Graphics with Processing

2010-11 シェーディングとマッピング

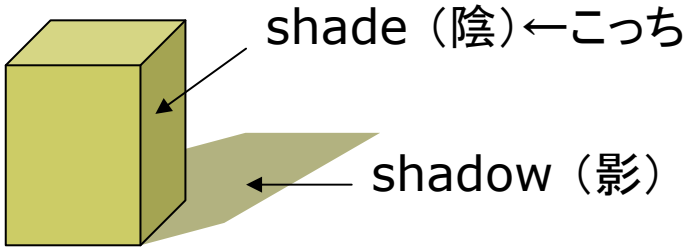
http://vilab.org

塩澤秀和

11.1 シェーディング

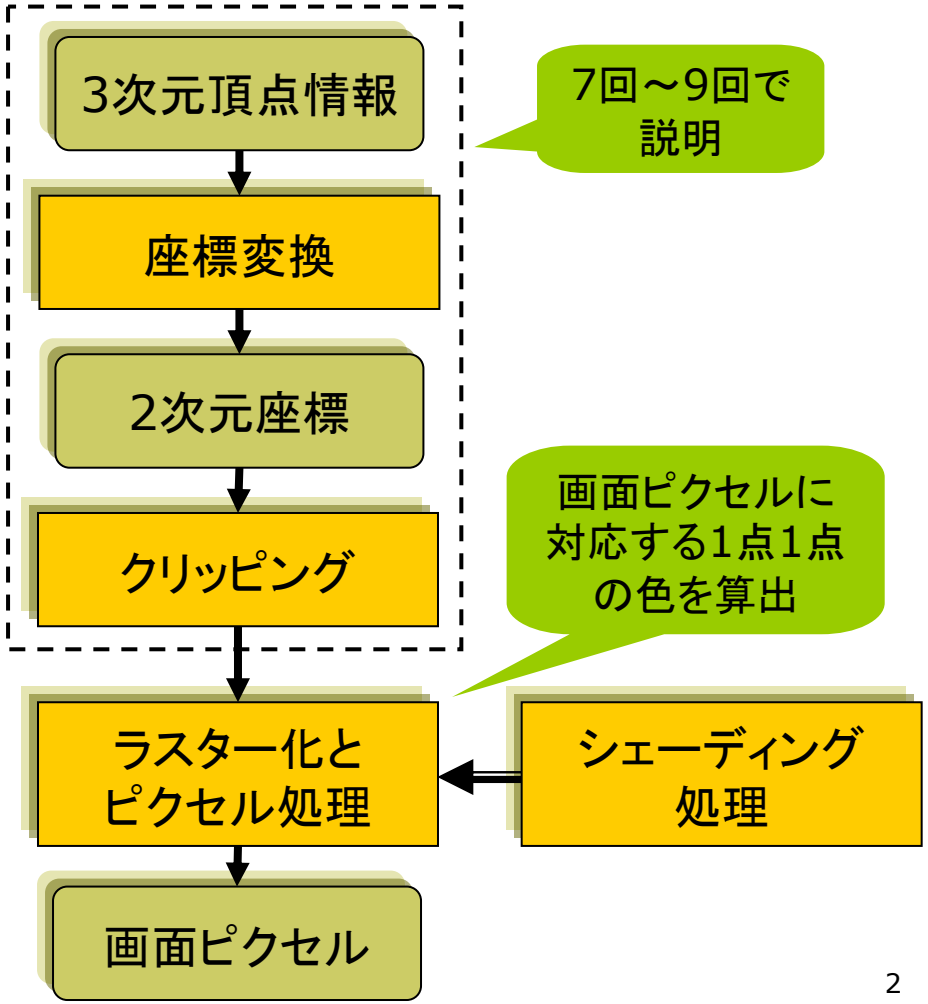
シェーディング（shading）

- シェーディングモデル
  - 光の反射・材質のモデル（前回）
  - ポリゴンの明暗の計算モデル



- フラットシェーディング
  - ポリゴンを単一色で塗りつぶし
- スムーズシェーディング
  - ポリゴン内を滑らかに明暗付け
  - グローシェーディング
  - フォンシェーディング

