

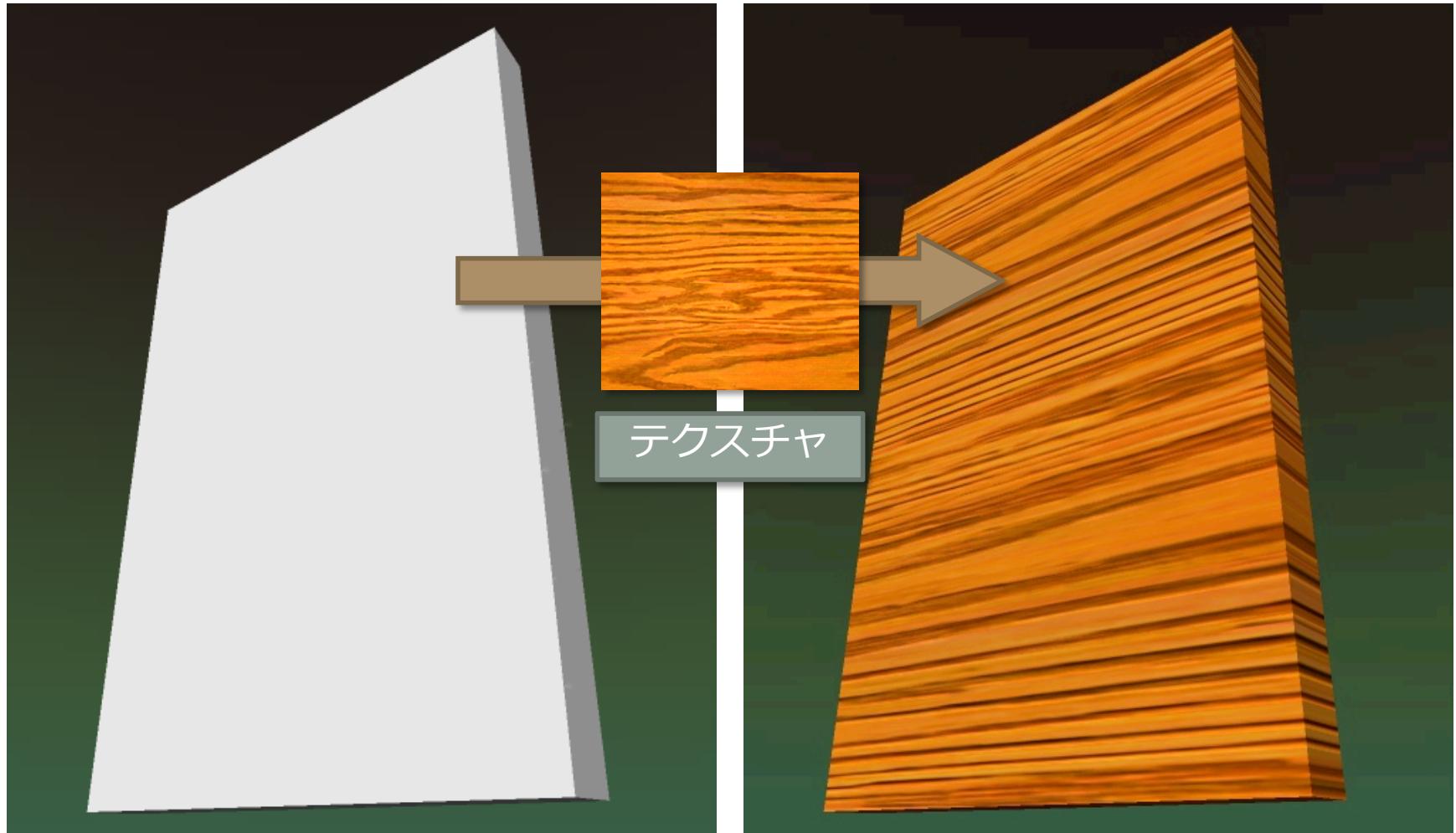
ゲームグラフィックス特論

第8回 テクスチャマッピング (1)

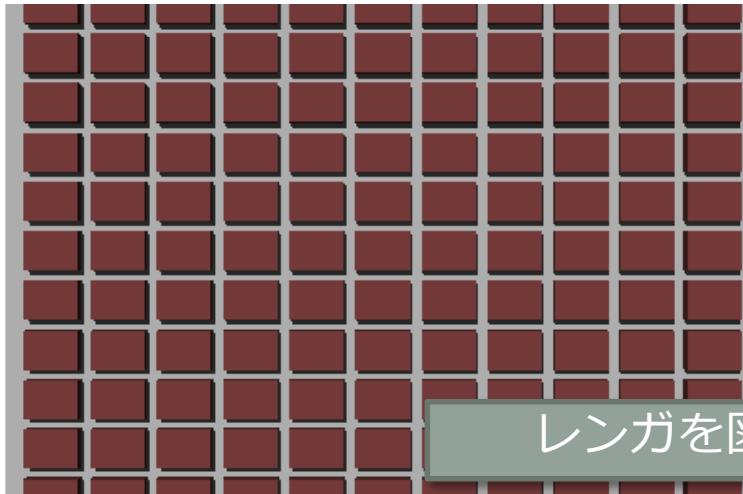
物体表面に画像を貼付ける

表面材質の画像による制御

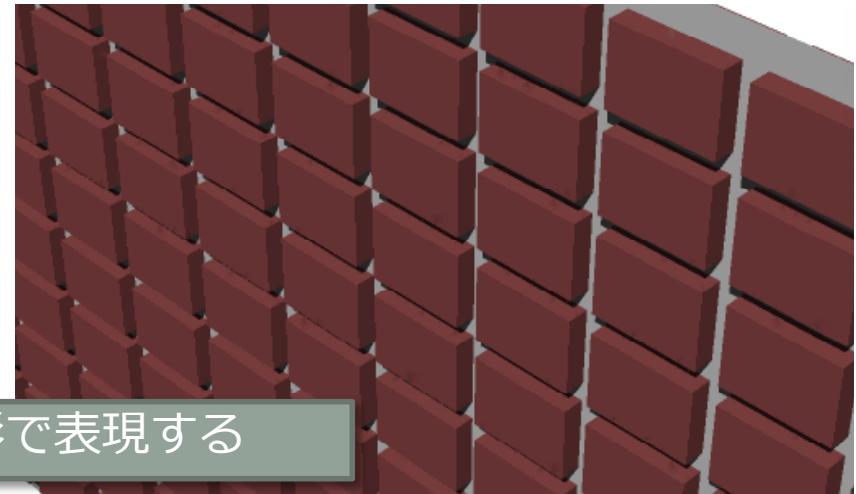
テクスチャマッピング (Texturing)



レンガの壁の表現



レンガを図形で表現する



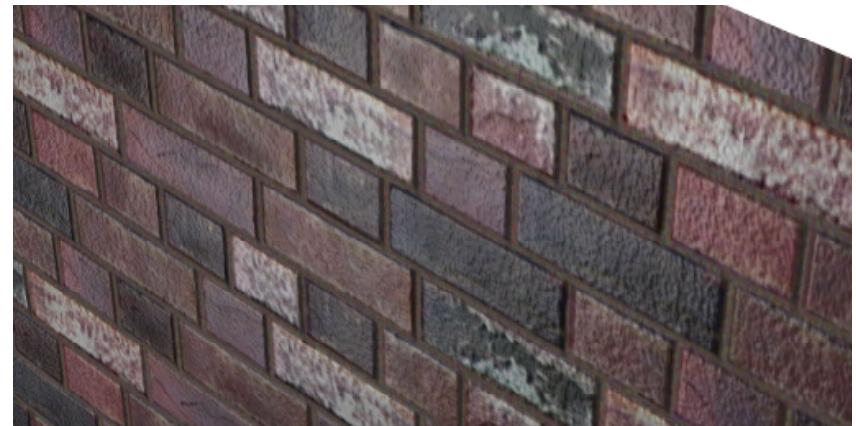
レンガをテクスチャで表現する



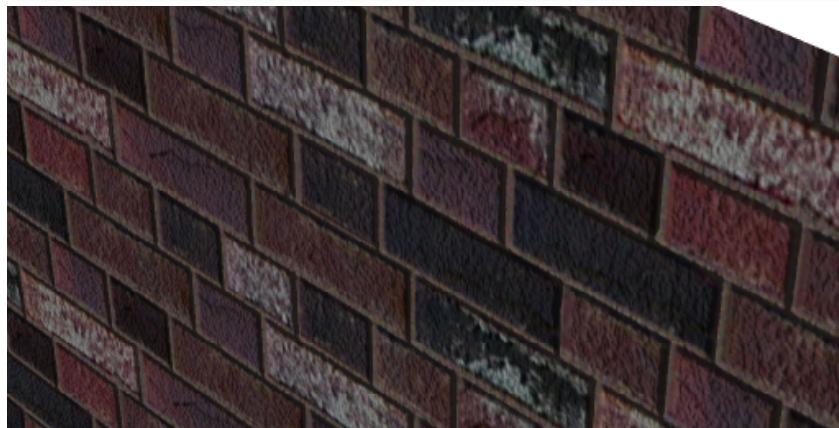
テクスチャを使ったレンガの壁



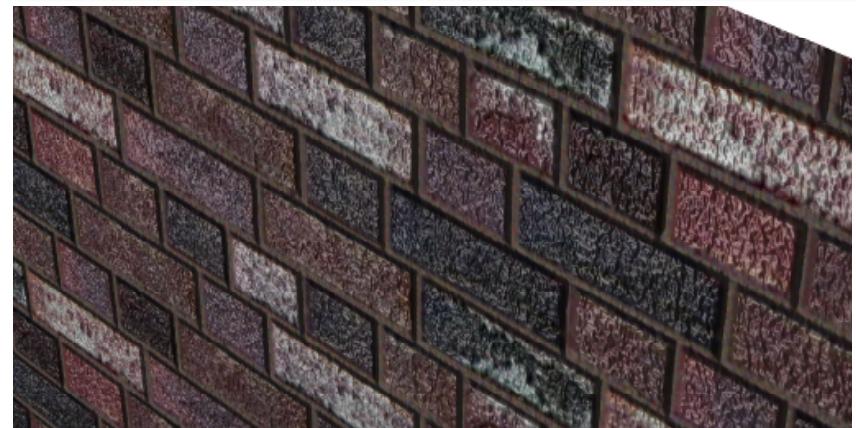
Without bump/highlight mapping



With highlight mapping



With bump mapping



With bump/highlight mapping

Parallax Occlusion Mapping with Silhouettes (Crysis)



テクスチャマッピングの手順

画像の読み込みと標本化方法の設定

テクスチャマッピングの手順

1. テクスチャの作成
 - i. テクスチャオブジェクトの作成と結合
 - ii. テクスチャメモリの確保と画像の読み込み
2. テクスチャ座標の生成
 - i. 頂点属性にテクスチャ座標を追加する
 - ii. テクスチャ座標をバーテックスシェーダの `in (attribute)` 変数に渡す
3. バーテックスシェーダの処理
 - i. `in (attribute)` 変数のテクスチャ座標値を必要に応じて座標変換する
 - ii. 変換後の座標値を `out (varying)` 変数に格納して次のステージに送る
4. フラグメントシェーダの処理
 - i. `in (varying)` 変数で受け取った座標値をもとにテクスチャを標本化する
 - ii. 標本化したテクスチャの画素値を使って陰影を計算する
 - iii. 求めた陰影をカラー値としてカラーバッファに格納する

テクスチャオブジェクトの準備

- N 個のテクスチャオブジェクトを作成する
 - GLuint `tex[N];`
 - `glGenTextures(N, tex);`
- i 番目のテクスチャオブジェクトを結合する
 - `glBindTexture(GL_TEXTURE_2D, tex[i]);`

二次元テクスチャの場合

テクスチャに用いる画像の準備

- 配列変数に画像を格納する
 - GLubyte **image[HEIGHT][WIDTH][CHANNEL];**
 - HEIGHT, WIDTH は画像の幅と高の画素数
 - 初期の OpenGL では 2 の整数乗 2^n にする必要があったが今はこの制限はない
 - CHANNEL は画像のチャンネル数
 - カラー (GL_RGB) なら3, アルファチャンネル付き (GL_RGBA) なら4
 - 配列変数 (**image**) の各要素に画素の色を格納する
 - **image[y][x][0] = <赤色のレベル (0~255)>;**
 - **image[y][x][1] = <緑色のレベル (0~255)>;**
 - **image[y][x][2] = <青色のレベル (0~255)>;**
 - **image[y][x][3] = <不透明度 (0~255)>;**

GL_RGBA のとき, 0: 透明, 255: 不透明

画像の読み込み

- ・テクスチャメモリを確保して、そこに画像データを格納する
 - ・`glTexImage2D(GL_TEXTURE_2D, 0, internalFormat, WIDTH, HEIGHT, 0, format, type, image);`
 - ・*internalFormat*: GPU 内でのデータの形式. 1チャンネル (モノクロ) なら `GL_RED`, 2チャンネルなら `GL_RG`, カラーなら `GL_RGB`, アルファチャンネル付きなら `GL_RGBA`. 他, 多数.
 - ・*format*: 画像データ (*image*) の形式, 1チャンネル (モノクロ) なら `GL_RED`, 2チャンネルなら `GL_RG`, カラーなら `GL_RGB` または `GL_BGR`, アルファチャンネル付きなら `GL_RGBA` または `GL_BGRA`.
 - ・*type*: 画像を格納している配列変数 (*image*) のデータ型, `GLubyte` (符号なし 8bit) なら `GL_UNSIGNED_BYTE`, `GLfloat` なら `GL_FLOAT`.
 - ・2次元テクスチャなら第1引数は `GL_TEXTURE_2D`, 第2引数は ~~ミツ~~ プマップのレベル (後述) で基本は `0`, 第6引数はテクスチャの境界線の太さの指定だったが今は使われないので常に `0`.

internalFormat, format, type

- internalFormat と format が一致している必要はない
 - 変換して読み込まれる
- format が **GL_BGR, GL_BGRA**
 - 画像のチャネルの R と B が入れ替わっているとき
 - OpenCV, ARToolKit 等でキャプチャした画像をテクスチャとして使うとき便利
- type が **GL_UNSIGNED_BYTE** のとき
 - 0~255 の整数値が 0~1 の実数値に変換されて読み込まれる
- type が **GL_FLOAT** のとき
 - internalFormat が **GL_RGB/GL_RGBA**
 - 配列 image の各要素の値は [0, 1] にクランプされて読み込まれる
 - internalFormat が **GL_RGB32F/GL_RGBA32F**
 - 配列 image の各要素の値はそのままシェーダで参照される
 - **GL_RGB16F/GL_RGBA16F** (16bits 浮動小数点もあり)
 - OpenEXR と組み合わせて HDRI を扱うときや GPGPU で便利

