

Processingユーザーのための シェーダーアート入門

Atsushi Asakura / aadebdeb, @PCD Tokyo, 2020/02/01

資料URL: ここに資料のURLを貼る

自己紹介

- Atsushi Asakura / aadebdeb
 - クリエイティブコーダー / プログラマー
 - 3D CG、シミュレーション、ジェネティブアートに特に興味がある
-
- Twitter
 - https://twitter.com/aa_debdeb
 - OpenProcessing
 - <https://www.openprocessing.org/user/51764/>
 - NEORT
 - <https://neort.io/itjyV7OFFeaAOMQKr6tkgIED8TE3>

内容

- 1. 基礎
 - シェーダーとは？ シェーダーアートとは？
 - シェーダー言語 GLSL
 - 作例 (基礎編)
- 2. ランダム
 - 一様乱数、ブロックノイズ、バリューノイズ、fbm
 - 作例 (ランダム編)
- 3. 図形
 - 符号付き距離関数とは？
 - 基礎的な図形 (円, 四角形)
 - 移動・回転・スケール, ブーリアン演算
 - 作例 (図形編)
- 4. 3D
 - レイマーチング

目標

シェーダーアートを自分で調べながら
制作できるようになる

なので、気軽に質問してください！！

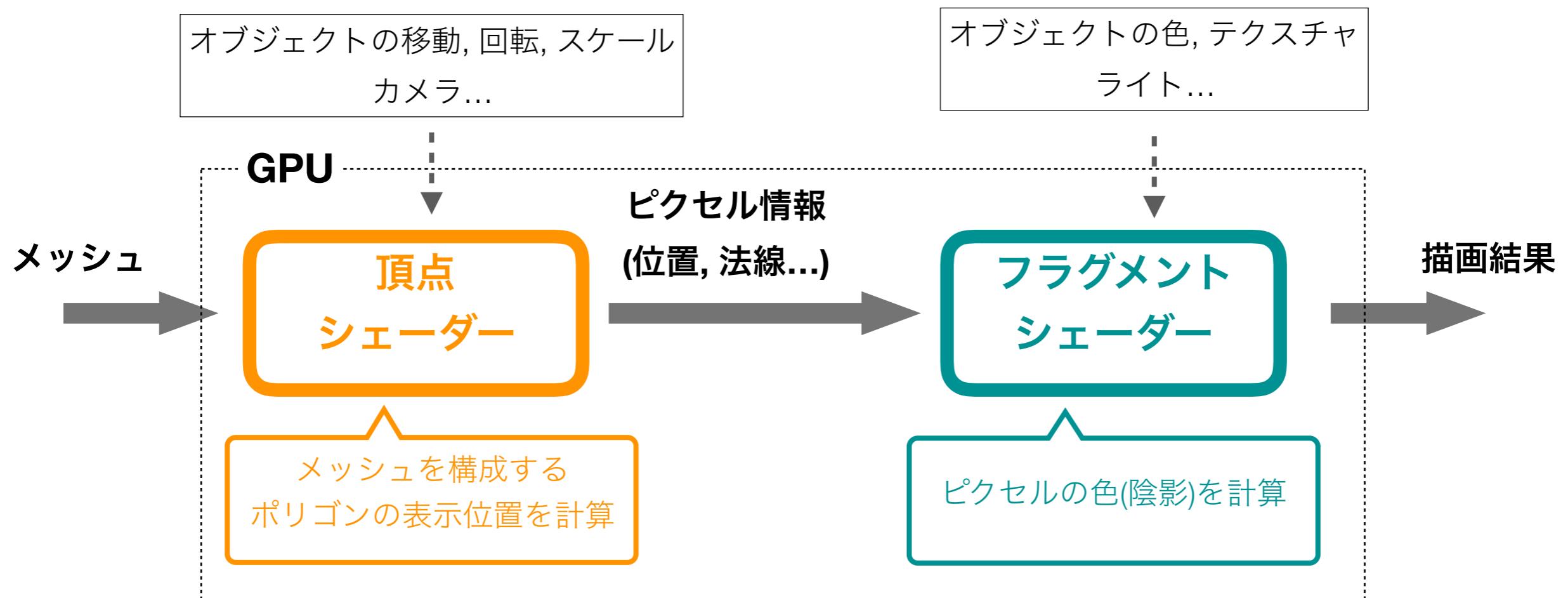
1. 基礎

シェーダーとは

GPUで実行されるプログラム

頂点計算や陰影(ライティング)計算に利用される

3Dレンダリングパイプライン (よくあるリアルタイム3D)