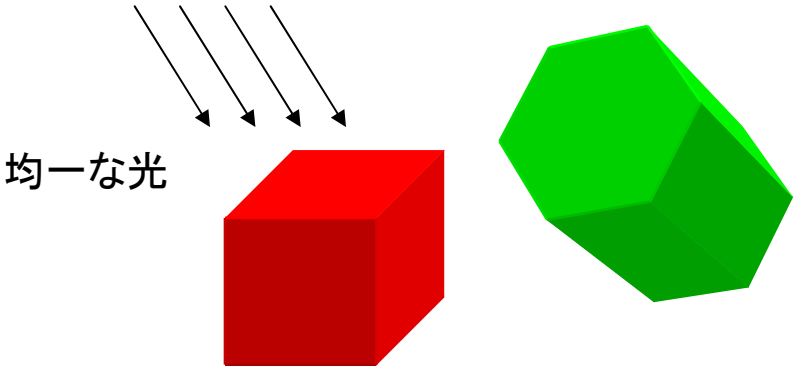
シェーディング処理のタイミング



11.2 フラットシェーディング

フラットシェーディング（p.133）

- 各ポリゴンを単一色で描画
  - ポリゴンの代表点（例：重心）の法線ベクトルを面の向きとする
  - 法線ベクトルと入射光の角度から色（反射光）を計算する
  - 面全体を同一色で描画する
  - 均一な平行光が平面に当たる場合は光学的に正しい
  - 高速だがリアリティには欠ける



法線ベクトル（p.101）

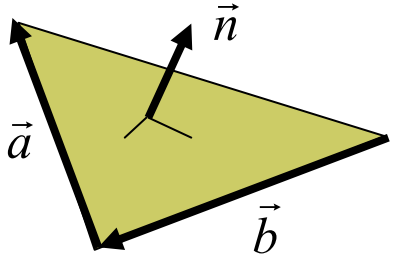
平面の方程式から

$$ax + by + cz + d = 0$$

$$\vec{N} = (a, b, c)$$

曲面の場合は？

$$\left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right)$$



ポリゴンの辺（ベクトル）から

$$\vec{N} = \vec{a} \times \vec{b}$$

外積（クロス積）については教科書等参照

単位法線ベクトル

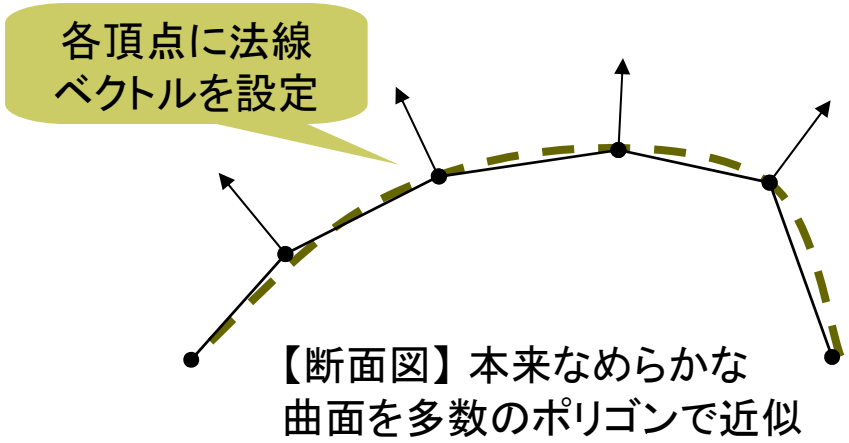
$$\vec{n} = \vec{N} / |\vec{N}|$$

（大きさを1にする）

11.3 ポリゴン曲面

ポリゴン曲面(p.78)

- ポリゴンの集合で曲面を近似
  - 三角形を使うことが多い(頂点が必ず同一平面上にあるから)
  - ポリゴンをつなぐ頂点には元の曲面の法線ベクトルを設定する
  - スムーズシェーディングで面の色を滑らかにつなげて描画することで，曲面に見せかける



