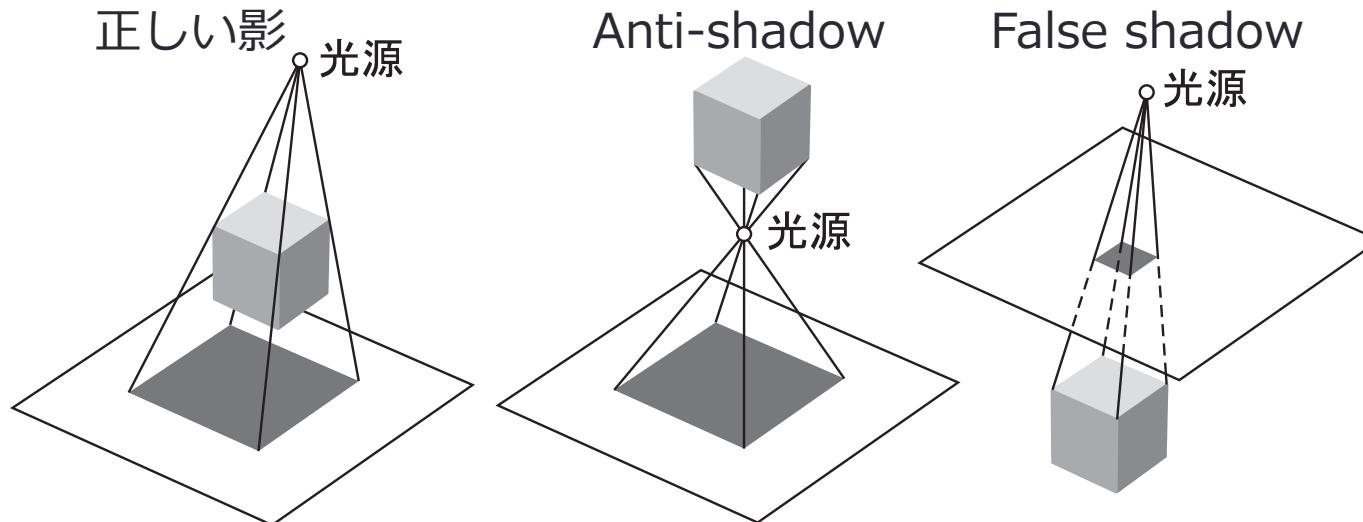
Projection Shadows の欠点

- ・反射（映り込み）と衝突してしまう
 - ・影のポリゴンにも映り込みの処理が必要
- ・影が受光面からはみ出てしまう
 - ・ステンシルバッファを使って削り取る
- ・不透明の物体の影しか処理できない
 - ・半透明の物体を取り扱うには特別な処理が必要
 - ・凸形状の物体なら影のポリゴンは常に2枚重なる
 - ・背面ポリゴンを除去するなら1枚
 - ・凹部をもつ物体はこの性質が保障されない
 - ・半透明の影がおかしくなる
 - ・ステンシルバッファを使って各ポリゴンを1回だけ描くようにする



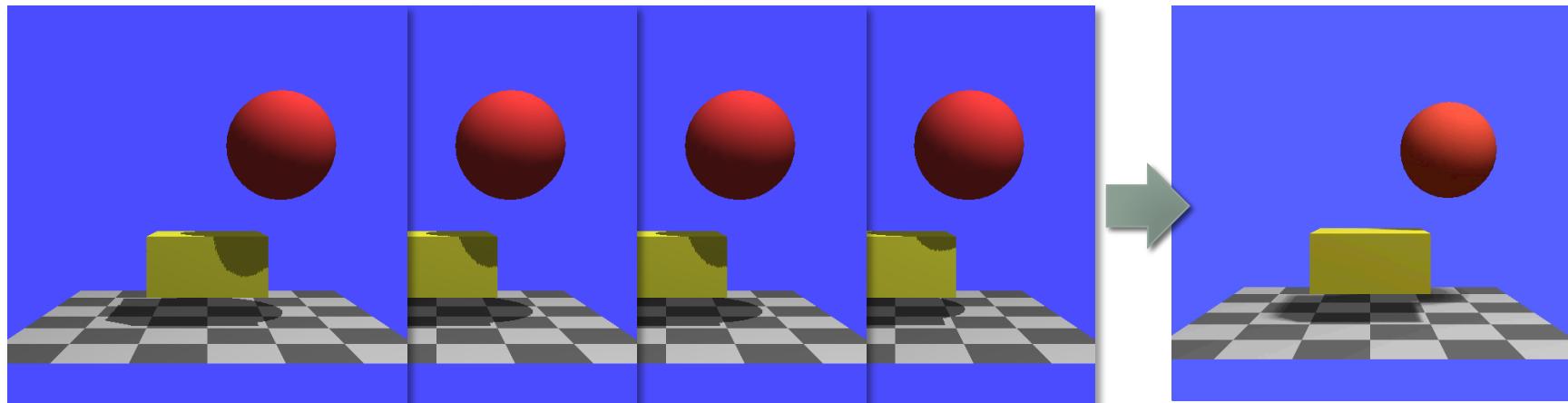
Antishadow と False shadow

- Anti-shadow
 - 光源と遮蔽物の反対側の面に影が落ちてしまう
- False Shadow
 - 遮蔽物が受光面の反対側にあるときにも影が落ちてしまう
- いずれも光源と受光面を含む空間をクリッピングして対処



ソフトシャドウ

- ・ソフトシャドウ は光源が面積を持っているときに発生する
- ・光源の領域内に複数の点光源を置いて近似することができる
 - ・個々の点光源による影をアキュムレーションバッファに積算し、平均を求めればソフトシャドウが得られる
 - ・ハードシャドウ を求めるいかなるアルゴリズムを用いても、この手法により半影を生成することが可能
 - ・メモリの制限などにより一般的な実装は難しい



アキュムレーションバッファ

- カラーバッファの内容を累積する別のバッファ
 - レンダリング後にカラーバッファの内容を加算する
 - 複数のレンダリング結果の合計を得ることができる
 - 値をシフトした結果をカラーバッファに書き戻すことができる
 - 例えばカラーバッファの内容を 4 回加算した結果を 2 ビット右にシフト（4 で割ったことと同じ）して書き戻せば 4 回のレンダリング結果の平均が得られる
- 古い機能
 - 現在は非推奨 (deprecated) となっている
 - GL_RGB32F / GL_RGBA32F などの浮動小数点テクスチャをカラーバッファに使って FBO (Frame Buffer Object, 13回目に説明予定) を作成すれば同じことができる

Heckbert と Herf のアルゴリズム

$$\mathbf{e}_w = \mathbf{b} - \mathbf{a}$$

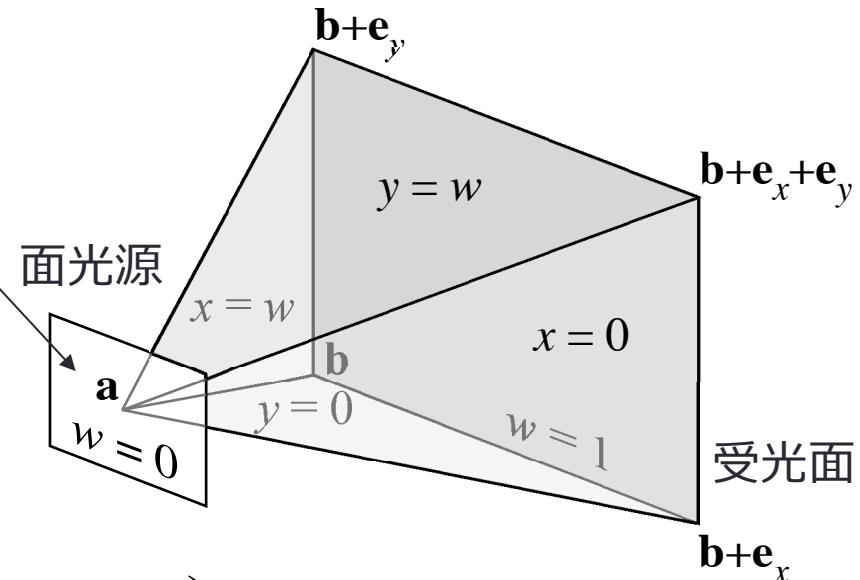
$$\mathbf{n}_u = \mathbf{e}_w \times \mathbf{e}_y \quad q_u = 1 / \mathbf{n}_u \cdot \mathbf{e}_x$$

$$\mathbf{n}_v = \mathbf{e}_x \times \mathbf{e}_w \quad q_v = 1 / \mathbf{n}_v \cdot \mathbf{e}_y$$

$$\mathbf{n}_w = \mathbf{e}_y \times \mathbf{e}_x \quad q_w = 1 / \mathbf{n}_w \cdot \mathbf{e}_w$$

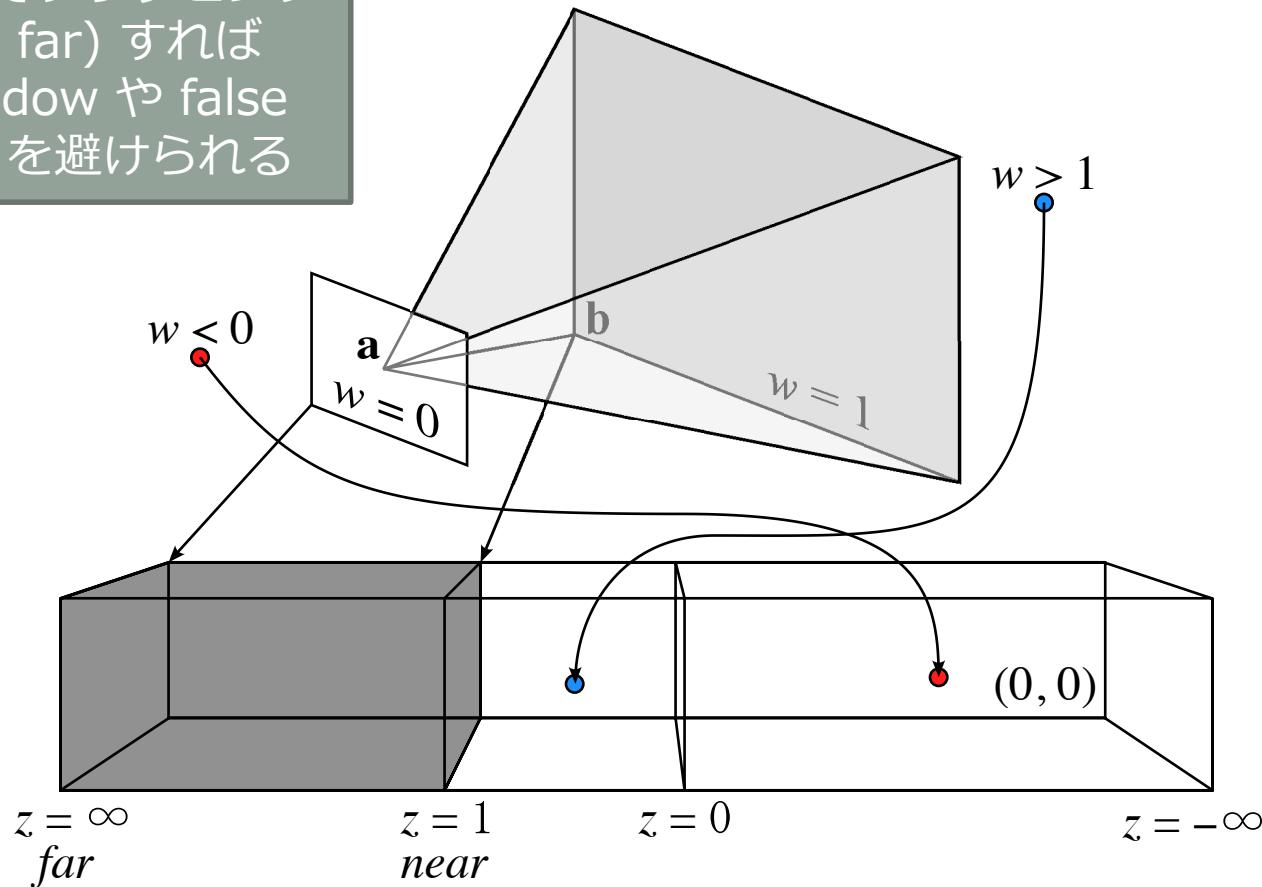
$$\mathbf{M} = \begin{pmatrix} q_u n_{ux} & q_u n_{uy} & q_u n_{uz} & -q_u \mathbf{n}_u \cdot \mathbf{b} \\ q_u n_{vx} & q_u n_{vy} & q_u n_{vz} & -q_v \mathbf{n}_v \cdot \mathbf{b} \\ 0 & 0 & 0 & 1 \\ q_w n_{wx} & q_w n_{wy} & q_w n_{wz} & -q_w \mathbf{n}_w \cdot \mathbf{a} \end{pmatrix}$$

