



コンピュータアニメーション特論

第15回 レンダリング関連技術

九州工業大学 情報工学研究院 尾下真樹

今回の内容

- ・ 影の描画
- ・ イメージベースド・レンダリング
- ・ BRDFによる質感の表現
- ・ HDRレンダリング
- ・ 講義のまとめ



レンダリングの最新技術

- ・ レンダリングの最新技術

- 基礎的なレンダリング技術だけでは、写実的な画像の生成は難しい
- より写実的な画像を生成するための最新技術が開発されている
 - ・ オフライン・アニメーション、オンライン・アニメーションの両方の用途に適用可能な技術



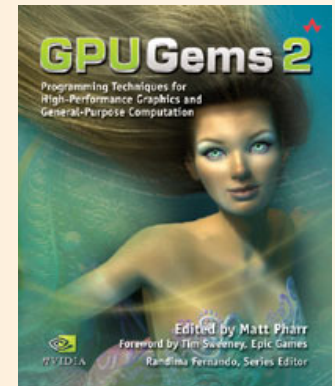
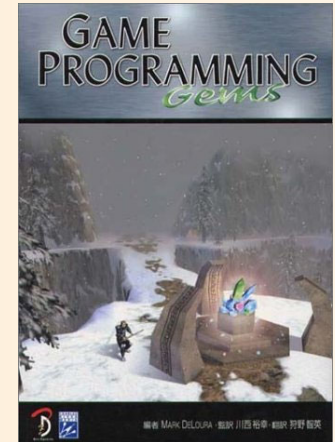
今回の内容

- ・ 影の描画
- ・ イメージベースド・レンダリング
- ・ BRDFによる質感の表現
- ・ HDRレンダリング
- ・ 講義のまとめ



参考資料（影の描画）

- ・ テクスチャマッピング
 - 参考書（赤本）を参照
- ・ 平面へのポリゴン投影
 - Game Programming Gems I
 - 参考書（赤本）を参照
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング
 - Game Programming Gems II
 - GPU Gems I ~ II
- ・ どの方法も、ネットで検索すると参考資料が見つかる



参考書（他のレンダリング技術）

- ・ 「コンピュータグラフィックス」
CG-ARTS協会 編集・出版（3,200円）
- ・ 「3DCGアニメーション」
栗原恒弥 安生健一 著、技術評論社
出版（2,980円）
- ・ Image-Based Rendering / BRDF
– Computer Graphics Gems JP シリーズ

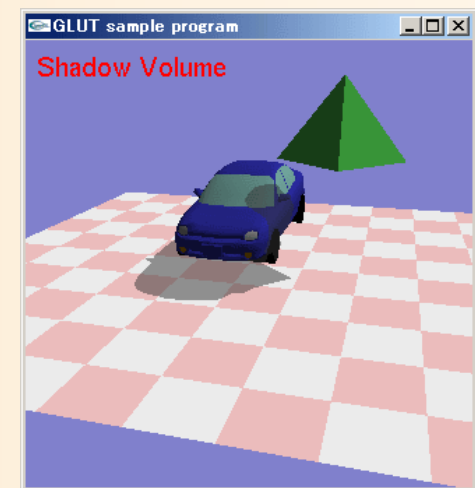




影の描画

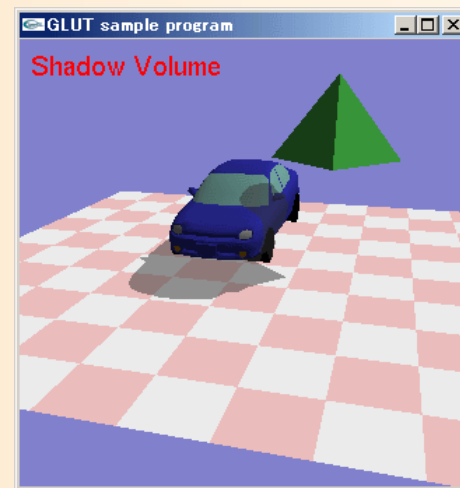
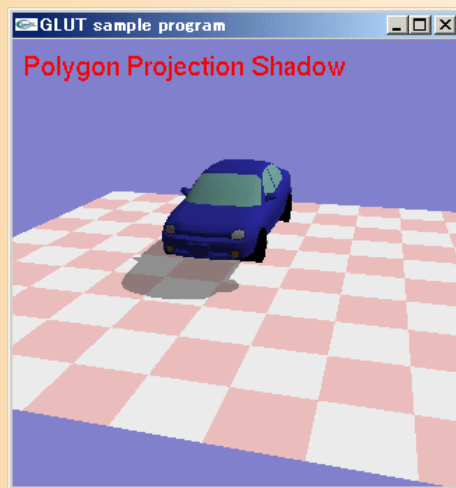
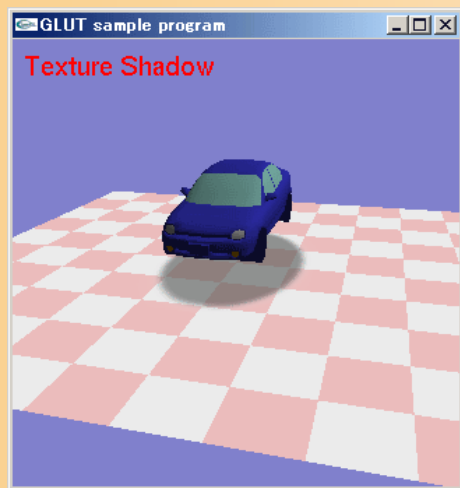
影の表現

- レンダリング画像の現実感（リアリティ）を出す上で、影の描画は不可欠
 - 影の有無は、画面の自然さに大きく影響
 - 特に空中に浮いている物体を描画するようときには、影があると、高さが把握しやすい
- 影の描画の技術
 - いくつかの方法が利用されている
 - 高度な描画技術が必要となる
 - ・ アルファブレンディング（半透明描画）
 - ・ ステンシルバッファ



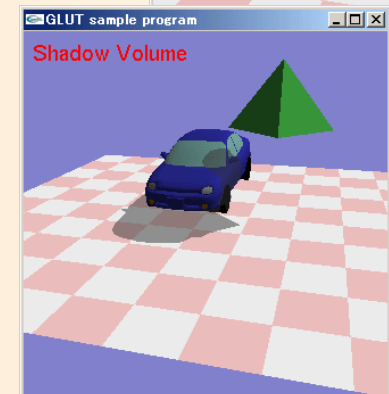
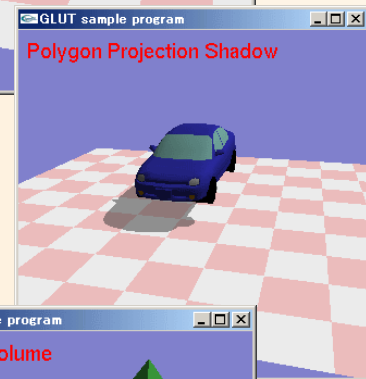
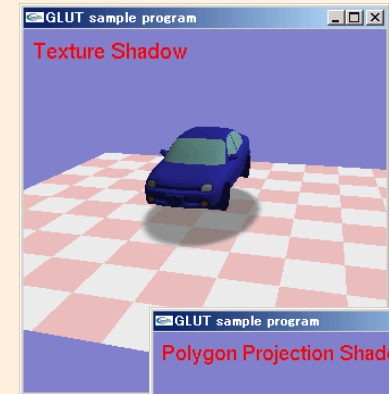
影の表現方法

- ・ テクスチャマッピング
- ・ 平面へのポリゴン投影
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング



各表現方法の比較

- ・ テクスチャマッピング
 - 高速、近似形状
 - 他の物体・自分自身への投影不可
- ・ 平面へのポリゴン投影
 - 中速
 - 他の物体・自分自身への投影不可
- ・ シャドウ・ヴォリューム
 - 低速
 - 他の物体への投影可、自分自身への投影可
- ・ シャドウ・マッピング
 - 低速、ハードウェアにより高速化可能
 - 他の物体への投影可、自分自身への投影不可



影の描画

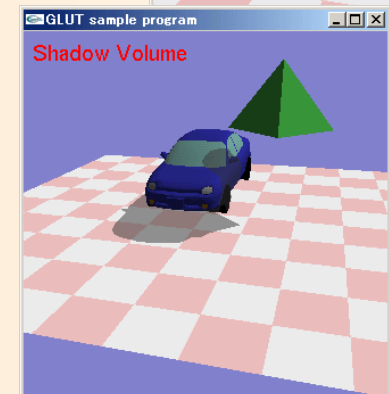
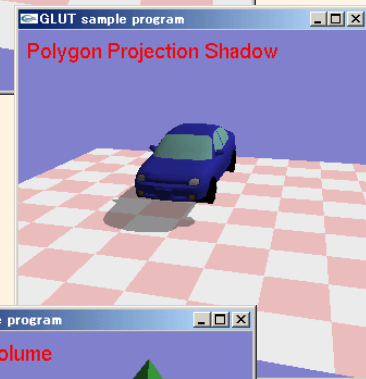
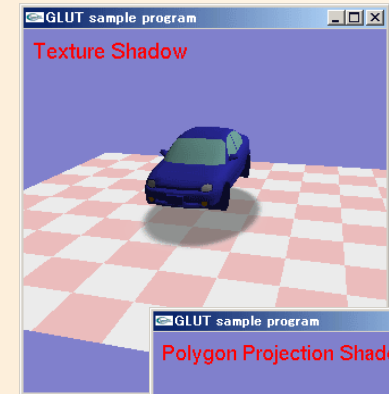
- ・ 影の描画のプログラム
- ・ テクスチャマッピングによる影の描画
- ・ 平面へのポリゴン投影による影の描画
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング
- ・ 高度な影の描画技術
- ・ 高度な描画技術
- ・ 演習課題



デモプログラム

・ 影の描画

- 3種類の方法を切り替え可能
 - ・ テクスチャマッピング
 - ・ 平面へのポリゴン投影
 - ・ シャドウ・ボリューム
- 物体の表示・非表示
 - ・ 物体同士の影の確認
- 視点操作・形状データの読み込みは、これまでの講義で扱った技術を利用





サンプルプログラムの構成

- ・ デモプログラムの一部を実装したサンプルプログラム (shadow_sample.cpp)
 - レポート課題のもとになるプログラム
- ・ 幾何形状の読み込み・描画 (Obj.h, Obj.cpp)
 - 過去の授業で扱った内容
- ・ ビットマップ画像の読み込み (bitmap.h, bitmap.cpp)
 - テクスチャ画像の読み込みに使用
 - BMP (24ビット、非圧縮) 形式の読み込み関数



幾何形状の定義

// 幾何形状モデル (Obj形式)

struct Obj

{

int num_vertices; // 頂点数

Vector * vertices; // 頂点座標配列

int num_normals; // 法線ベクトル数

Vector * normals; // 法線ベクトル配列

int num_textures; // テクスチャ座標数

Vector * textures; // テクスチャ座標配列

int num_triangles; // 三角面数

int * tri_v_no; // 三角面の頂点座標番号の配列

int * tri_vn_no; // 三角面の法線ベクトル番号の配列

int * tri_vt_no; // 三角面のテクスチャ座標番号の配列

Mtl * tri_material; // 三角面の素材の配列

};



幾何形状の読み込み・描画関数

```
// Objファイルの読み込み
Obj * LoadObj( const char * filename );

// Mtlファイルの読み込み
void LoadMtl( const char * filename, Obj * obj );

// 幾何形状モデルのスケーリング(スケーリング後のサイズを返す)
void ScaleObj( Obj * obj, float max_size,
               float * size_x, float * size_y, float * size_z );

// Obj形状データの描画
void RenderObj( Obj * obj );
```



サンプルプログラム (1)

- ・ 影の描画方法や光源位置を表す変数
 - マウス操作により変化

```
// 影の描画方法
enum ShadowModeEnum
{
    SHADOW_NONE,
    SHADOW_TEXTURE,
    SHADOW_PROJECTION,
    SHADOW_VOLUME,
    NUM_SHADOW_MODE
};

// 現在の影の描画方法
ShadowModeEnum shadow_mode = SHADOW_PROJECTION;

// 点光源の位置(影の投影方向)
Vector light_pos;
```



サンプルプログラム (2)

・ 幾何形状オブジェクトの情報

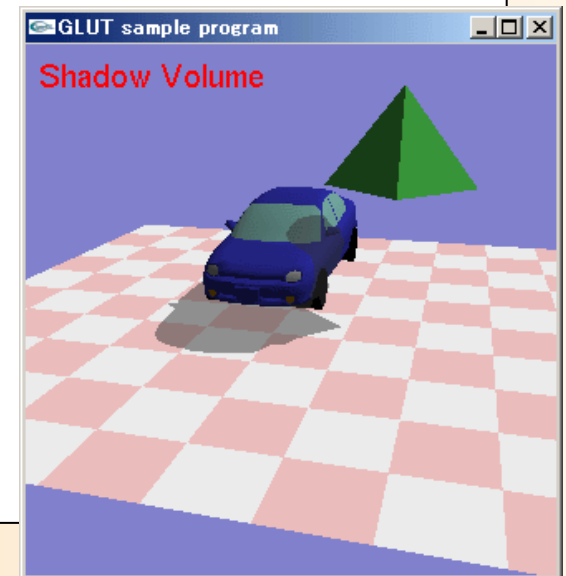
```
// 幾何形状オブジェクトの数
#define NUM_OBJECTS 2

// 幾何形状オブジェクト
Obj *   object[ NUM_OBJECTS ];

// 位置
Vector  object_pos[ NUM_OBJECTS ];

// 水平向き
float   object_ori[ NUM_OBJECTS ];

// 大きさ(テクスチャマッピングによる影の描画用)
Vector  object_size[ NUM_OBJECTS ];
```



サンプルプログラム (3)

・ 幾何形状オブジェクトの読み込み・初期化

```
void LoadObjects()
{
    // オブジェクトの読み込み
    object[ 0 ] = LoadObj( "Car.obj" );
    ScaleObj( object[ 0 ], 5.0f,
        &object_size[ 0 ].x, &object_size[ 0 ].y, &object_size[ 0 ].z );
    object[ 1 ] = LoadObj( "Pyramid.obj" );
    ScaleObj( object[ 1 ], 2.0f,
        &object_size[ 1 ].x, &object_size[ 1 ].y, &object_size[ 1 ].z );

    object_pos[ 0 ].x = 0.0f;
    object_pos[ 0 ].y = 2.0f;
    object_pos[ 0 ].z = 0.0f;
    object_ori[ 0 ] = 180.0f;
    ...
}
```



サンプルプログラム (4)

・ 描画処理 (オブジェクト + 影の描画)

```
// 画面描画時に呼ばれるコールバック関数
void DisplayCallback()
{
    // 画面をクリア

    // 変換行列(カメラ座標系からワールド座標系への変換行列)を設定

    // 光源位置を設定
    float light0_position[] = { light_pos.x, light_pos.y, light_pos.z, 1.0 };
    glLightfv( GL_LIGHT0, GL_POSITION, light0_position );

    // 格子模様の床を描画

    // それぞれの幾何形状モデル + 影を描画
    for ( int i=0; i<NUM_OBJECTS; i++ )
    {
```



サンプルプログラム (5)

・ 描画処理 (オブジェクト + 影の描画)

// モデル座標系からワールド座標系への変換行列を計算

float matrix[16];

glPushMatrix();

glLoadIdentity();

glTranslatef(object_pos[i].x, object_pos[i].y, object_pos[i].z);

glRotatef(object_ori[i], 0.0f, 1.0f, 0.0f);

glGetFloatv(GL_MODELVIEW_MATRIX, matrix);

glPopMatrix();

// 物体を描画

glMultMatrixf(matrix);

RenderObj(object[i]);

// 影を描画

// 現在の影の描画方法に応じて処理を呼び出し

...