【断面図】本来なめらかな曲面を多数のポリゴンで近似

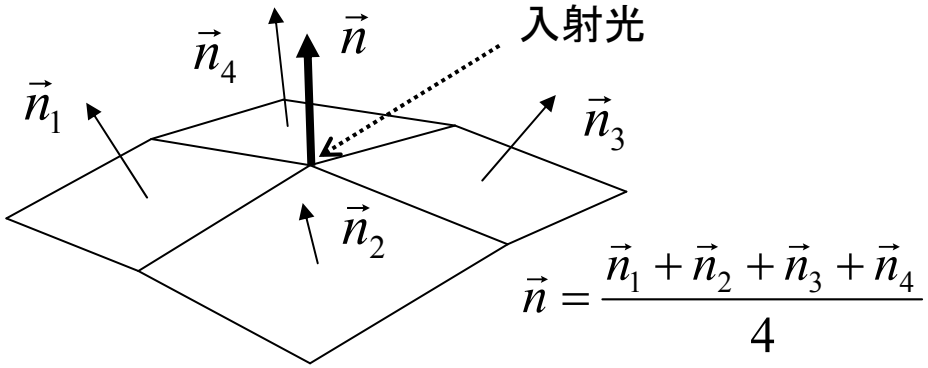
```
void draw() {
  background(0);
  directionalLight(255,255,255,1,0,-1);
  noStroke();
  translate(width/2, height/2, 0);

  beginShape(QUAD_STRIP);
  for (int a = 0; a <= 360; a += 20) {
    float x = 100 * cos(radians(a));
    float z = 100 * sin(radians(a));
    if (mousePressed) normal(x, 0, z);
    vertex(x, -100, z);
    if (mousePressed) normal(x, 0, z);
    vertex(x, 100, z);
  }
  endShape();
}
```

- normal(nx, ny, nz)
  - 曲面近似等のために，頂点位置の法線ベクトルを明示的に設定
  - 対応するvertexの直前で使う

11.4 グローシェーディング (p.134)

グロー(Gouraud)シェーディング

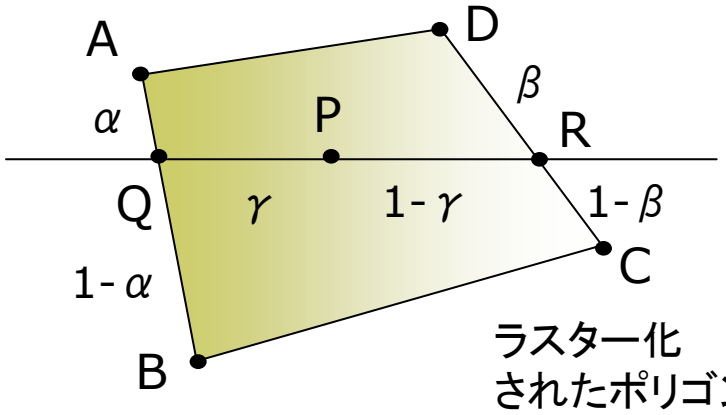
- 頂点間の色を補間
  - 頂点位置の法線ベクトルを求め，光の反射を計算してポリゴンのすみの色を決定する
  - ポリゴン内部は，色が滑らかにつながるように線形補間する
- 頂点の法線ベクトル
  - 曲面の方程式から計算できない場合，周囲の面の法線ベクトルを平均化して求める



$$\vec{n} = \frac{\vec{n}_1 + \vec{n}_2 + \vec{n}_3 + \vec{n}_4}{4}$$

バイリニア補間

- 2段階の線形補間
  - ラスター化の過程で，画面上の各ピクセルの色を補間する



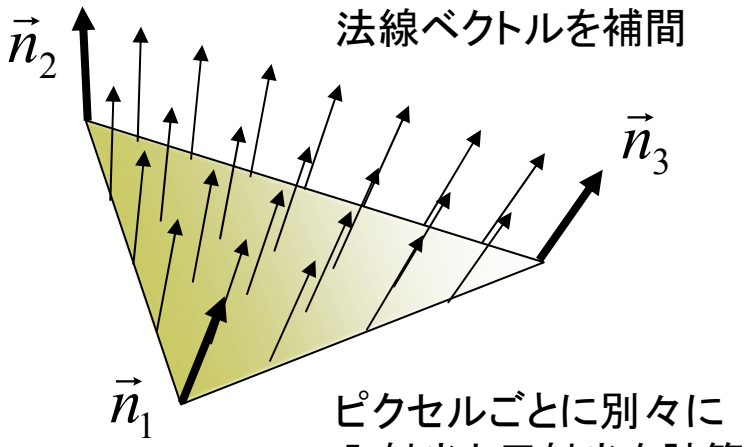
$$\left.\begin{aligned} C_{\mathrm{Q}} &= (1-\alpha)\,C_{\mathrm{A}} + \alpha\,C_{\mathrm{B}} \\ C_{\mathrm{R}} &= (1-\beta)\,C_{\mathrm{D}} + \beta\,C_{\mathrm{C}} \end{aligned}\right\} \text{1段階目}$$

$$C_{\mathrm{P}} = (1-\gamma)\,C_{\mathrm{Q}} + \gamma\,C_{\mathrm{R}} \quad \text{— 2段階目}$$