面光源上の点光源



Heckbert と Herf のアルゴリズムの特徴

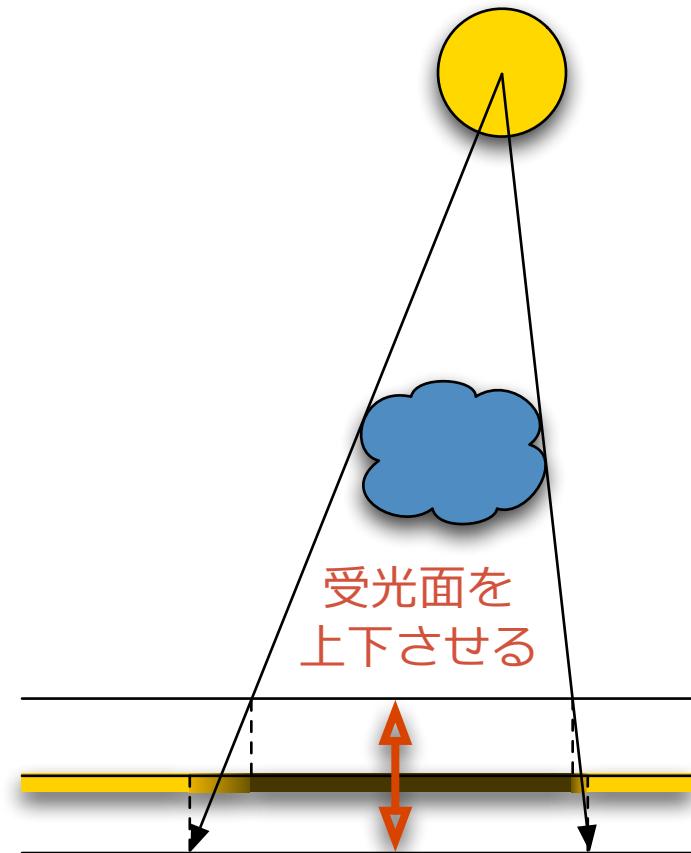
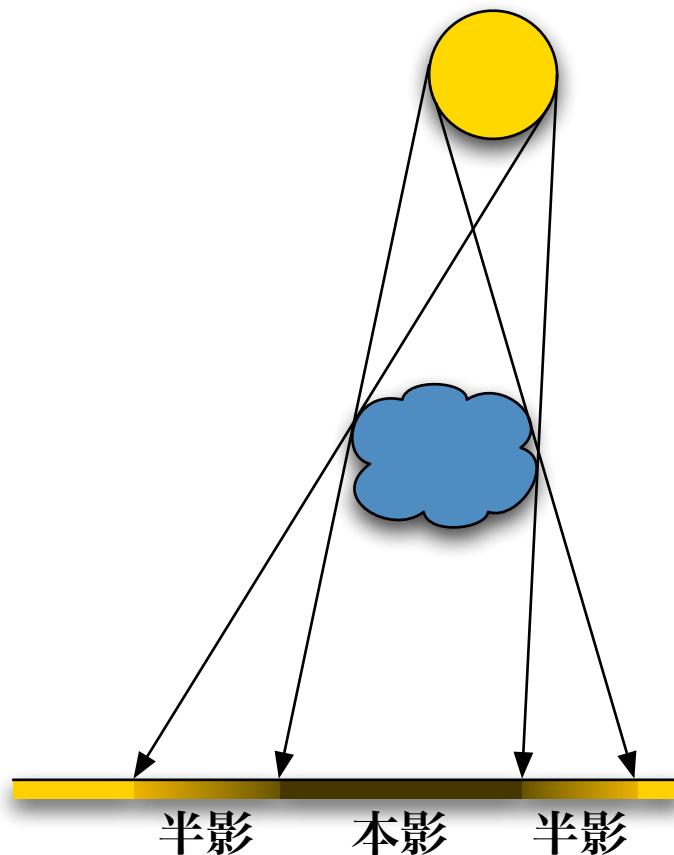
Z軸の値でクリッピング
(near と far) すれば
anti-shadow や false
shadow を避けられる



Heckbert と Herf のアルゴリズムの手順

1. 光源上の個々のサンプル点を点光源に用いる
2. 最初に受光面をひとつのサンプル点でレンダリングする
3. M を使って四角錐内にある物体をすべてレンダリングする
 - 影なので黒色でレンダリングする
 - デプスバッファ, テクスチャリング, ライティングは行わない
4. 影の画像はキュムレーションバッファに積算する
5. 最後に平均値を求めれば影のテクスチャが得られる

Gooch らのアルゴリズム

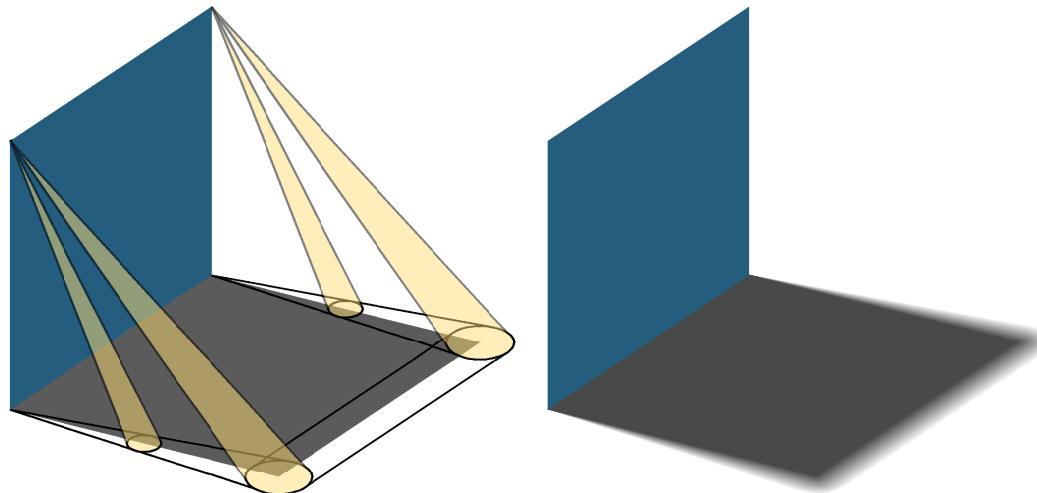


Gooch らのアルゴリズムの特徴

- 生成される影が入れ子になっている
 - 一般的に見かけがよいので少ないサンプル数ですむ
- 物体が受光面と接しているときは影を正しく再現できない
 - 「暗さ」が物体の下から外にはみ出てしまう
 - 低い面だけで影のレンダリングや平均を求めれば解消される
 - 影のリアリティは損なわれるが、はみ出しありは阻止できる

Hains の方法

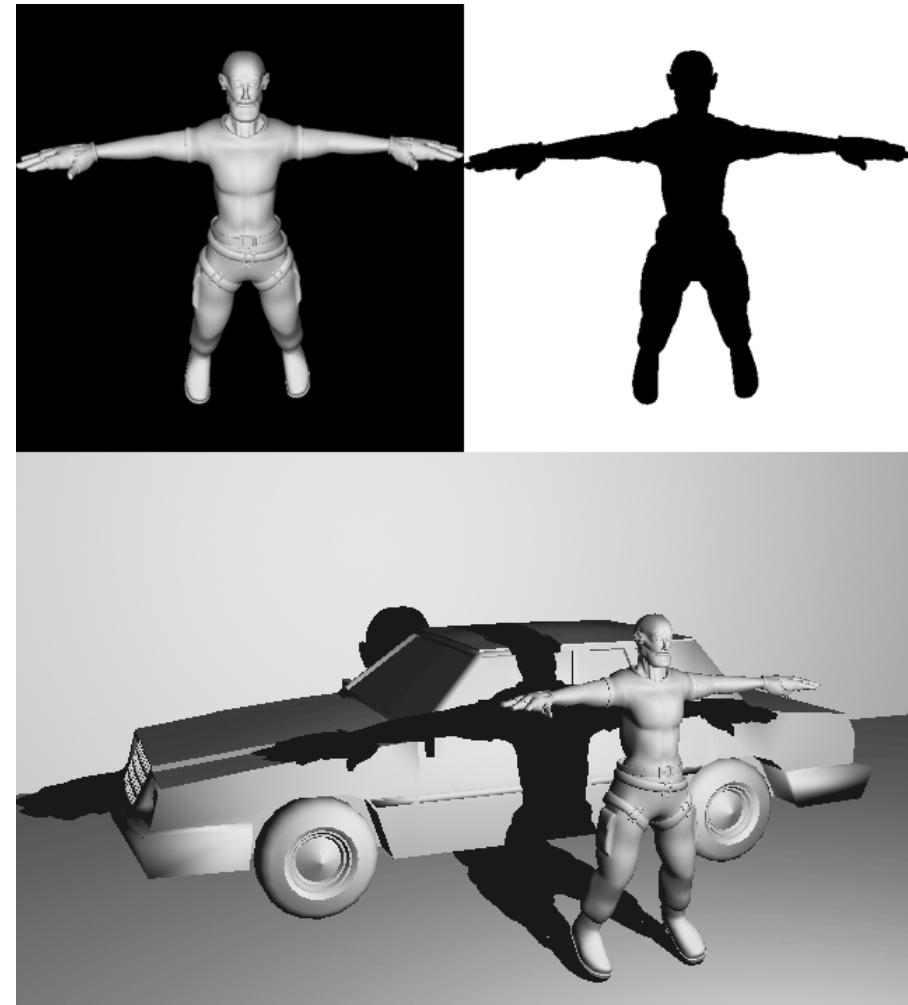
- 円形のエリアライトを対象にして 1 パスでソフトシャドウを生成する
 - ハードシャドウと同様な手法で求めた影の外形（シルエットエッジ）に沿って、中央から外周に向かってグラデーションをつけた円を描く



