

Graphics with Processing



2021-14 プログラマブルシェーダ

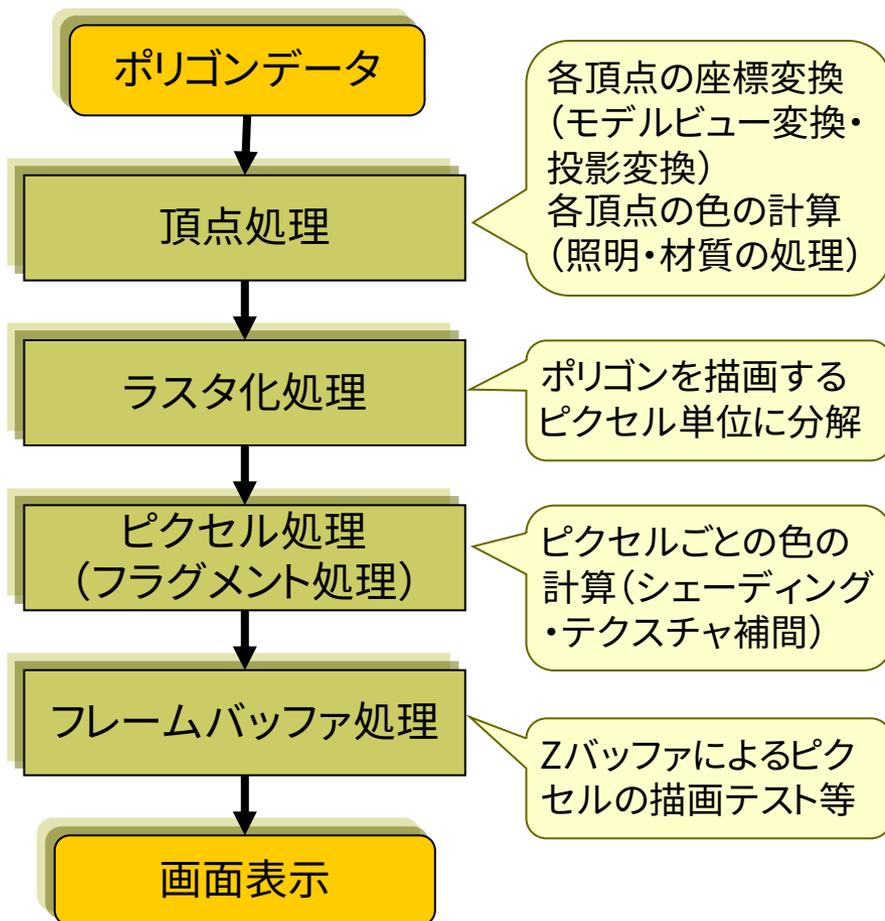
<http://vilab.org>

塩澤秀和

14.1* シェーダプログラミング

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



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

14.2 GLSLによる2D描画の例

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

void setup() {
  size(600, 600, P2D);
  // フラグメントシェーダを読み込み有効化
  circle = loadShader("circle.glsl");
  // シェーダにfloat型変数を渡す
  circle.set("radius", 20.0);
}

void draw() {
  // シェーダに2Dベクトルの値を渡す
  circle.set("center", (float)mouseX,
            (float)(height - mouseY));
  // シェーダを有効にして全ピクセルを描画
  shader(circle);
  rect(0, 0, width, height); // 全ピクセル
}
```

```
/* シェーダプログラム circle.glsl */
// 渡された変数(中心と半径)を受け取る
uniform vec2 center; // 2Dベクトル
uniform float radius; // スカラー(float)

// 定数の定義(前景色と背景色のRGBA)
const vec4 bg = vec4(1.0, 1.0, 1.0, 1.0);
const vec4 fg = vec4(0.2, 0.5, 0.2, 1.0);

// 各ピクセルで別々に実行されるメイン関数
void main() {
  // そのピクセルから中心までの距離を取得
  float d = length(gl_FragCoord.xy - center);