影の描画に必要

単位行列で初期化

位置と水平向きにもとづいて
変換行列を設定

変換行列を取得

ワールド座標系からカメラ座標系への変換行列が
設定されている状態で、モデル座標系からワ
ールド座標系への変換行列を右からかける

引数として matrix を渡す

サンプルプログラム (6)

- ・ テクスチャマッピングによる影の描画
- ・ ポリゴン投影による影の描画
- ・ (シャドウ・ボリュームによる影の描画)

```
// テクスチャマッピングによる影の描画(位置・向き、大きさ、高さを指定)  
void RenderTextureShadow(  
    float obj_matrix[ 16 ], float size_x, float size_z, float shadow_y )
```

```
// ポリゴン投影による影の描画  
void RenderProjectionShadow(  
    Obj * obj, float obj_matrix[ 16 ], Vector & light_dir,  
    float color_r, float color_g, float color_b, float color_a )
```

各関数の引数の定義は後程説明



演習課題

- ・ 今回のプログラミング演習は自由演習課題とするので、興味があれば各自で取り組む
 - 成績評価には考慮しない
- ・ 演習課題の詳細は後述



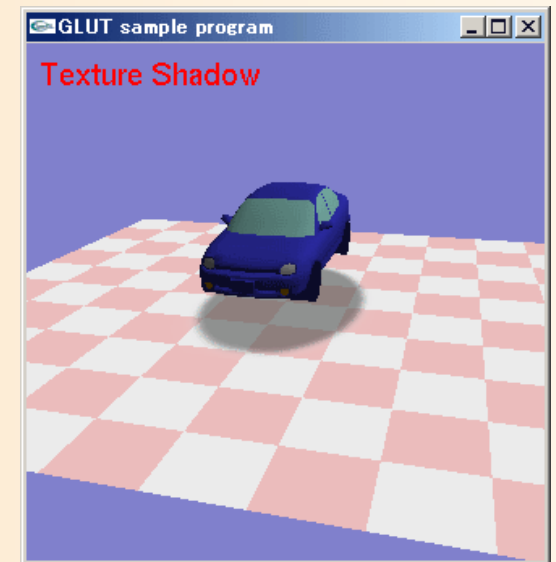
影の描画

- ・ 影の描画のプログラム
- ・ テクスチャマッピングによる影の描画
- ・ 平面へのポリゴン投影による影の描画
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング
- ・ 高度な影の描画技術
- ・ 高度な描画技術
- ・ 演習課題



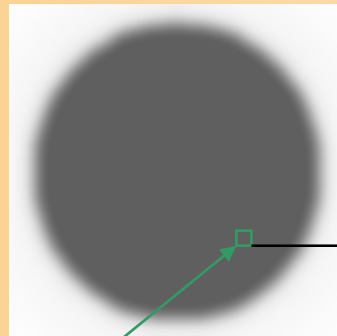
テクスチャマッピングによる 影の描画

- 適当な影の画像を用意
- 物体の下に影の画像をテクスチャマッピング
 - 単純に貼り付けるとおかしくなるので、ブレンディング（半透明描画）を行いながら貼り付ける



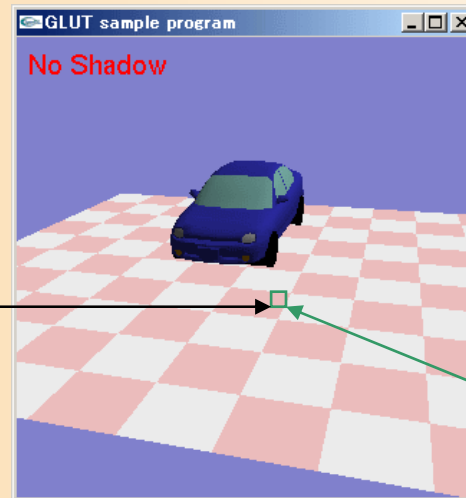
ブレンディングの方法

- glEnable(GL_BLEND)
 - glBlendFunc(Fsrc, Fdest)
 - 描画色（この例ではテクスチャ）と画面のもとのピクセル色をどのように混ぜ合わせるかを設定
- $$C = C_{src} * F_{src} + C_{dest} * F_{dest}$$



$C_{src} (R, G, B, A)$

?



$C_{dest} (R, G, B, A)$

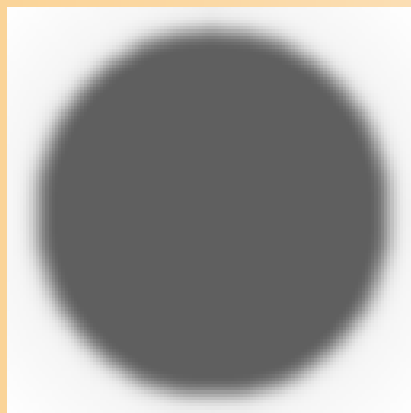
ブレンディングの方法

- `glBlendFunc( Fsrc, Fdest )` の引数の種類
 - `GL_ZERO`
 - `GL_ONE`
 - `GL_DEST_COLOR`
 - `GL_SRC_COLOR`
 - `GL_ONE_MINUS_DEST_COLOR`
 - `GL_ONE_MINUS_SRC_COLOR`
 - `GL_SRC_ALPHA`
 - `GL_DEST_ALPHA`
 - `GL_ONE_MINUS_SRC_ALPHA`
 - `GL_ONE_MINUS_DEST_ALPHA`
 - `GL_SATURATE`
- 色を係数として使用
- アルファ値を係数として使用

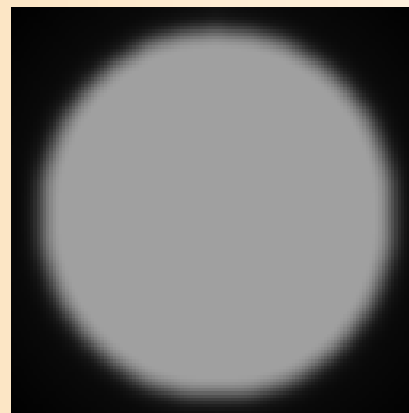


影テクスチャの貼り付け（1）

- ・ 方法1：アルファプレーンを使用
 - はりつける影の部分を指定してアルファプレーンを作成しておく
 - ・ グレースケールBMPとして用意して読み込んでも良いし、テクスチャ画像から自動的に生成しても良い



+



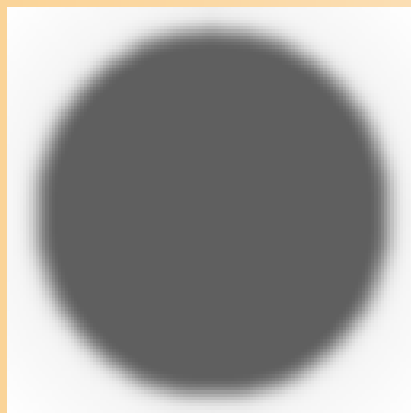
どの程度テクスチャを画面に混ぜ合わせるかを表している
0 の範囲は全く描画されない

カラープレーン（R, G, B）

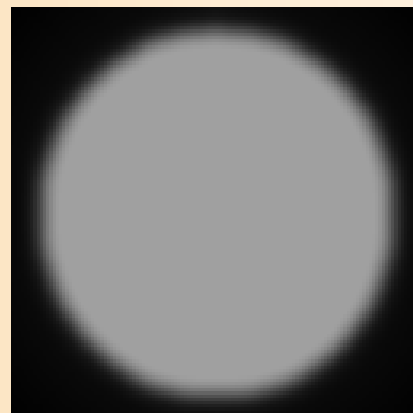
アルファプレーン（A）

影テクスチャの貼り付け (2)

- ・ 方法1：アルファプレーンを使用 (続き)
 - テクスチャ側のアルファ値を使ってブレンド
`glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);`



+

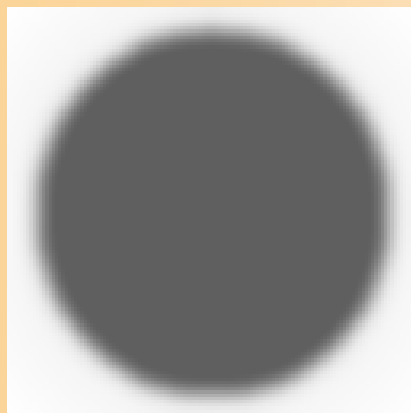


カラープレーン (R, G, B)

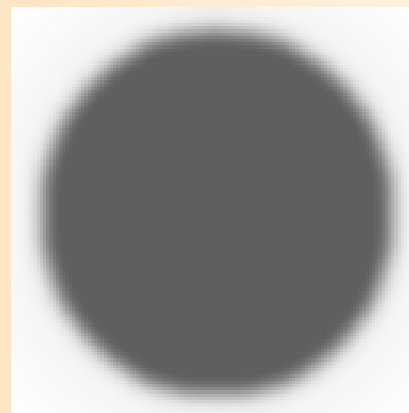
アルファプレーン (A)

影テクスチャの貼り付け (3)

- ・ 方法1：アルファプレーンを使用 (続き)
 - 下図のようなアルファプレーンを使用するときは、2つの引数を入れ替える
`glBlendFunc(GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);`



+



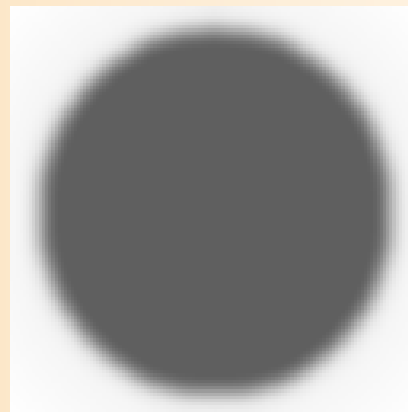
カラープレーン (R, G, B)

アルファプレーン (A)

影テクスチャの貼り付け (4)

・ 方法2：カラープレーンのみを使用

- 今回は、テクスチャの色によってブレンド比率が決まるので、アルファプレーンを使わなくて済む
- テクスチャ側のカラー値を使ってブレンド
`glBlendFunc( GL_ONE_MINUS_SRC_COLOR, GL_SRC_COLOR );`



カラープレーン (R, G, B)



参考：テクスチャマッピングの 手順

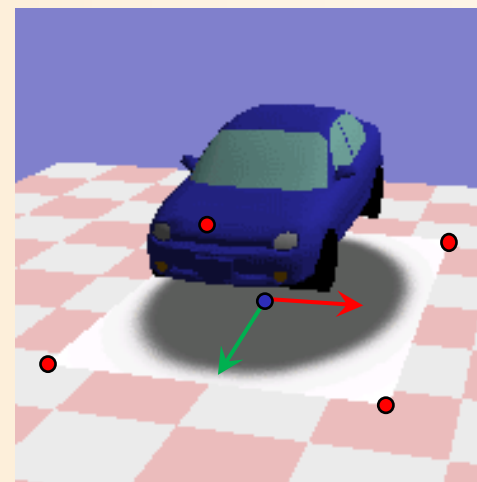
1. テクスチャ画像の読み込み
2. テクスチャ画像を登録
3. テクスチャマッピングのパラメタを設定
4. テクスチャ画像の適用方法を設定
5. テクスチャマッピングを用いてポリゴンを描画
 - テクスチャマッピングを有効に設定
 - 各頂点ごとにテクスチャ座標 (u,v) を指定

※ 詳細は参考書・テキストを参照



影画像の描画処理の手順

1. テクスチャ画像の読み込み・設定
2. テクスチャ画像の描画位置（四角形ポリゴンの四隅の位置）を計算
3. テクスチャマッピングの設定
4. テクスチャ画像の描画（四角形ポリゴンを描画）



描画処理の作成 (1)

```
// テクスチャマッピングによる影の描画  
void RenderTextureShadow( float obj_matrix[ 16 ],  
                           float size_x, float size_z, float shadow_y )
```

```
{  
    // テクスチャ画像の読み込みと設定  
    // 最初に一度だけ行われる  
    if ( shadow_texture == 0 )  
    {  
        if ( ! LoadShadowTexture() )  
            return;  
    }  
    ...  
}
```

影を描画する物体の水平位置・向き、
影の前後・左右方向の大きさ、
影の高さ、を引数として受け取る

テクスチャ画像の読み込みと設定
詳しい処理はサンプルプログラムを参照



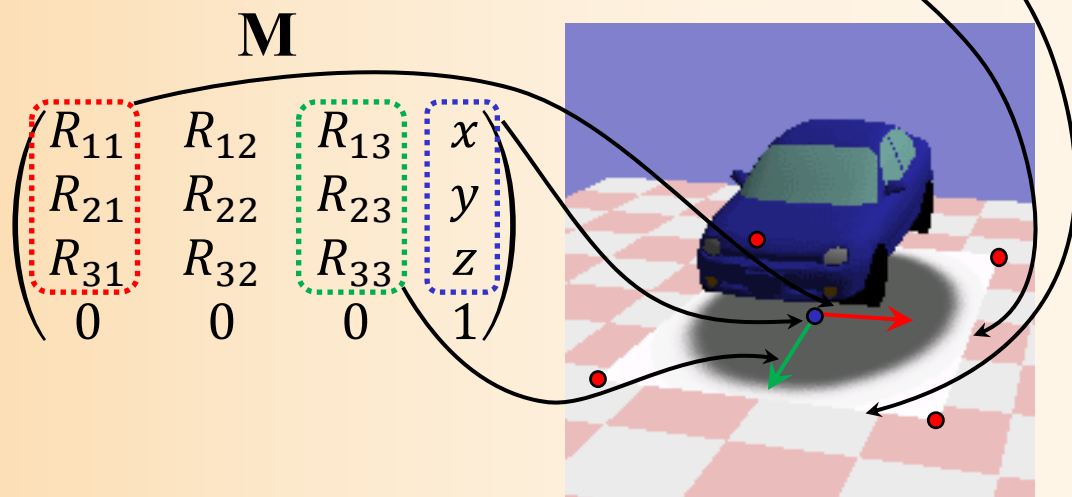
描画処理の作成 (2)

- ・ テクスチャ画像の描画 (四隅) 位置の計算
- ・ 入力情報

- オブジェクトの位置・向きを表す変換行列 M
- 影の左右・前後方向の大きさ $size_x, size_z$

- ・ 計算方法

- 変換行列から
水平方向の
位置・X軸・
Z軸を取得
→ **四隅の位置**
の計算



描画処理の作成 (3)

```
// 影のテクスチャ画像を描画する四隅の水平位置 + 高さ
float x0, z0, x1, z1, x2, z2, x3, z3, y;
```

```
// オブジェクトの水平方向の中心位置・x軸方向・z軸方向を使用
float center_x, center_z, x_axis_x, x_axis_z, z_axis_x, z_axis_z;
```

```
center_x = obj_matrix[ 12 ];
```

```
center_z =
```

```
...
```

?

```
// テクスチャ画像を描画する四隅の水平置を計算
```

```
x0 = center_x + (0.5 * size_x) * x_axis_x
      + (0.5 * size_z) * z_axis_x;
```

```
z0 =
```

