

# Graphics with Processing



2015-14 CGシステムとCGの応用

<http://vilab.org>

塩澤秀和

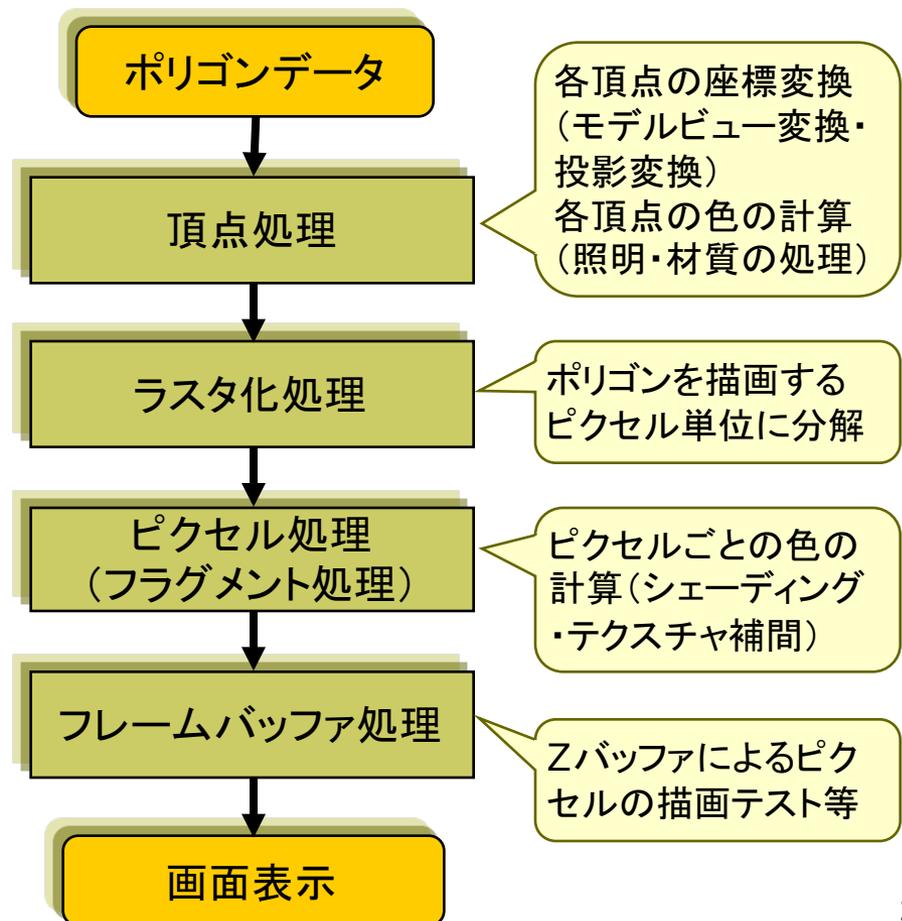
# 14.1 3DCGシステム

## 3DCG API(p.280)

- OpenGL (旧Silicon Graphics)
  - リアルタイムCG初期から
  - ハイエンドCGソフト, UNIX, iPhone, PS3, Wii(類似)など
- DirectX (Microsoft)
  - リアルタイムCG(特にゲーム)
  - Windows, Xbox
  - 高速性重視, 対応ハードが安い
- その他のリアルタイムAPI
  - Java3D, GNMX(PS4)
  - Mantle(AMD), Metal(iOS)
- RenderMan (Pixar)
  - 非リアルタイムCG(映像製作)
  - 映画製作で標準的

## レンダリングパイプライン(p.284)

- 専用ハードウェアの処理手順



# 14.2 シェーダプログラミング

## プログラマブルGPU

- GPU
  - Graphics Processing Unit
  - 3DCG計算(特にリアルタイム)のための専用ハードウェア
  - 今ではほぼすべてのPCが搭載
- プログラマブルシェーダ
  - GPUのレンダリング処理を用途に応じてカスタマイズできる機能
  - CG技術が高度化・多様化し, 固定機能では対応できなくなった
- シェーディング言語
  - GLSL - OpenGL, WebGL
  - HLSL - DirectX, Xbox
  - Cg - NVIDIA, PS3
  - PPSL(PS4), Metal(iOS),...