  // 円の内外を判定し、ピクセルの色を設定
  if (d < radius) gl_FragColor = fg;
  else            gl_FragColor = bg;
}
```

14.3 GLSL入門

変数の種類

uniform	全画素/頂点で使える共通変数
in	画素/頂点ごとの入力変数
out	画素/頂点ごとの出力変数

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
演算の例	a*v, v*v (対応要素同士の掛算)

行列

データ型	mat2, mat3, mat4
演算の例	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

頂点シェーダの入出力

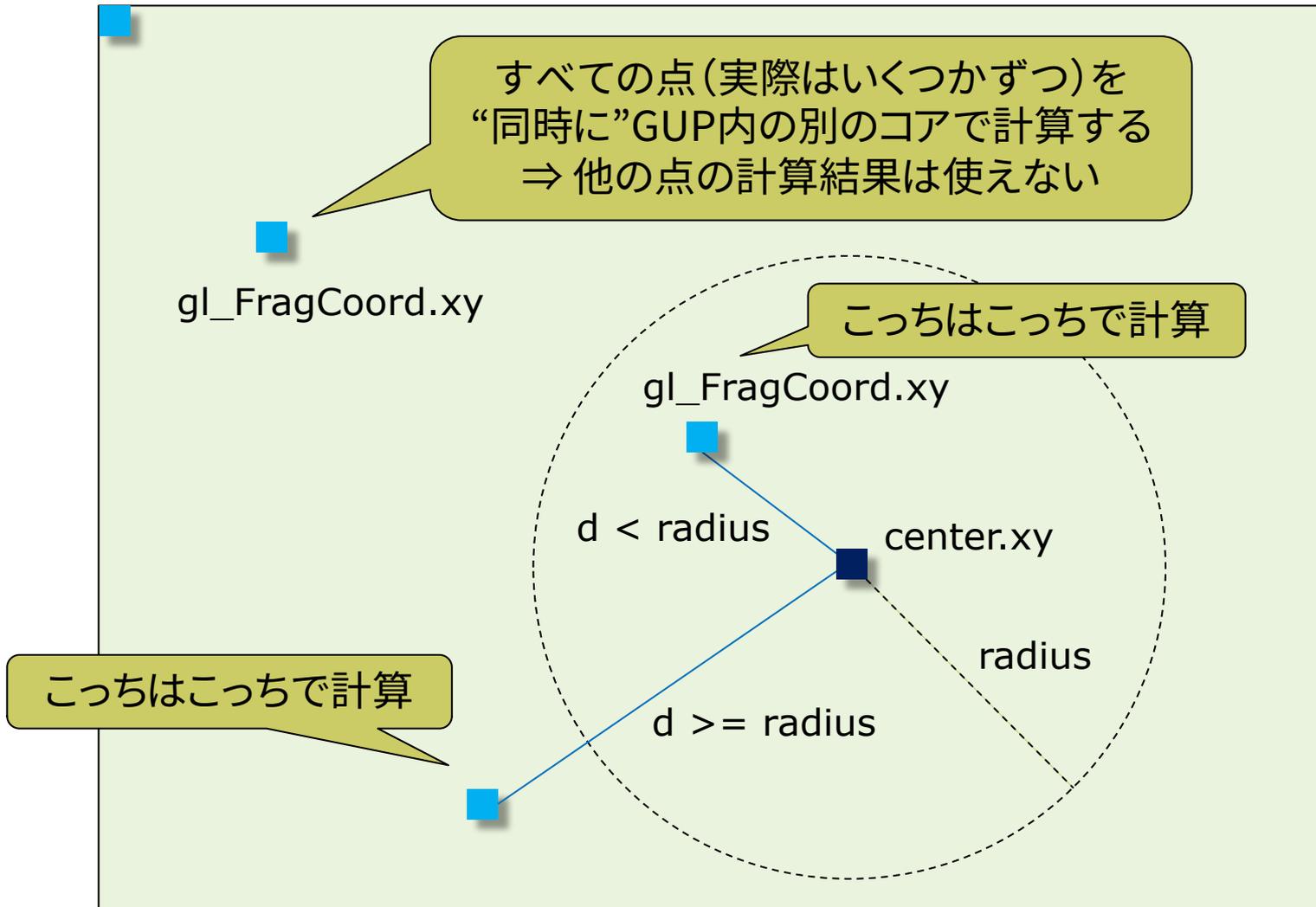
入力	uniform, in (attribute※) 変数
出力	gl_Position.xyzw (視点座標系での頂点座標) 任意の out (varying※) 変数

フラグメントシェーダの入出力

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

14.4 並列処理の考え方



14.5 波紋のような効果

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

void setup() {
  size(600, 600, P2D);
  ripple = loadShader("ripple.glsl");
  ripple.set("rmax", 300.0);
}

void draw() {
  background(0, 0, 255);
  ripple.set("time", millis() / 1000.0);
  ripple.set("center", (float)mouseX,
               (float)(height - mouseY));
  // シェーダを適用した円を描画
  shader(ripple);
  ellipse(mouseX, mouseY, 600, 600);
  // 独自シェーダを解除
  resetShader();
}
```

```
/* 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; // 経過時間(秒)
uniform float rmax; // 最大半径
uniform vec2 center; // 中心(マウス座標)

const float gap = 60.0; // 波紋の間隔

// 各ピクセルで別々に並列実行される処理
void main() {
  float d = length(gl_FragCoord.xy - center);
  float c = 0.0;
  // gap間隔で複数のリング(半径r)を生成し、
  // すべてのリング(光源)からの寄与を合計
  for (float r = gap * fract(time); r < rmax;
       r += gap) {
    c += 2.0 / abs(d - r); // 各リングからの距離
  }
  gl_FragColor = vec4(0.0, 0.0, c, 1.0);
}
```