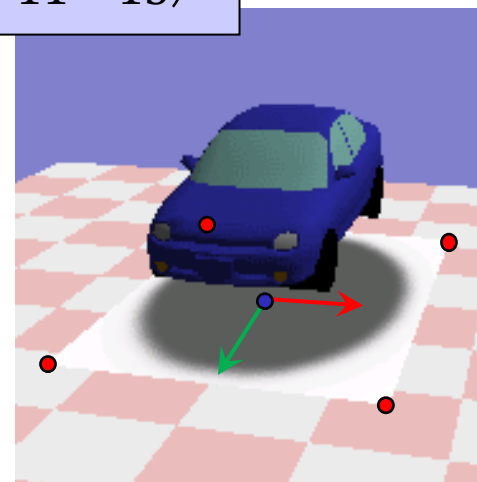
```
...
```

?

```
// 高さは引数として渡された値をそのまま使用
```

```
y = shadow_y;
```

中心位置のx座標・z座標、
X軸のx座標成分・z座標成分、
Z軸のx座標成分・z座標成分

$$\begin{pmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{pmatrix}$$


描画処理の作成 (4)

```
// 現在の描画設定を取得(描画終了後に元の設定に戻すため)
```

```
...
```

```
// 描画オプションの設定
```

```
glDisable( GL_LIGHTING ); // ライティングは無効に設定
```

```
glEnable( GL_BLEND ); // ブレンディングを有効に設定
```

```
glEnable( GL_TEXTURE_2D ); // テクスチャマッピング
```

```
// テクスチャマッピングの設定
```

```
glBindTexture( GL_TEXTURE_2D, shadow_texture );
```

```
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL );
```

```
// ブレンディングの設定(方法2を使用)
```

```
glBlendFunc( ? );
```

アルファプレーンは設定されていないため、
カラープレーンのみを用いる方法を使用

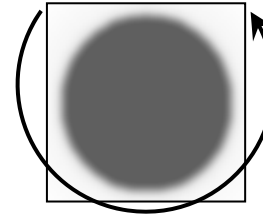
描画処理の作成 (5)

// 影テクスチャの描画(四角形のポリゴンを描画)

```
glBegin( GL_POLYGON );  
glNormal3f( 0.0, 1.0, 0.0 );  
glTexCoord2f( 1.0, 0.0 );  
glVertex3f( x0, y, z0 );  
...
```

どの点から始めても構わないので、半時計周りに、四隅のテクスチャ座標・位置を指定

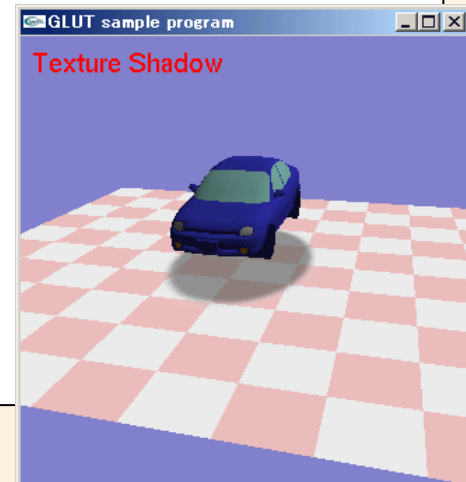
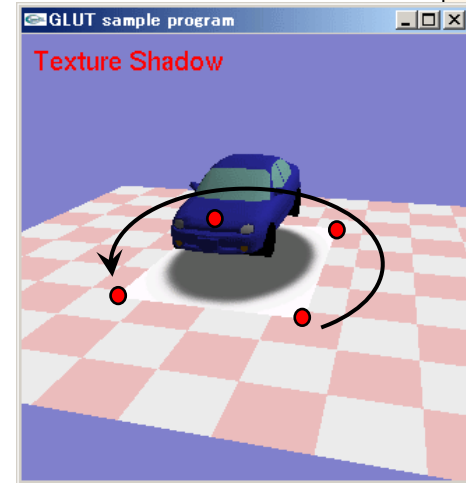
?



```
glEnd();
```

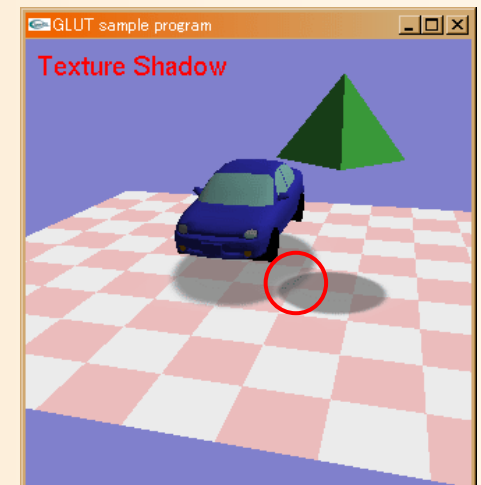
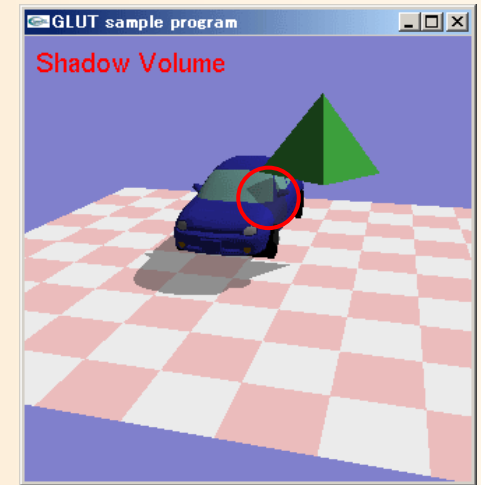
// 描画設定を復元

...



テクスチャマッピングによる 影の描画の問題点

- 影の形が単純
- 水平面にしか影を投影できない
 - 他の物体や自分自身への影の投影はできない
- 物体同士が近くにあるときに、影テクスチャ同士が重なるとおかしくなる
 - ステンシルバッファを使った解決方法を次で説明



影の描画

- ・ 影の描画のプログラム
- ・ テクスチャマッピングによる影の描画
- ・ 平面へのポリゴン投影による影の描画
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング
- ・ 高度な影の描画技術
- ・ 高度な描画技術
- ・ 演習課題



ポリゴン投影による影の描画

- 物体を構成する各ポリゴンを、地面に投影して、灰色（半透明）で描画
 - 物体の形状を反映した影を描画できる
 - 単純計算で、2倍の量のポリゴンを描画する必要がある
 - テクスチャマッピングによる影の描画と比べると、描画処理に時間がかかる



地面への投影 (1)

- ・ 視野変換行列の計算に、投影行列を追加
 - モデルからワールドへの変換行列 M
 - ワールドからカメラへの変換行列 C
 - ワールド座標系での地面への投影行列 P

影を描画するときの視野変換行列 $= C P M$

- ・ 単純な投影行列 P

- 真下に投影
 - ・ y 座標の値を常に 0 にする

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



地面への投影 (2)

- 任意の方向への投影

- 光源の方向を ($light_x, light_y, light_z$) とする

$$\mathbf{P} = \begin{pmatrix} 1 & -light_x / light_y & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -light_z / light_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- 任意の方向+任意の平面への投影

- 複雑になるが、同様に変換行列を計算できる
- 赤本を参照



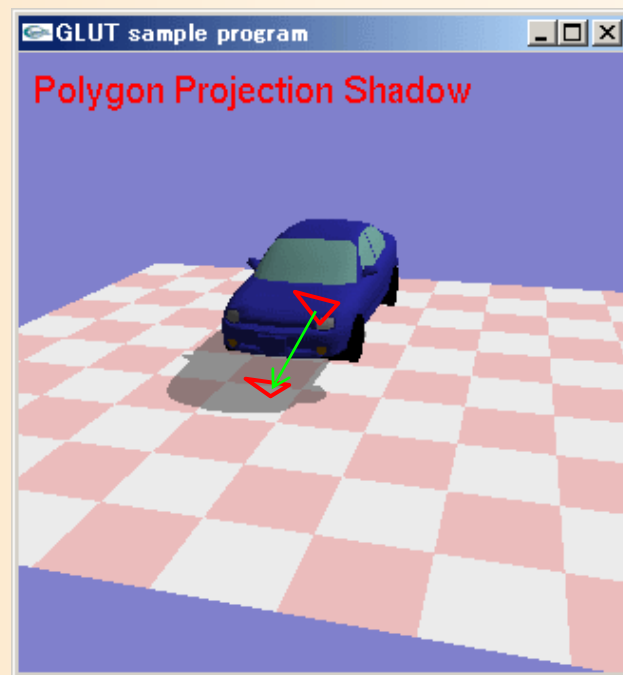
地面以外への投影

- 各平面ごとにポリゴンを投影して描画すれば、地面以外の影も表現できる
 - ただし、影が平面からはみ出る場合は、切り取りのための処理が必要
 - クリッププレーンを追加すれば、OpenGLが処理してくれる



投影した幾何形状の影の描画

- ・ 投影のための変換行列を設定した状態で、物体の幾何形状モデルを描画
 - 3次元の頂点座標は自動的に地面に投影される
- ・ 頂点の色を全て黒（灰色）
半透明で描画する
 - ブレンディングにより半透明で描画
- ・ ライティング（光源処理）はオフにして描画する



描画処理の作成 (1)

```
// ポリゴン投影による影の描画
void RenderProjectionShadow( const Obj * obj,
                             const float obj_matrix[ 16 ], const Vector & light_dir,
                             float color_r, float color_g, float color_b, float color_a )
{
    // 描画オプションの設定
    glDisable( GL_LIGHTING ); // ライティングは無効に設定
    glEnable( GL_BLEND ); // ブレンディングを有効に設定
    glEnable( GL_STENCIL_TEST ); // ステンシルバッファを使用するよう設定

    // ブレンディングの設定
    glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );

    // ステンシルバッファの設定
    ... (後から追加、後述)

    // 動作確認のための描画オプションの変更
    ...
}
```

物体の幾何形状モデル、
物体の位置・向き、
光源の方向、影の色

描画処理の作成 (2)

```
// 現在の変換行列を一時保存  
glPushMatrix();
```

```
// ポリゴンモデルを地面に投影して描画するための変換行列を設定  
// ワールド→カメラ変換 × 地面への投影変換 × モデル→ワールド変換  
// この時点で、ワールド→カメラ変換の変換行列が設定されているものとする
```

```
// 地面への投影行列を計算
```

```
float mat[ 16 ];
```

```
mat[ 0 ] =  ;
```

```
...
```

```
mat[ 15 ] =  ;
```

$$\mathbf{P} = \begin{pmatrix} 1 & -light_x/light_y & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -light_z/light_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
// 変換行列を設定
```

```
glMultMatrixf(  );
```

```
glMultMatrixf(  );
```

描画処理の作成 (3)

```
...
```

```
// 変換行列の設定
```

```
...
```

```
// 影の描画、幾何形状モデルを指定色で描画
```

```
RenderObjUnicolor( obj, color_r, color_g, color_b, color_a );
```

```
// 一時保存しておいた変換行列を復元
```

```
glPopMatrix();
```

```
// 描画設定を復元
```

```
...
```

```
}
```



描画処理の作成 (4)

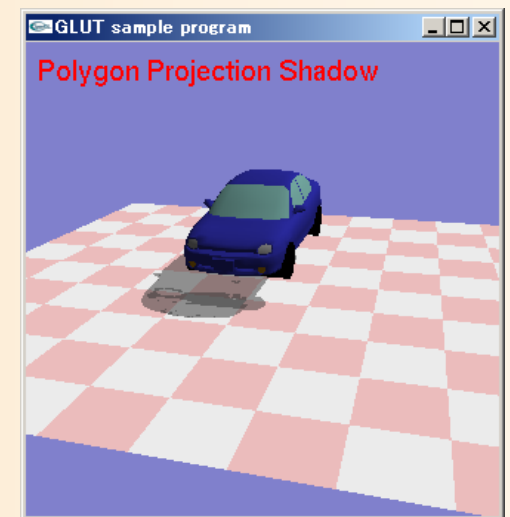
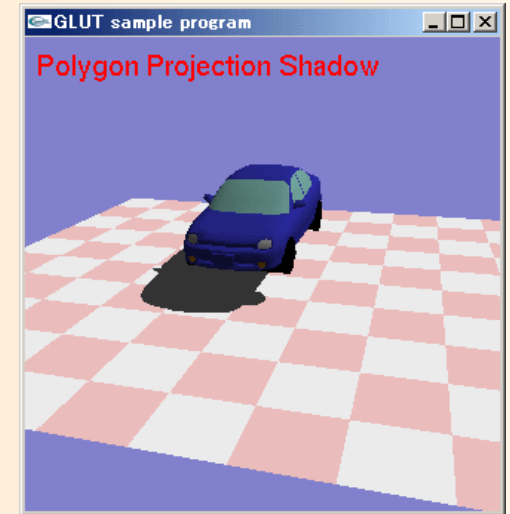
・ 影の描画

- 幾何形状モデルを指定色で描画
 - ・ 幾何形状モデルが持っている色の情報は使用せずに、全ての頂点を指定された色で描画する
- 通常の描画関数 (RenderObj関数) を参考に作成

```
//  
// 幾何形状モデル(Obj形状)の描画(固定色で描画)  
//  
void RenderObjUnicolor( const Obj * obj,  
                        float color_r, float color_g, float color_b, float color_a )  
{  
      
}
```

ブレンディング使用時の問題

- ブレンディングなし
 - 影が塗りつぶされて不自然
- ブレンディングあり
 - 複数のポリゴンが重なる箇所が暗くなってしまう
 - Zバッファが有効になっていれば同じ位置にポリゴンは重ならないはずだが、微妙な誤差のため複数回描画される点が生じる
 - ステンシルバッファを用いて回避



ステンシルバッファ

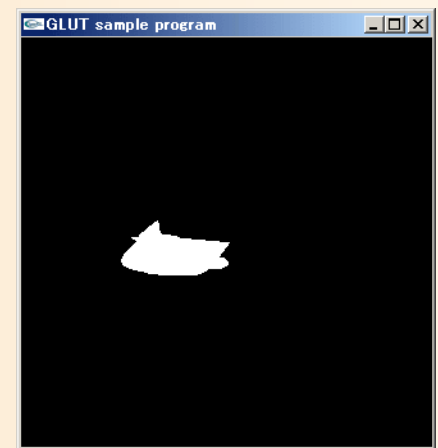
- ・ 各ピクセルへの描画の可否を制御できる
 - Zバッファと同じく、画面と同サイズの領域を持つ
 - 各ピクセルには整数値を書き込むことができる
 - Zバッファと同じく、ある条件を満たすときだけ描画が行えるように、条件を設定できる



+



+



フレームバッファ (R, G, B, A)

Z バッファ

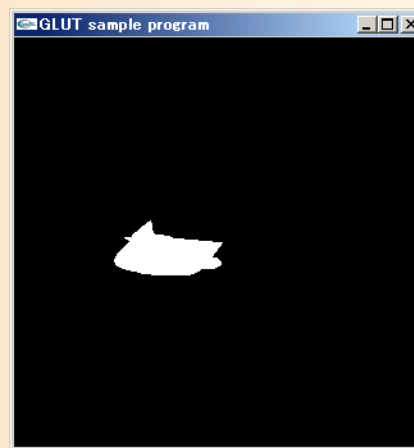
ステンシルバッファ

ステンシルバッファを使った描画

- ・ 影の重ね描きを防ぐためのフラグとして使う
 - 影のポリゴンの各ピクセルを描画するときに、ステンシルバッファに値を書き込む
 - 既にステンシルバッファに値が書き込まれていれば、そのピクセルには描画しない



カラーバッファ (R, G, B, A)



ステンシルバッファ



ステンシルバッファの初期化

- ステンシルバッファの初期化
 - 通常はステンシルバッファを持たない
 - 初期化時に指定する必要がある
 - 通常、ピクセル当たり 1bit~8bit 程度を使用
(色情報の一部を利用することもある)
- GLUTでのステンシルバッファの利用方法
 - glutInit 関数の引数に GLUT_STENCIL を指定
 - グラフィックカード・ドライバ・画面モードによっては、必ずしも成功するとは限らない



ステンシルバッファの利用

- ステンシルバッファのクリア
 - `glClear( GL_STENCIL_BUFFER_BIT );`
- ステンシルテストの有効化
 - `glEnable( GL_STENCIL_TEST );`
- ステンシルテストの設定
 - 各ピクセルのステンシルバッファの値にもとづいて描画の可否を判定
- ステンシルバッファへの書き込みの設定



ステンシルテストの設定

- `glStencilFunc( func, ref, mask )`
 - **func** には比較関数の種類を指定
 - `GL_EQUAL`, `GL_NOTEQUAL`, `GL_LESS`, `GL_GREATER`, `GL_LEEQUAL`, `GL_GEEQUAL`, `GL_NEVER`, `GL_ALWAYS`
 - 現在のステンシルバッファの値と **ref** の値を比較して、条件を満たすときにのみ書き込みを行う
 - 判定を行う前に、ステンシル値に **mask** との論理積を適用（一部のビットのみを参照できる）
 - 例： `glStencilFunc( GL_NOTEQUAL, 1, 1 );`
 - ステンシル値が 1 以外の場合のみ、描き込み



ステンシルバッファへの書き込み

- `glStencilOp( fail, zfail, zpass )`
 - それぞれ、ステンシルテストに失敗 (**fail**)、ステンシルテストは通ったがZテストに失敗 (**zfail**)、どちらも成功してピクセルを更新 (**zpass**) したときに、ステンシルバッファをどうするかを設定
 - `GL_KEEP`, `GL_ZERO`, `GL_REPLACE`, `GL_INCR`, `GL_DECR`, `GL_INVERT`
 - `GL_REPLACE` では、参照値 `ref` を書き込む
 - 例 : `glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE );`
 - ピクセル描画時に、ステンシル値に `ref` を代入

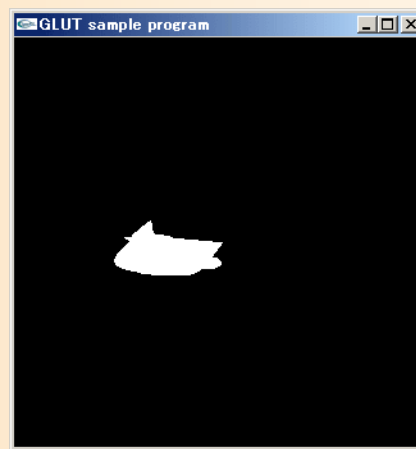


ステンシルバッファを使った描画

- ・ 影の重ね描きを防ぐためのフラグとして使う
 - 既に影が描かれたピクセルのステンシル値を 1 とする
 - `glStencilFunc( GL_NOTEQUAL, 1, 1 );`
 - ・ 既にステンシル値が 1 のピクセルには描画を行わない
 - `glStencilOp( GL_KEEP, GL_KEEP, GL_REPLACE );`
 - ・ ピクセルに描き込むときにステンシル値を 1 に設定



カラーバッファ (R, G, B, A)



ステンシルバッファ



描画処理の変更