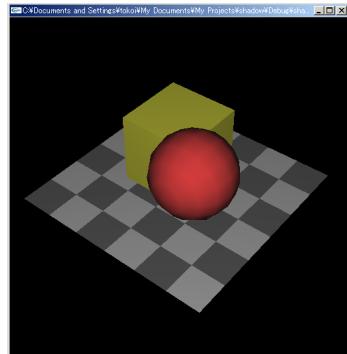
曲面に落ちる影

- シャドウテクスチャ

- 平面に落ちる影を求めてテクスチャとして物体表面に投影する
- 平面に落ちる影は、光源の位置から遮蔽物をレンダリングして、そのシルエットを求めるべき
- 何が遮蔽物で何が受光面なのかをモデリング時に決めておかなければならぬ

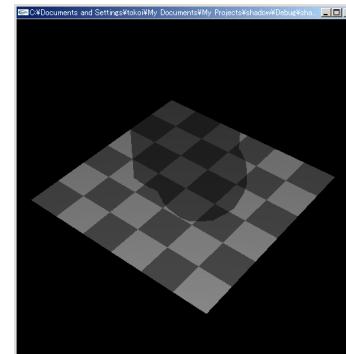
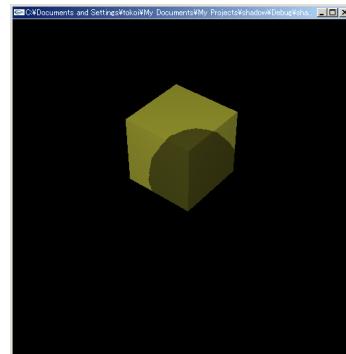
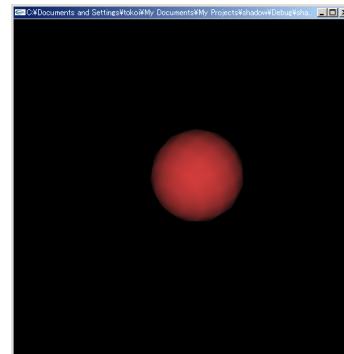
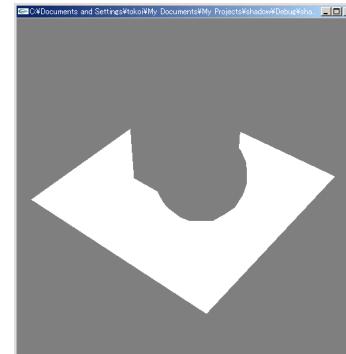
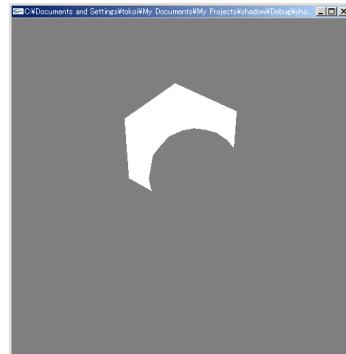
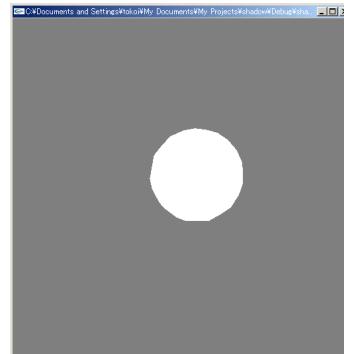


シャドウテクスチャの手順



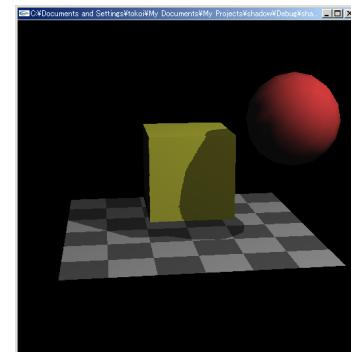
光源位置から
見たシーン

シルエット画像

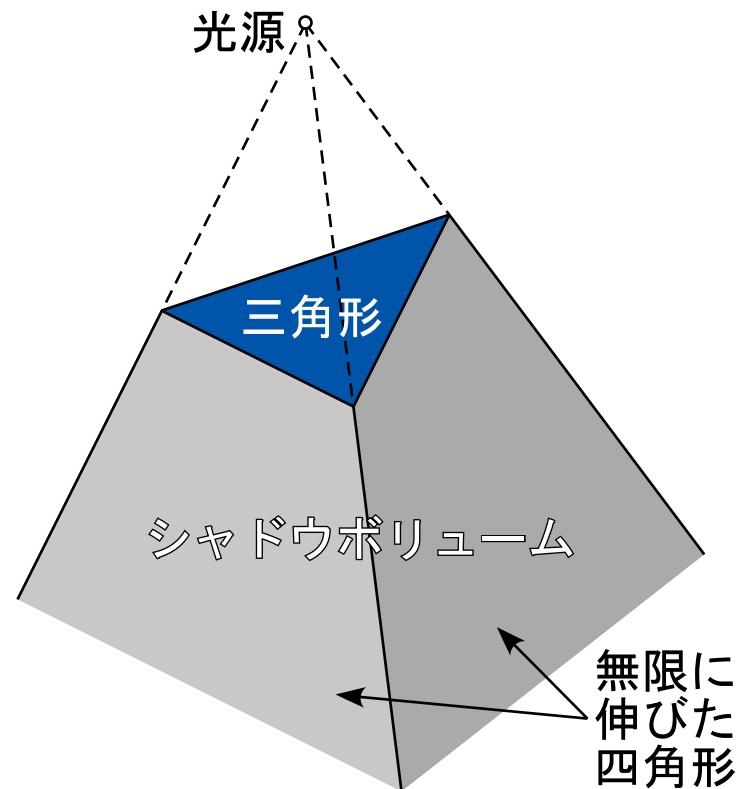


シルエット画像を物体にマッピング

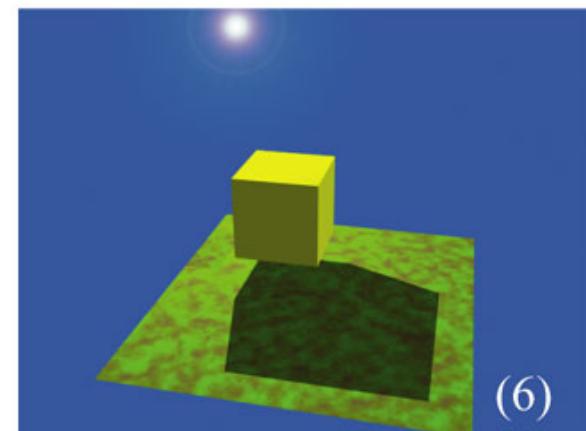
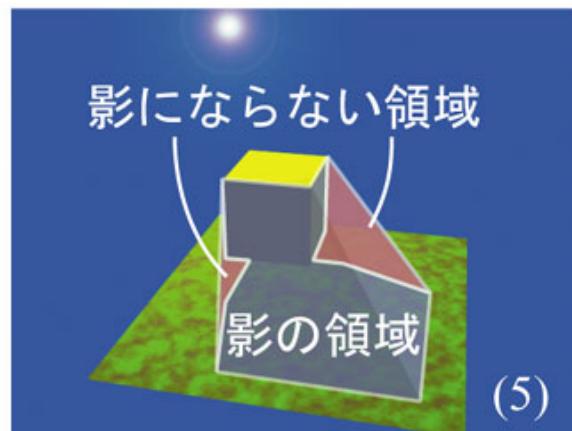
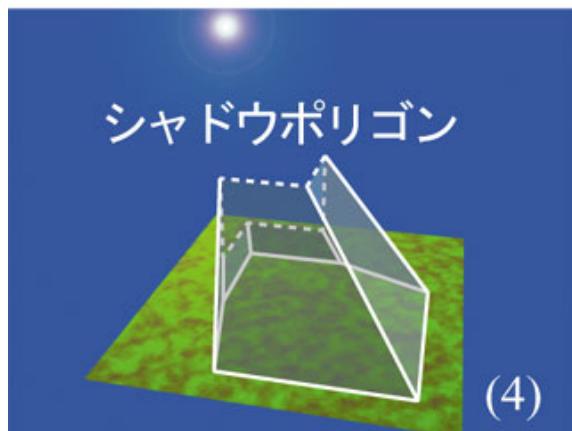
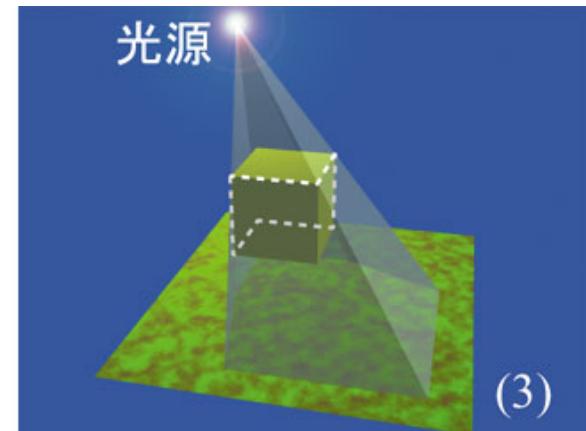
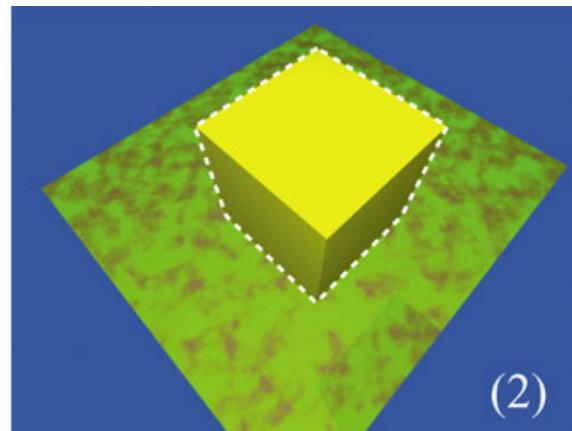
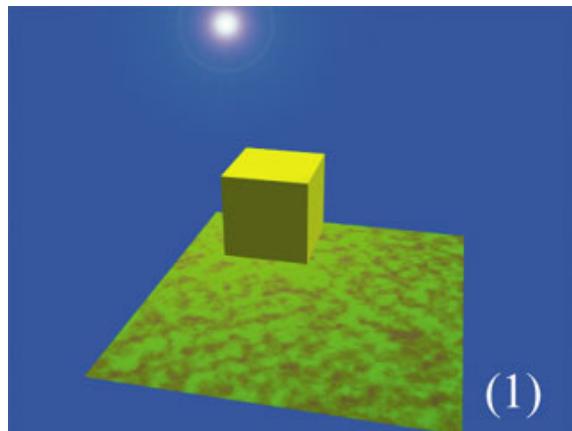
完成シーン