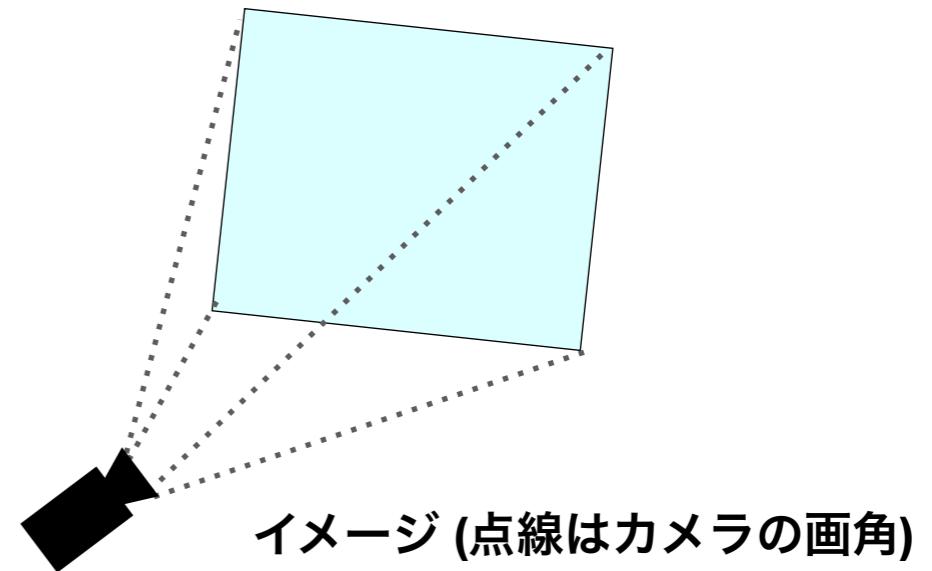


シェーダーアートとは

画面を覆う四角形メッシュにフラグメントシェーダーだけで絵を描く表現

v.s. Processing / p5.js

- Pros
 - 滑らかな表現や複雑な表現が得意
 - グラデーション, 3D
 - リアルタイムに動く表現が得意
- Cons
 - 状態を持てない
 - e.g. 位置と速度を持ったパーティクルみたいな表現はできない
 - 幾何学的な図形表現が得意ではない
 - 直感的ではない