```
// 描画オプションの設定
glDisable( GL_LIGHTING ); // ライティングは無効に設定
glEnable( GL_BLEND ); // ブレンディングを有効に設定
glEnable( GL_STENCIL_TEST ); // ステンシルバッファを使用するよう設定

// ブレンディングの設定
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );

// ステンシルバッファの設定
glStencilFunc(  );
glStencilOp(  );

// 変換行列の設定
...

// 幾何形状モデルを描画(指定色で描画)
RenderObjShadow( obj, color_r, color_g, color_b, color_a );
```

実行結果

- ・ 影のポリゴンが重なることなく描画される
- ・ 複数オブジェクトの影の重なりにも対応可能
- ・ テクスチャマッピングによる影の描画と同じく、基本的に地面にしか投影できないという問題がある



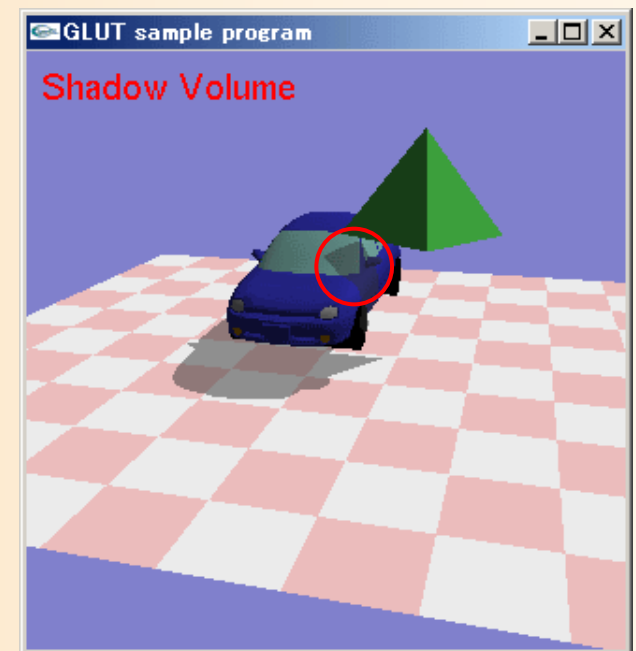
影の描画

- ・ 影の描画のプログラム
- ・ テクスチャマッピングによる影の描画
- ・ 平面へのポリゴン投影による影の描画
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング
- ・ 高度な影の描画技術
- ・ 高度な描画技術
- ・ 演習課題



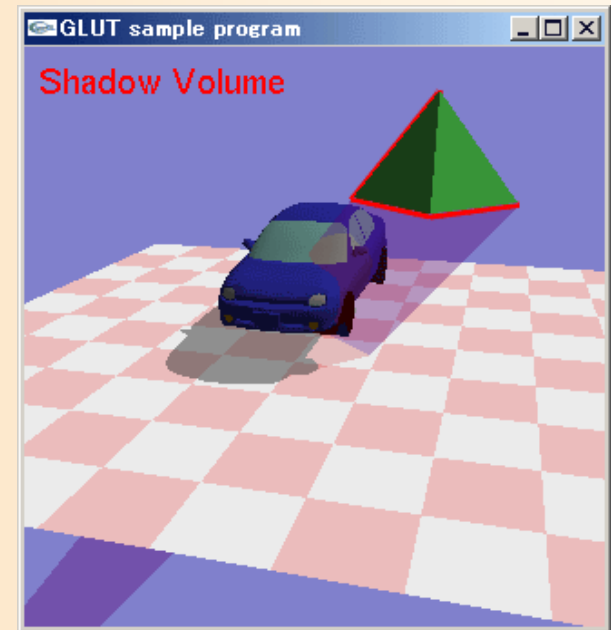
シャドウ・ヴォリューム

- ・ 他の物体や自分自身に投影される影も実現
- ・ 影になる空間領域（シャドウ・ヴォリューム）を求める
 - ステンシルバッファを利用



描画手順 (1)

- ・ 光源から見て物体の輪郭になる辺を求める
- ・ 輪郭になる辺を光の伸びる方向に拡張し、シャドウ・ボリュームを作成

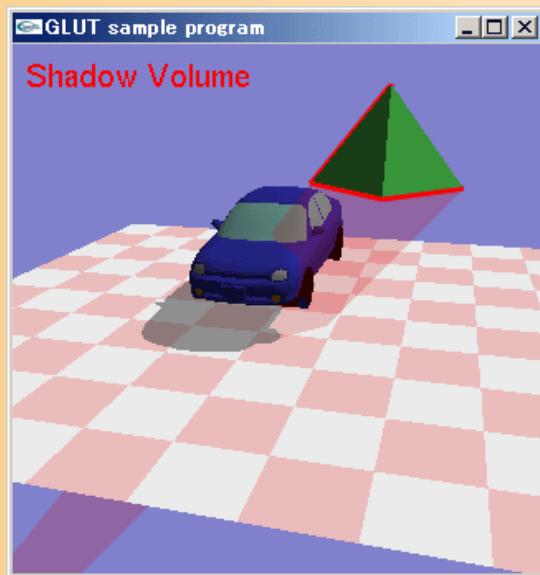


描画手順 (2)

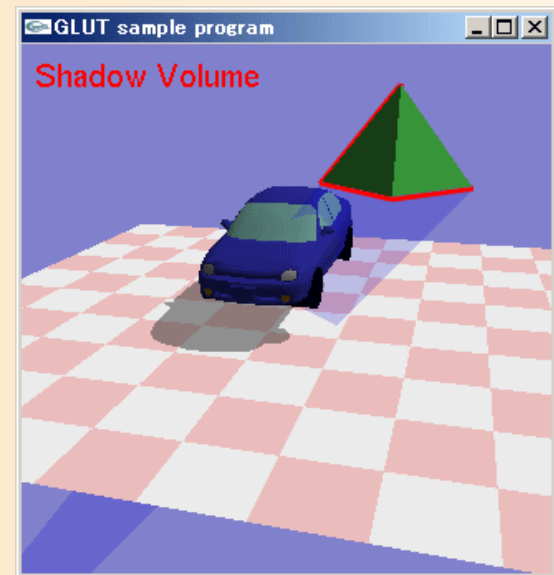
- ・ シャドウ・ボリュームの表の面の描画処理を行い、ステンシルバッファの値を加算
 - 実際の描画は行わない
- ・ 裏の面も同様に描画処理を行い、減算



表面



—

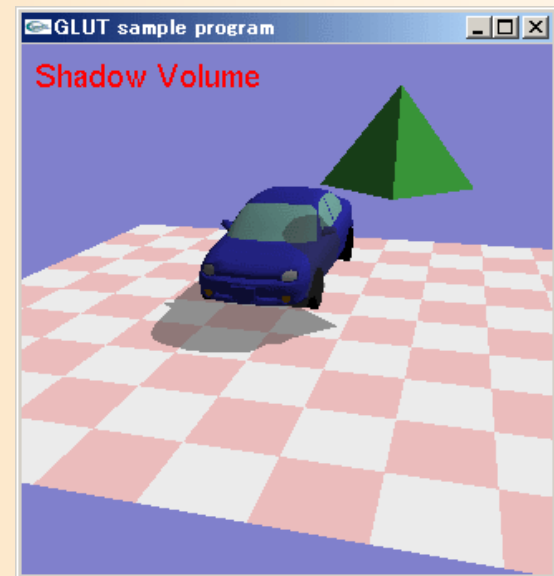
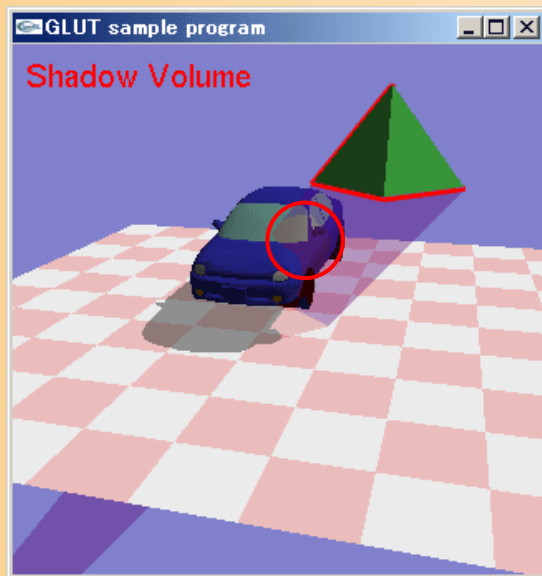


=

裏面

描画手順 (3)

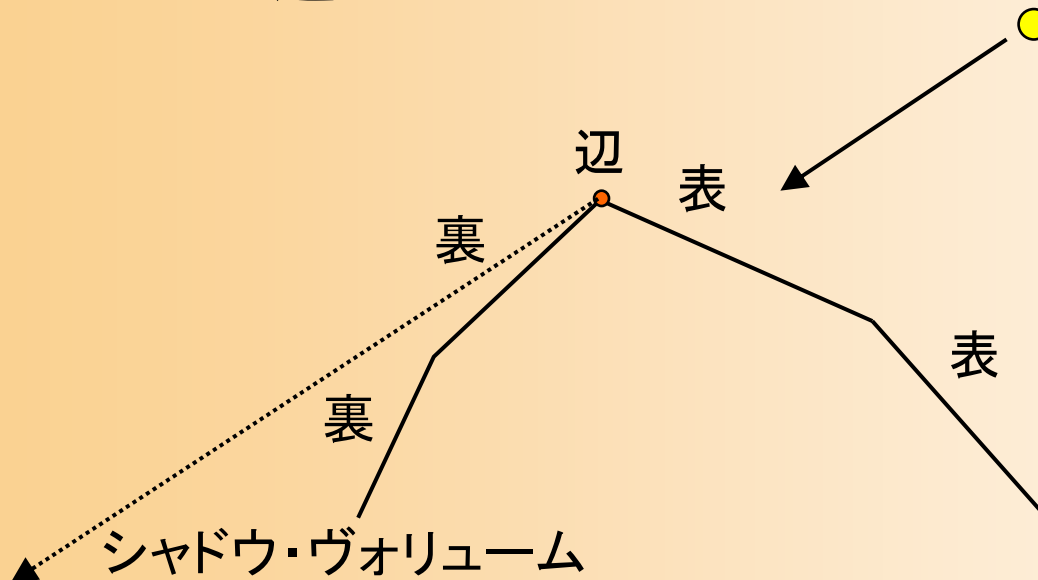
- ・ Zバッファ法による描画の結果、影の領域のみステンシルバッファの値が1以上になる
 - 表面よりも後ろで、裏面よりも前にある領域
- ・ その領域にのみアルファブレンドを適用



輪郭辺の計算 (1)

- 輪郭辺の定義

- 光源から見た物体の輪郭辺
- 辺の両側の面のうち、片側の面が光源方向を向いており、もう片側の面が光源と反対方向を向いているような辺



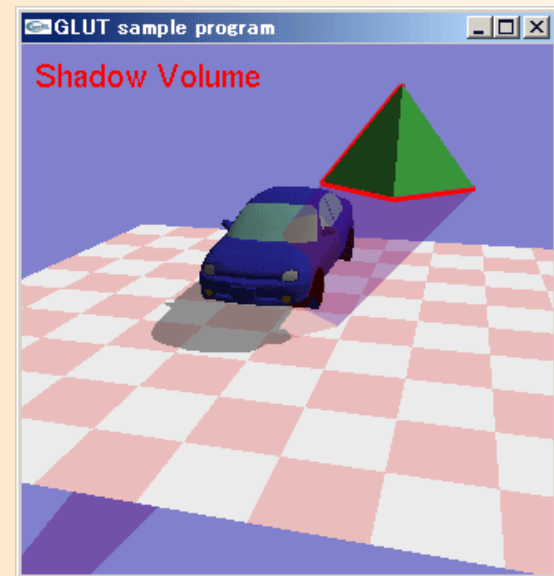
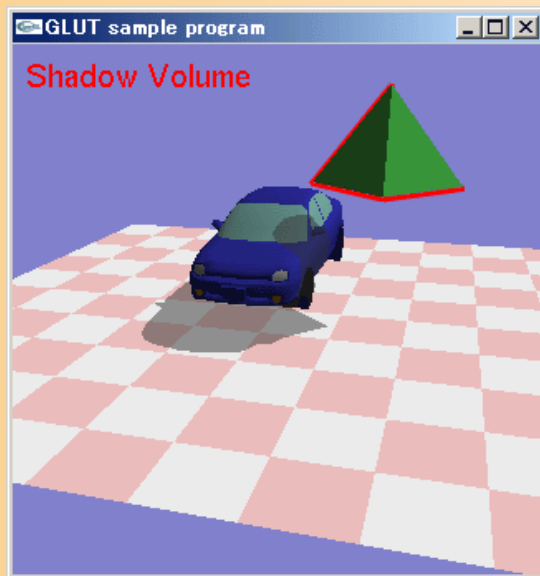
輪郭辺の計算 (2)

- ・ 各辺と各面の対応関係を前計算しておく
 - 各辺に通し番号をつけて、各辺の両側の面の番号を記録しておく
 - (通常のポリゴンモデルは辺の情報は持たない)
- ・ 物体が移動する度に、各面が光源方向を向いているかどうかを判定して記録
- ・ 上記の2つの情報をもとに、各辺が輪郭辺かどうかを判定して記録
 - このとき、辺のどちら側が表かを記録しておく



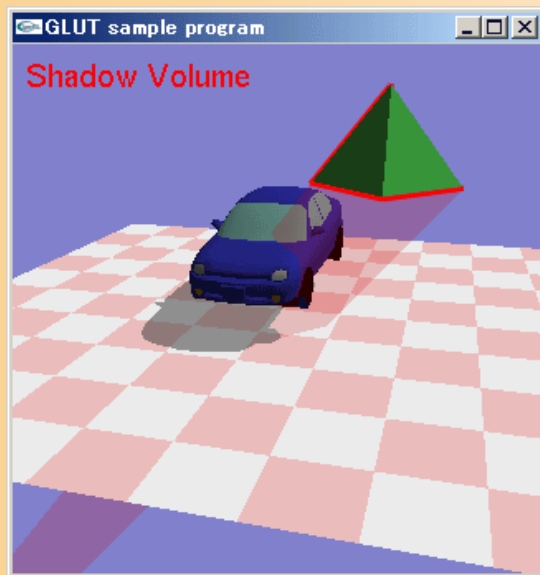
シャドウ・ボリュームの生成

- ・ 輪郭辺を光源と反対方向に延長
 - 各辺から四角面を生成
 - 四角面が表向きになるように、頂点の順番を合わせる

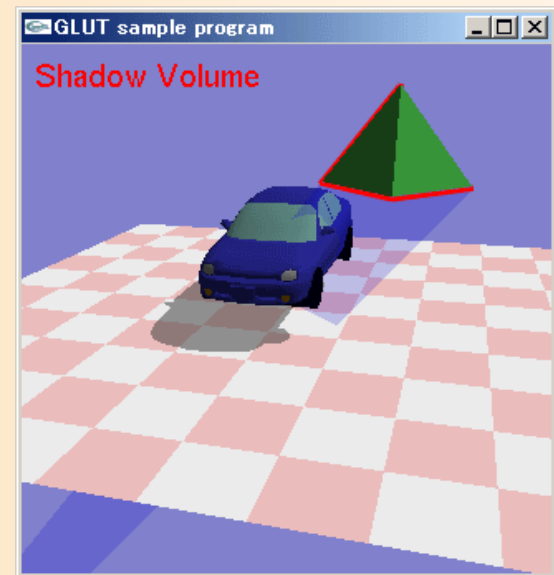


シャドウ・ボリュームの描画

- ・ 背面除去の機能を利用する
 - 表の面だけを描画（ステンシルバッファ加算）
 - 裏の面だけを描画（ステンシルバッファ減算）
 - `glCullFace( GL_FRONT or GL_BACK );`



—

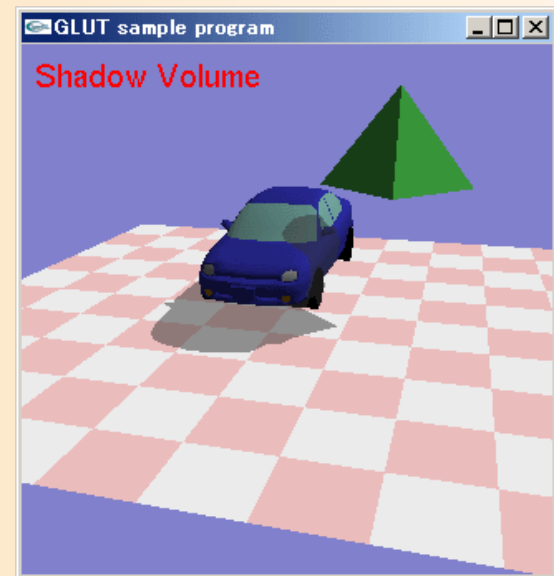
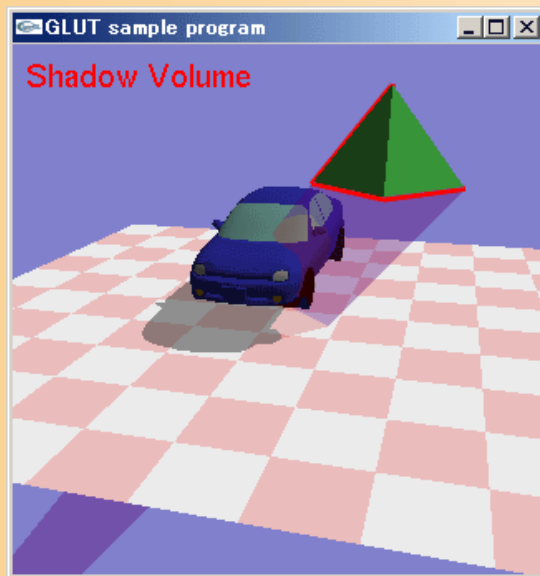


=



影の領域を暗くする

- 画面全体に、黒（灰色）の四角形を、半透明（アルファブレンディング）で描画
 - 平行投影を行うように設定（以前のテキスト描画と同様）
 - ステンシルテストを有効にして、画面全体に描画

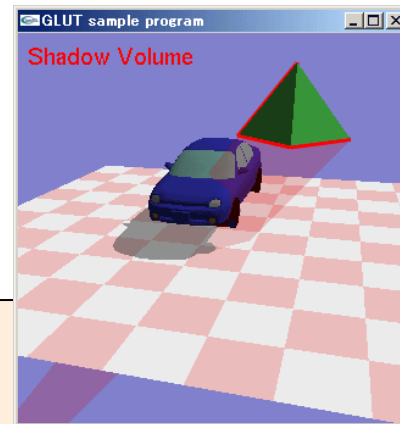
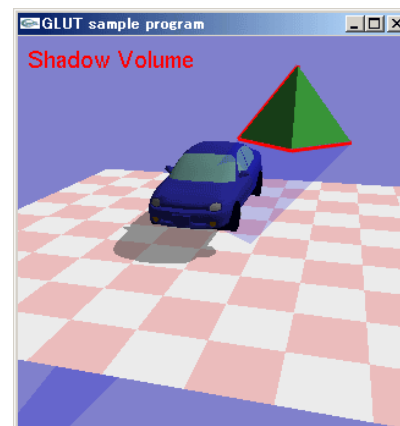


描画処理の例 (1)