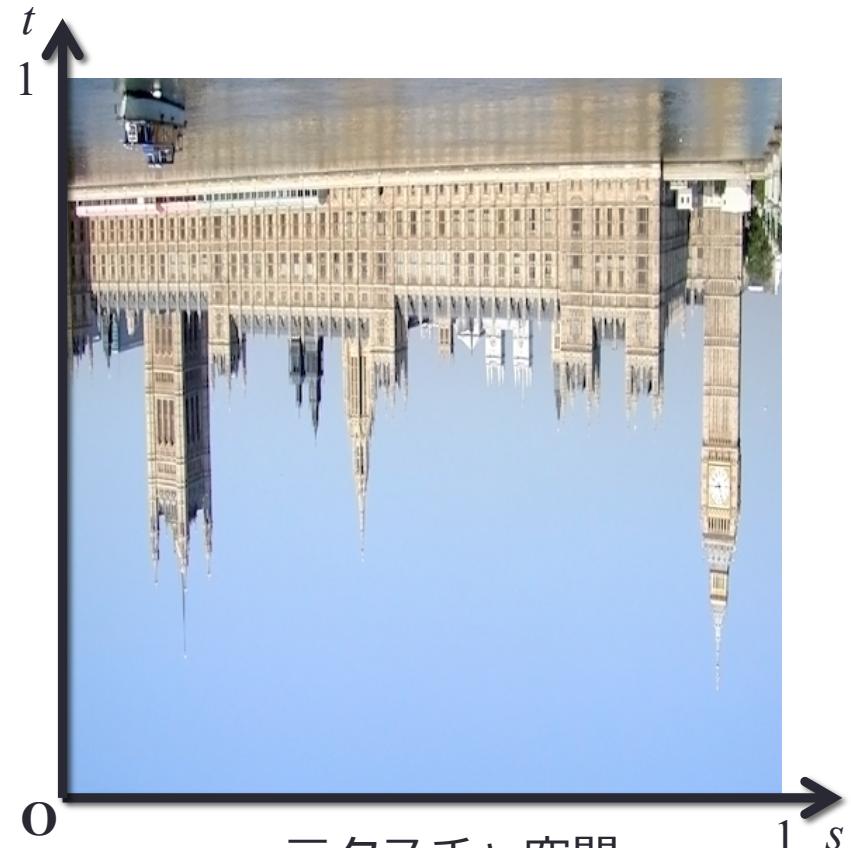
テクスチャ空間

画像の原点

$(0, 0)$



テクスチャ画像

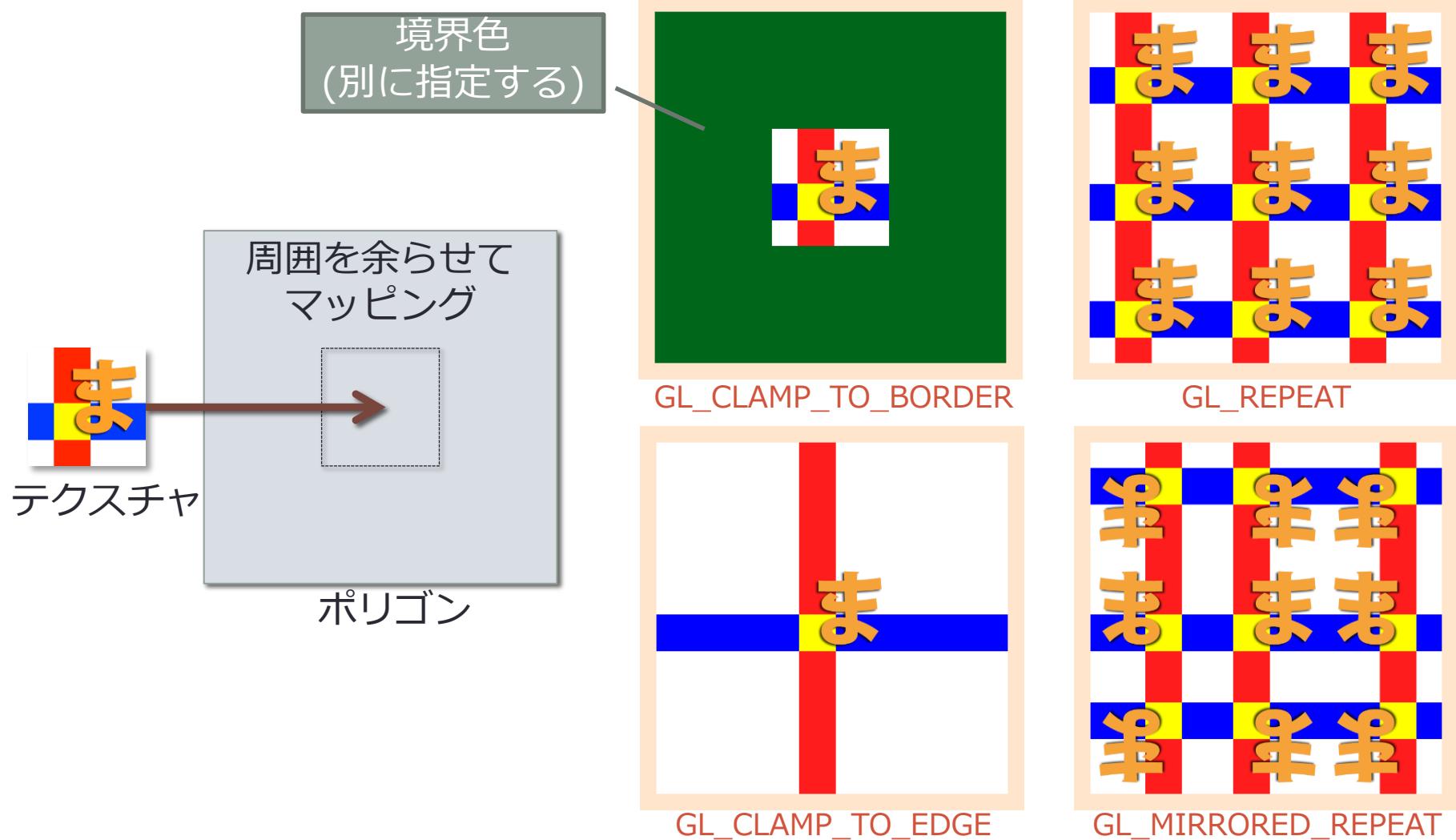


テクスチャ空間は読み込まれた画像は
サイズに関係なく $[0, 1]$ の範囲になる

テクスチャ範囲外の標本点の取り扱い

- テクスチャのラッピングモードの選択
 - `glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, mode);`
 - `glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, mode);`
 - *mode*: ラッピングモード, テクスチャ座標がテクスチャ空間の
`GL_TEXTURE_WRAP_S` あるいは `GL_TEXTURE_WRAP_T` 方向の範囲を
はみ出た時の処理
 - `GL_CLAMP_TO_BORDER` – 境界色を延長する
 - `GL_CLAMP_TO_EDGE` – テクスチャの最外周の画素の色を延長する
 - `GL_MIRRORED_REPEAT` – 同じテクスチャを反転しながら繰り返す
 - `GL_REPEAT` – 同じテクスチャを繰り返す

ラッピングモード



テクスチャの標本化方法の設定

- テクスチャの補間方法の選択
 - `glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, filter);`
 - *filter*: テクスチャが拡大される時に用いるフィルタの指定
 - `GL_NEAREST` – 最近傍法, `GL_LINEAR` – 双線形補間
 - `glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, filter);`
 - *filter*: テクスチャが縮小される時に用いるフィルタの指定
 - `GL_NEAREST` – 最近傍補間, `GL_LINEAR` – 双線形補間
 - `GL_NEAREST_MIPMAP_NEAREST` – 最近傍法の結果に最近傍法を適用
 - `GL_LINEAR_MIPMAP_NEAREST` – 双線形補間の結果に最近傍補間を適用
 - `GL_NEAREST_MIPMAP_LINEAR` – 最近傍補法の結果に線形補間を適用
 - `GL_LINEAR_MIPMAP_LINEAR` – 双線形補間の結果に線形補間を適用

ミップマップ
使用時

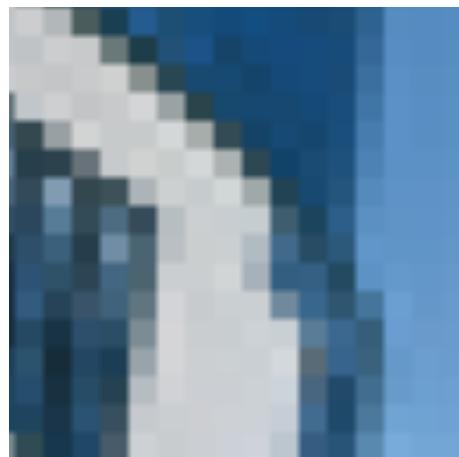
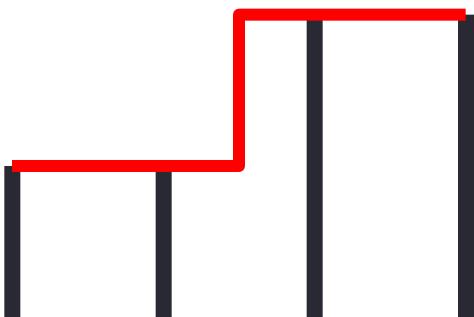
テクスチャの拡大



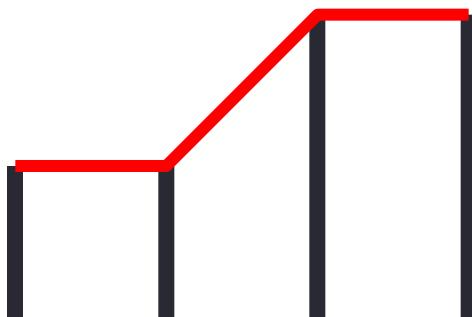
最近傍法
(nearest neighbor)

双線形補間
(bilinear)

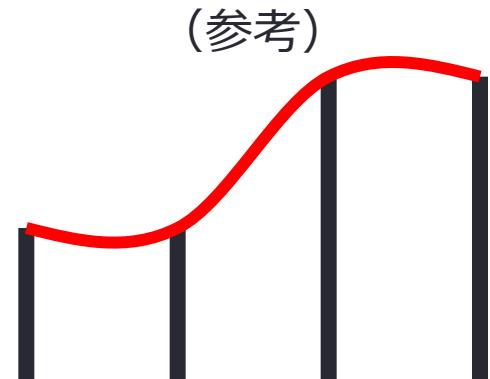
拡大フィルタ



最近傍法
(nearest neighbor)

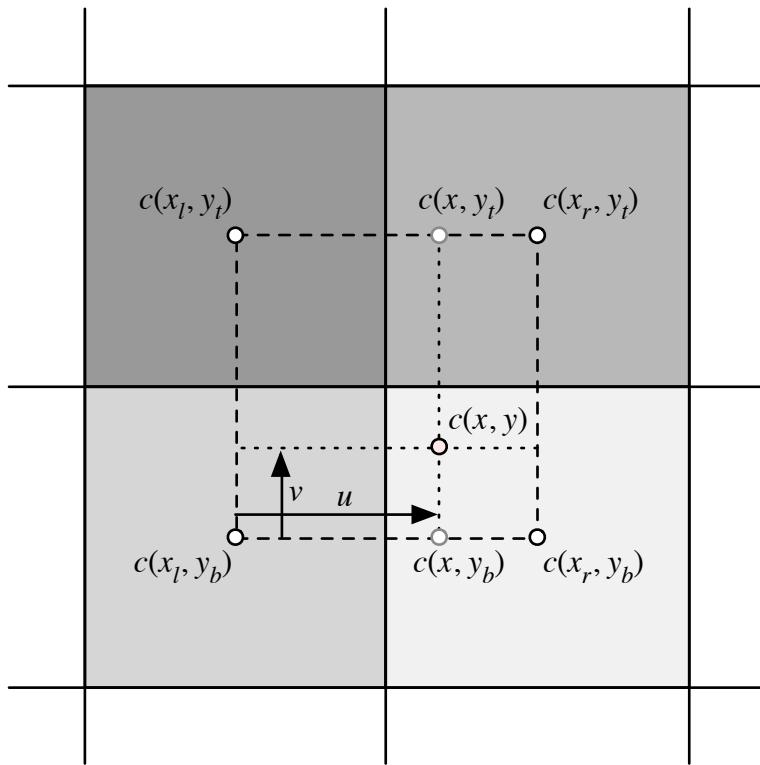


双線形補間
(bilinear)



双三次補間
(bicubic)

テクスチャの双線形（バイリニア）補間



$$\begin{aligned}
 (x_l, y_b) &= (\lfloor x \rfloor, \lfloor y \rfloor) \\
 (u, v) &= (x, y) - (x_l, y_b) \\
 c(x, y_b) &= c(x_l, y_b)(1-u) + c(x_r, y_b)u \\
 c(x, y_t) &= c(x_l, y_t)(1-u) + c(x_r, y_t)u \\
 c(x, y) &= c(x, y_b)(1-v) + c(x, y_t)v \\
 &= (1-u)(1-v)c(x_l, y_b) \\
 &\quad + u(1-v)c(x_r, y_b) \\
 &\quad + (1-u)v c(x_l, y_t) \\
 &\quad + uv c(x_r, y_t)
 \end{aligned}$$

テクスチャの縮小



最近傍法
(nearest neighbor)

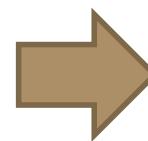


双線形補間
(bilinear)