## プログラマブルシェーダの機能

- 頂点シェーダ
  - 頂点処理(単一頂点の座標や色の処理)をプログラミング
    - ⇒ モデルビュー変換, 投影変換, 頂点色の計算(照明・材質処理), テクスチャ座標の算出
- ジオメトリ(プリミティブ)シェーダ
  - 頂点処理後, プリミティブ(点, 線分, 三角形)単位の処理を追加
    - ⇒ 頂点の増減, プリミティブの変更
- ピクセル(フラグメント)シェーダ
  - 頂点シェーダ等の結果を利用し, ピクセル処理をプログラミング
    - ⇒ シェーディング・マッピング処理, 画像処理的エフェクト

## 14.3 Processing+GLSLの例(2D)

```
/* 本体プログラム ripple.pde */
PShader ripple; // シェーダオブジェクト

void setup() {
  size(600, 600, P3D);
  ripple = loadShader("ripple.glsl");
  // シェーダへの変数受け渡し
  ripple.set("diagonal", 1.414 * 600);
}

void draw() {
  float time = millis() / 1000.0;
  ripple.set("time", time);
  ripple.set("mouse", (float)mouseX,
            (float)(height - mouseY));
  shader(ripple);
  // 全ピクセルに対してシェーダのみで描画
  rect(0, 0, width, height);
}
```

```
/* シェーダプログラム ripple.glsl */
// 渡された変数の受け取り
uniform float time, diagonal;
uniform vec2 mouse;

const float gap = 100.0;

// 各ピクセルに対する処理
void main() {
  float d = length(gl_FragCoord.xy - mouse);
  float c = 0.0;
  float r = mod(time * gap, gap);
  while (r < diagonal) {
    // いわゆるネオン管のようなエフェクト
    c += 2.0 / abs(d - r);
    r += gap;
  }
  gl_FragColor = vec4(0.0, 0.0, c, 1.0);
}
```

# 14.4 GLSL入門

## 変数の種類

uniform	全画素/頂点の共通変数
in	画素/頂点ごとの入力変数
out	画素/頂点ごとの入力変数

## ベクトル(座標/色)

データ型	vec2, vec3, vec4 vec3 v = vec3(0.1,0.2,0.3);
要素名	xyzw または rgba
要素の参照	v.x, v.z, v.xy, v.rgb, v.a など
演算	a*v, v*v(要素同士の掛算)

## 行列

データ型	mat2, mat3, mat4
演算	a*mat, mat*mat, mat*vec

## 関数

スカラー	sin, cos, atan, pow, sqrt, abs, min, clamp, mix(線形補間) など
ベクトル	dot(内積), length, distance, normalize, reflect(反射方向) など

## 頂点シェーダ

入力	uniform, in(attribute※) 変数
出力	gl_Position.xyzw (視点座標系での頂点座標) 任意の out(varying※) 変数

## フラグメントシェーダ

入力	gl_FragCoord.xy(ピクセル座標) uniform, in(varying※) 変数
出力	gl_FragColor.rgb(ピクセルの色)

※ 以前のバージョンでの書き方

## 14.5 GLSLでフォンシェーディング

```
PShader phong; // シェーダクラス
```

```
void setup() {  
  size(600, 600, P3D);  
  noStroke();  
  // dataフォルダに入れてあるフラグメント  
  // シェーダと頂点シェーダを読み込む  
  phong = loadShader("fshader.glsl",  
                    "vshader.glsl");  
}
```

```
void draw() {  
  background(0);  
  translate(width/2, height/2, 0);  
  rotateX(radians(-30));  
  
  float angle = radians(frameCount);  
  float x = 200 * cos(angle);  
  float z = 200 * sin(angle);
```

```
  shader(phong); // シェーダの有効化
```

```
  // 簡単のため、照明は点光源1つだけとし、  
  // 環境光はシェーダに直接記述している  
  lightSpecular(50, 50, 50);  
  pointLight(200, 200, 200, x, -200, z);
```

```
  // この例では、fillとshininessにのみ対応し、  
  // specularは無視され、fillと同一色となる  
  fill(180, 180, 180); shininess(50);
```

```
  // 1枚の板で床を表示しても大丈夫
```

```
  beginShape(QUADS);  
  vertex(-300, 0, -300); vertex(300, 0, -300);  
  vertex(300, 0, 300); vertex(-300, 0, 300);  
  endShape();
```

```
  fill(220, 180, 80); shininess(100);  
  sphere(100);
```

```
}
```