```
// 光源から見たときの物体の輪郭線を計算  
vector< Vector >  contour_edges; // 輪郭線を構成する各辺の頂点座標の配列  
ComputeContourEdges( obj, contour_edges ); // 実装の詳細は省略
```

```
// シャドウ・ヴォリュームの前方の面を描画  
glStencilFunc( GL_ALWAYS, 0, 0 );  
glStencilOp( GL_KEEP, GL_KEEP, GL_INCR );  
glCullFace( GL_BACK );  
DrawVolume( contour_edges ); // 実装の詳細は省略
```

```
// シャドウ・ヴォリュームの後方の面を描画  
glStencilFunc( GL_GREATER, 0, 0xff );  
glStencilOp( GL_KEEP, GL_KEEP, GL_DEC );  
glCullFace( GL_FRONT );  
DrawVolume( contour_edges ); // 実装の詳細は省略
```



描画処理の例 (2)

```
// 画面全体に描画するための射影行列を設定(演習資料の文字描画の解説を参照)
```

```
// ブレンディングの設定
```

```
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
```

```
// ステンシルバッファの設定
```

```
glStencilFunc( GL_GREATER, 0, 0xff );
```

```
glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
```

```
// 画面全体を黒く描画
```

```
glBegin( GL_QUADS );
```

```
    glColor3f( 0.0f, 0.0f, 0.0f, 0.5f ); // α値で半透明度を指定
```

```
    glVertex3f( 0.0f, 0.0f, 0.0f );    glVertex3f( 0.0f, 1.0f, 0.0f );
```

```
    glVertex3f( 1.0f, 1.0f, 0.0f );    glVertex3f( 1.0f, 0.0f, 0.0f );
```

```
glEnd();
```

```
// 射影行列・描画設定を復元
```

問題点

- ・ 複雑な物体ではうまくいかないときがある
 - 輪郭が曲面になっている場合などで、輪郭線が正しく判定されないことがある
- ・ カメラがシャドウ・ヴォリュームの中に入ると、正しい画像が生成されない
- ・ 処理時間がかかる
 - 輪郭辺の計算
 - ・ ハードウェアでの実現が困難
 - シャドウ・ヴォリュームの描画
 - 画面全体への四角形の描画



輪郭辺の計算の応用

- ・ 視線から見た輪郭を計算して描画することで、アニメ絵風の効果が出せる（トゥーン・レンダリング）



視線方向から見た輪郭を計算



影の描画

- ・ 影の描画のプログラム
- ・ テクスチャマッピングによる影の描画
- ・ 平面へのポリゴン投影による影の描画
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング
- ・ 高度な影の描画技術
- ・ 高度な描画技術
- ・ 演習課題



シャドウ・マッピングの概要

- ・ マルチパス・レンダリング

1. まず光源から物体を見た画像をレンダリング

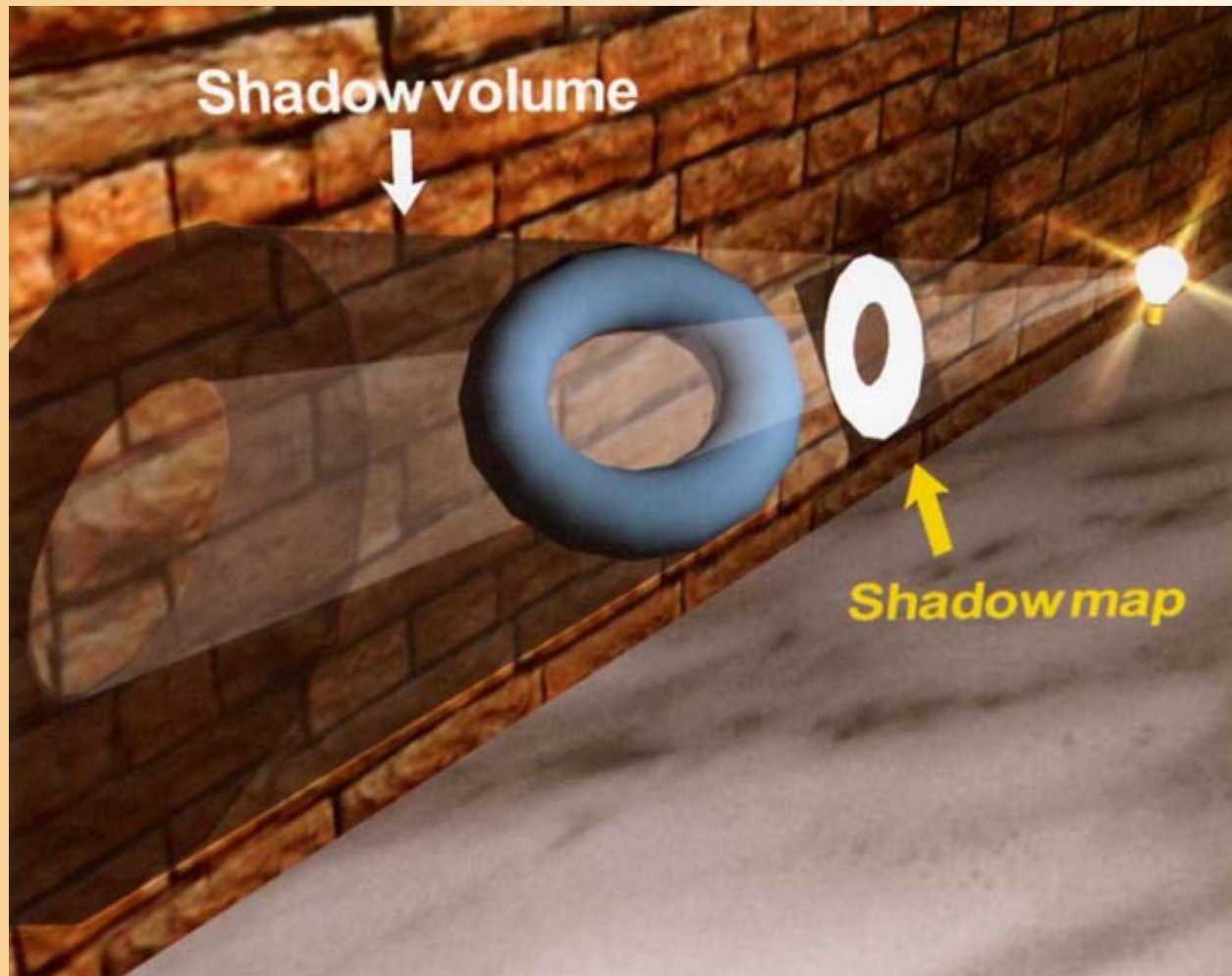
- この結果をシャドウマップとする

2. 物体が投影される面を描画するときに、シャドウマップをテクスチャマッピングする

- 適切な位置に投影されるように、各頂点のテクスチャ座標を計算



シャドウ・マッピングの図解



[Game Programming GemsII, Gabor Nagy]

シャドウ・マッピングの詳細

- ・ 詳しい実装方法については省略
 - 基本的にはこれまでに紹介した技術の組み合わせで実現できる
 - テクスチャへのレンダリングが必要になる
- ・ シャドウ・マッピングの種類の違い
 - シャドウ・マッピング（奥行き値なし）
 - 深度マッピング（奥行き値あり）
- ・ スキャンライン法と組み合わせて、オフラインでの高品質なレンダリングにも用いられる



影の描画

- ・ 影の描画のプログラム
- ・ テクスチャマッピングによる影の描画
- ・ 平面へのポリゴン投影による影の描画
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング
- ・ 高度な影の描画技術
- ・ 高度な描画技術
- ・ 演習課題



セルフ・シャドウ

- ・ 自分自身への影
- ・ 実現方法
 - シャドウ・ボリュームを使えば、実現可能
 - シャドウ・マッピングで実現することは困難
(腕と胴体を別の物体として描画するなどの工夫が必要)



[SCE, ワンダと巨像]

参考資料：

3Dゲームファンのための「ワンダと巨像」グラフィックス講座

<http://www.watch.impress.co.jp/game/docs/20051207/3dwa.htm>



ソフト・シャドウ

- ・ 輪郭がぼやけたような影
 - 現実世界では、ひとつの点光源ではないので、本来は影の輪郭はぼやける



[GPU GemsII, Yury Uralsky]

- ・ 実現方法
 - シャドウ・ボリュームでは、実現は困難
(光源を微妙にずらして複数回レンダリングなどすれば、時間はかかるが可能)
 - シャドウ・マッピングでは、シャドウマップをぼかしたりすることで、実現可能