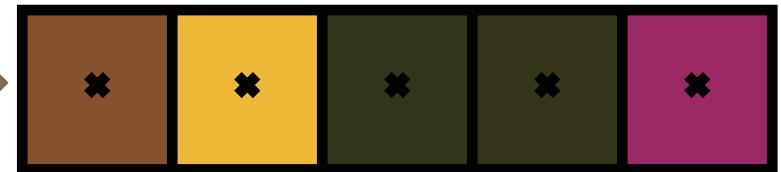


エイリアシングの発生

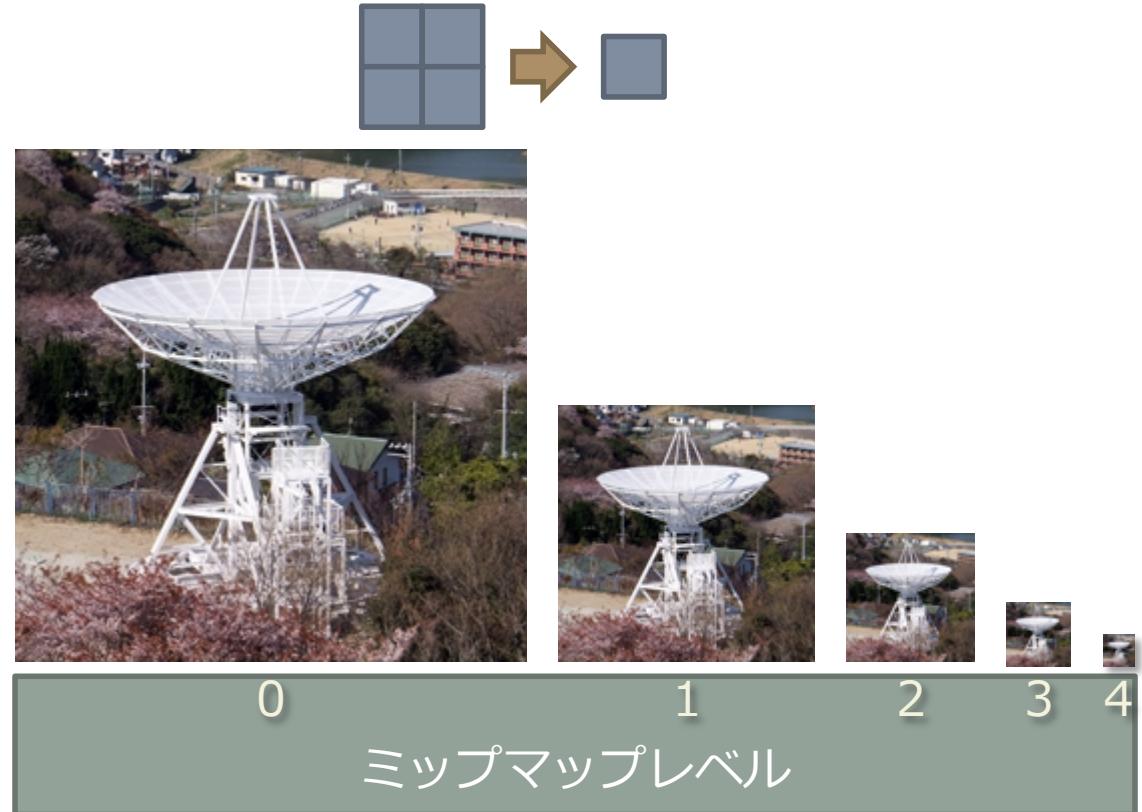
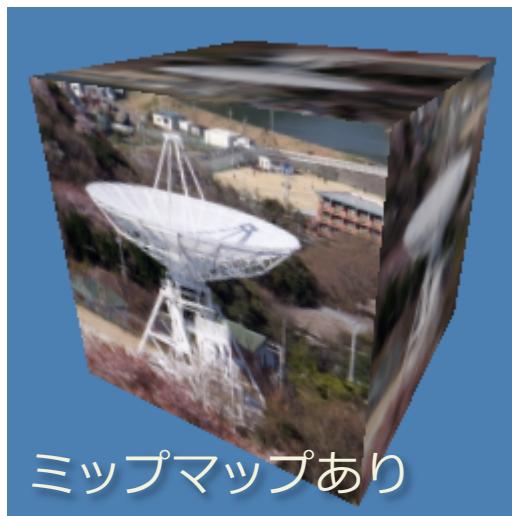
- 縮小では複数の画素が1画素にまとめられる
 - 画素の中の1箇所で標本化するとエイリアシングが発生する



標本化した画素の色がその領域の代表色（平均色）とはならない



ミップマップ



複数の解像度のテクスチャを用意しておいて画面上に現れるテクスチャの縮小率に応じて切り替えて使う

ミップマップのテクスチャの読み込み

```
glTexImage2D(GL_TEXTURE_2D, 0, internalFormat,
    WIDTH,      HEIGHT,      0, format, type, image0);
```

:

:

```
glTexImage2D(GL_TEXTURE_2D, 1, internalFormat,
    WIDTH / 2, HEIGHT / 2, 0, format, type, image1);
```

:

:

```
glTexImage2D(GL_TEXTURE_2D, 2, internalFormat,
    WIDTH / 4, HEIGHT / 4, 0, format, type, image2);
```

:

:

(以下、繰り返し - 繰り返しの最大回数は解像度が1になるまで)

またはレベル 0 のテクスチャ (image0) を読み込んだあと
image1, image2 に nullptr を指定しておいて

```
glGenerateMipmap(GL_TEXTURE_2D);
```

image0: 本来の解像度 (レベル 0, ベース) のテクスチャ

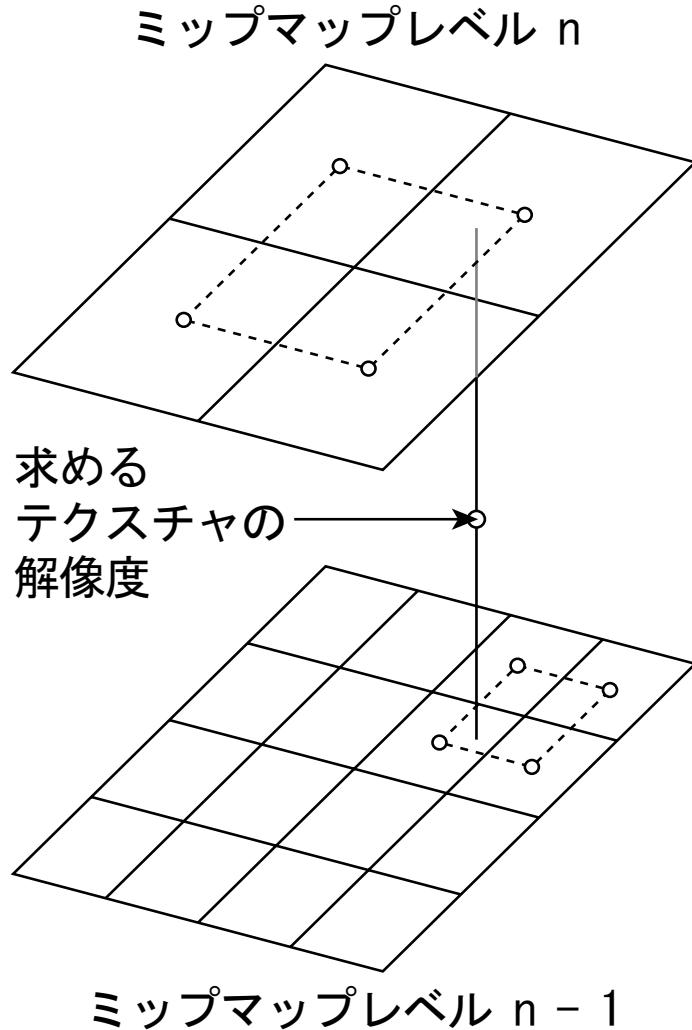
image1: image0 の解像度を縦横 1 / 2 に縮小したもの

image2: image0 の解像度を縦横 1 / 4 に縮小したもの



一発!

ミップマップの標本化



- 目標の解像度に一致するミップマップレベルがあるわけではない
 - 前後のレベルで標本化
 - 標本化には**最近傍法**と**双線形補間**を選択できる
 - それぞれレベルの標本化結果をさらに補間する
 - これにも**最近傍法**と**双線形補間**を選択できる
 - レベルの切り替わり場所の境界を目立たなくする

テクスチャの使用

- 使用するテクスチャユニットを指定する

- `glActiveTexture(GL_TEXTURE0);`

テクスチャユニット0

- 使用可能なテクスチャユニットの数は次の手順で units に取得できる

```
GLint units;  
glGetIntegerv(GL_ACTIVE_TEXTURE, &units);
```

- または

```
GLint units;  
glGetIntegerv(GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS, &units);
```

- 作成したテクスチャオブジェクトを結合する

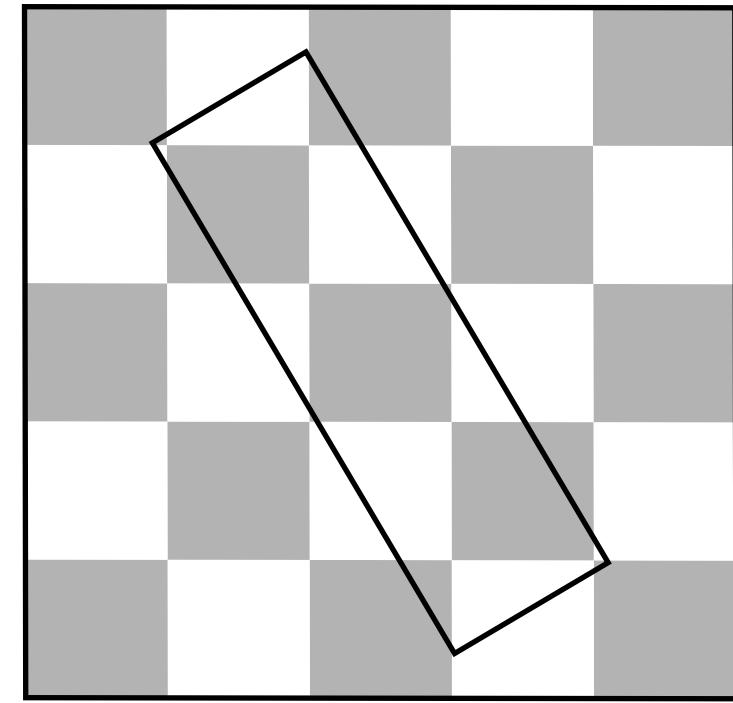
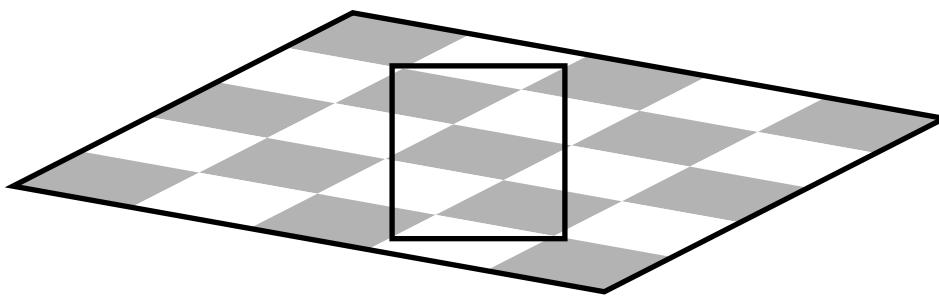
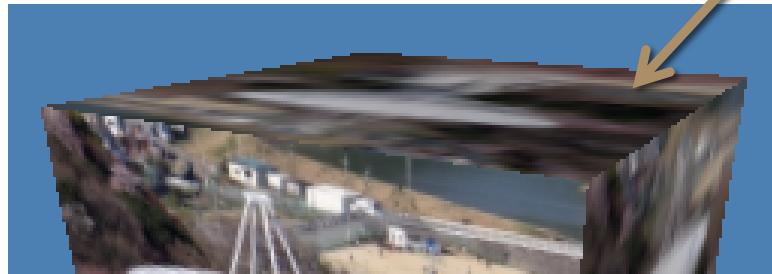
- `glBindTexture(GL_TEXTURE_2D, tex);`

テクスチャの標本化の改良

OpenGL の標準機能には実装されていない手法

視線の入射角が浅い場合

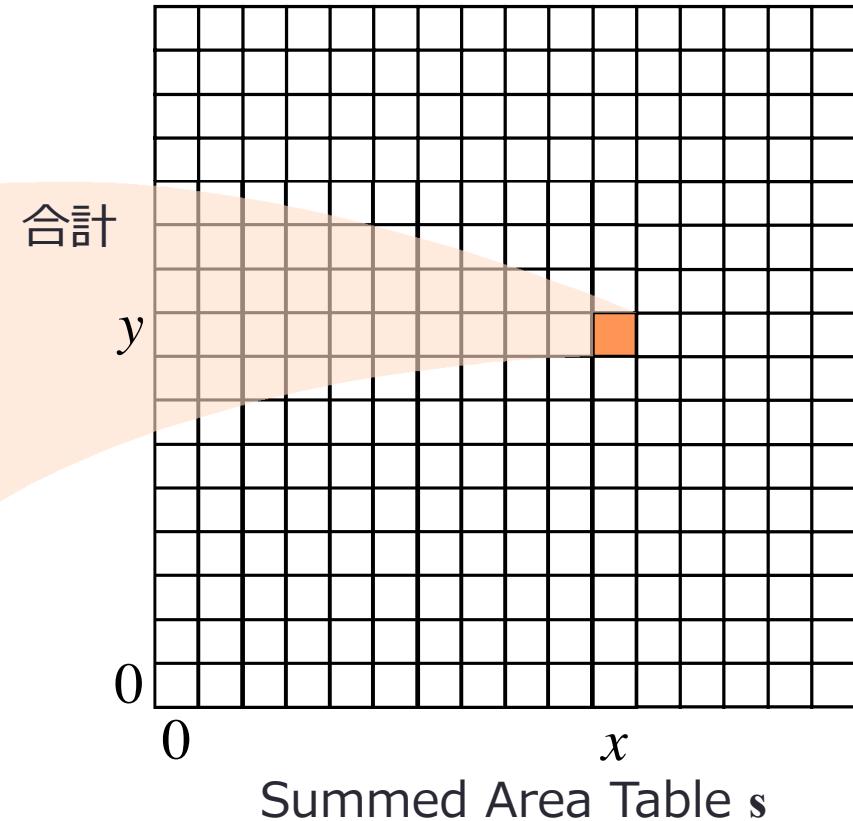
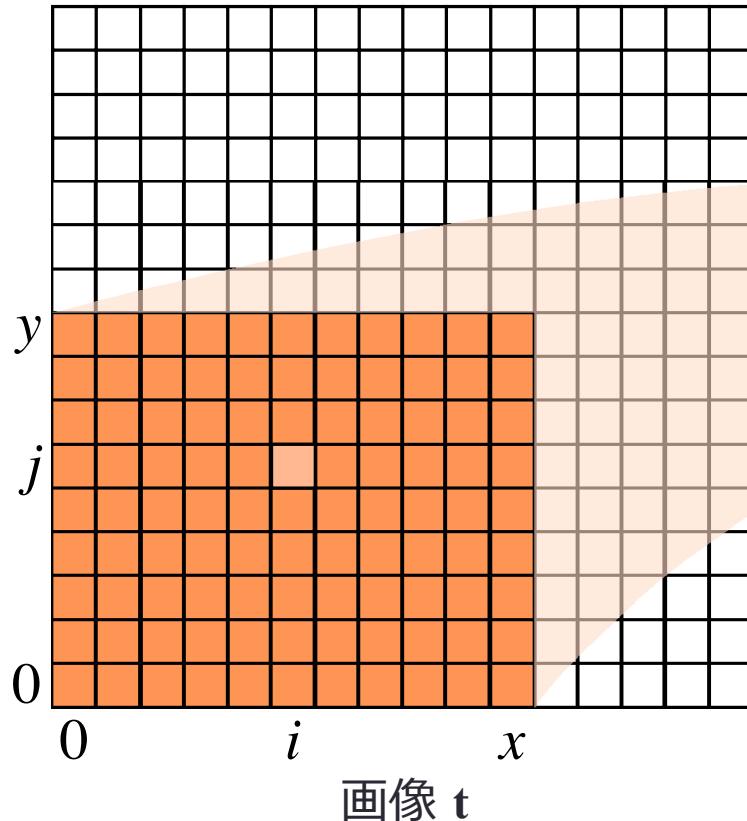
- 1画素に収まる縦方向と横方向の画素数が異なる
 - エイリアシングを防ぐために多い方に合わせてテクスチャを選択すると、マッピングされるテクスチャがぼけ過ぎる



非対称ミップマップ (Ripmap)