11.5 フォンシェーディング (p.135)

フォン(Phong)シェーディング

- 法線ベクトル自体を補間
  - 色を補間するのでなく，面全体の法線ベクトルを補間する
  - 画面上の各ピクセルに対応する法線ベクトルを計算し，その点での光の反射から色を決定する
  - 鏡面反射の光沢をリアルに表現



シェーディング機能の実現

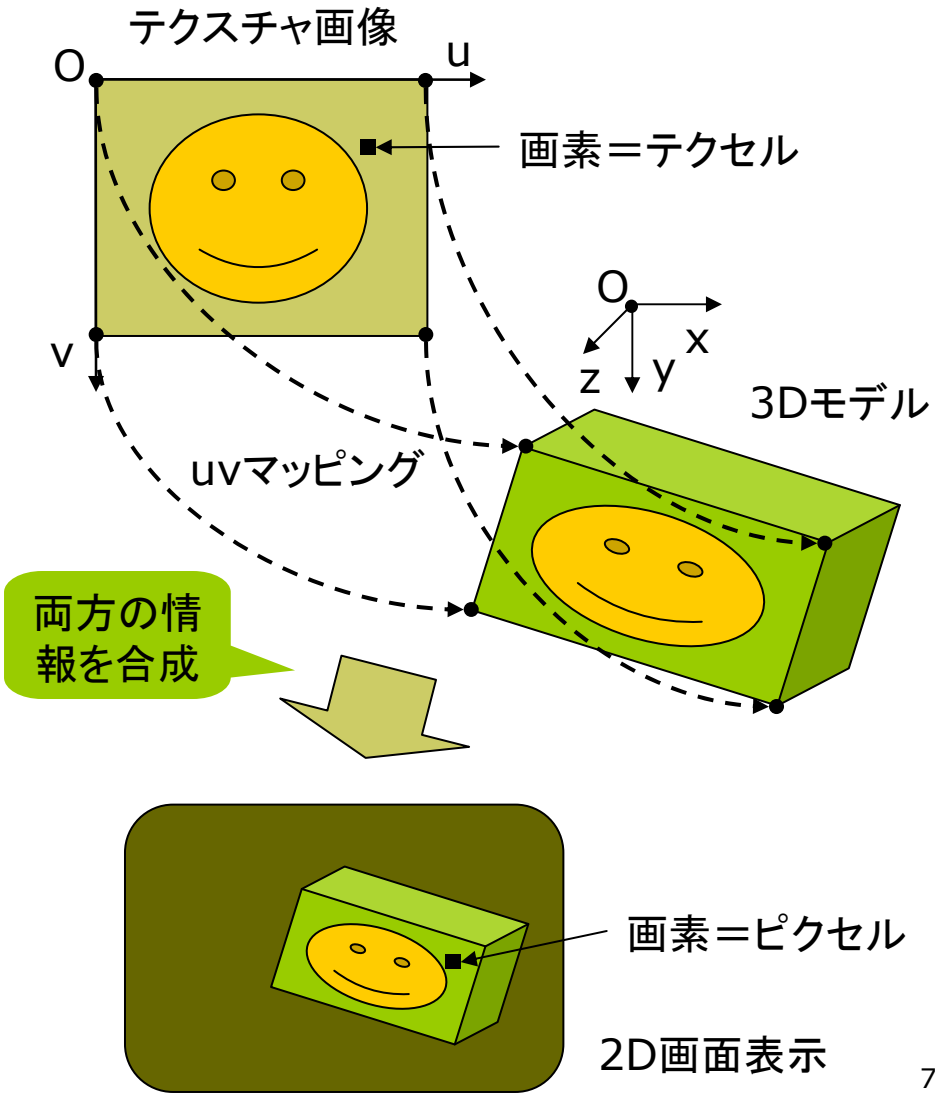
- 固定機能シェーダー
  - 従来型システムは代表的な機能だけをシステムが用意

    環境光 ⇒ 一様な定数
    拡散反射 ⇒ ランバートの式
    鏡面反射 ⇒ フォンのモデル
    シェーディング ⇒ グロー
  - これでは，影・液体・ゲームでの特殊効果などに対応しづらい
- プログラマブルシェーダー
  - 近年，水面・人の肌・動物の毛並みなど，対象ごとに多数のシェーディング技法が開発
  - それに対応して，最近のGPUはプログラム言語で内部処理を変更可能になっている