シャドウボリュームによる影の生成



シャドウボリュームによる影の生成手順



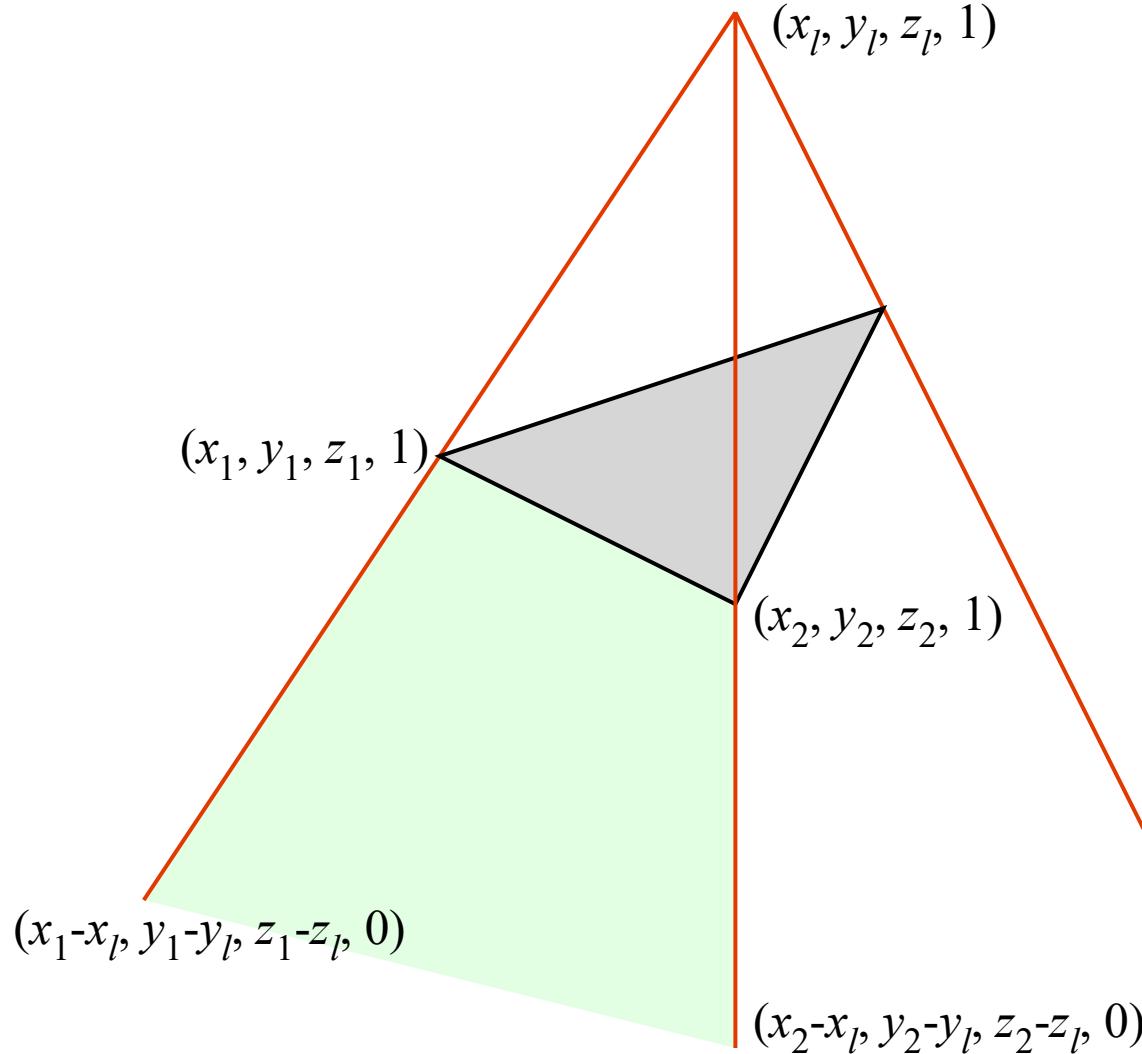
ステンシルバッファを使う

- ・フレームバッファを構成するもう一つのバッファ
 - ・カラーバッファやデプスバッファと重なっている
- ・フラグメントごとに描画回数をカウントできる
 - ・フラグメント単位に値を設定することができる
 - ・デプステストの結果によって異なる値を設定できる
- ・フラグメントごとに描画の可否を制御できる
 - ・設定された値をもとにフラグメントの表示・非表示を制御できる
 - ・描画するポリゴンの「型抜き」ができる

シャドウボリュームによる影付け手順

1. シャドウポリゴンを生成する
2. 光が当たっていない状態でシーンを描画する
3. シャドウポリゴン描画時の設定を行う
4. 視点側を向いたシャドウポリゴンを描画する
5. 視点と反対側を向いたシャドウポリゴンを描画する
6. 光が当たった状態でシーンを描画する

シャドウポリゴンを生成する



光が当たっていない状態で描画する

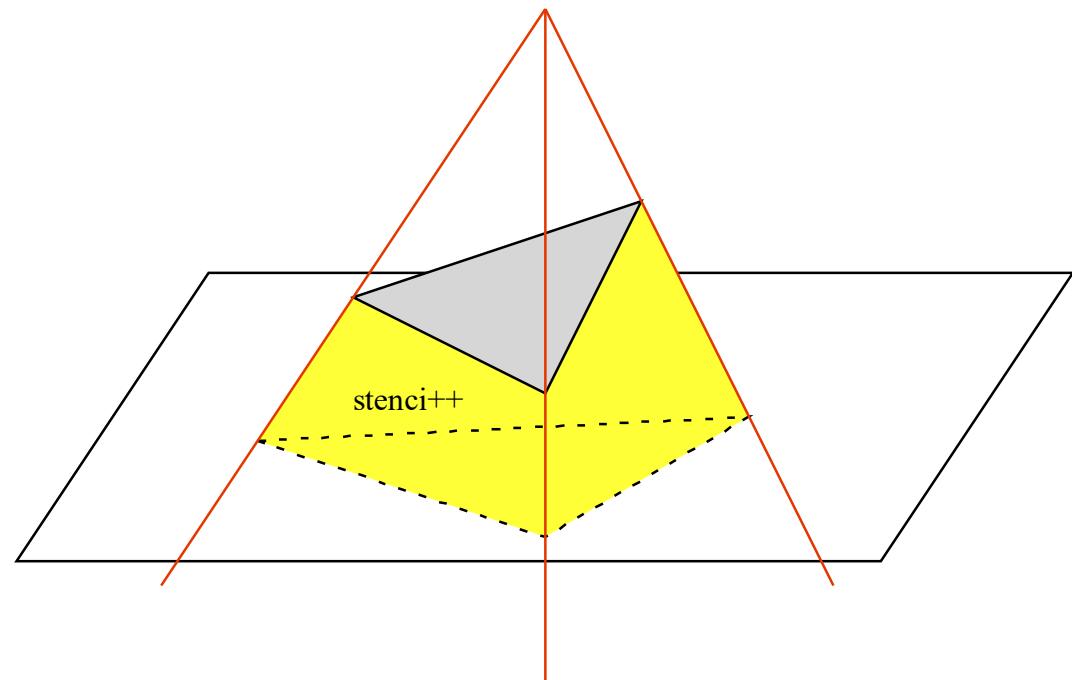
- 光源強度の拡散反射光成分と鏡面反射光成分を0にしてレンダリングする
 - カラーバッファに光源の光が当たっていないシーンの画像を得る
 - デプスバッファにシーンのデプス値を得る

シャドウポリゴン描画時の設定

- ・シャドウポリゴンは表示しない
 - ・カラーバッファへの書き込みを禁止する
 - ・デプスバッファへの書き込みを禁止する
 - ・陰影付けは行わない
- ・ステンシルテストを有効にする
 - ・ステンシルテストは常に成功するようにしておく
 - ・デプステストが成功したときにステンシルバッファをインクリメントする