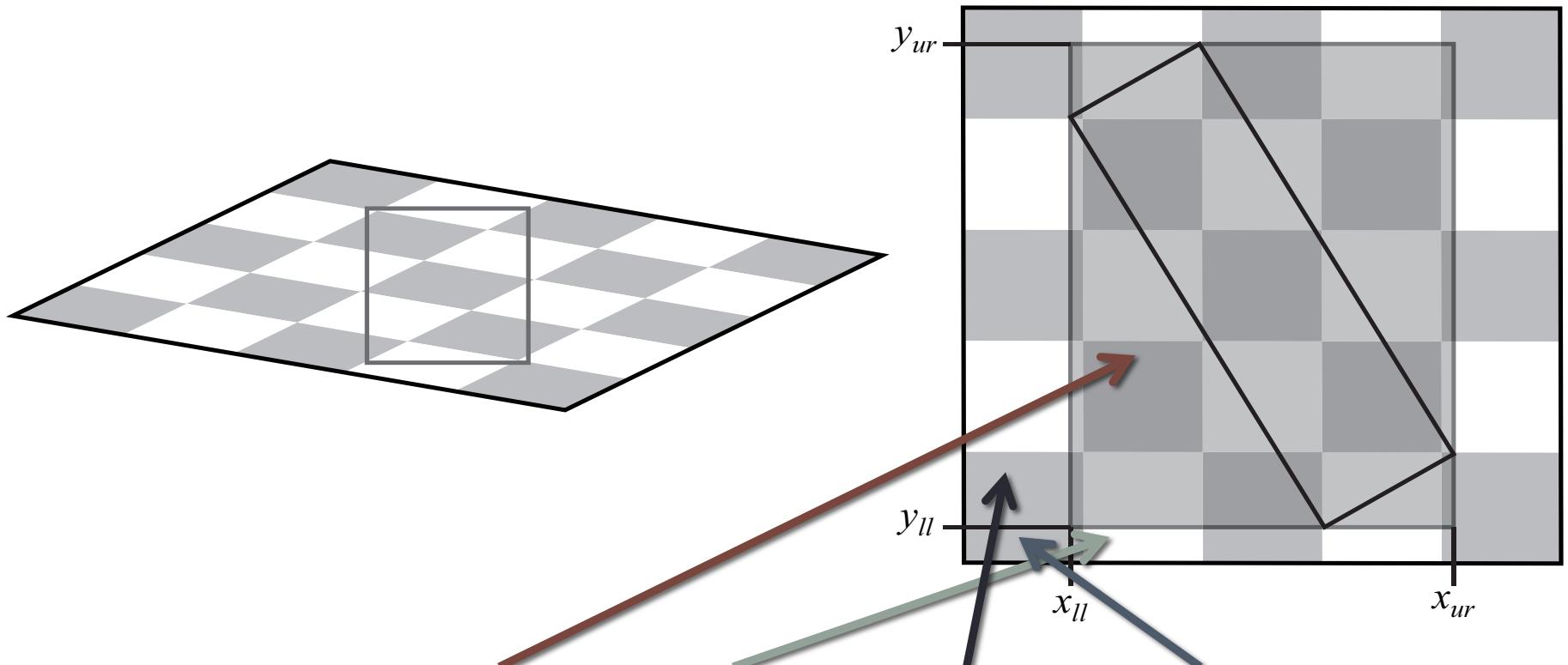


Summed-Area Table (SAT)



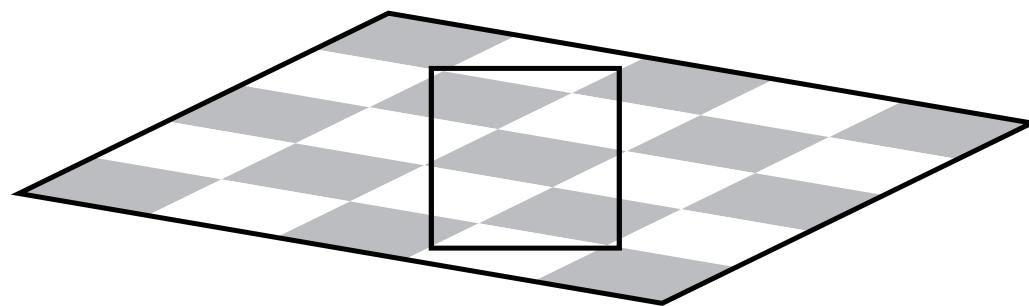
$$s[x, y] = \sum_{j=0}^y \sum_{i=0}^x t[i, j]$$

SATを使った領域の標本化



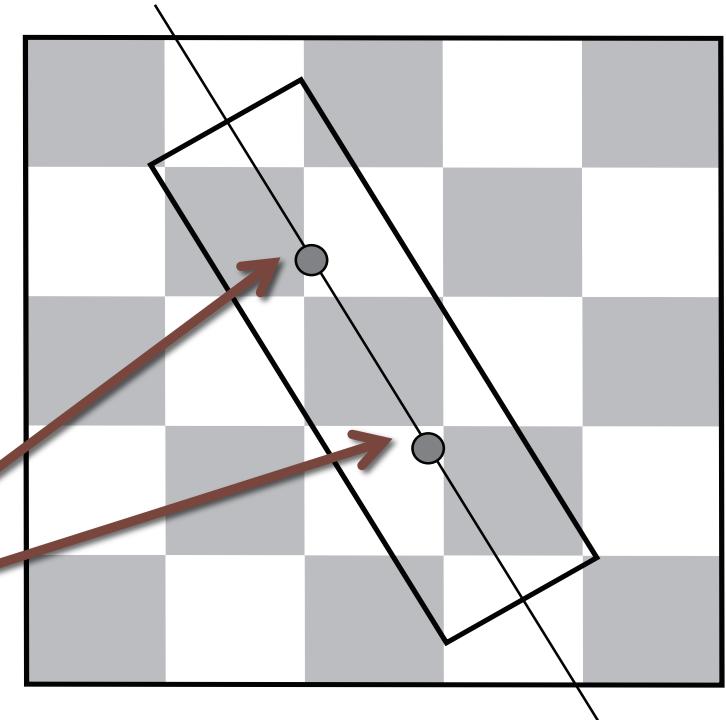
$$\mathbf{c} = \frac{\mathbf{s}[x_{ur}, y_{ur}] - \mathbf{s}[x_{ur}, y_{ll}] - \mathbf{s}[x_{ll}, y_{ur}] + \mathbf{s}[x_{ll}, y_{ll}]}{(x_{ur} - x_{ll})(y_{ur} - y_{ll})}$$

Unconstrained Anisotropic Filtering



ミップマップの標本

非対称の軸



矩形テクスチャ

- GL_TEXTURE_RECTANGLE
 - GL_TEXTURE_2D の代わりに使用する
 - テクスチャ空間の大きさは [0, 1] ではなく画像のサイズに一致する
 - テクスチャ座標が画素位置と一致するので画像処理などに便利
 - ラッピングモードに GL_REPEAT, GL_MIRRORED_REPEAT が使えない
 - GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER だけ
 - テクスチャの補間方法としてミップマップが使えない
 - GL_TEXTURE_MIN_FILTER は GL_NEAREST, GL_LINEAR だけ
 - 今はあまり使う意味はない
 - GL_TEXTURE_2D が 2^n 以外の画素数に対応する前に代わりに使われた
 - https://www.opengl.org/wiki/Rectangle_Texture の Purpose 参照
 - これを使った画像処理のサンプルあります (OpenCV 必要)
 - <https://github.com/tokoik/filtertest>

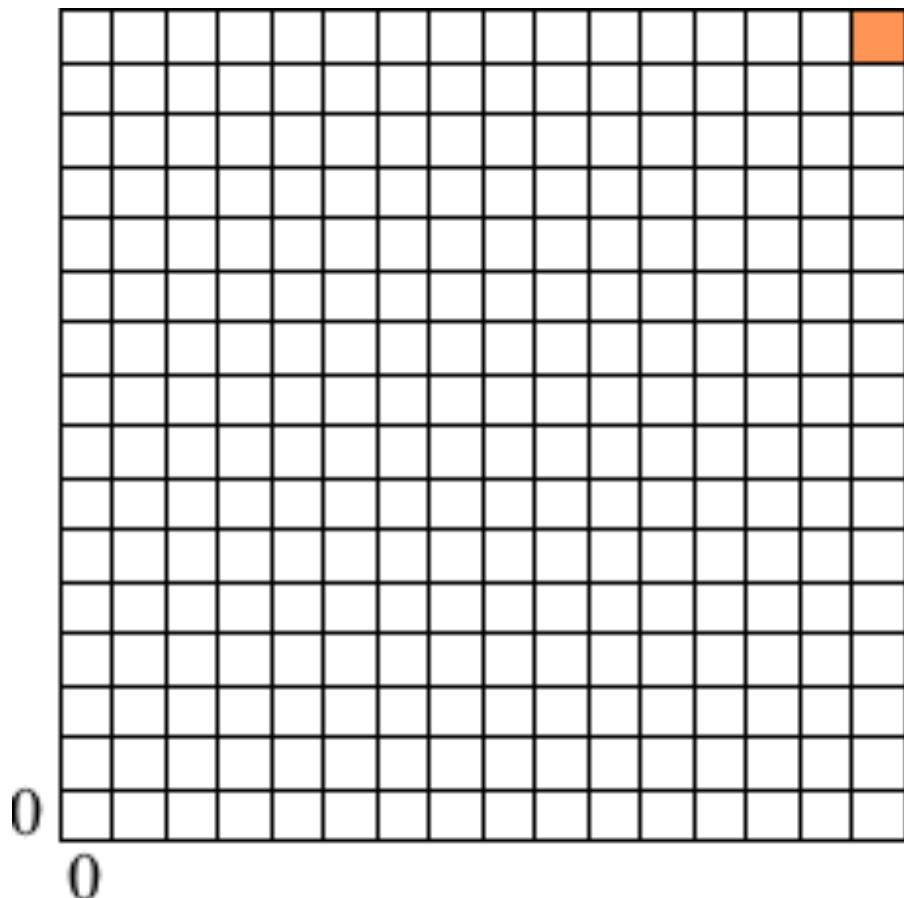
glTexStorage2D()

- ・ミップマップの複数レベルのテクスチャメモリを同時に確保
 - ・`glTexStorage2D(target, levels, internalFormat, width, height)`
 - ・`target`: `GL_TEXTURE_2D`, `GL_TEXTURE_RECTANGLE` など
 - ・`levels`: 確保するミップマップのレベル, `0` ならミップマップなし
 - ・`internalFormat`: GPU 内でのデータの形式. 1チャンネル (モノクロ) なら `GL_RED`, 2チャンネルなら `GL_RG`, カラーなら `GL_RGB` または `GL_BGR`, アルファチャンネル付きなら `GL_RGBA` または `GL_BGRA`.
 - ・`width, height`: テクスチャのサイズ
 - ・テクスチャの転送は行わない

```
for (i = 0; i < levels; i++) {
    glTexImage2D(target, i, internalformat, width, height, 0, format, type, NULL);
    width = max(1, (width / 2));
    height = max(1, (height / 2));
}
```
 - ・OpenGL 4.2 以降 (macOS では使えない)

課題

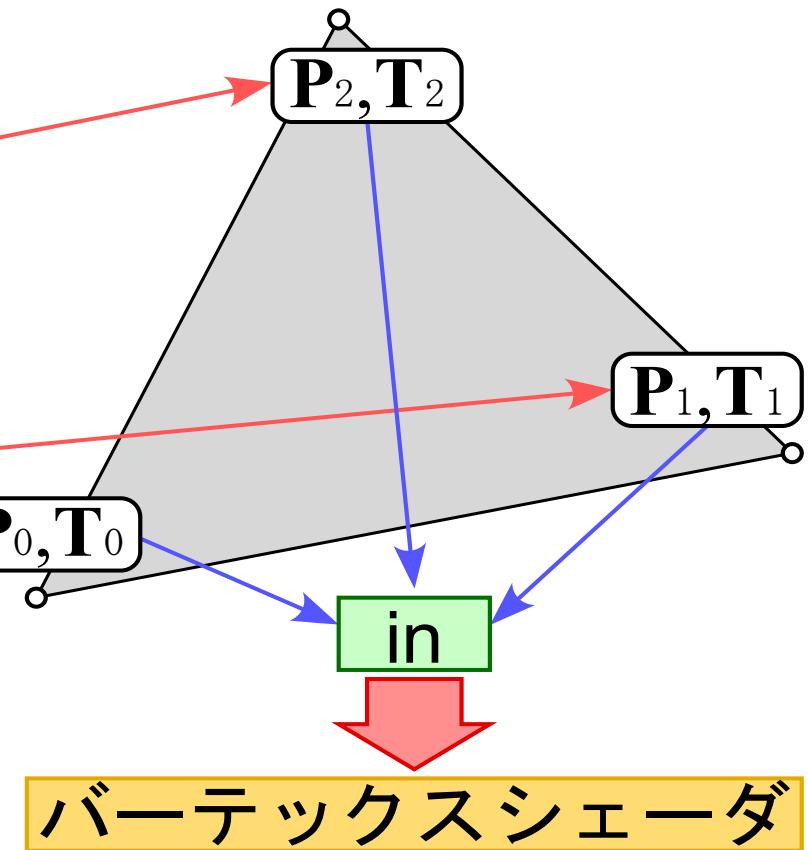
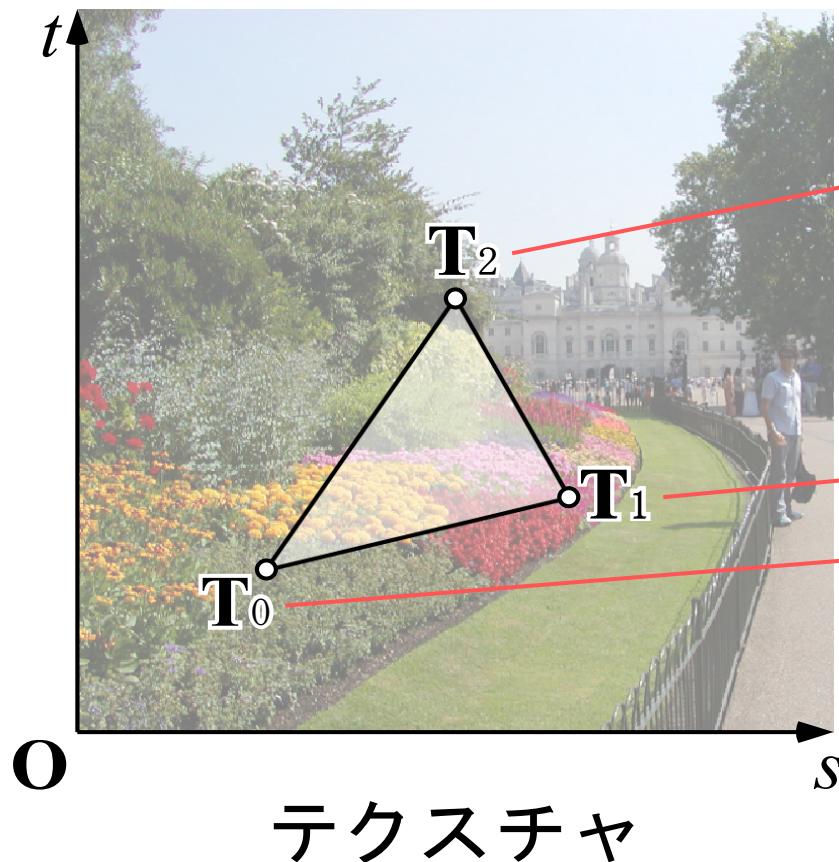
- Summed Area Table の右上隅の画素にはどのような値が入るか。