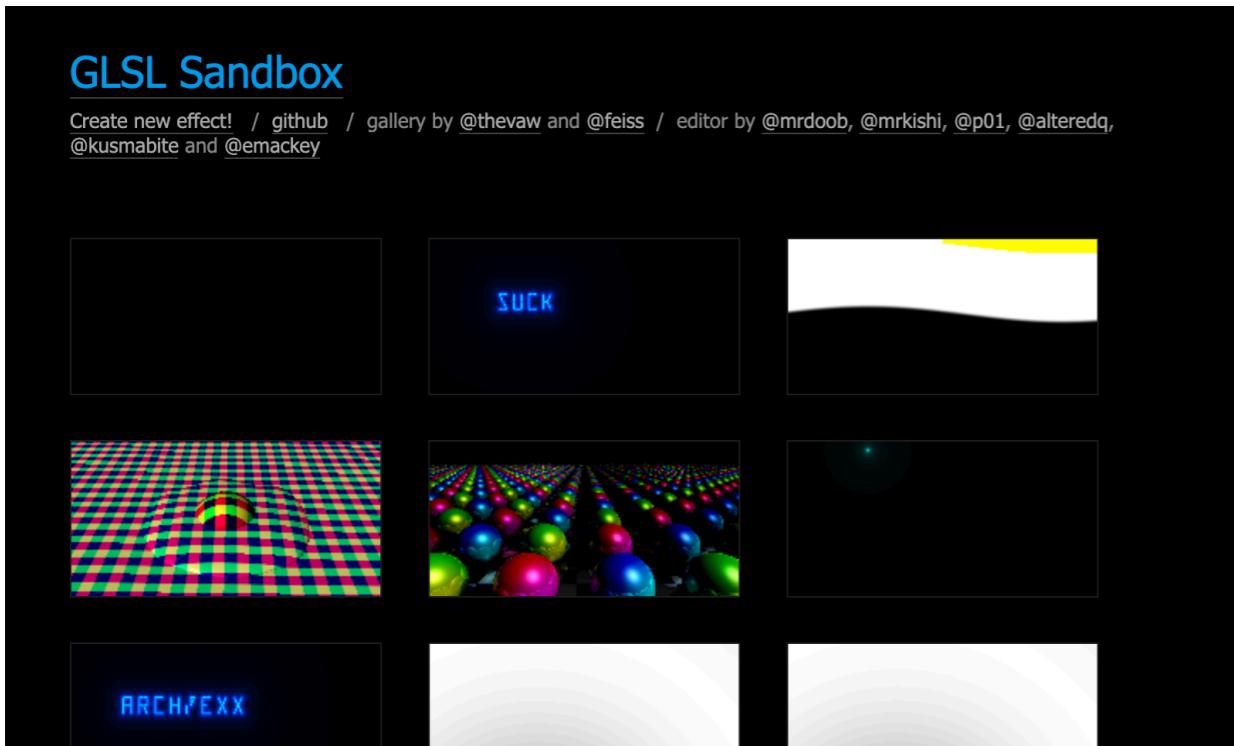
プラットフォーム / ツール

- GLSL Sandbox
 - 気軽にシェーダーを書けるWebサービス
 - <http://glslsandbox.com/>
- Shadertoy
 - シェーダー界隈の猛者が集う場所
 - <https://www.shadertoy.com/>
- NEORT
 - デジタルアートプラットフォーム、GLSL Sandbox互換
 - <https://neort.io/>
- glslfan
 - コーディング画面をリアルタイム共有、GLSL Sandbox互換
 - <https://glslfan.com/>
- Visual Studio Code Shader Toy Extension
 - VS CodeのShadertoy拡張、GLSL Sandbox形式もサポート
 - <https://marketplace.visualstudio.com/items?itemName=stevenson.shader-toy>

GLSL (OpenGL Shading Language)

- OpenGLで使われるシェーダー言語
- C言語をベースにしている
- 制御構文
 - if, for...
- 組み込み型
 - 整数: int
 - 浮動小数点数: float
 - ベクトル: vec2, vec3, vec4
 - 行列: mat2, mat3, mat4
 - テクスチャ: sampler2D
 - ...
- 組み込み関数
 - sin, cos, min, max, step, smoothstep, mix ...

GLSL Sandbox



The screenshot shows the GLSL Sandbox code editor. At the top right are buttons for "hide code", "fullscreen", and "gallery". The status bar indicates "compiled successfully". The code editor displays the following GLSL shader code:

```
1 #ifdef GL_ES
2 precision mediump float;
3 #endif
4
5 #extension GL_OES_standard_derivatives : enable
6
7 uniform float time;
8 uniform vec2 mouse;
9 uniform vec2 resolution;
10
11 void main( void ) {
12
13     vec2 position = ( gl_FragCoord.xy / resolution.xy ) + mouse / 4.0;
14
15     float color = 0.0;
16     color += sin( position.x * cos( time / 15.0 ) * 80.0 ) + cos( position.y * cos( time / 15.0 ) * 10.0 );
17     color += sin( position.y * sin( time / 10.0 ) * 40.0 ) + cos( position.x * sin( time / 25.0 ) * 40.0 );
18     color += sin( position.x * sin( time / 5.0 ) * 10.0 ) + sin( position.y * sin( time / 35.0 ) * 80.0 );
19     color *= sin( time / 10.0 ) * 0.5;
20
21     gl_FragColor = vec4( vec3( color, color * 0.5, sin( color + time / 3.0 ) * 0.75 ), 1.0 );
22 }
23 }
```