11.6 テクスチャマッピング (p.146)

テクスチャマッピング

- texture＝布目・模様
  - 立体の表面に画像をシールのように貼りつける
    (例)球に世界地図を貼りつける
  - 質感を表すのに効果てきめん
  - テクスチャ画像を構成する画素をテクセル(texel)という
- uv座標(テクスチャ座標)
  - テクスチャ画像の2次元座標
  - モデリング座標と区別するため，(u, v) (またはs,t)で表す
- uvマッピング
  - 3次元座標に2次元のテクスチャ座標を対応づけること
  - 画像(u, v) → 空間(x, y, z)



11.7 テクスチャマッピング関数

テクスチャマッピング関数

- texture(画像)
  - 画像：PImage型（4.6参照）
  - テクスチャ画像の設定
  - beginShape(), endShape()の中で指定する
- vertex(x, y, z, u, v)
  - 通常のvertex(x, y, z)の処理に加え，その点をテクスチャ座標(u, v)に対応づける
  - 2次元での画像変形にも使えるvertex(x, y, u, v)
- textureMode(座標モード)
  - uv座標の指定モード
  - IMAGE: 実際の画像の座標
  - NORMALIZED: 0.0～1.0
- smooth()
  - テクスチャマッピングで使うと，透視投影におけるテクスチャのゆがみを防ぐ
    ⇒「パースペクティブ補正」参照
  - OPENGLの場合はデフォルトでゆがまないので不要
- サンプルプログラム
  - Examples → 3D → Textures → 4つのプログラム
  - Examples → Library → OpenGL→ TexturedSphere

テクスチャの形とポリゴンの形は一致していなくてもいいし，大きな画像から一部を切り出して貼り付けてもいい

11.8 テクスチャマッピングの使用例

```
// 準備：画像ファイル（kouji50m.jpg）を
// あらかじめ講義ページからダウンロードして
// スケッチのdataフォルダに入れておく
// （メニューで Sketch → Add File...）
import processing.opengl.*;

PImage tex;

void setup() {
  size(300, 300, OPENGL);
  tex = loadImage("kouji50m.jpg");
}

void draw() {
  background(0);
  translate(width/2, height/2, 0);
  rotateY(-radians(frameCount));

  // smooth();
  noStroke();
  beginShape(QUADS);
  texture(tex);
  textureMode(NORMALIZED);
  vertex(-20,-50,  0, 0, 0);
  vertex( 20,-50,  0, 1, 0);
  vertex( 20, 50, 15, 1, 1);
  vertex(-20, 50, 15, 0, 1);
  endShape();

  fill(#ffffff, 128);
  stroke(#555555);
  beginShape(QUADS);
  vertex(-20,-50,   0);
  vertex( 20,-50,   0);
  vertex( 20, 50, -15);
  vertex(-20, 50, -15);
  endShape();
}
```