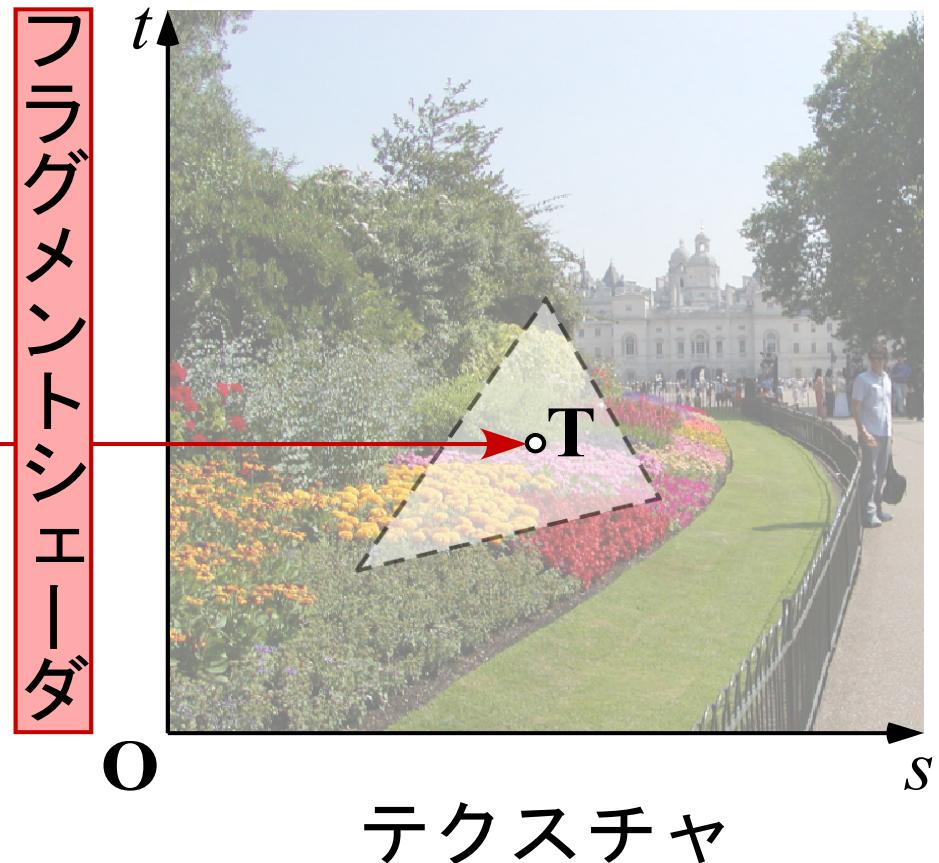
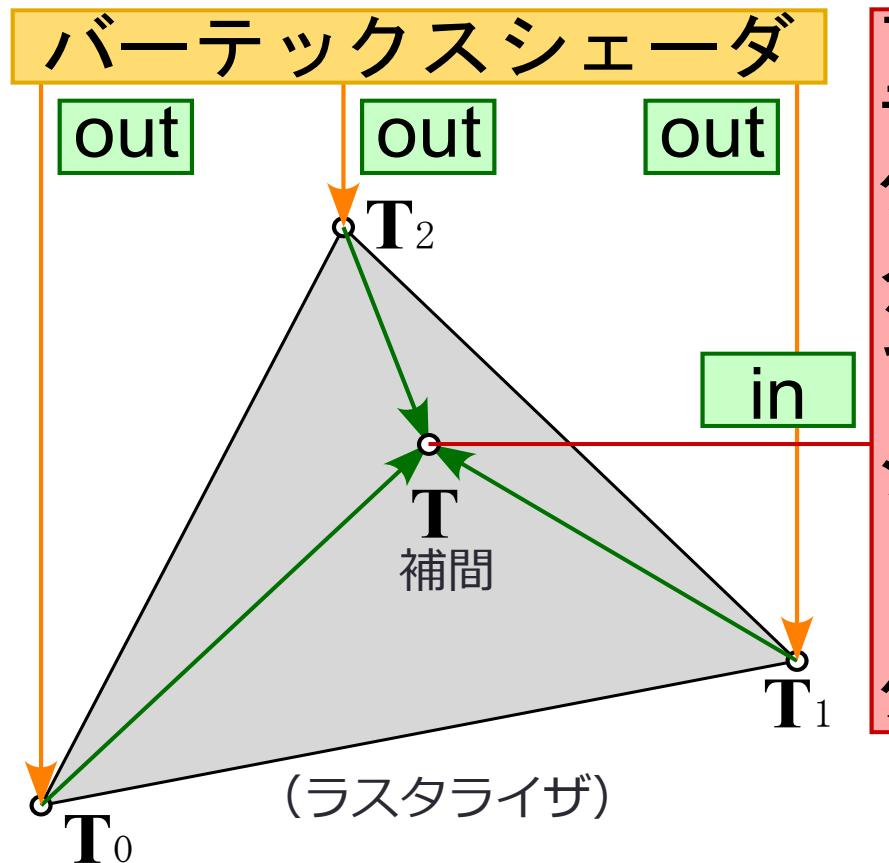
テクスチャ座標の設定

in (attribute) 変数にテクスチャ座標を指定する

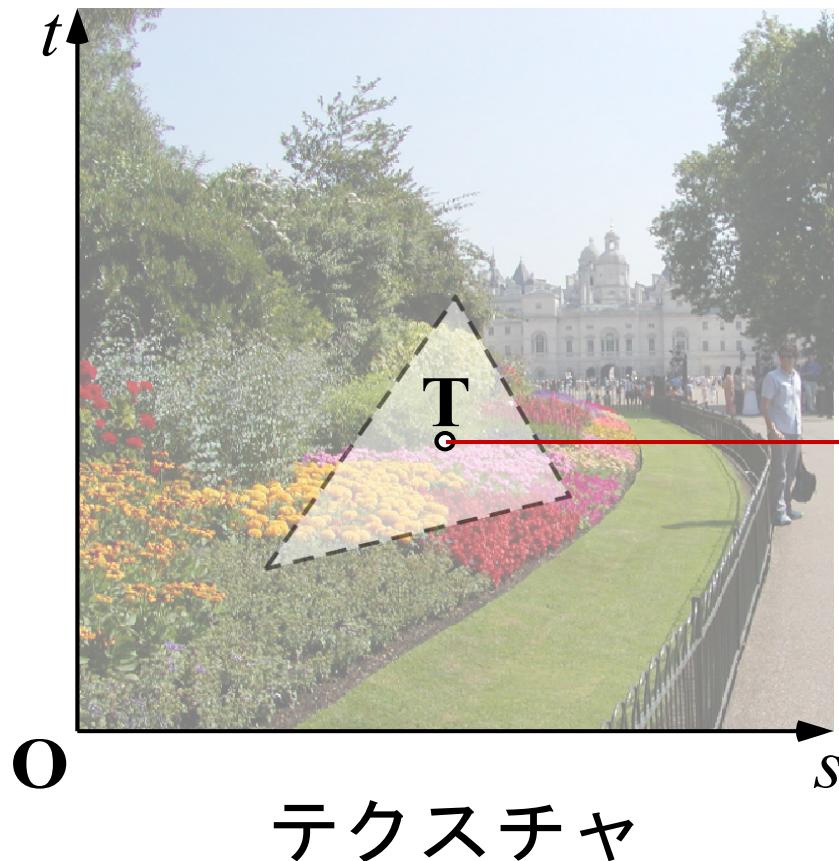
テクスチャ座標を in 変数に設定



テクスチャの参照

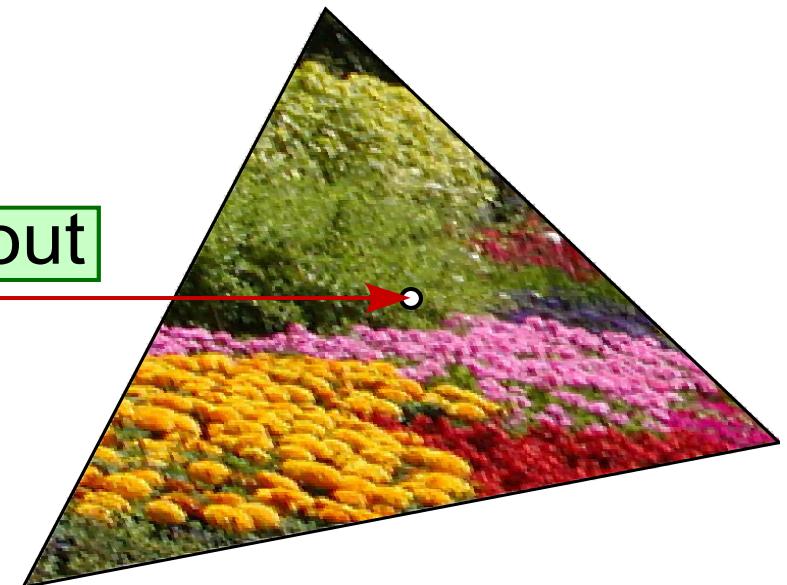


参照した画素値を使って陰影付け

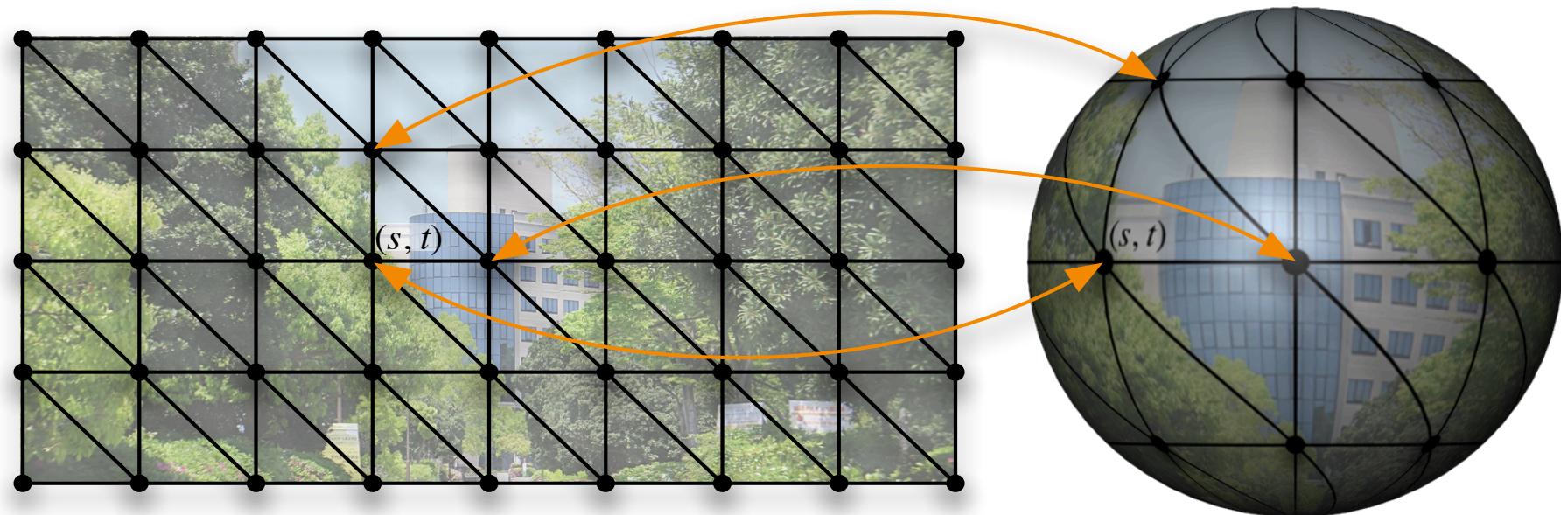


フレグメントシェーダ

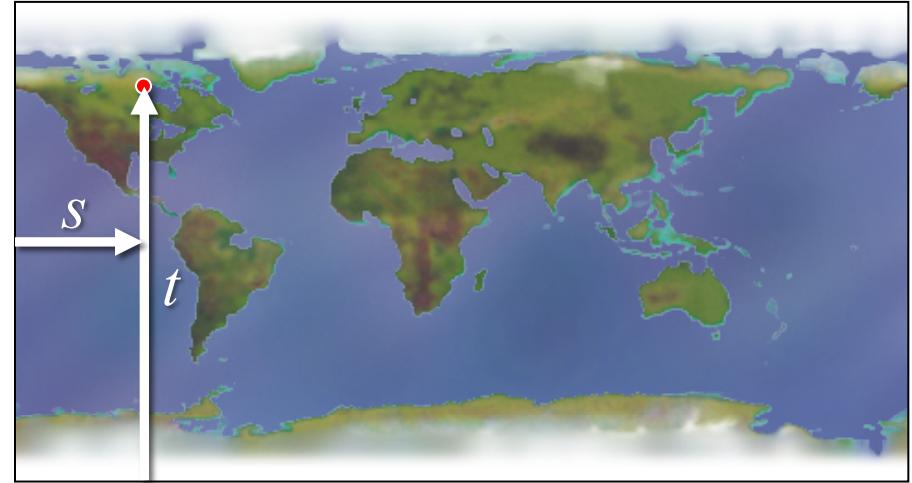
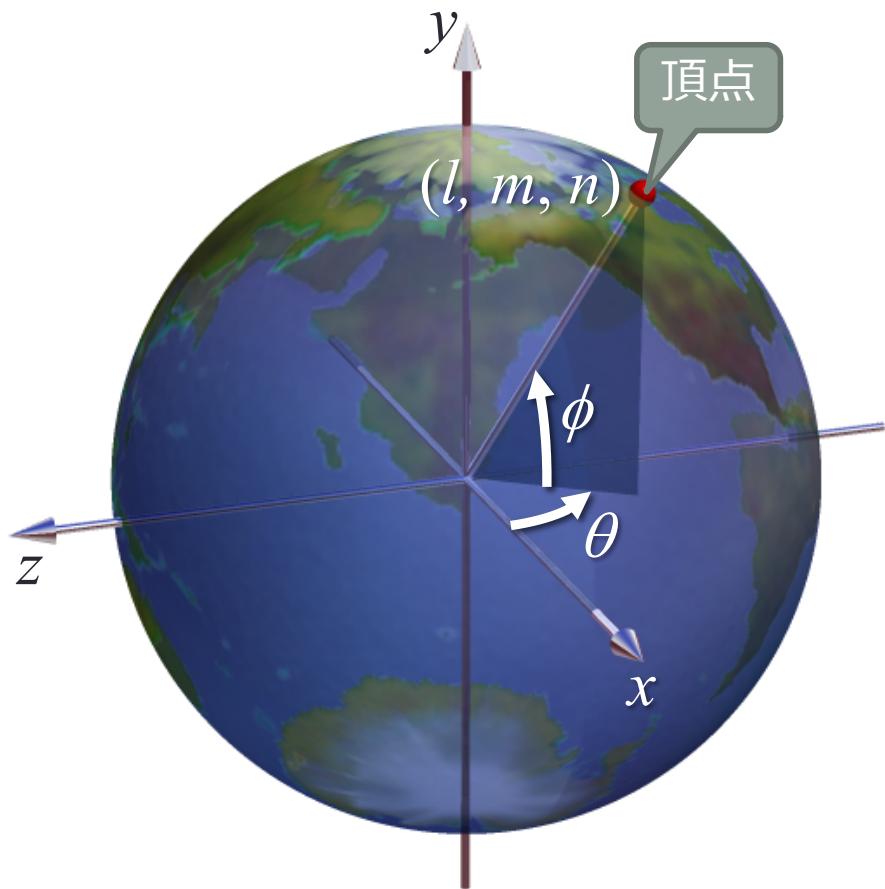
out