 - ソフト・シャドウの描画に特化した手法もある



影の実現方法の比較

・ シャドウ・ヴォリューム

- シーンのポリゴン数に大きく影響を受ける
- ある程度高いフィルレートが必要
- 機能自体は、古いハードウェアでも実行可能
- ソフトシャドウの実現は困難

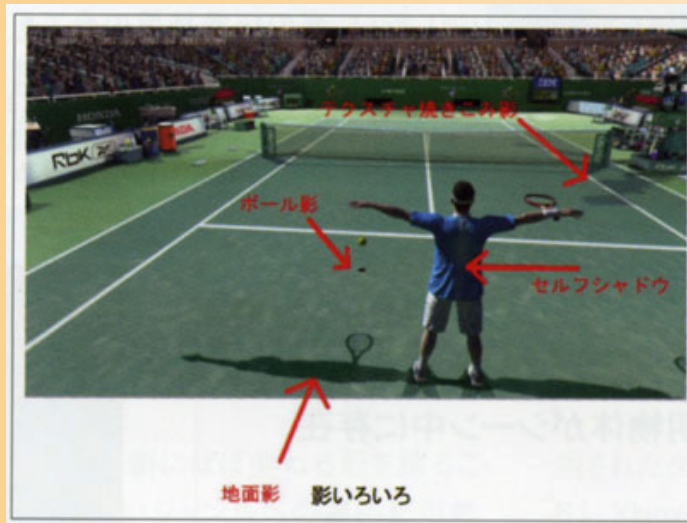
・ シャドウ・マップ

- シーンのポリゴン数にはあまり影響は受けない
- オフスクリーンレンダリングやマルチテクスチャに対応した環境が必要
- セルフシャドウの実現は困難



影の描画方法の使い分け

・ コンピュータゲームでの影の使い分けの例



影の描画における種類分け。動くか動かないか、などの性質によってそれぞれに異なる描画手法が採られた

[セガ, パワースマッシュ3]

参考資料:
CG WORLD 2007年
12月号
「CEDEC 2007 技術
トラック解説」

- シャドウマップによるセルフシャドウ (人物)
- 動く影 (人物やボールから地面への影) は、ポリゴン投影 (テニスコートは平面であることを利用)
- 動かない影は地面のテクスチャに焼き込み



影の描画

- ・ 影の描画のプログラム
- ・ テクスチャマッピングによる影の描画
- ・ 平面へのポリゴン投影による影の描画
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング
- ・ 高度な影の描画技術
- ・ 高度な描画技術
- ・ 演習課題



高度な描画技術

- ・ 今回は影の描画のみを扱ったが、自然な画像を生成するための高度な描画技術は多く開発・利用されている
 - 本科目では扱わない
- ・ 一般的な描画技術について、簡単に紹介



高度な描画技術

- ・ バンプマッピング

- 表面の凹凸を表す画像を適用し、法線を変化させることで、凹凸を表現

- ・ 環境マッピング

- 周囲の風景の映り込みを、半透明のテクスチャマッピングにより表現

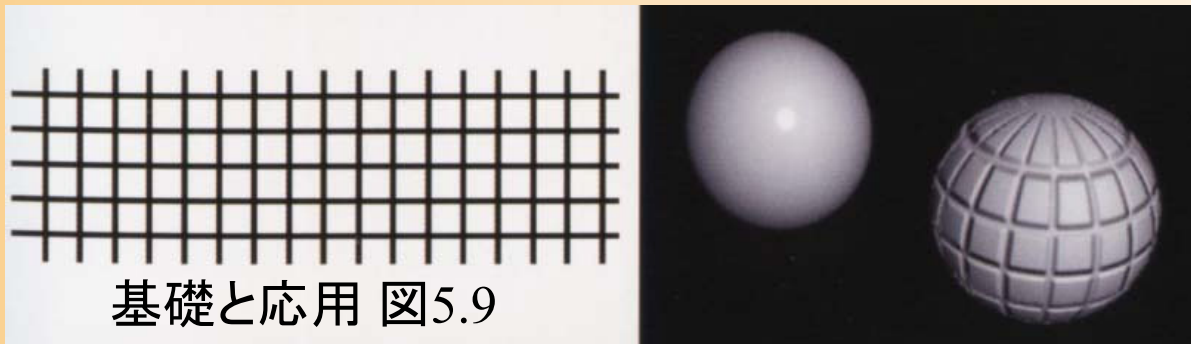
- ・ 大域照明（ラジオシティ、フォトンマップ）

- 各面に当たる環境光をより正確に計算
- 事前計算しておいたデータを適用



高度なマッピング（復習）

- 凹凸のマッピング（バンプマッピング）



- 周囲の風景のマッピング（環境マッピング）

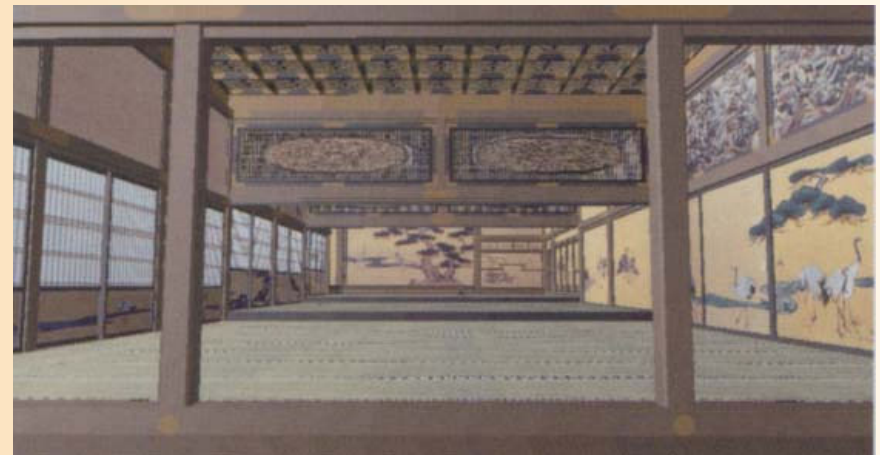


大域照明の効果の例（復習）

- 大域照明を考慮して描画することで、より写実的な画像を得ることができる



映り込み(大域照明)を考慮
基礎と応用 図8.9



環境光(大域照明)を考慮
基礎と応用 図9.1, 9.2

大域照明の効果の例（復習）

- 大域照明を考慮して描画することで、より写実的な画像を得ることができる



映り込み(大域照明)を考慮
基礎と応用 図8.9



環境光(大域照明)を考慮
基礎と応用 図9.1, 9.2

影の描画

- ・ 影の描画のプログラム
- ・ テクスチャマッピングによる影の描画
- ・ 平面へのポリゴン投影による影の描画
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング
- ・ 高度な影の描画技術
- ・ 高度な描画技術
- ・ 演習課題



演習課題

- ・ 今回のプログラミング演習は自由演習課題とするので、興味があれば各自で取り組む
 - 成績評価には考慮しない
- ・ 影の描画を実現するプログラムを作成せよ
 1. テクスチャマッピングによる影の描画
 2. ポリゴン投影による影の描画
 - サンプルプログラム (shadow_sample.cpp) をもとに作成する



自由演習課題の発展

- ・ より高度な技術に興味があれば、以下の処理を実現するような拡張が可能
- ・ 頂点配列を使った幾何形状モデルの描画
 - 頂点配列の使用に適したデータ表現への変換
- ・ シャドウ・ヴォリュームによる影の描画
- ・ シャドウ・マップによる影の描画



影の描画のまとめ

- ・ 影の描画のプログラム
- ・ テクスチャマッピングによる影の描画
- ・ 平面へのポリゴン投影による影の描画
- ・ シャドウ・ヴォリューム
- ・ シャドウ・マッピング
- ・ 高度な影の描画技術
- ・ 高度な描画技術
- ・ 演習課題

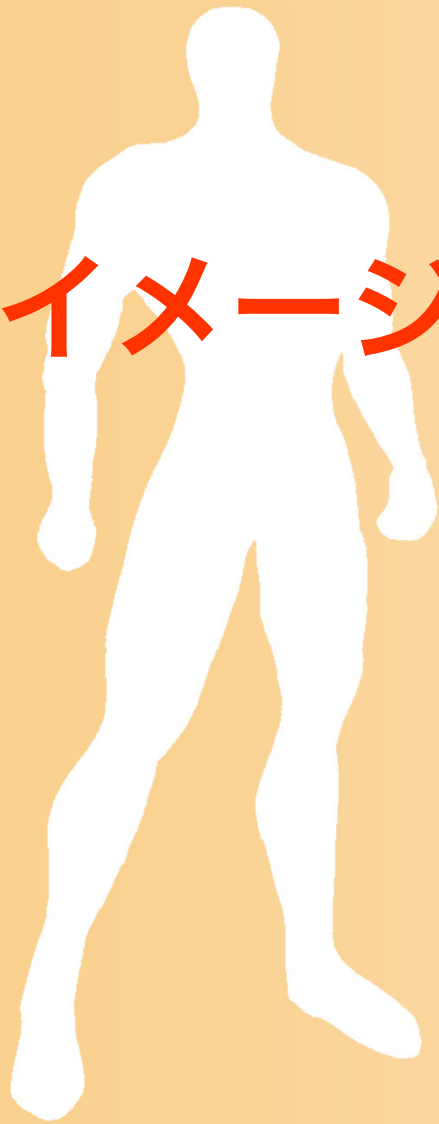


今回の内容

- ・ 影の描画
 - ・ イメージベースド・レンダリング
 - ・ BRDFによる質感の表現
 - ・ HDRレンダリング
-
- ・ 講義のまとめ



イメージベースド・レンダリング (モデリング)



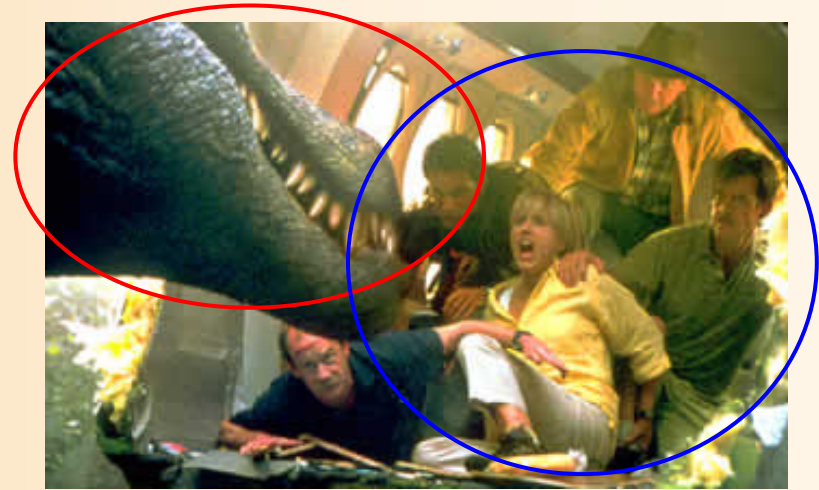
3Dグラフィックスと実写の関係

・ 3Dグラフィックス

- 制作には労力がかかる
- 存在しないものも表現可
- 人間などの再現は難しい

・ 実写

- 実物をそのまま撮影できる
- 人間などは実写の方が向いている



Jurassic Park III
Universal Pictures

- ・ 従来は、両者をうまく使い分けて合成する、という方法がとられてきた



問題点

- ・ 3Dグラフィックスでは、現実世界同様の形状・テクスチャ・素材（反射特性）を作成するのに多くの時間がかかってしまう
- ・ グラフィックスだけ、実写だけでは作成が難しい映像ものもある
 - － 例： 激しく吹き飛ばされる人間など

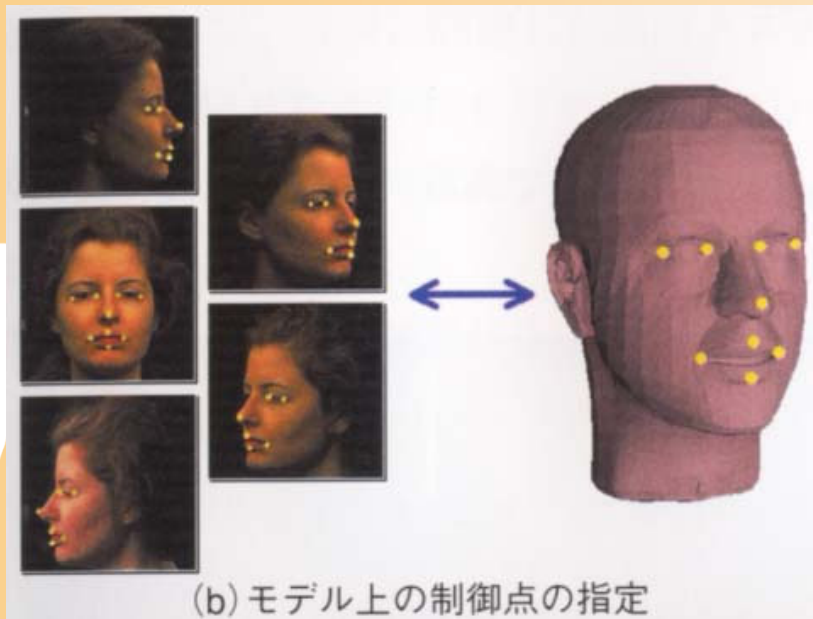
→ 解決方法

- － 現実世界のデータを取り込んで、3Dグラフィックスの世界を作る



イメージベースド・レンダリング (モデリング) (1)