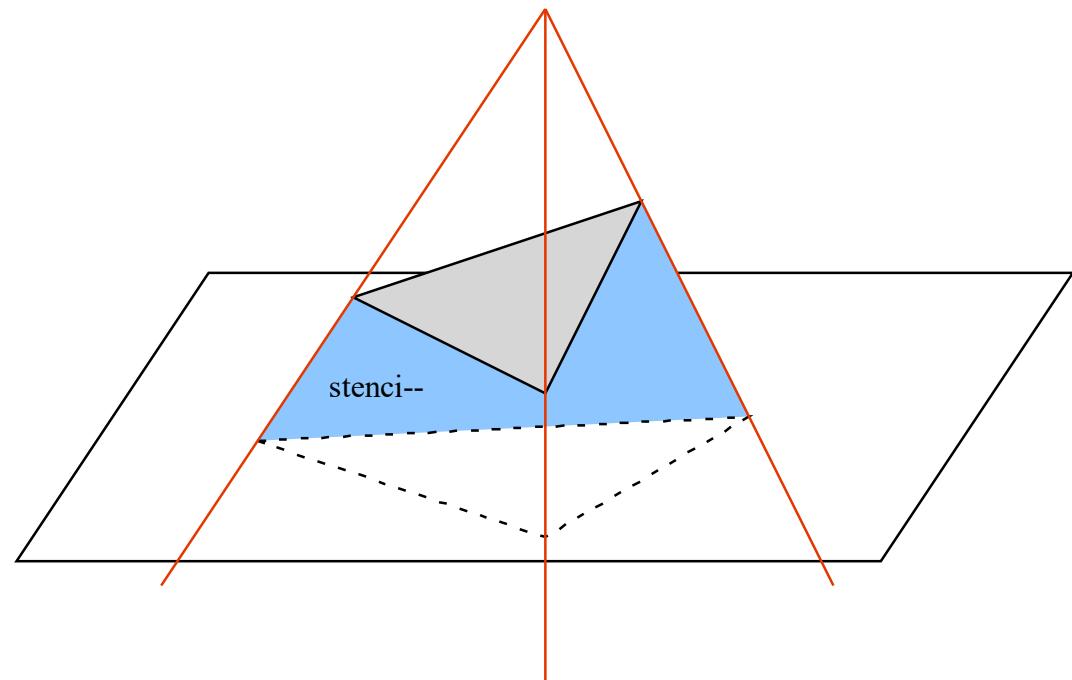
視点側を向いたシャドウポリゴンを描画

- ・デプステストが成功したときにステンシルバッファをインクリメントする



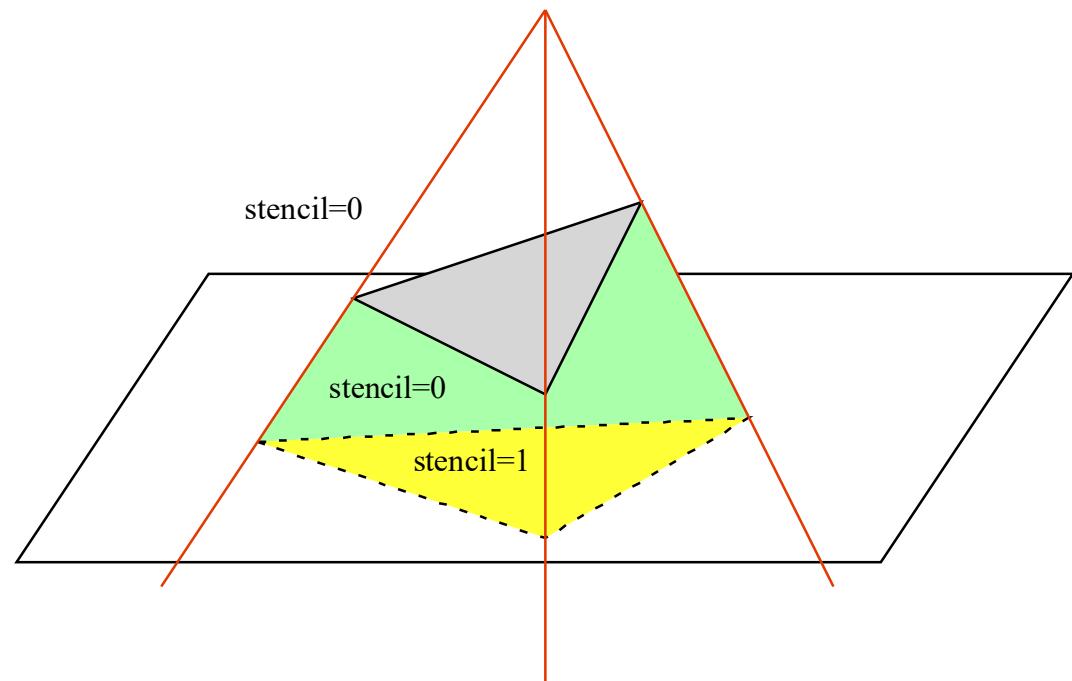
反対側を向いたシャドウポリゴンを描画

- ・デプステストが成功したときにステンシルバッファをデクリメントする



ステンシルバッファが0の部分を描画

- ・ステンシルバッファが0のときステンシルテストが成功する
ようにする



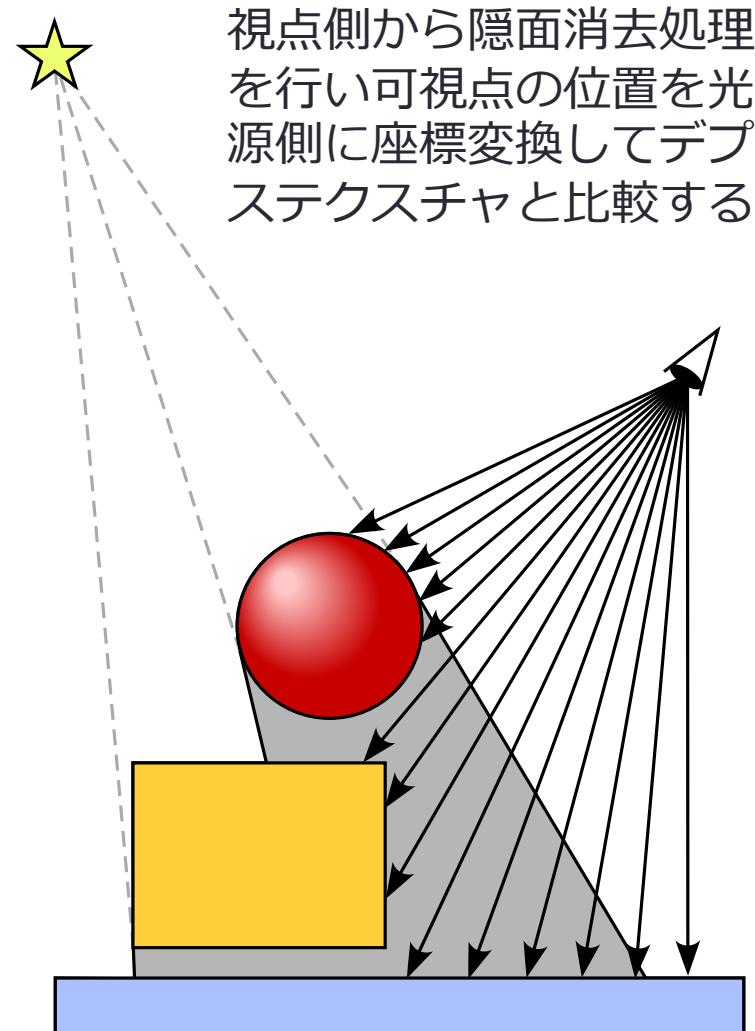
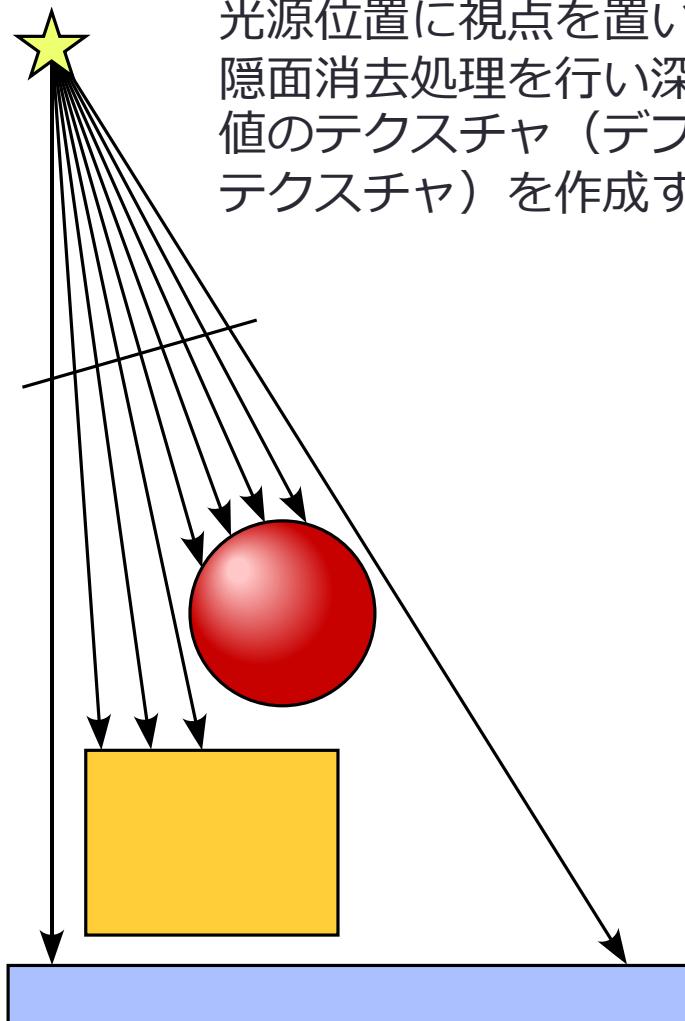
光が当たった状態でシーンを描画

- ・光源強度の拡散反射光成分と鏡面反射光成分を1にする
- ・ライティングを有効にする
- ・シーンをレンダリングする
 - ・カラーバッファのステンシルバッファが0の部分に光源の光が当たっているシーンの画像を得る

この方法の問題点

- オブジェクトのポリゴン数の3倍（オブジェクトが三角形で構成されている場合）のシャドウポリゴンをレンダリングしなければならない
 - シルエットエッジに対してのみシャドウポリゴンを生成する
- 視点がシャドウボリュームの中に入ったときに正しく処理ができない
 - ステンシルバッファを0ではなく視点を含むシャドウボリュームの数で初期化する
- 前方面がシャドウボリュームと交差するときに正しく処理できない
 - Z-fail 法 (Carmack の方法)

シャドウマップ法（シャドウマッピング）

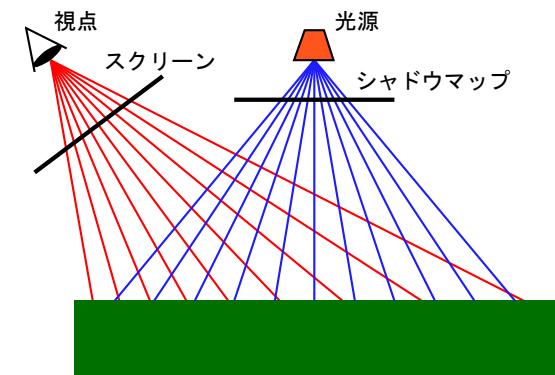
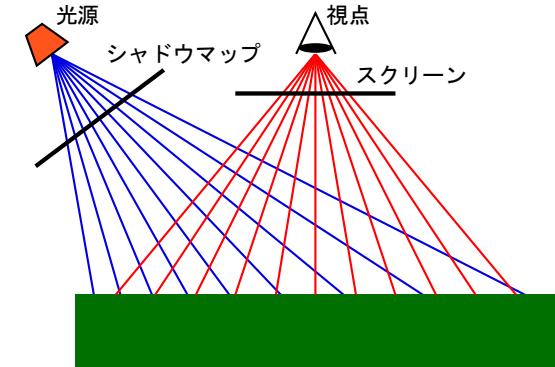
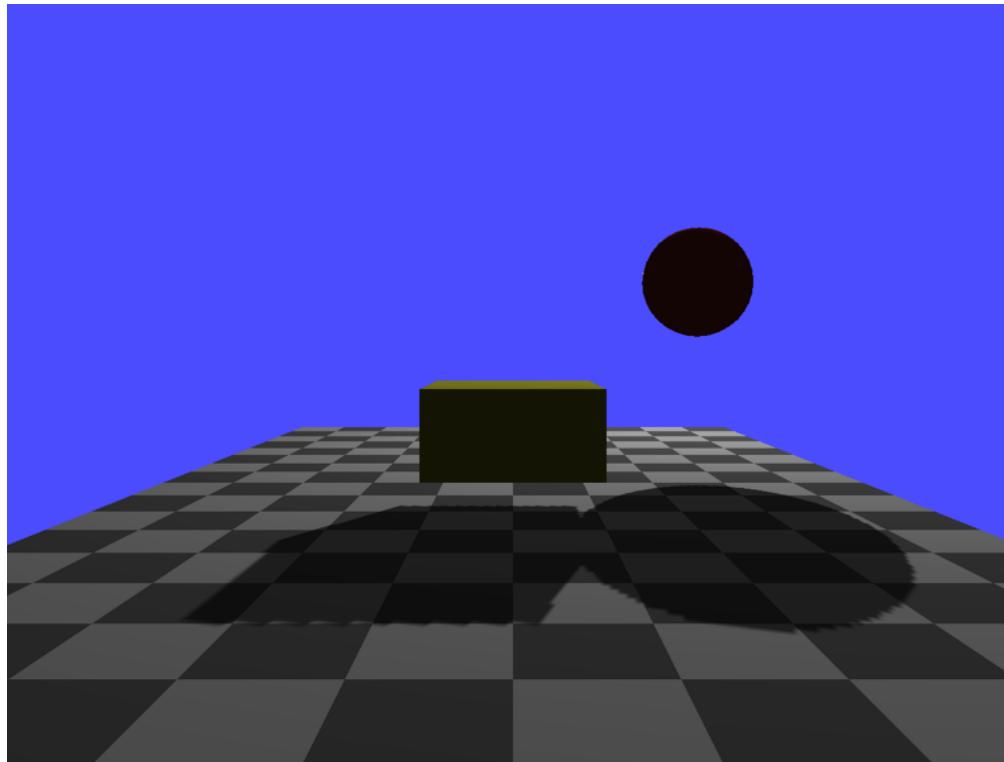


シャドウマップ法の手順 (宿題のヒントに実装例)