テクスチャ座標の割り当て



球面マッピング



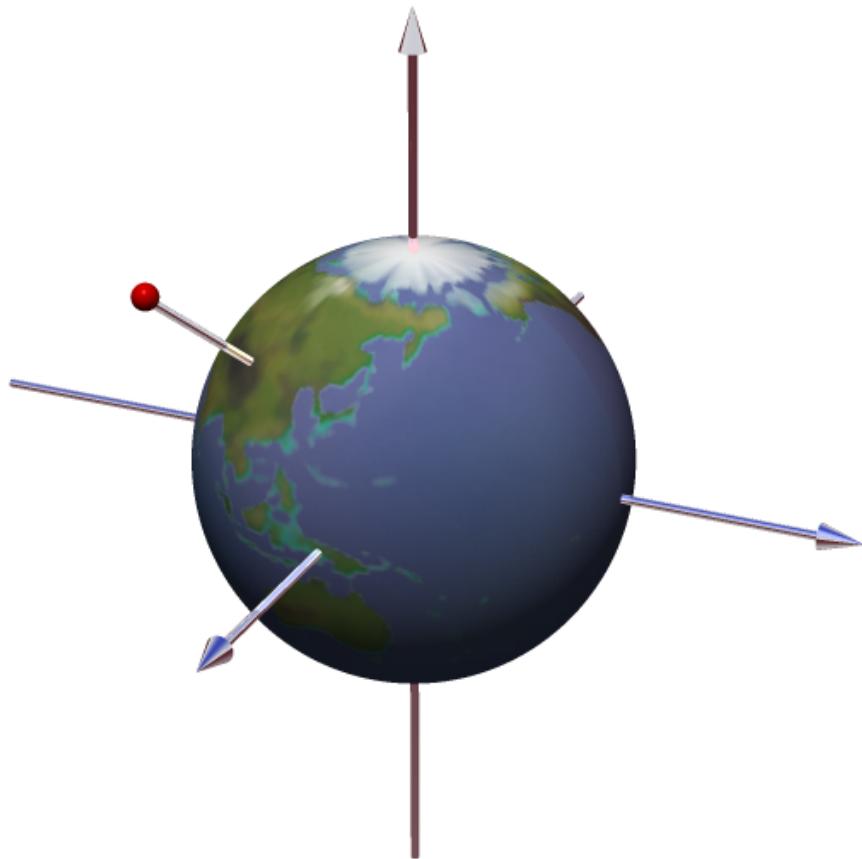
図形の表面上の頂点位置 (l, m, n)

$$s = \frac{\theta + \pi}{2\pi} = \frac{1}{2\pi} \tan^{-1}\left(\frac{n}{l}\right) + \frac{1}{2}$$

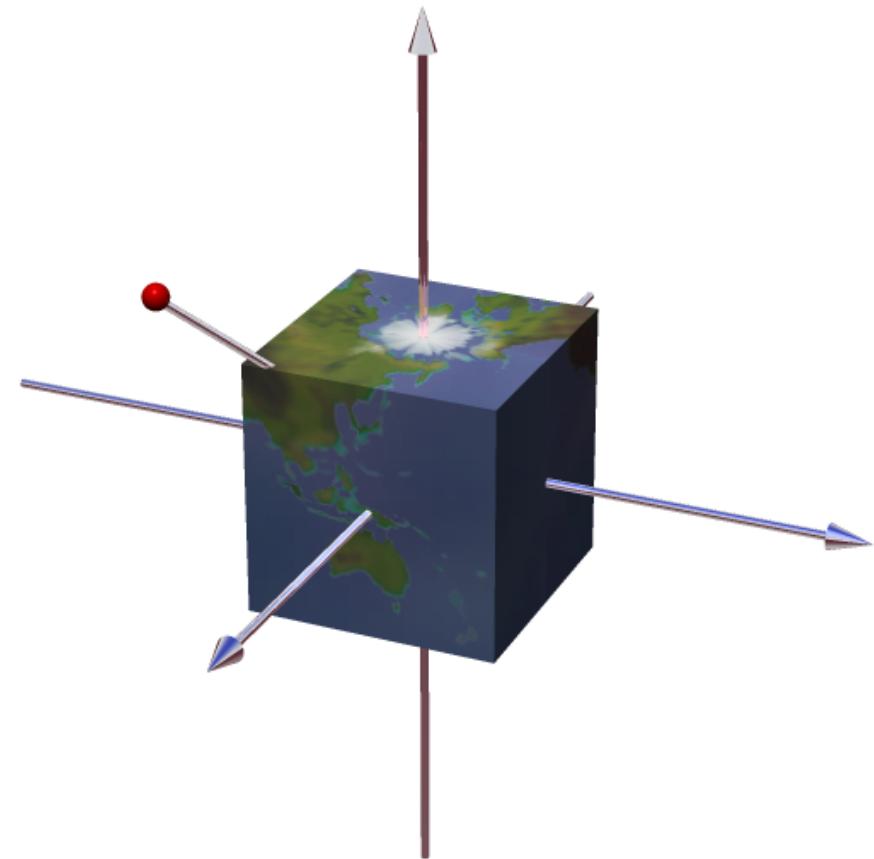
$$t = \frac{\phi + \pi/2}{\pi} = \tan^{-1}\left(\frac{m}{\sqrt{l^2 + n^2}}\right) + \frac{1}{2}$$

球面マッピングされた物体

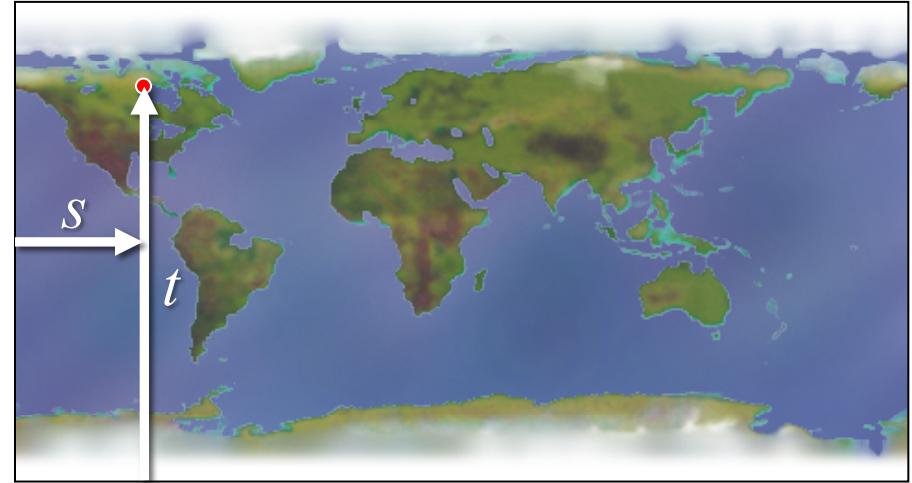
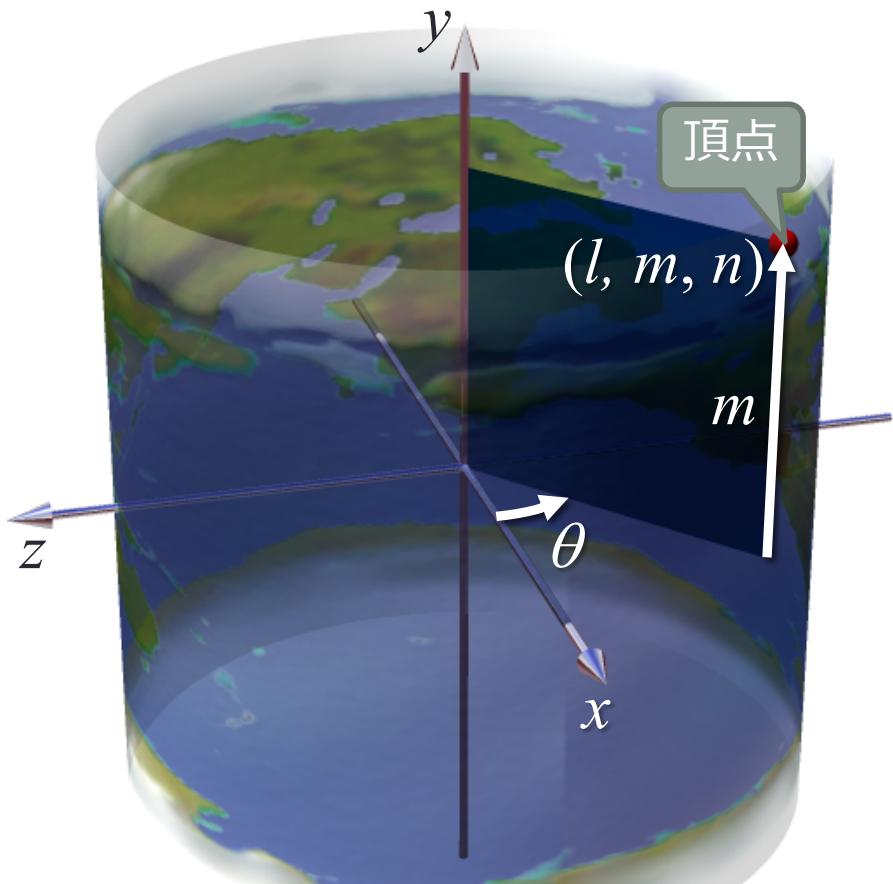
球に球面マッピング



立方体に球面マッピング



円柱マッピング



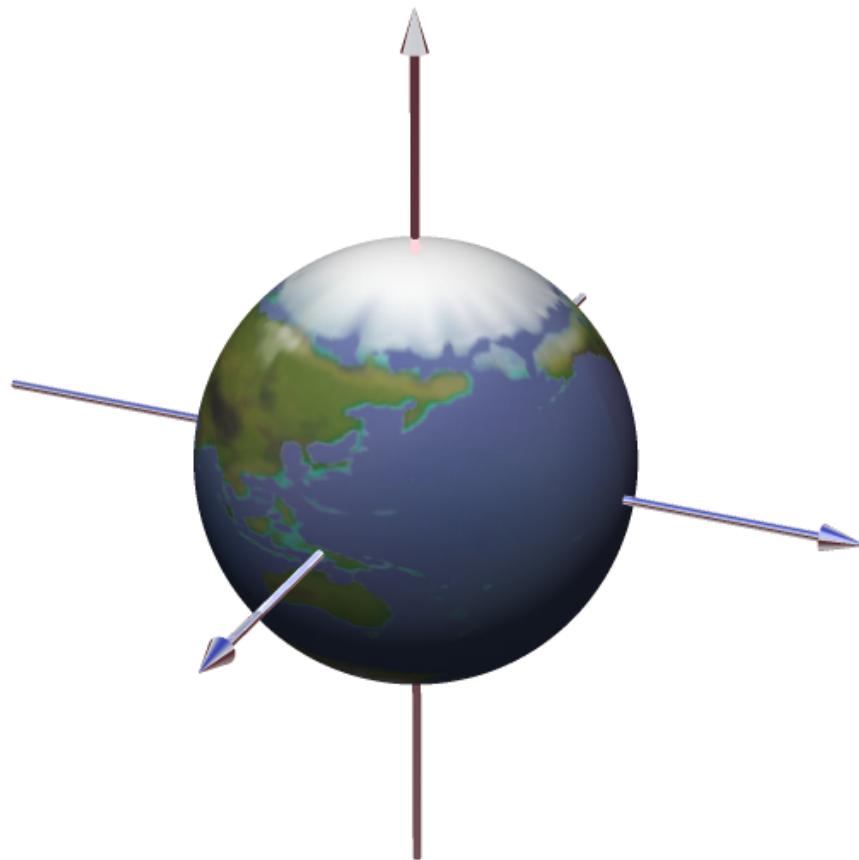
図形の表面上の頂点位置 (l, m, n)
図形の高さ h

$$s = \frac{\theta + \pi}{2\pi} = \frac{1}{2\pi} \tan^{-1}\left(\frac{n}{l}\right) + \frac{1}{2}$$