- ・ 実際の画像から、3Dモデルの情報を取得
 - 形状、テクスチャ、反射特性
 - ・ 入力画像は、必ずしも、実写画像である必要はない
 - ・ 下は、複数画像から、1枚のテクスチャを取り出す例



イメージベースド・レンダリング (モデリング) (2)

- ・ 主に2つの方向性がある
- ・ 3Dモデルの正確な形状・素材・反射特性などを画像から取得する
 - レンダリングには、一般的な 3Dグラフィックスの方法を使う
- ・ 取得した画像を使って、物体をそのまま描画
 - 画像を使ったレンダリングに特化した、モデル表現・レンダリング手法を使う



3Dモデル・テクスチャ取得の例

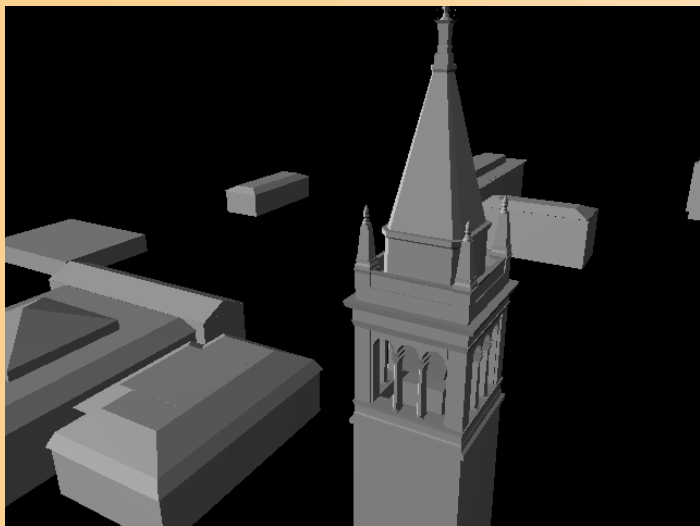
- ・ 有名な UC Berkeley の塔の例（1997年）
 - Debevec ら
 - <http://www.debevec.org/>



Tower Photographs

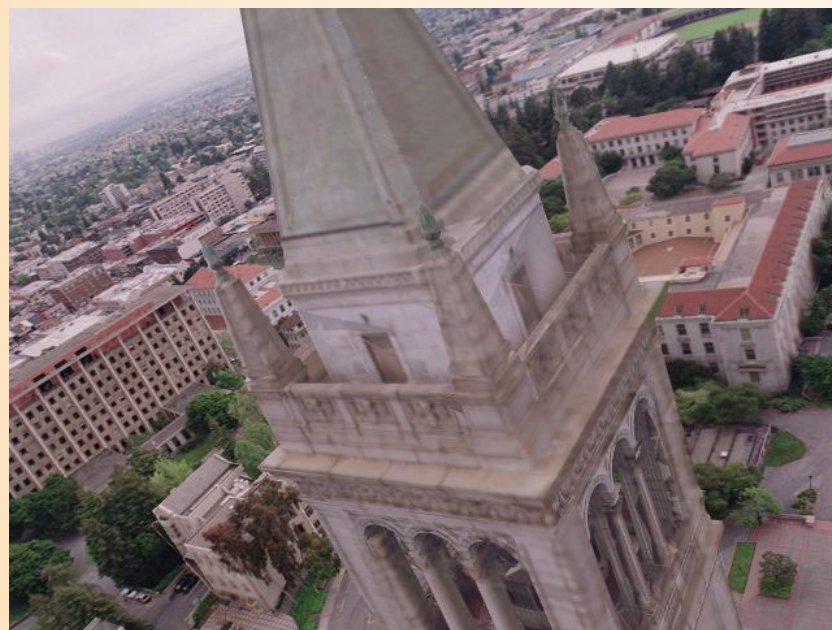


Environment Photographs



形状データの推定

取得した形状データ+テクスチャを使って描画



映画などへの応用

- **Matrixの有名なシーン**

- 背景はすべてCGで描画

- 前のスライドで紹介した手法が使われている

- 人物については実写

- 俳優の周囲に並べた多数のカメラで撮影
 - 各カメラの画像を順に補間



Matrix
Warner Bros.



周囲の環境（光）を取得

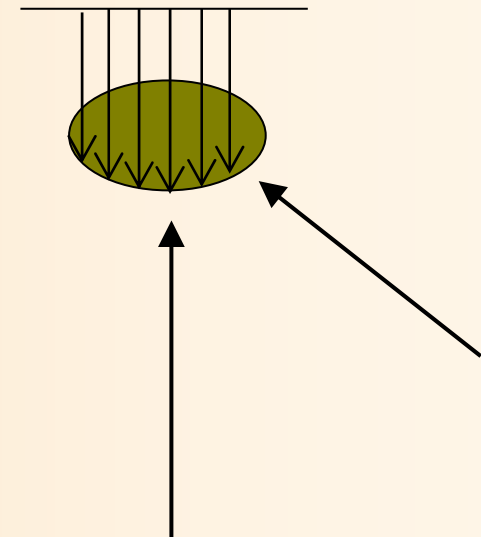
- ・ 周囲から得た環境光の情報を人物の顔（CG）のレンダリングに適用



教科書 図5.46
Debevec 2002

画像をそのまま描画する方法

- ・ 画像から奥行き値を計算
- ・ 奥行き値を使い、異なる視点の画像を生成
 - もとの画像で隠れている箇所は描画できないことに注意



BRDF



BRDF

- **BRDF**
 - Bi-directional Reflectance Distribution Function
 - 双方向反射分布
- 現実世界の物質の反射特性を正確に再現するための技術
- イメージベースドレンダリングの考え方を発展させたもの



反射特性のモデル

・ フォンのモデル

– 拡散反射光

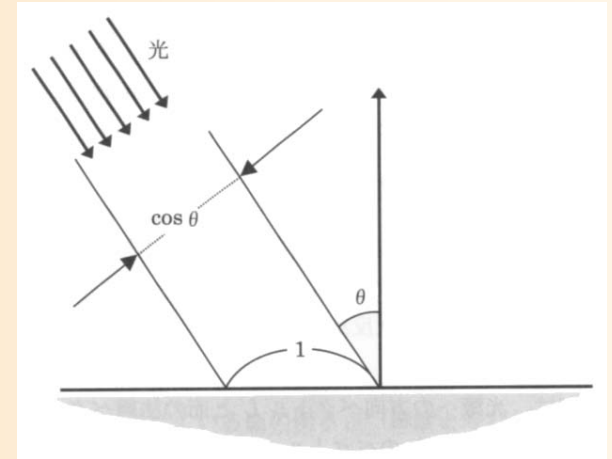
$$I_d = k_d I_l (N \cdot L)$$

– 鏡面反射光

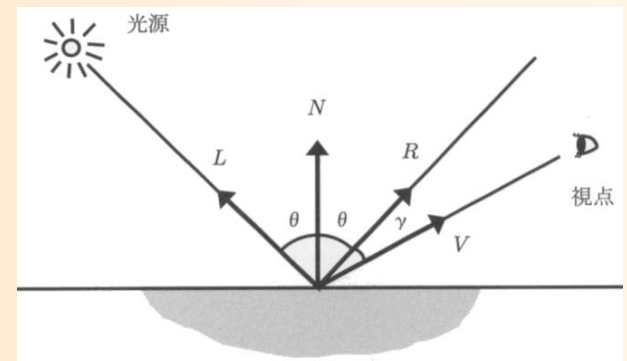
$$I_d = k_s I_l (R \cdot V)^n$$

・ 現実の物体

- 表面は平らではなく、乱反射などが起こる
- モデルとのずれが生じる



基礎と応用 図2.9

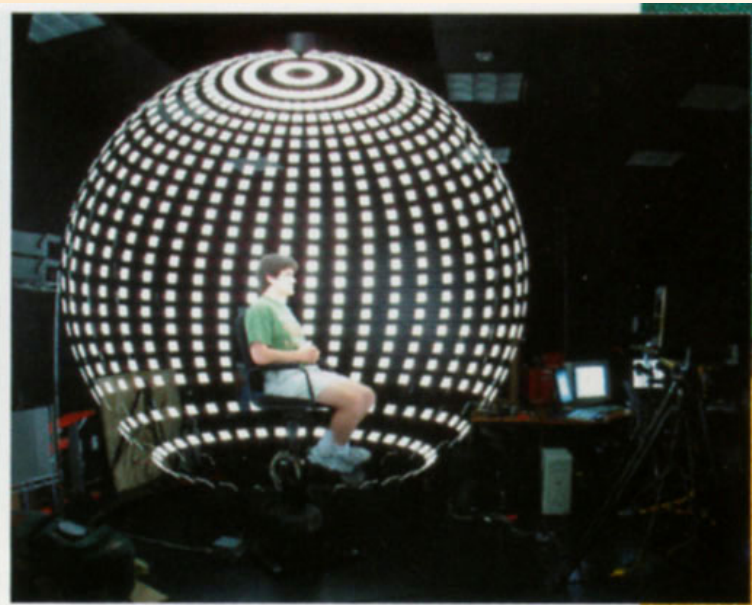
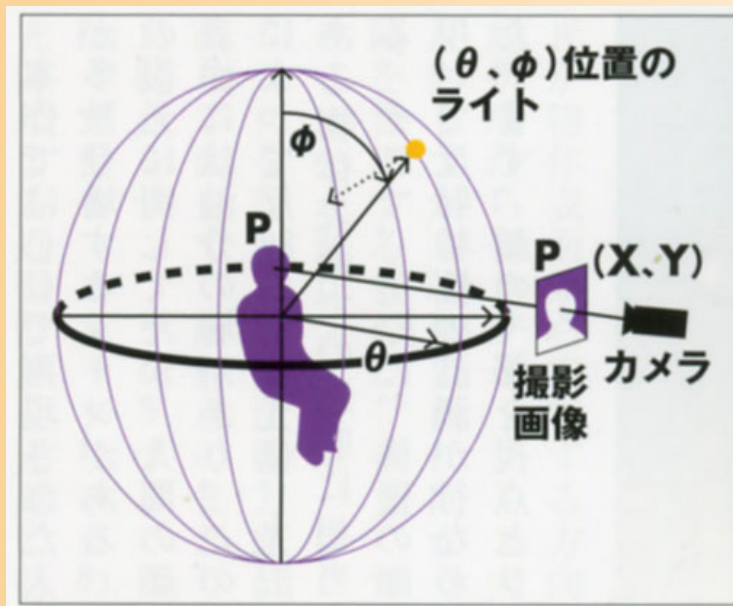


基礎と応用 図2.10

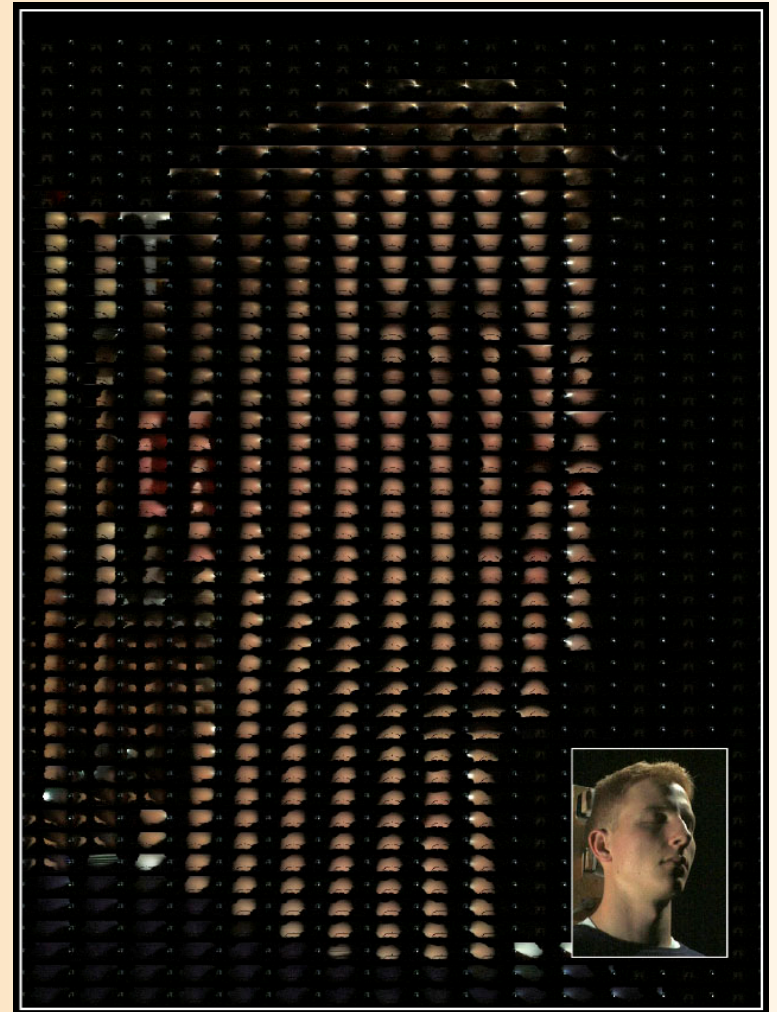


BRDFの考え方

- ・ 反射特性を表現
 - 視点方向・光源方向の関数によって表される
 - ・ 法線に対する視点方向・光源方向
 - 特殊な装置を使って実際の素材から計測

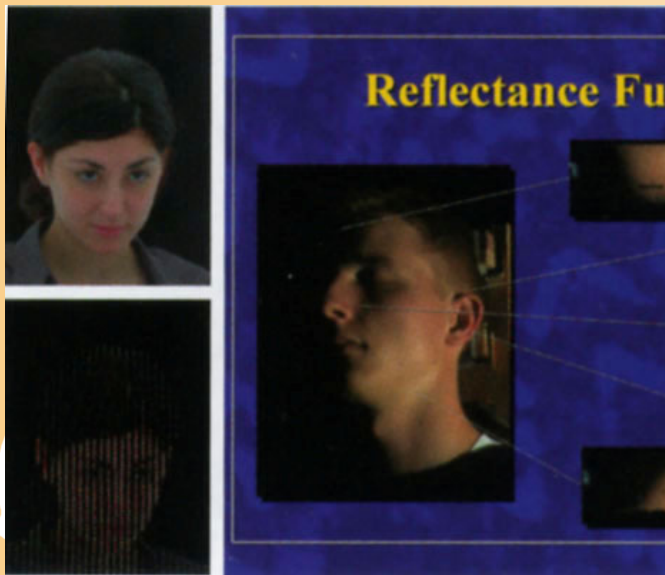


サンプル画像の取得



映画への応用 (1)

- Spider-Man 2 (2004)
 - 俳優を再現したCGキャラクターの、顔の皮膚の質感を再現



CG WORLD Spider-Mann 2
2004年12月号 Sony Pictures

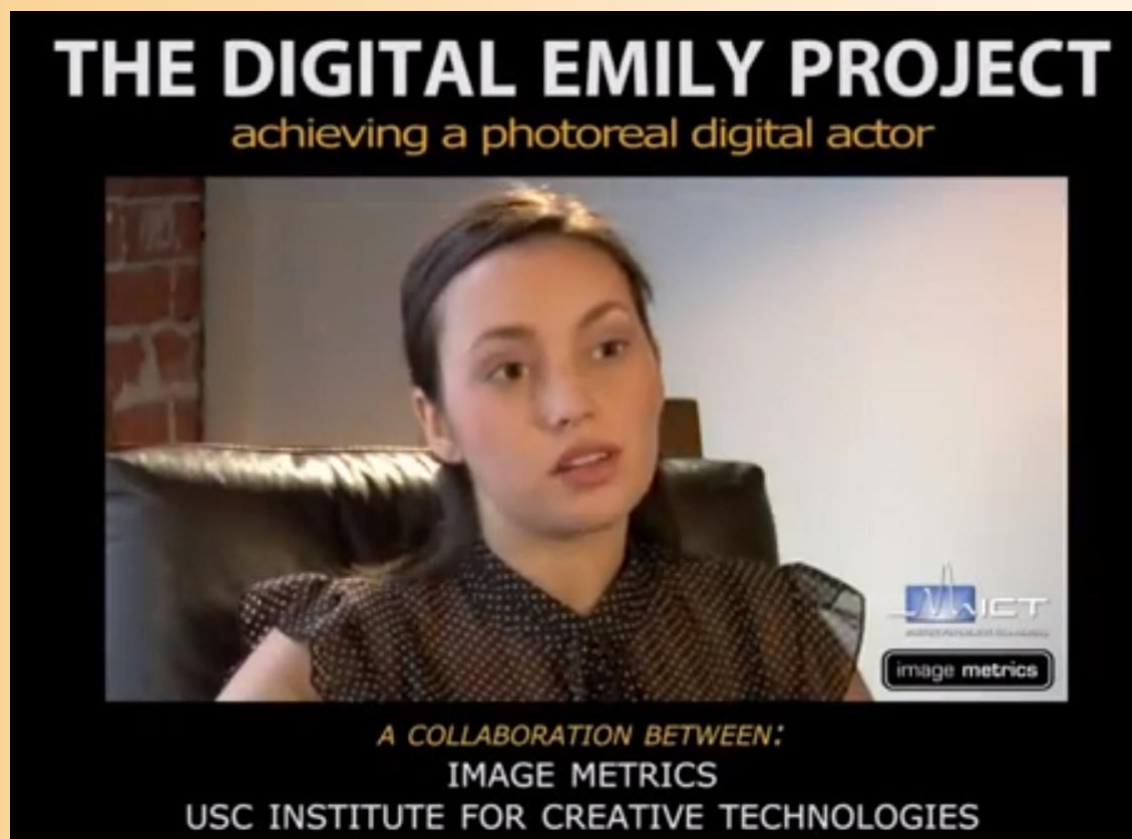
映画への応用 (2)

- **Matrix Reloaded / Revolutions (2003)**
 - 俳優を再現したCGキャラクタにBRDFが使われた最初の例



デジタル・エミリー

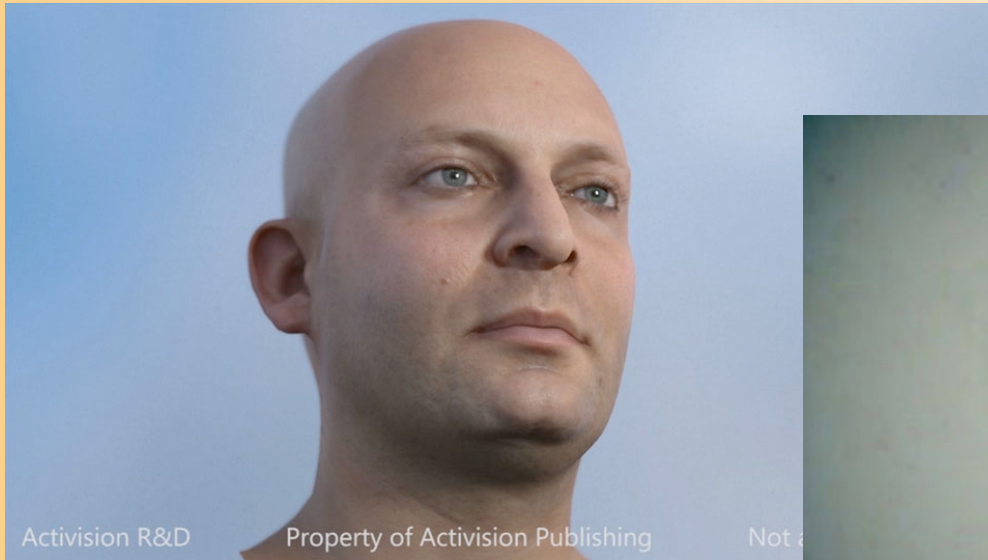
- 俳優の表情・質感をCGで再現（2010）



<http://gl.ict.usc.edu/Research/DigitalEmily/>

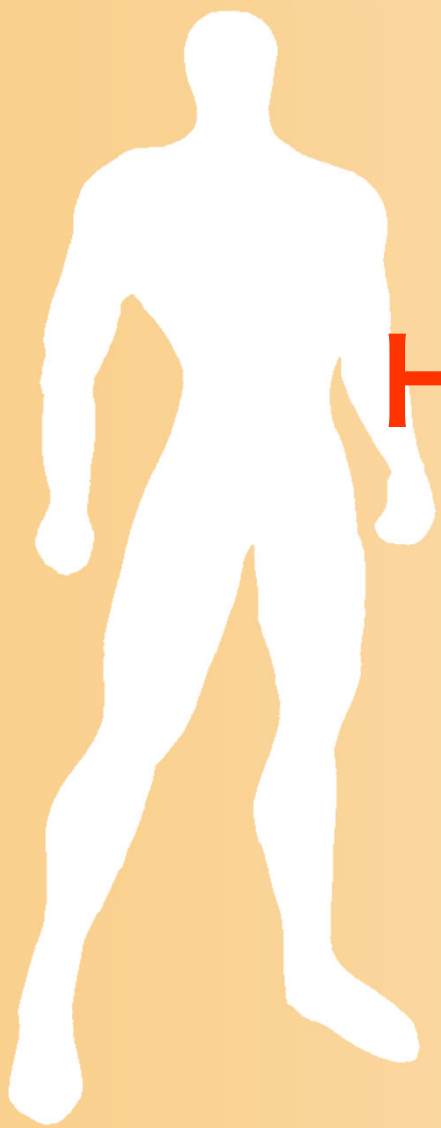
リアルタイム・レンダリング

- Digital Ira (2012)
- Luminous Studio Square Enix



2015 © SQUARE ENIX CO., LTD

<http://ict.usc.edu/prototypes/digital-ira/>



HDRレンダリング

HDRレンダリング

- High Dynamic Range (HDR)

- 人間の目がとらえることのできる輝度の範囲は、かなり広い
 - ・ R:G:B = 8:8:8 = 24ビットで表せる範囲より広い
- 浮動小数点を使って、各ピクセルのより正確な輝度（色）を表現する

- HDRレンダリング (HDRR)

- High Dynamic Range を考慮して、浮動小数点バッファ（16ビット or 32ビット）にレンダリング
- 画面に表示する際に、24ビット画像に変換
 - ・ HDRモニタであれば、浮動小数点のまま表示可能



視覚のダイナミックレンジ

月なし

月夜

薄明かり

屋内

日照

円錐体

桿状体

16bit 浮動小数点

8bit 整数

1E-6

1E-4

1E-2

1E+0

1E+2

1E+4

1E+6

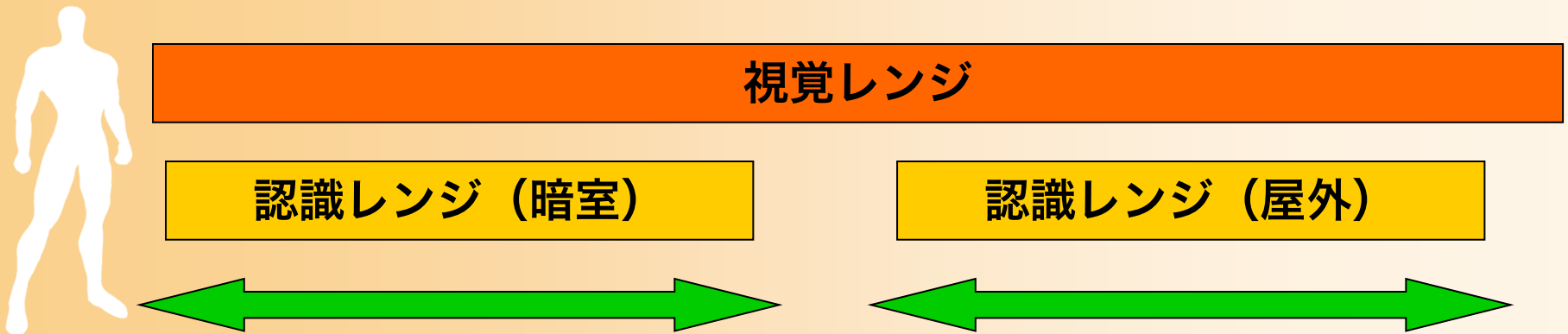