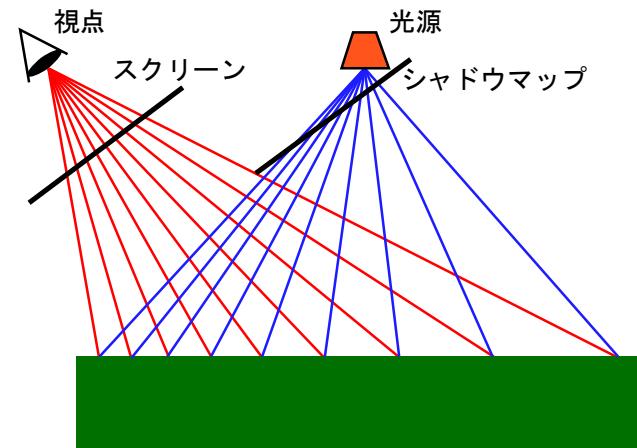
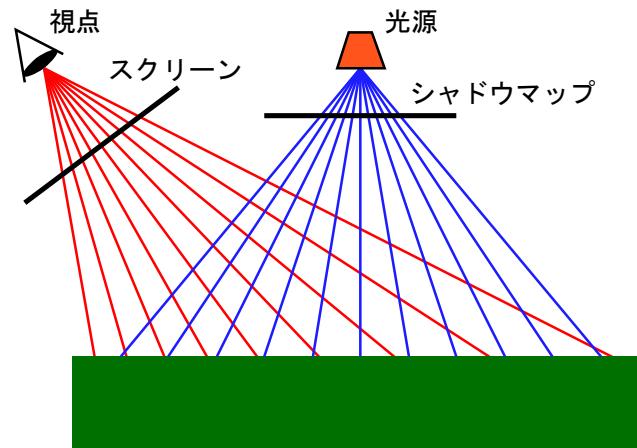
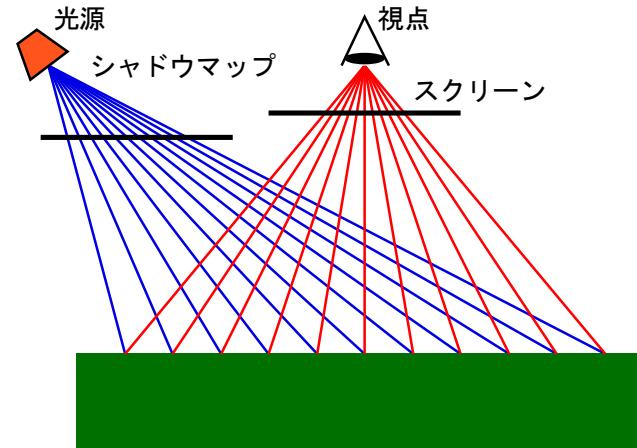
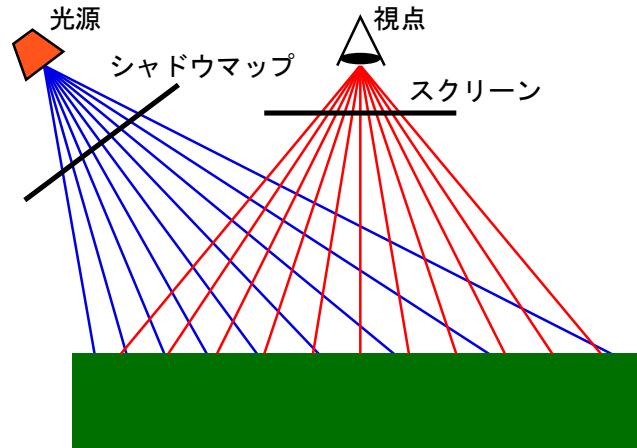
1. 光源位置に視点を置いてデプスバッファ法による隠面消去処理を行う
2. 得られたデプスバッファの内容 (デプステクスチャ, デプスマップ) をテクスチャメモリに転送する
3. 視点位置から見たシーンをレンダリングする
 - i. バーテックスシェーダで光源を視点にした時の頂点の位置を求める
 - ii. フラグメントシェーダに渡されたこの頂点の位置の補間値の xy 成分を使ってデプステクスチャをサンプリングする
 - iii. その補間値の z 成分とサンプリングした値を比較する
 - iv. 補間値の z 成分の方が大きければ、そのフラグメントを影とする
- GPU は ii ~ iv の処理を実行する機能を備えている

シャドウマップの解像度の影響

- ・シャドウマップの解像度が一様
 - ・影が拡大される部分でエリアシングが発生する



シャドウマップの補正



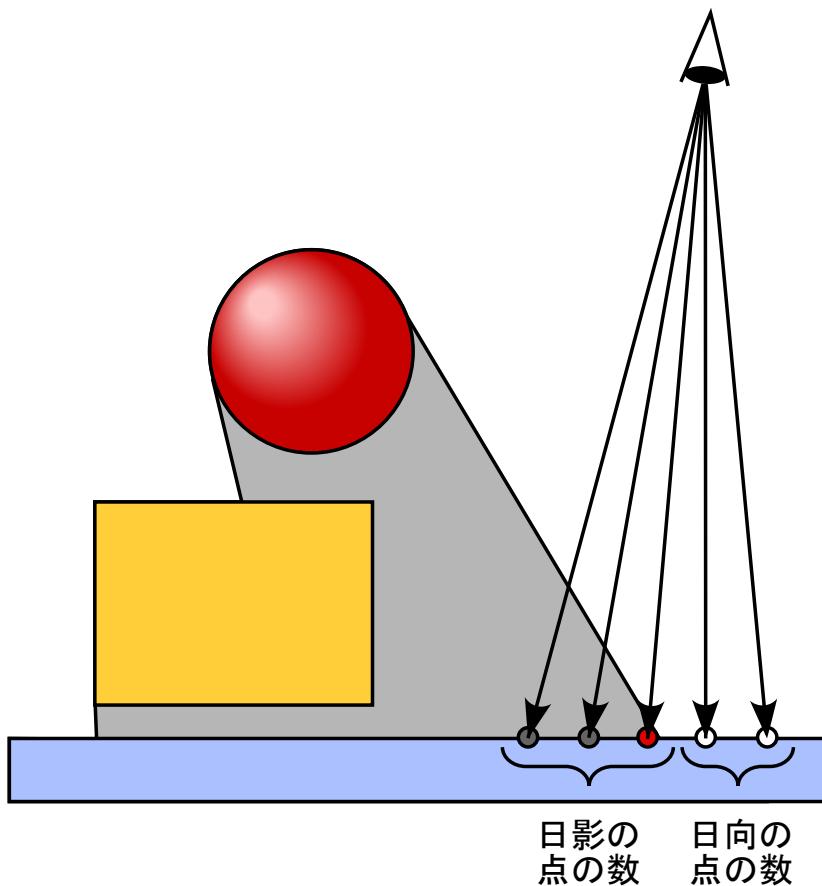
シャドウマップの補正手法

- PSM (Perspective shadow maps)
 - Stamminger, Marc, and George Drettakis. "Perspective shadow maps." *ACM transactions on graphics (TOG)*. Vol. 21. No. 3. ACM, 2002.
 - 可視物体が視点からの透視変換結果に一致するよう光源の変換行列を設定する
- LiSPSM (Light space perspective shadow maps)
 - Wimmer, Michael, Daniel Scherzer, and Werner Purgathofer. "Light space perspective shadow maps." *Rendering Techniques 2004* (2004): 15th.
 - 光源の変換行列に透視変換を適用する
- TSM (Trapezoidal Shadow Maps)
 - Martin, Tobias, and Tiow Seng Tan. "Anti-aliasing and Continuity with Trapezoidal Shadow Maps." *Rendering techniques 2004* (2004): 15th.
 - 台形変換を用いる
- CSM (Cascade Shadow Maps)
 - Dimitrov, Rouslan. "Cascaded shadow maps." *Developer Documentation, NVIDIA Corp* (2007).
 - PSM の階層化

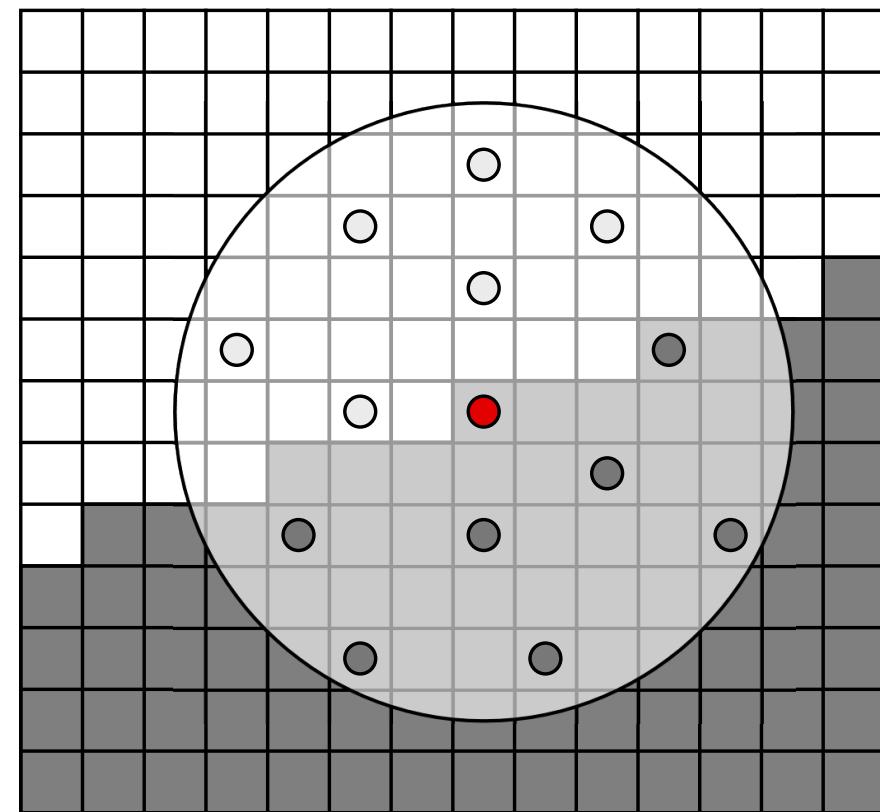
シャドウマップによるソフトシャドウ

- PCF (Percentage Closer Filtering)
 - Fernando, Randima. "Percentage-closer soft shadows." *ACM SIGGRAPH 2005 Sketches*. ACM, 2005.
 - 可視点の周囲のデプステクスチャを複数個サンプリングして、サンプリング点の影日向の割合で影の濃さを調整する
- VSM (Variance Shadow Maps)
 - Donnelly, William, and Andrew Lauritzen. "Variance shadow maps." *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006.
 - 可視点の周囲のデプステクスチャの深度値の平均と分散から、その可視点が影になる確率の上限値で影の濃さを調整する

PCF (Percentage Closer Filtering)



$$\text{影の濃さ} = \frac{\text{日影の点の数}}{\text{日向の点の数} + \text{日影の点の数}}$$



VSM (Variance Shadow Maps)

- ・深度値ではなく深度の平均値と μ その分散 σ^2 を用いる
 - ・チェビシェフの不等式

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2} \quad t > \mu \quad \Rightarrow \quad P(X \geq t) \leq \frac{\sigma^2}{\sigma^2 + (t - \mu)^2}$$