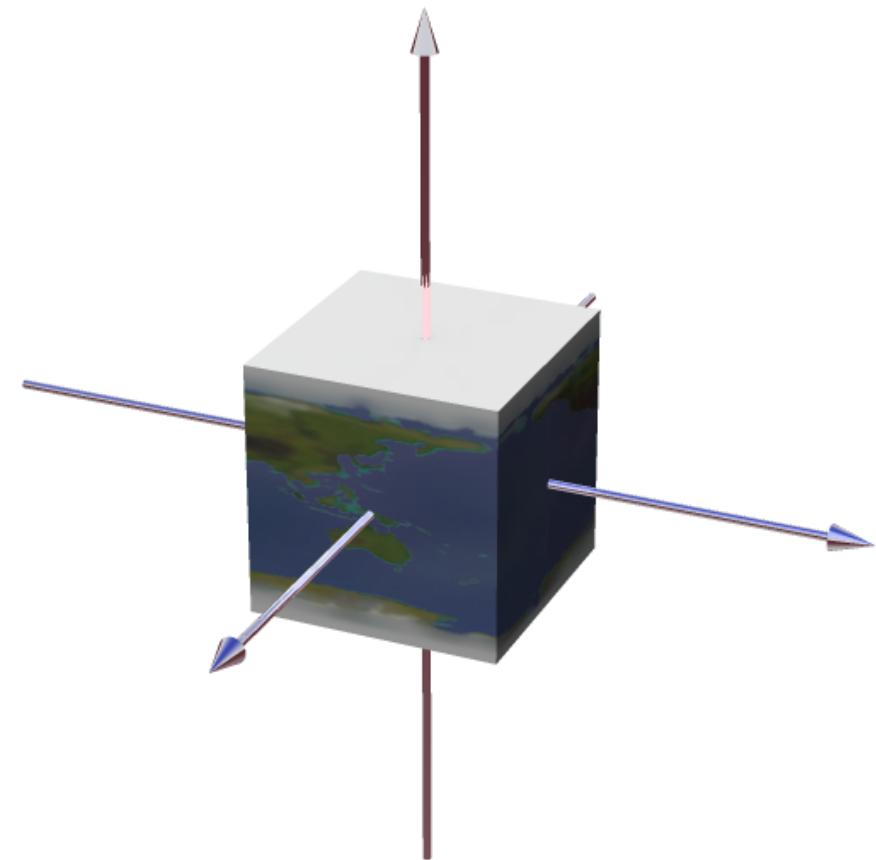
$$t = \frac{m}{h} + \frac{1}{2}$$

円柱マッピングされた物体

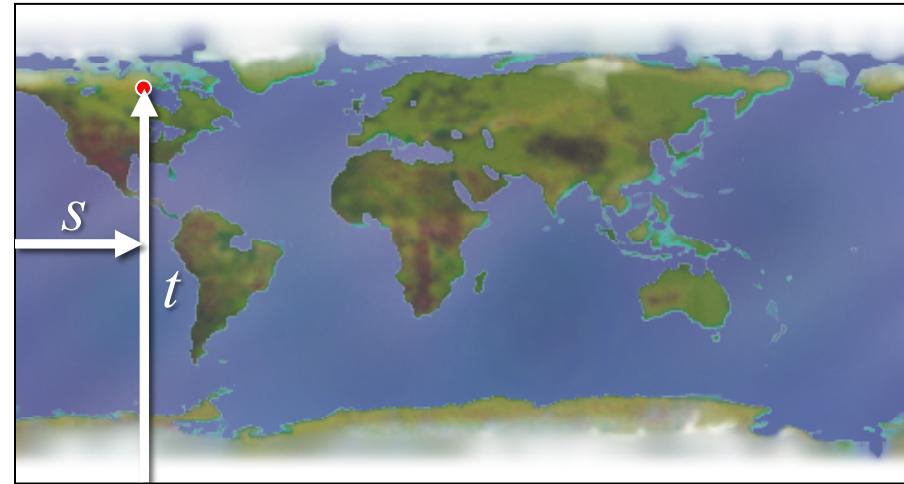
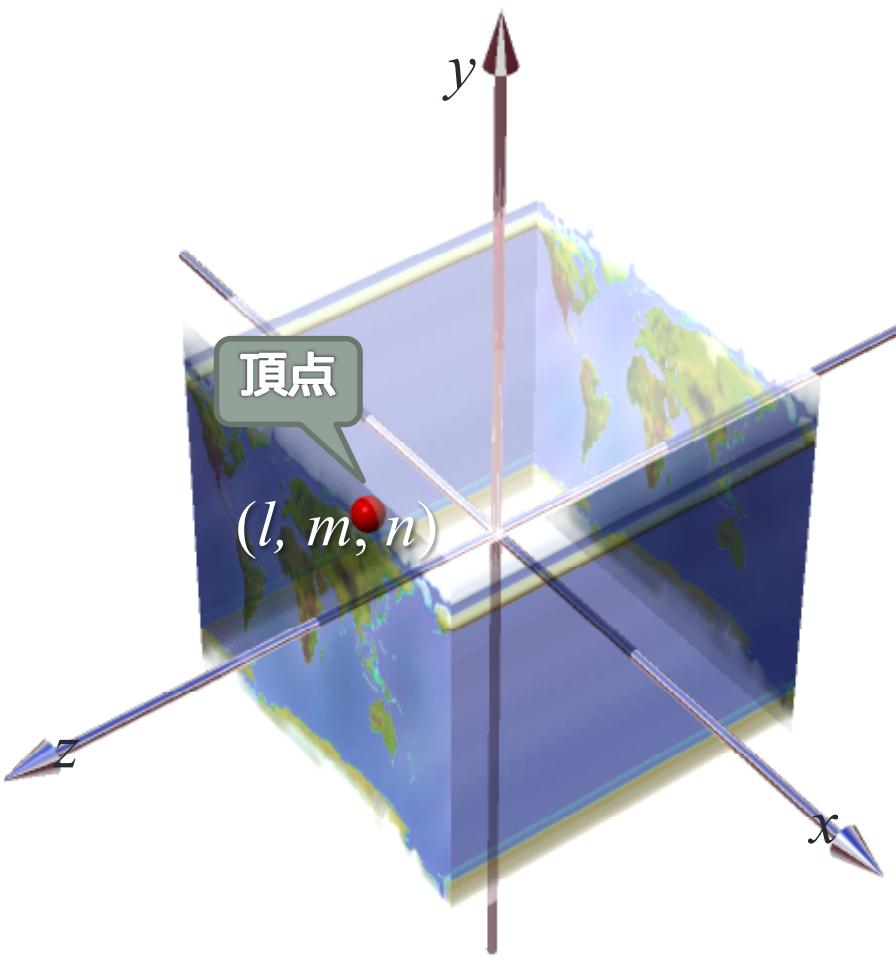
球に円柱マッピング



立方体に円柱マッピング



平行マッピング



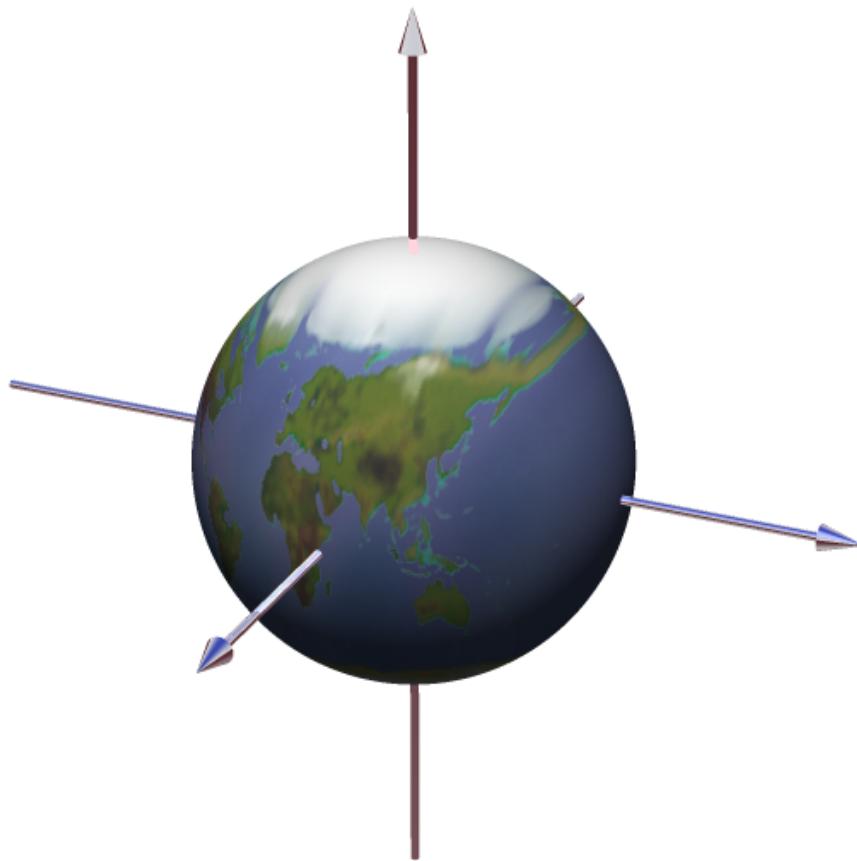
図形の表面上の頂点位置 (l, m, n)
図形の幅 w , 高さ h

$$s = \frac{l}{w} + \frac{1}{2}$$

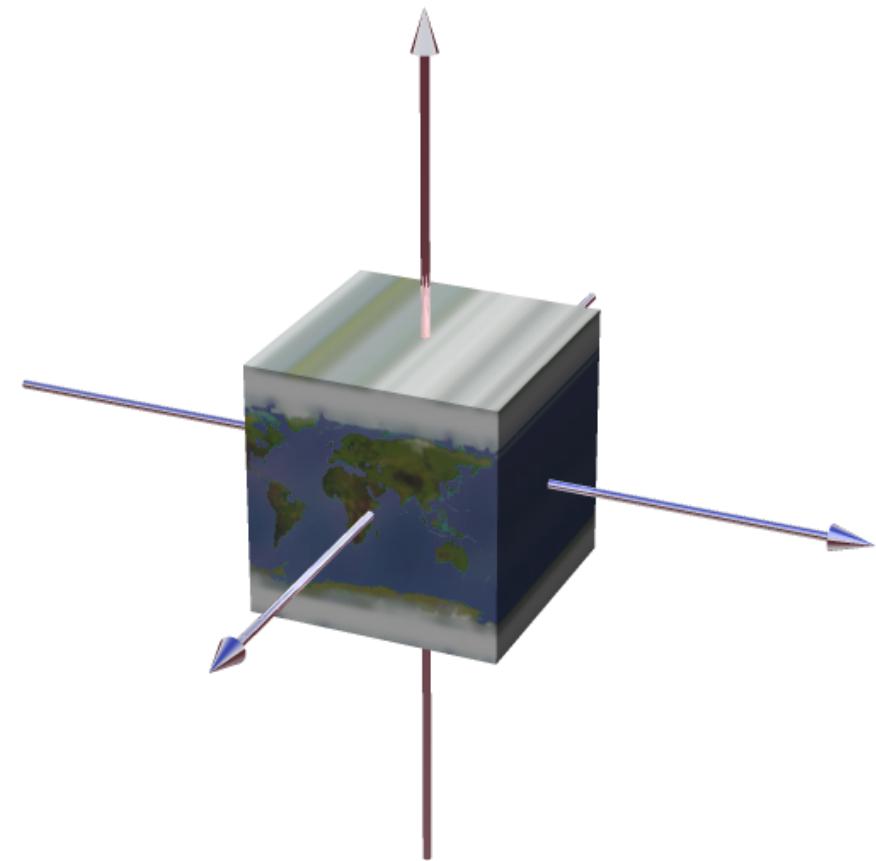
$$t = \frac{m}{h} + \frac{1}{2}$$

平行マッピングされた物体

球に平行マッピング



立方体に平行マッピング



UV マッピング



(提供: デザイン情報学科15期 篠原史典氏)

UV マッピングの結果



テクスチャマッピングして描画

テクスチャ座標とテクスチャオブジェクト

テクスチャ座標の in (attribute) 変数

```
// プログラムオブジェクトの作成
GLuint program = glCreateProgram();

... (ソースプログラムの読み込み, コンパイル, 取り付け, リンク等)

// in (attribute) 変数のインデックスの検索 (見つからなければ -1)
nvLoc = glGetAttribLocation(program, "nv"); // 頂点法線
tvLoc = glGetAttribLocation(program, "tv"); // テクスチャ座標
```

テクスチャユニットの uniform 変数

```
// 視点座標系への変換行列（モデルビュー変換行列）
mwLoc = glGetUniformLocation(program, "mw");

// クリッピング座標系への変換行列（モデルビュー・投影変換行列）
mcLoc = glGetUniformLocation(program, "mc");

// 法線変換行列
mgLoc = glGetUniformLocation(program, "mg");

// テクスチャユニット番号
texLoc = glGetUniformLocation(program, "tex");
```

頂点配列オブジェクトを作成する

```
// 頂点配列オブジェクト  
GLuint vao;  
glGenVertexArrays(1, &vao);  
 glBindVertexArray(vao);
```

頂点バッファオブジェクトを作成する

```
// 頂点バッファオブジェクト  
GLuint vbo[4];  
glGenBuffers(4, vbo);
```

位置, 法線, テクスチャ座標, 頂点のインデックスの四つ分

形状データ

```
// 形状データ
static GLfloat pv[VERTICES][3];    // 頂点位置
static GLfloat nv[VERTICES][3];    // 頂点法線
static GLfloat tv[VERTICES][2];    // テクスチャ座標
static GLuint face[FACES][3];    // 三角形の頂点インデックス

... (形状データの作成等)
```



二次元のテクスチャ座標

位置と法線の頂点バッファオブジェクト

```
// 頂点の座標値 pv 用の頂点バッファオブジェクト
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof pv, pv, GL_STATIC_DRAW);

// 頂点バッファオブジェクトを in 変数 pv から参照できるようにする
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
 glEnableVertexAttribArray(0);

// 頂点の法線ベクトル nv 用のバッファオブジェクト
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof nv, nv, GL_STATIC_DRAW);

// 頂点バッファオブジェクトを in 変数 nv から参照できるようにする
glVertexAttribPointer(nvLoc, 3, GL_FLOAT, GL_FALSE, 0, 0);
 glEnableVertexAttribArray(nvLoc);
```

テクスチャ座標と頂点インデックス

```
// 頂点のテクスチャ座標値 tv 用のバッファオブジェクト  
glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);  
glBufferData(GL_ARRAY_BUFFER, sizeof tv, tv, GL_STATIC_DRAW);
```

```
// 頂点バッファオブジェクトを in 変数 tv から参照できるようにする  
glVertexAttribPointer(tvLoc, 2, GL_FLOAT, GL_FALSE, 0, 0);  
glEnableVertexAttribArray(tvLoc);
```

二次元のテクスチャ座標

```
// 頂点のインデックス face 用のバッファオブジェクト  
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);  
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof face, face,  
GL_STATIC_DRAW);
```

頂点のインデックス用の
バッファオブジェクト

テクスチャオブジェクト

```
// テクスチャオブジェクトの作成  
GLuint tex;  
glGenTextures(1, &tex);  
 glBindTexture(GL_TEXTURE_2D, tex);
```

マッピングする画像の読み込み

```
// テクスチャメモリの確保と画像の読み込み
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
    format, GL_UNSIGNED_BYTE, image); // 内部フォーマット
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MAG_FILTER, GL_LINEAR);      // 拡大時に線形補間
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER, GL_LINEAR);      // 縮小時に線形補間
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);   // エッジでクランプ
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);   // エッジでクランプ
```

width, height, fomat は、それぞれ画像 image の幅、高さ、書式
image は unsigned char 型の配列

テクスチャをマッピングして描画する

```
// シェーダプログラムを選択  
glUseProgram(program);  
  
// uniform 変数を設定する  
glUniformMatrix4fv(mwLoc, 1, GL_FALSE, mw);  
glUniformMatrix4fv(mcLoc, 1, GL_FALSE, mc);  
glUniformMatrix4fv(mgLoc, 1, GL_FALSE, mg);  
glUniform1i(texLoc, 0);  
  
// 使用するテクスチャユニットの指定  
glActiveTexture(GL_TEXTURE0);  
    ↓  
    テクスチャユニット 0 のユニット番号  
    テクスチャユニット 0  
  
// マッピングするテクスチャの指定  
glBindTexture(GL_TEXTURE_2D, tex);  
  
// 描画に使う頂点配列オブジェクトの指定  
glBindVertexArray(vao);  
  
// 図形の描画  
glDrawElements(GL_TRIANGLES, FACES * 3, GL_UNSIGNED_INT, face);
```

バーテックスシェーダ

```
#version 150 core  
...  
  
in vec4 pv;           // ローカル座標系の頂点位置  
in vec4 nv;           // 頂点の法線ベクトル  
in vec2 tv;           // 頂点のテクスチャ座標値  
  
uniform mat4 mw;      // 視点座標系への変換行列  
uniform mat4 mc;      // クリッピング座標系への変換行列  
uniform mat4 mg;      // 法線ベクトルの変換行列
```