輝度 (lum/m²/sr)

DIRECTX

参考資料: <http://www.watch.impress.co.jp/game/docs/20050525/3dh121c.htm>

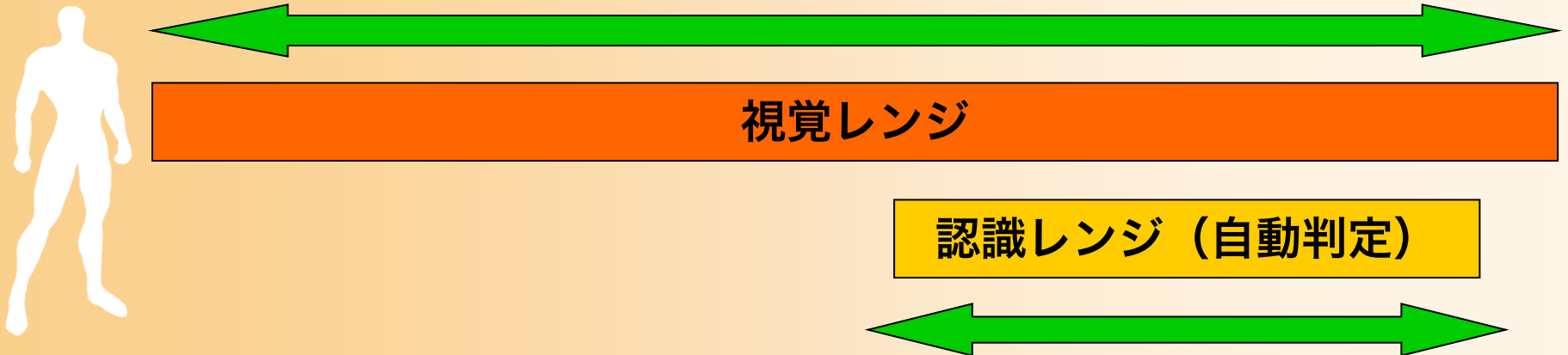
通常のレンダリング

- ・ 人間の視覚レンジは広い
- ・ 周囲の明るさに応じて、そのうち一部を認識
- ・ 認識レンジ（範囲）は動的に調節される
- ・ 通常は、適切な固定の認識レンジに24ビットを割り当てることで、レンダリング



HDRレンダリングの概要

- ・ レンダリングの段階では、HDRで処理を行う
 - 24ビットよりも高い精度・広い範囲を使用
- ・ 適切な範囲を表示色にマッピングする
 - 従来のモニタでは、24ビットに落とす必要がある
 - シーンの明るさに応じて認識レンジを動的に調節



HDRレンダリングによる効果

- ・ 周囲の明るさの変化によるHDRRの変化
 - 暗いところから明るいところへ出たときなどの、徐々に環境に目が慣れる効果が再現できる
- ・ 激しく明るい箇所の表現
 - まぶしく輝くような効果を再現できる
- ・ 計算の精度が上がる
 - いままでの方法では暗くて描画されなかった物体も、強い光が当たると見えるようになる、など
 - ・ 暗い素材 × 非常に明るい光 = 普通の色に見える



明るさの変化による効果

視覚レンジ

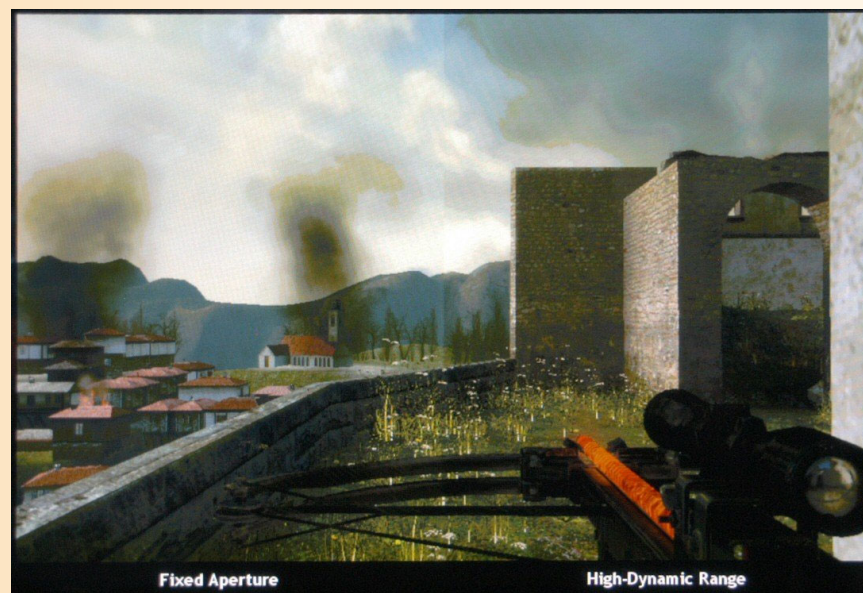
認識 →

認識レンジ（自動判定）

急に明るいところに出る（右半分がHDR）



徐々に目が慣れる



激しく明るい箇所の表現

- ・ 太陽光源、太陽光源が直接反射する場所
 - 激しく輝いて見える
 - 周囲のピクセルまでぼんやりと明るくなる
 - ・ 周囲のピクセルの輝度も何らかのルールで明るくする



「DOUBLE S.T.E.A.L.」(ぶんか社)



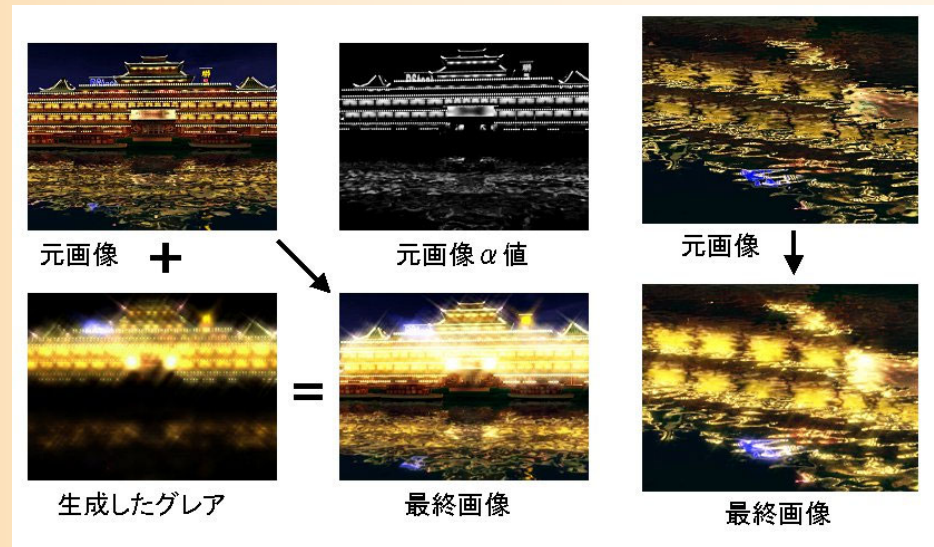
Hal Life 2

疑似HDRレンダリング

- ・ 激しく明るい箇所に特殊な効果を適用する手法は、これまでも使われていた
 - フレア効果（光源が直接見えるときに描画）
 - グレア効果（画像を混合して描画）



「Incoming」(Rage)



川瀬正樹 (ぶんか社)

HDRレンダリングの他の応用例



vVidia

[http://www.nzone.com/object/
nzone_timbury_downloads.html](http://www.nzone.com/object/nzone_timbury_downloads.html)

Age of Empires III

今回の内容

- ・ 影の描画
 - ・ イメージベースド・レンダリング
 - ・ BRDFによる質感の表現
 - ・ HDRレンダリング
-
- ・ 講義のまとめ



レンダリングの最新技術

・ 最新技術の実用

- 今回紹介したような技術は、実際の映像作品やゲーム制作で、実用的に使われている
 - ・ 市販のレンダリングソフトウェアやゲームエンジンでの実装、各自での実装
 - GPUを使った実装、リアルタイム（高速）描画の実現
 - 高度なプログラミングが必要になる
 - ・ データの準備など、実際の利用には手間がかかる
- 十分に写實的（リアル）な表現が実現されている

- ・ 精度（リアリティ）・速度・利用のしやすさなどの点では、まだ課題が残されている



今回の内容

- ・ 影の描画
- ・ イメージベースド・レンダリング
- ・ BRDFによる質感の表現
- ・ HDRレンダリング

- ・ 講義のまとめ





講義のまとめ

本科目の達成目標

- ・ コンピュータアニメーションの応用技術を学習する
- ・ コンピュータアニメーションを利用したソフトウェアを開発する上で役に立つ最新技術を、実際の演習を交えながら習得する

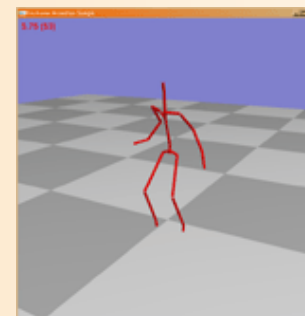
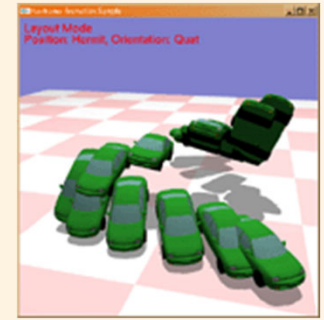


本科目の内容

• 本科目で学ぶ応用技術

- 視点操作
- 幾何形状データの読み込み
- キーフレームアニメーション
- 物理シミュレーション
- 衝突判定、ピッキング
- キャラクタアニメーション
- レンダリング技術

特にキャラクタアニメーション
関連の技術を重点的に扱う



成績評価

- ・ **プログラミング演習課題（80%）**
 - 演習課題のプログラムを作成し、レポートを提出
 - レポート提出後に演習問題（ミニテスト）を実施
- ・ **演習問題（20%）**
 - 毎回の授業に出席して、講義内容を理解する
 - 授業中に演習問題（ミニテスト）を実施
 - 授業後に復習用の演習問題を公開する
 - ・ 復習問題の点数も成績に考慮する
 - **上記以上の詳細や、個別の成績評価に関する質問には回答しない**



参考：CGエンジニア検定

- ・ CGエンジニア検定（CG-ARTS協会）
 - ベーシック／エキスパート
 - 年2回実施、福岡でも受験可能
 - 学部のコンピュータグラフィックス相当の内容
 - ・ CG-ARTS協会のテキストの内容を理解していれば、合格できる
 - ・ エキスパートは、やや応用的な問題が出題される
 - 本学の受験者は割引を受けられる



授業アンケートへの回答

- 大学院 授業アンケート
 - LiveCampusにログインして回答
 - 回答期間は掲示を参照
- 成績評価には一切影響しません



まとめ

- ・ 影の描画
- ・ イメージベースド・レンダリング
- ・ BRDFによる質感の表現
- ・ HDRレンダリング
- ・ 講義のまとめ