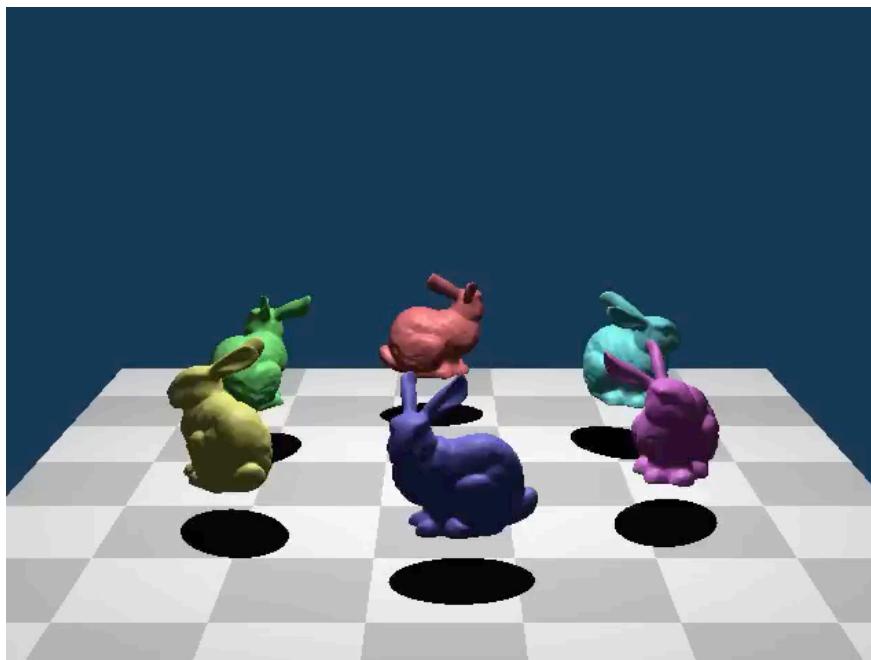
- ・ X の平均 μ と X との差が標準偏差 σ の k 倍以上になる確率は $1/k^2$ 以下
 - ・ある点が影になる確率の上限値を遮蔽度の近似値として用いる
- ・デプステクスチャ全体について
 - ・個々の画素について周囲の深度値の平均 $E(X) = \mu$ を求める
 - ・個々の画素について周囲の深度値の二乗の平均 $E(X^2)$ を求める
- ・画素のレンダリング時に
 - ・ $\sigma^2 = E(X^2) - E^2(X)$
 - ・ $t \leftarrow$ 現在の深度として
 - ・ $t > \mu$ のとき $P(X \geq t)$ の上限を求めて影の濃さとする
 - ・ $t \leq \mu$ なら本影

宿題

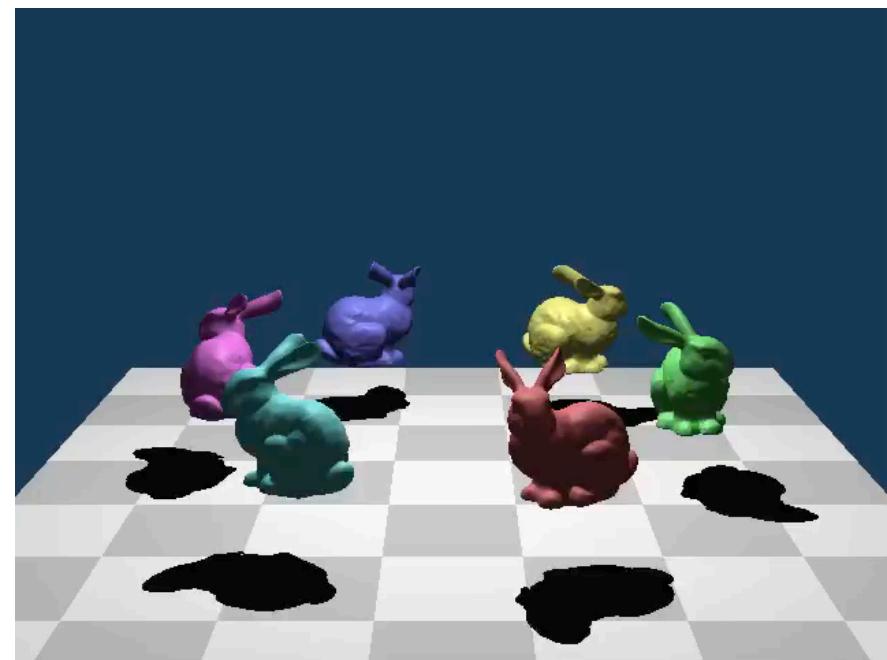
- 影付け処理を実装してください。
 - 次のプログラムは非常に単純な丸影を実装したものです。
 - <https://github.com/tokoik/ggsample11>
 - これを丸影以外の手法の影付け処理に変更してください。
 - 手法は任意です。
 - 根性があったら PCF や VSM を実装してみてください。
- 変更したソースプログラムをアップロードしてください

宿題プログラムの生成画像例

丸影



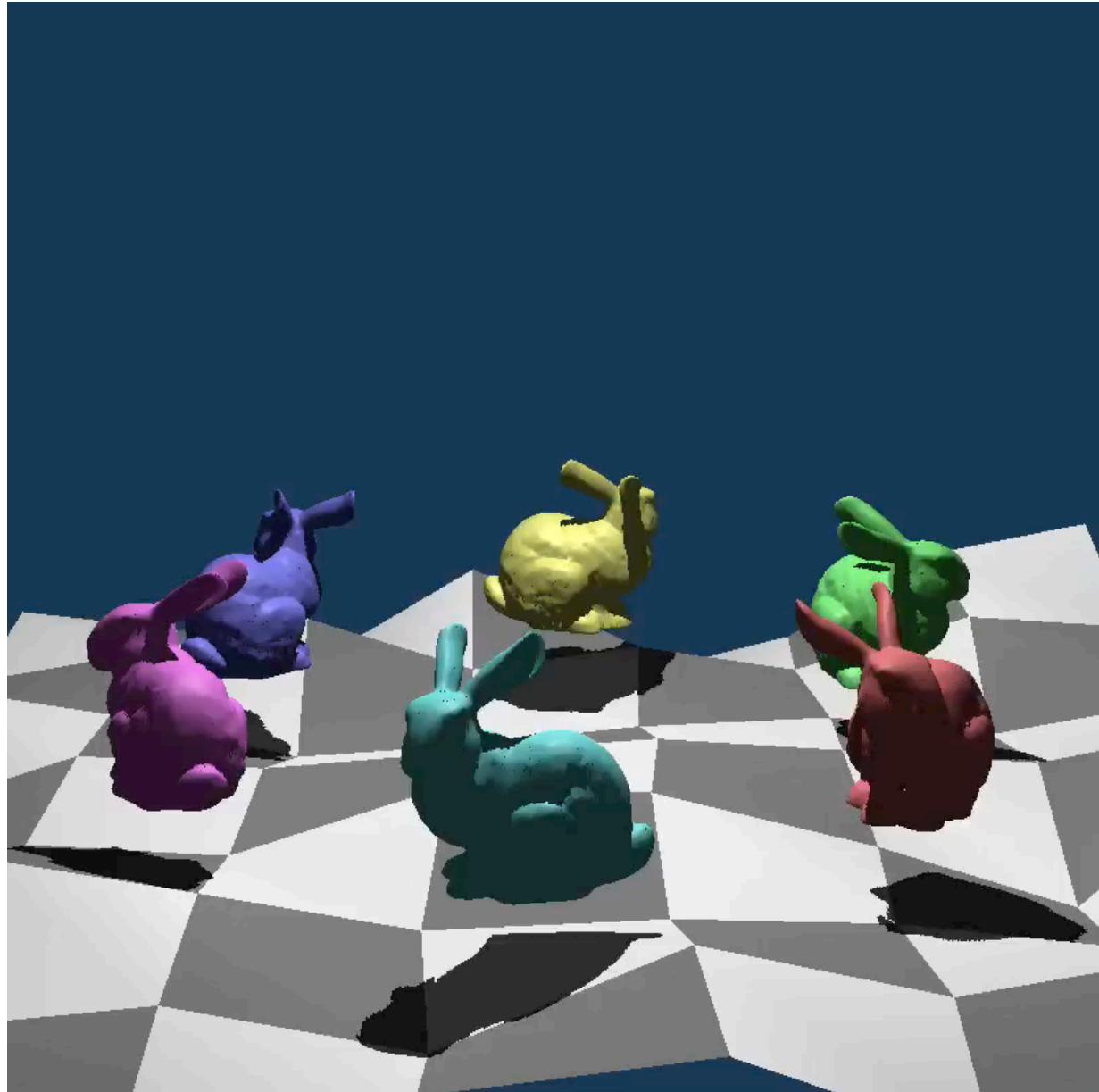
Projection Shadow



シャドウマップ

影の中に地面の色
を反映することができる

平面以外に影を落
とすことができる



影付け処理の実装のヒント

- Projection Shadow
 - 丸影の配置に用いている変換行列を Projection Shadow の変換行列 (M) に差し替える
 - 影として「丸」ではなく影を落とすオブジェクト自体を影の色で描く
- シャドウマップ
 - 後述

ウィンドウサイズ変更の抑制

```
// デプステクスチャの解像度  
const GLsizei dWidth(800), dHeight(800);
```

新たに開く
ウィンドウの幅と高さ

```
// ウィンドウを作成する
```

```
Window window("ggsample11", dWidth, dHeight);
```

ウィンドウの幅と高さの最小値

```
// ウィンドウサイズの変更を抑制する
```

```
glfwSetWindowSizeLimits(window.get(), dWidth, dHeight,  
GLFW_DONT_CARE, GLFW_DONT_CARE);
```

ウィンドウの幅と高さの最大値
GLFW_DONT_CARE なら制限しない

このへんは
宿題プログラムの
都合

デプステクスチャの作成

```
// デプステクスチャを作成する
GLuint dtex;
glGenTextures(1, &dtex);
glBindTexture(GL_TEXTURE_2D, dtex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
             dWidth, dHeight, 0, GL_DEPTH_COMPONENT, GL_FLOAT, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
```

影の有無を判断するだけなので線形補間する意味はない

シャドウマップ用のテクスチャパラメータ

```
// テクスチャ座標値の z 成分とデプステクスチャとの比較を行うようにする
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
    GL_COMPARE_REF_TO_TEXTURE);

// もし R の値がテクスチャの値以下なら真 (つまり日向)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC,
    GL_LEQUAL);

// デプステクスチャを一時保存するメモリを確保する
std::unique_ptr<GLfloat> depth(new GLfloat[dWidth * dWidth]);
```

スマートポインタ
スコープから外れた時に
delete される

↑
フレームバッファのデプスバッファの内容を
デプステクスチャにコピーする際に使う

光源の視点座標系での位置と目標点

```
// ワールド座標系の光源の目標位置  
const GLfloat lt[] = { 0.0f, 0.0f, 0.0f, 1.0f };
```