バーテックスシェーダ

```
out vec4 dc;           // フラグメントシェーダに送る頂点色の環境光+拡散光
out vec4 sc;           // フラグメントシェーダに送る頂点色の鏡面反射光
out vec2 tc;           // フラグメントシェーダに送るテクスチャ座標

void main(void)
{
    ...
    dc = iamb + idiff;   // 環境光+拡散光をフラグメントシェーダに送る
    sc = ispec;          // 鏡面反射光をフラグメントシェーダに送る
    tc = tv;              // テクスチャ座標はそのままフラグメントシェーダに送る

    gl_Position = mc * pv;
}
```

テクスチャの色は鏡面反射光
には影響させないので分ける

フラグメントシェーダ

```
#version 150 core  
  
in vec4 dc;           // 環境光 + 拡散反射光  
in vec4 sc;           // 鏡面反射光  
in vec2 tc;           // テクスチャ座標  
uniform sampler2D tex; // テクスチャユニット  
out vec4 fc;          // カラーバッファ
```

```
void main(void)  
{  
    fc = texture(tex, tc) * dc + sc;  
}
```

テクスチャユニット

テクスチャの色を
拡散反射係数として使う

補間されたテクスチャ座標

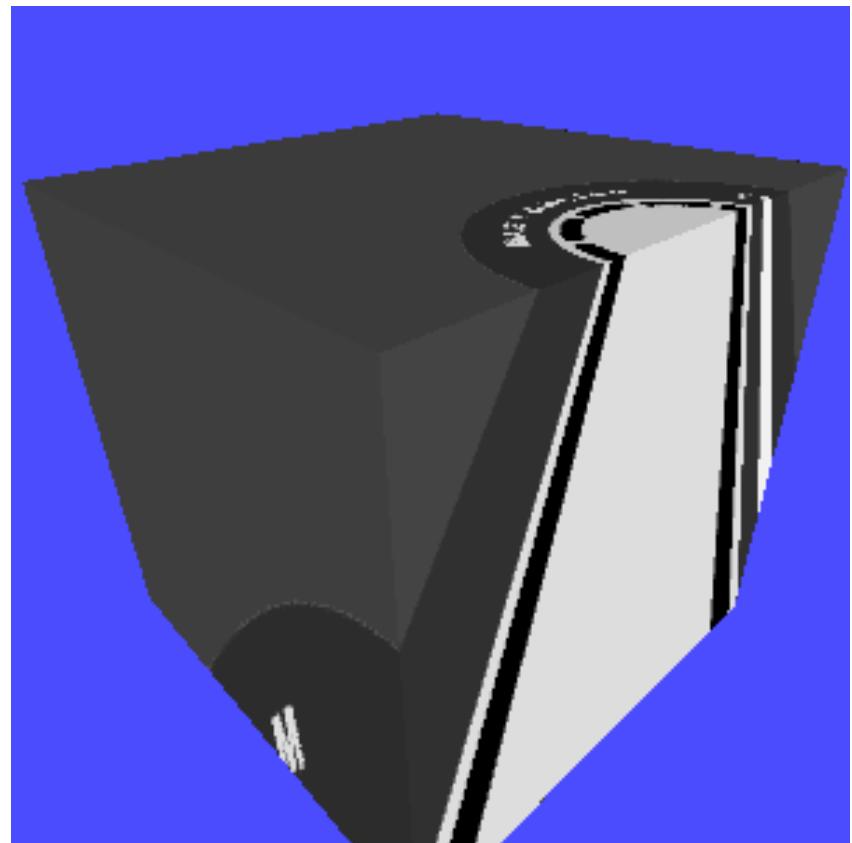
テクスチャを標本化する組み込み関数

3次元テクスチャ

テクスチャ座標も同次座標

3次元のパラメータ空間

- 頂点に対応付けるパラメータ座標を3次元で指定する
 - パラメータ座標を2次元のテクスチャ座標に変換する
 - テクスチャ自体を3次元にする（医用画像等）



同次座標のテクスチャ座標

```
#version 150 core  
  
in vec4 dc;           // 環境光 + 拡散反射光  
in vec4 sc;           // 鏡面反射光  
in vec4 tc;           // テクスチャ座標  
uniform sampler2D tex; // テクスチャユニット  
out vec4 fc;          // カラーバッファ
```

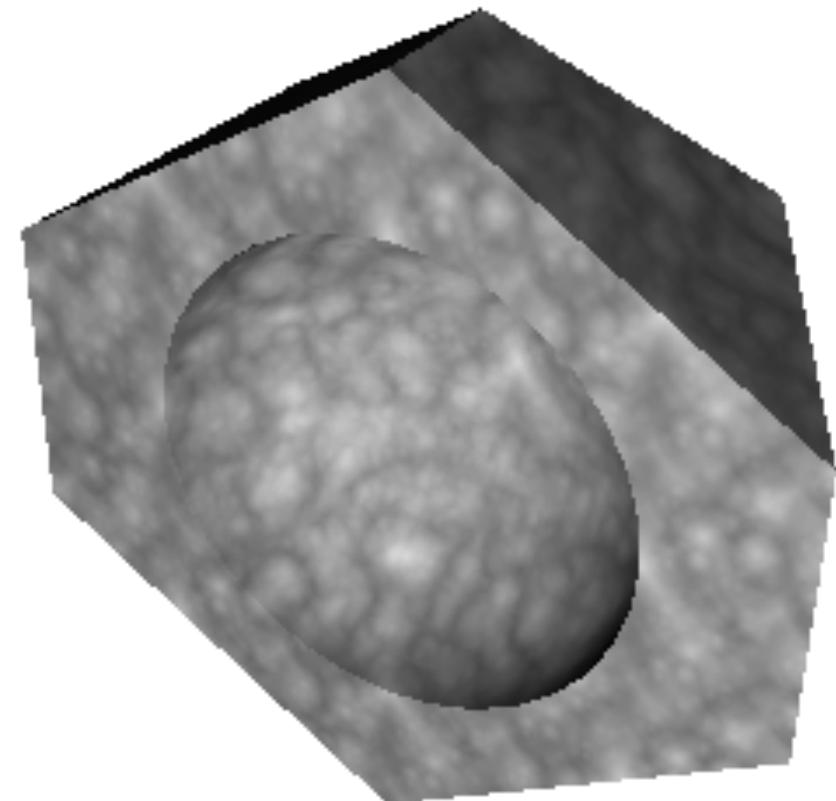
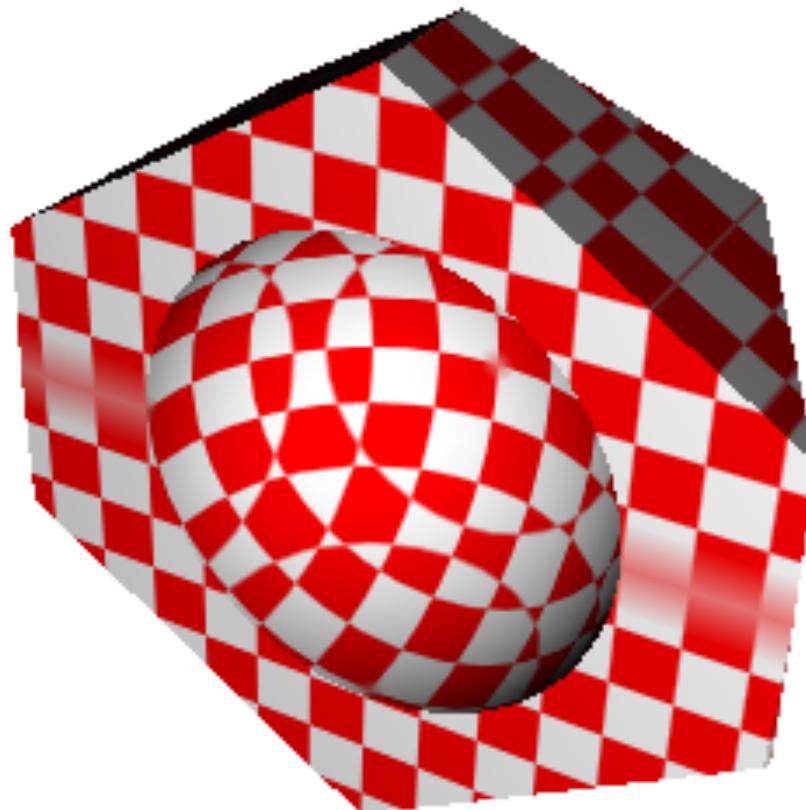
```
void main(void)  
{  
    fc = textureProj(tex, tc) * dc + sc;  
}
```

tc の XY 要素を W要素で割って二次元のテクスチャ座標で標本化する

同次座標のテクスチャ座標

テクスチャ座標を投影変換してテクスチャを標本化する組み込み関数

3D テクスチャ（ソリッドテクスチャ）



3次元テクスチャの読み込み

- テクスチャメモリを確保して、そこに画像データを格納する
 - `glTexImage3D(GL_TEXTURE_3D, 0, internalFormat, WIDTH, HEIGHT, DEPTH, 0, format, type, image);`
 - *internalFormat*: テクスチャにアルファチャネルを持たせるなら `GL_RGBA`, 持たせないなら `GL_RGB`
 - *format*: 画像データ (*image*) の形式, 画像がアルファチャンネルを持っているれば `GL_RGBA` または `GL_BGRA`, 持っていないなら `GL_RGB` または `GL_BGR`
 - *type*: 画像を格納している配列変数 (*image*) のデータ型, `GLubyte` なら `GL_UNSIGNED_BYTE`
 - 3次元テクスチャなら第1引数は `GL_TEXTURE_3D`, 第2引数はミップマップのレベルで基本は 0, 第7引数はテクスチャの境界線の太さの指定だったが今は使われないので常に 0.

3次元テクスチャのサンプリング

```
#version 150 core  
  
in vec4 dc;           // 環境光+拡散反射光  
in vec4 sc;           // 鏡面反射光  
in vec3 tc;           // テクスチャ座標  
uniform sampler3D tex; // テクスチャユニット  
out vec4 fc;          // カラーバッファ
```

```
void main(void)  
{  
    fc = texture(tex, tc) * dc + sc;  
}
```

3次元のテクスチャ座標

テクスチャを標本化する組み込み関数

複数のテクスチャの使用

テクスチャの合成・異なる材質に対するマッピング

Multipass Texture Rendering

- Multipass Rendering
 - 複数回に分けてレンダリングする
 - 照明方程式は多くの要素をもとに一度で結果を得る
 - 要因ごとに分けて別々にレンダリングし、合成することもできる
- 照明方程式の様々な要素はテクスチャで表現可能
 - 環境光の反射光・拡散反射光・鏡面反射光の和
 - テクスチャをつけたレンダリング結果を合成する
 - add
 - そのまま加算する
 - blend
 - アルファ値を使って合成する

マルチテクスチャ

- 複数のテクスチャを同時に使う

```
GLuint tex[2];  
glGenTextures(2, tex);
```

- サンプラーの uniform 変数 (uniform 変数の配列を使うこともできる)

```
GLint tex0Loc = glGetUniformLocation(program, "tex0");  
GLint tex1Loc = glGetUniformLocation(program, "tex1");
```

- テクスチャユニットごとにテクスチャオブジェクトを指定する

```
glUniform1f(tex0Loc, 0);           // tex0 のテクスチャユニットは 0  
glUniform1f(tex1Loc, 1);           // tex1 のテクスチャユニットは 1  
glActiveTexture(GL_TEXTURE0);      // テクスチャユニット 0  
glBindTexture(GL_TEXTURE_2D, tex[0]); // 一つ目のテクスチャ  
glActiveTexture(GL_TEXTURE1);      // テクスチャユニット 1  
glBindTexture(GL_TEXTURE_2D, tex[1]); // 二つ目のテクスチャ
```

- （複数のテクスチャをマッピングする図形の描画）

フラグメントシェーダでマルチテクスチャ

- テクスチャユニットごとにサンプラーの uniform 変数を用意する

```
uniform sampler2D tex0; // テクスチャユニット 0
```

```
uniform sampler2D tex1; // テクスチャユニット 1
```

```
in vec2 tc0; // 一つ目のテクスチャのテクスチャ座標
```

```
in vec2 tc1; // 一つ目のテクスチャのテクスチャ座標
```

- それぞれのサンプラー変数とテクスチャ座標を使う

```
vec4 c0 = texture(tex0, tc0),
```

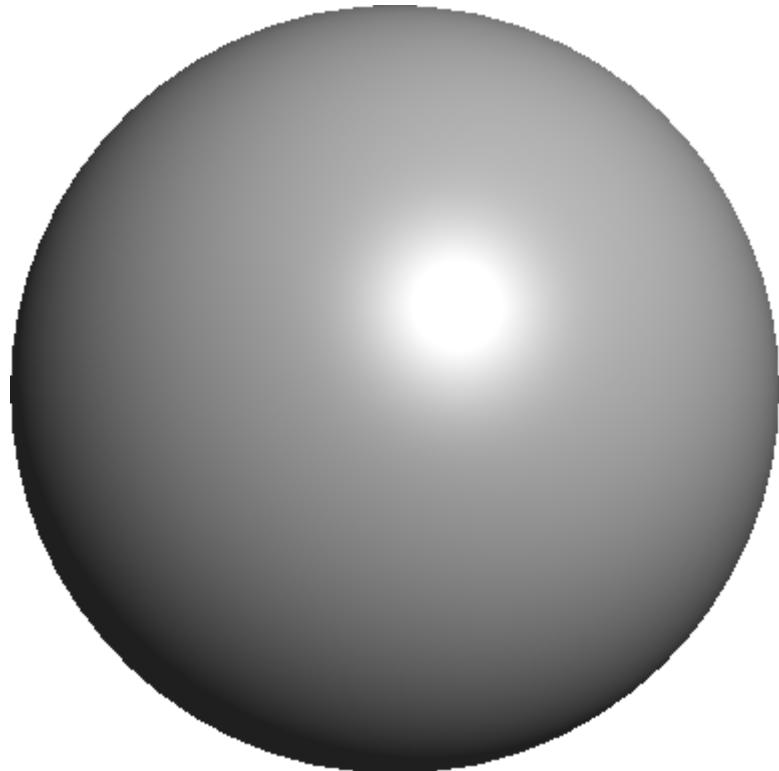
```
vec4 c1 = texture(tex1, tc1);
```

宿題

- ・テクスチャをマッピングしてください。
 - ・次のプログラムは**画素単位に陰影をつけた球を回転するアニメーション**を表示します。
 - ・<https://github.com/tokoik/ggsample08>
 - ・これにテクスチャユニット0に割り当てられたテクスチャを、陰影を付けてマッピングしてください。
 - ・テクスチャユニット0には既に画像を割り当てています。サンプラのuniform変数の変数名はcolorです。
 - ・in変数pv, nv, tvにはそれぞれ位置、法線、テクスチャ座標が入っています。
 - ・tv.sとtv.tには0~1の値が入っています。
- ・ggsample08.frag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

マッピング前



マッピング後