画像はグローバル変数で定義する

画像はsetupの中で一度だけ読み込む

P3Dでゆがみ補正

長方形に画像texをテクスチャマッピング

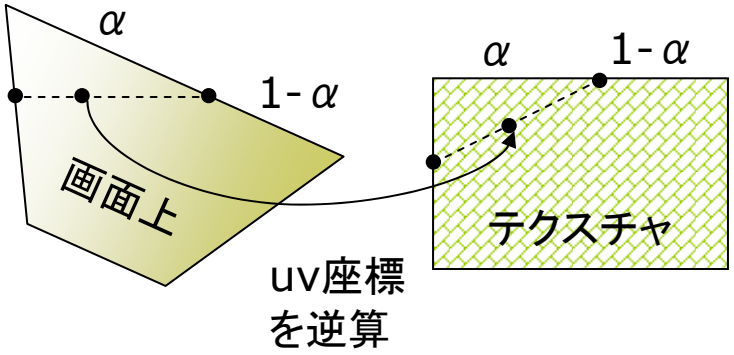
uv座標は0～1モード

半透明は必ず最後に描画（zバッファの問題を回避）

11.9 テクスチャの描画処理

テクスチャの描画処理（p.148）

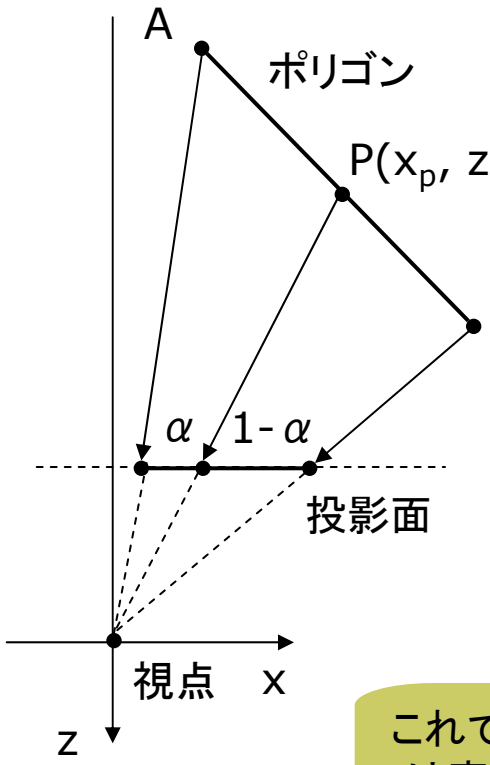
- 画面座標→uv座標
  - 画面上の各ピクセルに対応するuv座標を逆算する
  - ポリゴンの頂点に設定されたuv座標から補間する



- uv座標→描画色
  - 求めたuv座標の周辺テクセルから補間等で描画色を決める
  - 画像の拡大・変形技術と同じ

パースペクティブ補正

- 透視投影での補正
  - 遠くにいくほどテクスチャを縮めて“ゆがみ”を防ぐ



3Dと画面上とで長さの比が違う

テクスチャ座標を T(u, v) とすると

$$\frac{T_P}{z_P} = (1-\alpha)\frac{T_A}{z_A} + \alpha\frac{T_B}{z_B}$$

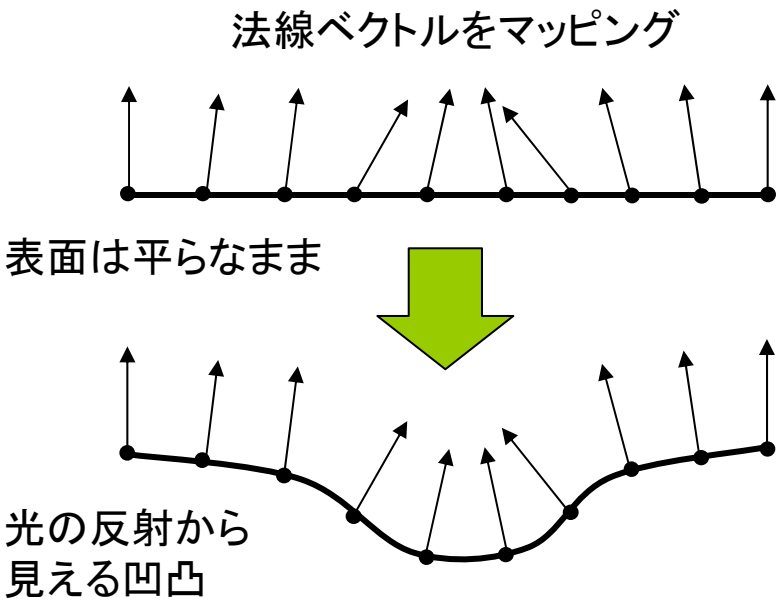
$$\frac{1}{z_P} = (1-\alpha)\frac{1}{z_A} + \alpha\frac{1}{z_B}$$

これで計算できる理由は専門書などを参照

11.10 その他のマッピング

バンプマッピング（p.151）

- 凹凸を表面にマッピング
  - 法線ベクトルの集合を面に貼り付けることで，まるで凹凸があるかのように見せる
  - 表面の細かい凹凸を簡単かつ少ない計算量で表現できる