光源の目標点は視点側から見た
オブジェクトの中心位置

```
// ビュー変換行列を mv に求める
```

```
const GgMatrix mv(ggLookat(  
    0.0f, 3.0f, 8.0f,  
    0.0f, 1.0f, 0.0f,  
    0.0f, 1.0f, 0.0f));
```

← 視点の位置
← 目標点の位置
← 上方向のベクトル

```
// 視点座標系の光源位置を求めて光源データに設定する  
mv.projection(light.position, lp);
```

影付け処理用の変換行列

```
// 影付け処理用の視野変換行列を求める
```

```
const GgMatrix mvs(ggLookat(lp[0], lp[1], lp[2],  
    lt[0], lt[1], lt[2], 0.0f, 0.0f, 1.0f));
```

ワールド座標系の目標点の位置

ワールド座標系の光源の位置

```
// 影付け処理用の投影変換行列を求める
```

```
const GgMatrix mps(ggPerspective(1.5f, 1.0f, 1.0f, 5.0f));
```

上(y軸)から見ているので
上方向のベクトルはz軸方向にする

```
// 影付け処理用の変換行列を求める
```

```
const GgMatrix ms(mps * mvs);
```

視錐台はシーン全体が収まるように設定する

シェーダに追加する uniform 変数の場所

```
// デプステクスチャのサンプラの uniform 変数の場所を取り出す
const GLint depthLoc(
    glGetUniformLocation(shader.get(), "depth"));

// シャドウマップ用の変換行列の uniform 変数の場所を取り出す
const GLint msLoc(
    glGetUniformLocation(shader.get(), "ms"));
```



宿題の補助プログラムで用意している
GgSimpleShader クラスのオブジェクトが保持する
プログラム名 (glCreateShader() の戻り値) を返す

シェーダプログラムの使用開始

```
// ウィンドウが開いている間くり返し描画する
while (window.shouldClose() == GL_FALSE)
{
    // 時刻の計測
    const float t(
        static_cast<float>(fmod(glfwGetTime(), cycle) / cycle));

    // 投影変換行列
    const GgMatrix mp(
        ggPerspective(0.5f, window.getAspect(), 1.0f, 15.0f));

    // シェーダプログラムの使用開始
    shader.use();
    shader.loadLight(light);
```

視点を光源位置に置いてレンダリング

```
// ビューポートをデプステクスチャのサイズに合わせる
glViewport(0, 0, dWidth, dHeight);

// デプスバッファのみ消去
glClear(GL_DEPTH_BUFFER_BIT);

// デプステクスチャの作成
for (int i = 1; i <= objects; ++i)
{
    // アニメーションの変換行列
    const GgMatrix ma(animate(t, i));

    // 遮蔽物の描画
    shader.loadMatrix(mps, mvs * ma);
    object->draw();
}
```

オブジェクトを光源の視点座標系に配置

光源の視点座標系で投影変換

デプスバッファをテクスチャに転送

```
// デプスバッファの読み込み  
glReadPixels(0, 0, dWidth, dHeight,  
    GL_DEPTH_COMPONENT, GL_FLOAT, depth.get());  
  
// デプステクスチャに転送  
glActiveTexture(GL_TEXTURE0);  
glBindTexture(GL_TEXTURE_2D, dtex);  
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, dWidth, dHeight,  
    GL_DEPTH_COMPONENT, GL_FLOAT, depth.get());
```

スマートポインタの
生ポインタを取り出す

13回目に説明する Frame Buffer Object (FBO) を使えばデータ転送は不要でデプステクスチャのサイズもウィンドウサイズと独立して設定できる

デプステクスチャ作成専用のシェーダを用いれば計算量やデータ転送量を減らすことができる（ここではシェーダを通常の描画と共用している）

uniform 変数に値を設定する

```
// デプステクスチャのテクスチャユニットを指定する
glUniform1i(depthLoc, 0);

// シャドウマップ用の変換行列を設定する
glUniformMatrix4fv(msLoc, 1, GL_FALSE, ms.get());
```



宿題の補助プログラムで用意している
GgMatrix クラスのオブジェクトが保持する
GLfloat 型の 16 要素の配列のポインタを返す

視点の方向からレンダリング

```
// ビューポートをウィンドウのサイズに合わせる  
glViewport(0, 0, window.getWidth(), window.getHeight());
```

現在開いているのウィンドウの幅と高さ

```
// 画面消去  
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
// 床の描画
```

```
shader.loadMatrix(mp, mv);
```

```
floor.draw(shader);
```

オブジェクトを視点座標系に配置

```
// シェーダプログラムの使用開始 (時刻tにもとづく回転アニメーション)  
for (int i = 1; i <= objects; ++i)  
{
```

...

バーテックスシェーダの変更点

```
#version 150 core
#extension GL_ARB_explicit_attrib_location : enable
...
// 変換行列
...
uniform mat4 ms;           // シャドウマップの変換行列
// ラスタライザに送る頂点属性
...
out vec4 ps;               // 光源を視点に置いた時の頂点のスクリーン座標
void main(void)
{
...
    gl_Position = mp * p;    // 頂点のスクリーン座標
    ps = ms * pv;           // 光源を視点に置いた時の頂点のスクリーン座標
}
```

フラグメントシェーダの変更点

```

#version 150 core
#extension GL_ARB_explicit_attrib_location : enable

// デプステクスチャのサンプラー
uniform sampler2DShadow depth;

// ラスタライザから受け取る頂点属性の補間値
in vec4 iamb;                                // 環境光の反射光強度
in vec4 idiff;                                // 拡散反射光強度
in vec4 ispec;                                // 鏡面反射光強度
in vec4 ps;                                    // 光源を視点に置いた時のスクリーン座標

// フレームバッファに出力するデータ
layout (location = 0) out vec4 fc; // フラグメントの色

void main(void)
{
    fc = iamb
        + (idiff + ispec) * texture(depth, ps.xyz * 0.5 / ps.w + 0.5);
}

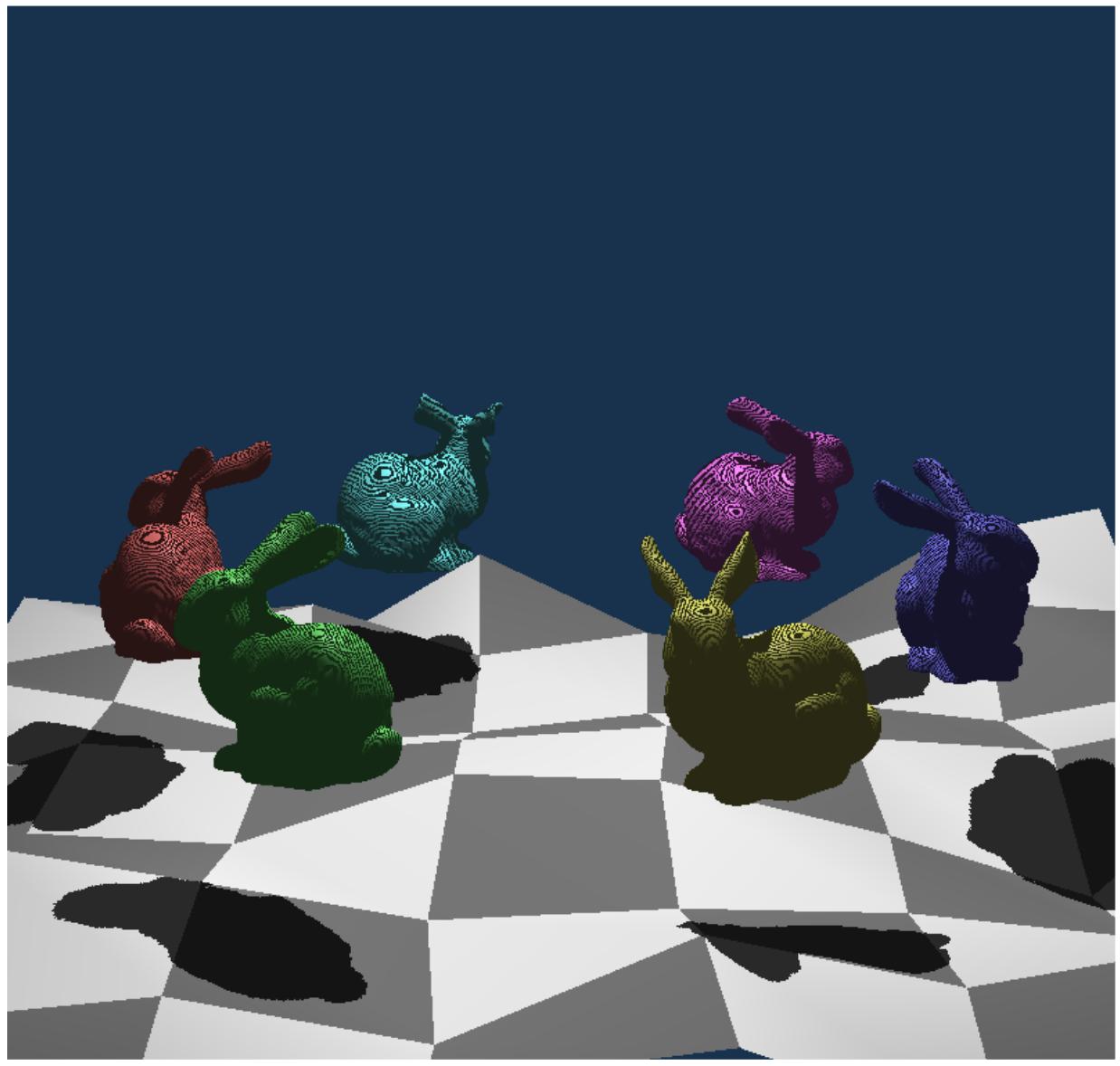
```

デプステクスチャのときデータ型は float

ps は投影変換で [-1,1] のクリッピング空間に配置されるので実座標に直して [0,1] のテクスチャ空間に収める

デプス ファイティング

ポリゴンをラスタライズする際の量子化誤差によりポリゴンとデプステクスチャの比較の結果が場所によって違ってしまう



ポリゴンオフセットの追加

```
// デプスファイティングを避けるためにポリゴンオフセットを設定する  
glPolygonOffset(1.0f, 1.0f);
```

ポリゴンの傾きに比例して追加する奥行きの係数

ポリゴンの奥行きにそのまま加える奥行きの係数

表現可能な最小の深度値の倍数

```
// ポリゴンの塗りつぶし領域においてポリゴンオフセットを有効にする  
glEnable(GL_POLYGON_OFFSET_FILL);
```

プログラミング上のヒント

- GgMatrix クラス
 - クラス定義
 - http://www.wakayama-u.ac.jp/~tokoi/lecture/gg/html/classgg_1_1GgMatrix.html
 - 用途
 - OpenGL で用いる変換行列を操作する
 - コンストラクタ
 - GLfloat 型の 16 要素の配列で初期化できる
 - GLfloat m[16] = { … }; GgMatrix mat(m);
 - 演算
 - かけ算 (*, *=) だけ
 - (GgMatrix) * (GgMatrix)
 - (GgMatrix) * (GLfloat *) → GLfloat 型の16要素の配列を直接かけられる
 - 関数
 - ggLookAt(), ggPerspective(), ggOrthogonal(), ggTranslate(), ggRotate()
 - それぞれ対応するメソッドによって生成された GgMatrix 型の値を返す