<http://glslsandbox.com/> にアクセス

Hello, World!

```
precision highp float;  
  
void main(void) {  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```



precision highp float;

浮動小数点数の精度 (コンパイルに必要)

void main(void) { ... }

main関数 (エントリーポイント)

gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);

ピクセルの色

red green blue alpha

- 色の範囲は 0~1 (Processingと違う!)
- alpha(透明度)は出力に影響しない
 - 1を入れておくことが多い

フラグメントシェーダーでやるべきことは

gl_FragColorに色(vec4)を設定すること

ベクトル

- 複数の値をまとめたもの
 - ProcessingのPVector, p5.jsのp5.Vectorに相当
- **vec2**
 - 2次元座標 (X, Y)
- **vec3**
 - 3次元座標 (X, Y, Z)
 - 色 (R, G, B)
- **vec4**
 - 同次座標 (シェーダーアートの文脈ではあまり使われない)
 - 色 + 透明度 (X, Y, Z, A)

ベクトルの作成・値の取得

```
vec4(1.0, 2.0, 3.0, 4.0);
vec4(vec2(1.0, 2.0), 3.0, 4.0);
vec4(vec3(1.0, 2.0, 3.0), 4.0);
vec4(vec2(1.0, 2.0), vec2(3.0, 4.0));
vec4(1.0, vec2(2.0, 3.0), 4.0);
vec4(1.0, vec3(1.0, 2.0, 3.0));
vec4(vec4(1.0, 2.0, 3.0, 4.0));
```

ベクトルの作成

vec2, vec3でも同様

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);
v.x; // = 1.0
v.y; // = 2.0
v.z; // = 3.0
v.w; // = 4.0
v.xy; // = vec2(1.0, 2.0)
v.xx; // = vec2(1.0, 1.0)
v.yx; // = vec2(2.0, 1.0)
v.xz; // = vec2(1.0, 3.0)
v.xyz; // = vec3(1.0, 2.0, 3.0)
v.xyy; // = vec3(1.0, 2.0, 2.0)
v.r; // 1.0
v.g; // 2.0
v.b; // 3.0
v.a; // 4.0
v.rgb; // vec3(1.0, 2.0, 3.0)
```

Swizzle演算子による値の取得

すべての組み合わせでない

vec2, vec3でも同様

ベクトルの演算

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0)
v + 1.0; // = vec4(2.0, 3.0, 4.0, 5.0)
v - 1.0; // = vec4(0.0, 1.0, 2.0, 3.0)
v * 2.0; // = vec4(2.0, 4.0, 6.0, 8.0)
v / 2.0; // = vec4(0.5, 1.0, 1.5, 2.0)
1.0 + v; // = vec4(2.0, 3.0, 4.0, 5.0)
1.0 - v; // = vec4(0.0, -1.0, -2.0, -3.0)
2.0 * v; // = vec4(2.0, 4.0, 6.0, 8.0)
2.0 / v; // = vec4(2.0, 1.0, 0.666..., 0.5)
```

スカラーとの演算

vec2, vec3でも同様

```
vec4 v1 = vec4(1.0, 2.0, 3.0, 4.0);
vec4 v2 = vec4(5.0, 6.0, 7.0, 8.0);
v1 + v2; // = vec4(6.0, 8.0, 10.0, 12.0)
v1 - v2; // = vec4(-4.0, -4.0, -4.0, -4.0)
v1 * v2; // = vec4(5.0, 12.0, 21.0, 32.0)
v1 / v2; // = vec4(0.2, 0.333..., 0.429..., 0.5)
```

//異なる大きさのベクトル同士では演算できない