その他のマッピング

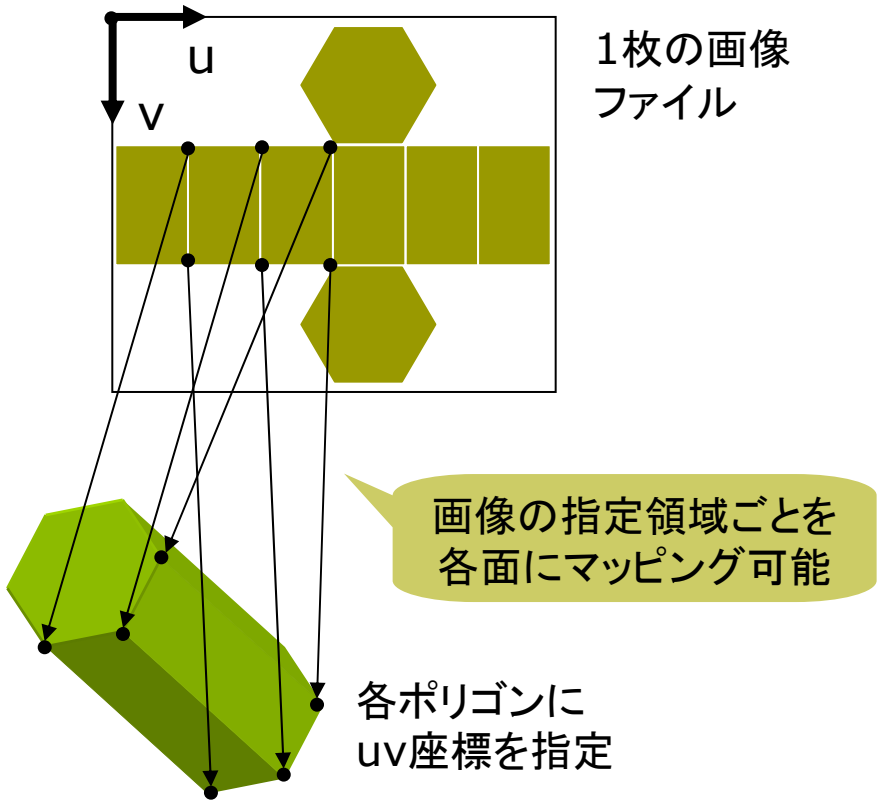
- 投影テクスチャマッピング(p.148)
  - プロジェクタで投影するように，物体にテクスチャを貼り付ける
  - 影の表現にも応用できる
- 環境マッピング（p.153）
  - 金属などへの景色の映り込みをテクスチャで表現する
  - 景色のテクスチャを作っておき，物体の表面に反射して視点から見える景色を計算して描画
- マッピングの利点
  - 頂点数（座標計算）を増やしたり，光の計算を複雑化させないで，リアリティを高めることができる
  - 多数の手法が開発されている

11.11 演習課題

課題

- 11.3のサンプルをもとに円筒の表面に画像をぐるりと貼り付けるプログラムを作成しなさい
  - アニメーション等によって円筒の全表面が見えるようにすること
  - さらに，PNG形式などで透過色（透明部分）がある画像を貼ってみると面白い
- 提出方法
  - プログラムを保存したら，Tools → Archive Sketch で，**画像もまとめたZIPファイルを作る**
  - Processingフォルダにできた ZIPファイルを提出する
  - アップロード時に種類で**「フォルダ圧縮ZIPファイル」**を選ぶこと

展開図画像などの利用



1枚のテクスチャから別々のオブジェクトに貼り付けることも可能 ⇒ “テクスチャアトラス”

11.12 参考:オフスクリーンレンダリング

```
import processing.opengl.*;

PGraphics pg;  // 隠し画面用変数

void setup() {
  size(400, 300, OPENGL);
  // 隠し画面を開く
  // 3つの引数の意味はsize関数と同じ
  pg = createGraphics(100, 100,
         JAVA2D);
}

void draw() {
  // 隠し画面上での描画処理
  pg.beginDraw();  // 開始
  pg.background(255);
  pg.translate(50, 50);
  pg.fill(240, 180, 180);
  pg.rotate(radians(frameCount));
  pg.rect(-100, -3, 200, 6);
  pg.endDraw();    // 終了
  // 表示画面での処理
  background(255);
  lights();
  translate(width / 2, height / 2, 0);
  rotateX(radians(frameCount) / 8);

  scale(90);
  beginShape(QUADS);
  texture(pg);  // 隠し画面を画像として使う
  textureMode(IMAGE);

  vertex(-1, 1, 1,   0,   0);
  vertex( 0, 1, 0,  50,   0);
  vertex( 0,-1, 0,  50, 100);
  vertex(-1,-1, 1,   0, 100);
  vertex( 0, 1, 0,  50,   0);
  vertex( 1, 1, 1, 100,   0);
  vertex( 1,-1, 1, 100, 100);
  vertex( 0,-1, 0,  50, 100);
  endShape();
}
```