## 14.6 (続き) 頂点シェーダ

```

/* ファイル名: vshader.glsl */
// ProcessingのLIGHTシェーダモード
#define PROCESSING_LIGHT_SHADER

// LIGHTシェーダモードで用意される共通変数
uniform mat4 modelview; // モデルビュー行列
uniform mat4 transform; // 合成変換行列
uniform mat3 normalMatrix; // 法線変換行列

// 簡単のため、点光源1つを前提としている
uniform vec4 lightPosition; // 視点座標系

// 頂点ごとに設定される変数(ローカル座標系)
in vec4 vertex; // 頂点座標
in vec3 normal; // 法線ベクトル
in vec4 color; // 頂点の材質色
in float shininess; // 輝き係数

// フラグメントシェーダに渡す補間変数
out vec3 fN; // 法線ベクトル
out vec3 fV; // 視点へのベクトル
out vec3 fL; // 光源へのベクトル
// 色関係はそのまま出力
out vec4 fColor = color;
out float fShininess = shininess;

void main() {
    // 入力頂点の座標を視点座標系に変換
    gl_Position = transform * vertex;

    // 視点座標系での各ベクトルを求める
    fN = normalMatrix * normal;
    fV = -(modelview * vertex).xyz;
    fL = lightPosition.xyz + fV;
}

```

この例では、簡単のため、拡散・鏡面・環境反射色をすべてcolor(本来は拡散反射色)を使って計算する

## 14.7 (続き)フラグメントシェーダ

```

/* ファイル名: fshader.glsl */
// 入射光の拡散反射成分と鏡面反射成分
uniform vec3 lightDiffuse, lightSpecular;

// 頂点シェーダの出力を補間(視点座標系)
in vec3 fN, fL, fV;
in vec4 fColor;
in float fShininess;

void main() {
    // 各ベクトルを単位ベクトル化する
    vec3 N = normalize(fN);
    vec3 L = normalize(fL);
    vec3 V = normalize(fV);
    // 反射方向のベクトル
    vec3 R = normalize(reflect(-L, N));

    vec3 diffuse = vec3(0.0, 0.0, 0.0);
    vec3 specular = vec3(0.0, 0.0, 0.0);

    // ランバートの式
    float LdotN = dot(L, N); // 内積 = |L| |N| cosθ
    if (LdotN > 0.0) {
        // 各ピクセルにおける拡散反射光と鏡面反射光
        // (材質色 × 照明色 × 係数)を求める
        diffuse = fColor.rgb * lightDiffuse * LdotN;
        specular = fColor.rgb * lightSpecular
            * pow(max(dot(R, V), 0.0), fShininess);
    }

    // 簡単のため、環境光は(0.2, 0.2, 0.2)に固定
    vec3 ambient = fColor.rgb * vec3(0.2, 0.2,
        0.2);

    // 減衰計算(逆2乗で計算してしまうと不自然)
    float fallOff = 1.0 / (1.0 + 0.001 * lenfth(fL));
    gl_FragColor.rgb =
        fallOff * (diffuse + specular) + ambient;
    gl_FragColor.a = fColor.a;
}

```

# 14.8 CGの応用

## 建築・設計

- CAD
  - CAD=コンピュータ支援設計
  - 製図・回路設計
  - 建築設計
  - 景観シミュレーション

## エンターテインメント

- コンピュータゲーム
  - ゲームはCGとともに発展
  - 2次元 → 3次元
- 映画・アニメーション(CGI)
  - SF映画
  - アニメーション
  - 実写映像への波や嵐の追加

## 人間との対話環境

- ユーザインタフェース
  - GUI, ウィンドウシステム
  - 3Dユーザインタフェース
- バーチャルリアリティ(VR)
  - 3次元仮想空間
  - オグメンテッドリアリティ(AR)  
(現実空間にCGを合成する)

## 可視化(visualization)

- 医療・科学・教育
  - データを見えるようにする
  - 科学データの分析
- 情報可視化
  - 情報分析のための可視化
  - 図解的利用, 「見える化」