14.6 レイキャスティング (13.6参照)

```
/* Processing 本体プログラム */
PShader sh;

void setup() {
  size(600, 600, P2D);
  sh = loadShader("raycast.glsl");
}

void draw() {
  sh.set("size", (float)width, (float)height);
  sh.set("r", sin(frameCount / 10.0) + 1.0);
  shader(sh);
  rect(0, 0, width, height);
}

/* シェーダプログラム raycast.glsl */
uniform vec2 size;
uniform float r;
vec3 kd = vec3(0.7, 1.0, 0.7);
vec3 light = normalize(vec3(1, 1, -3));
vec3 center = vec3(0, 0, -10);
```

```
void main() {
  gl_FragColor = vec4(0, 0, 0, 1);

  float scrX = (gl_FragCoord.x * 2.0
    - size.x) / size.x;
  float scrY = (gl_FragCoord.y * 2.0
    - size.y) / size.y;
  float scrZ = -2.0;
  vec3 ray = normalize(vec3(scrX, scrY, scrZ));
  vec3 nearest = ray * dot(center, ray);
  vec3 nc = nearest - center;
  float d2 = r * r - dot(nc, nc);

  if (d2 > 0) {
    vec3 p = nearest - sqrt(d2) * ray;
    vec3 n = normalize(p - center);
    vec3 color = kd * dot(n, -light);
    gl_FragColor = vec4(color, 1.0);
  }
}
```

14.7 GLSL参考サイト

- GLSLで簡単2Dエフェクト
 - <http://www.demoscene.jp/?p=1147>

- [連載]やってみれば超簡単! WebGL と GLSL で始める、はじめてのシェーダコーディング
 - <http://qiita.com/doxas/items/b8221e92a2bfdc6fc211>

- GLSL Sandbox超活用術
 - <http://www.demoscene.jp/?p=1154>

- GLSL Editor
 - <http://jp.wgld.org/js4kintro/editor/>

- Shadertoy
 - <https://www.shadertoy.com>

14.8 頂点シェーダの利用

頂点シェーダの役割

□ 頂点の座標変換

- 第8回～第9回の計算処理を実現
- 入力: position (ローカル座標系)
出力: gl_Position (視点座標系)
- 通常の場合

$gl_Position = transform * position$

□ 頂点でのパラメータ算出

- 第10回の色の計算(の一部)など
- 入力: color, normal など
- 出力: out変数 → 自動補間

□ 全頂点の共通変数

- 各種変換行列等を処理系が設定
- transform (合成変換行列)
- normalMatrix (法線変換行列)
- lightPosition (光源座標) など

□ 変数の自動補間

《3D空間》
頂点シェーダの出力
変数(out)

例: 3D空間のポリゴンの各頂点の色

ラスタライズ
補間

《2D画面》
フラグメントシェーダの
入力変数(in)

例: 2D画面のポリゴン内の各ピクセルの色

14.9 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);
```

```
}
```

14.10 (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 position; // 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 * position;

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

14.11 (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乗で計算すると不自然)
float fallOff = 1.0 / (1.0 + 0.001 * length(fL));
gl_FragColor.rgb =
    fallOff * (diffuse + specular) + ambient;
gl_FragColor.a = fColor.a;
}

```