```
vec2(1.0, 2.0) + vec3(1.0, 2.0, 3.0); // これはエラーになる
```

ベクトル同士の演算

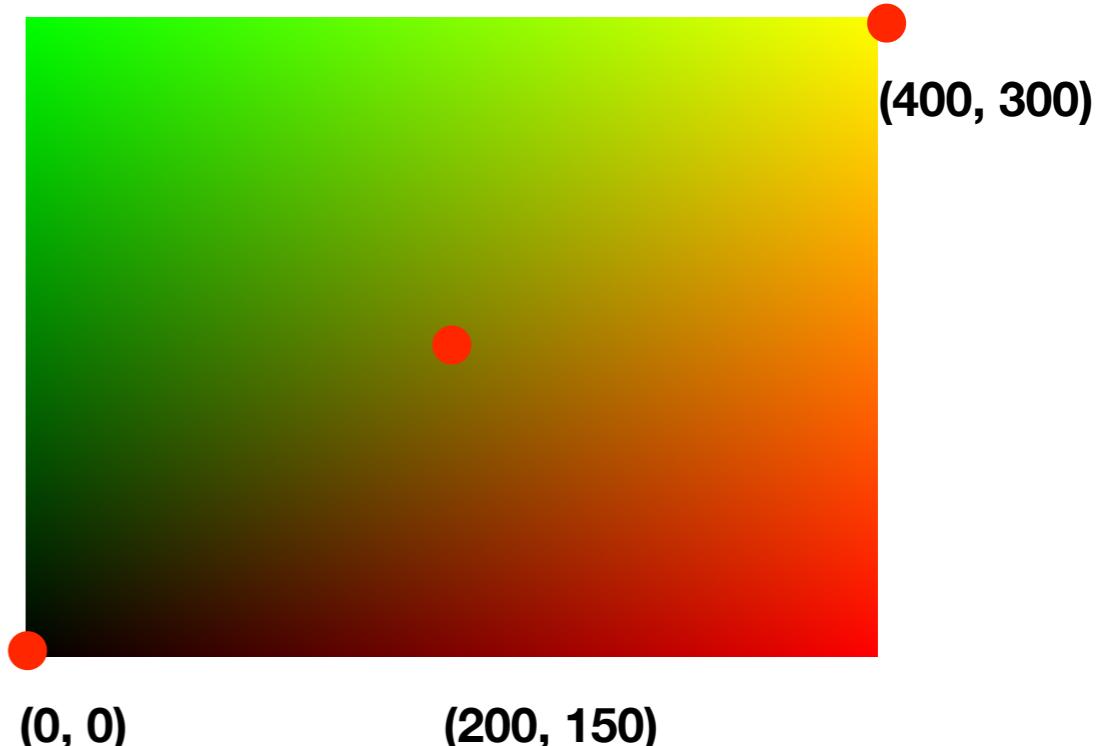
vec2, vec3でも同様

ピクセル位置の利用

```
precision highp float;  
  
uniform vec2 resolution;  
  
void main(void) {  
    vec2 pos = gl_FragCoord.xy / resolution;  
    gl_FragColor = vec4(pos, 0.0, 1.0);  
}
```

resolutionがvec2(400, 300)のとき

gl_FragCoord.xyは…



uniform vec2 resolution; 画面の大きさ

vec2 pos = gl_FragCoord.xy / resolution;

現在のピクセル位置

p5.jsで同じ処理を実装すると…

```
let img;
let resolution;

function setup() {
  createCanvas(640, 480);
  img = createImage(width, height);
  resolution = createVector(width, height);
}

function draw() {
  img.loadPixels();
  for (let w = 0; w < width; w++) {
    for (let h = 0; h < height; h++) {
      const coord = createVector(w, h);
      const c = getPixel(coord, resolution);
      img.set(w, h, c);
    }
  }
  img.updatePixels();
  image(img, 0, 0);
}

function getPixel(coord, resolution) {
  return color(
    255 * coord.x / resolution.x,
    255 * coord.y / resolution.y,
    0.0
  );
}
```

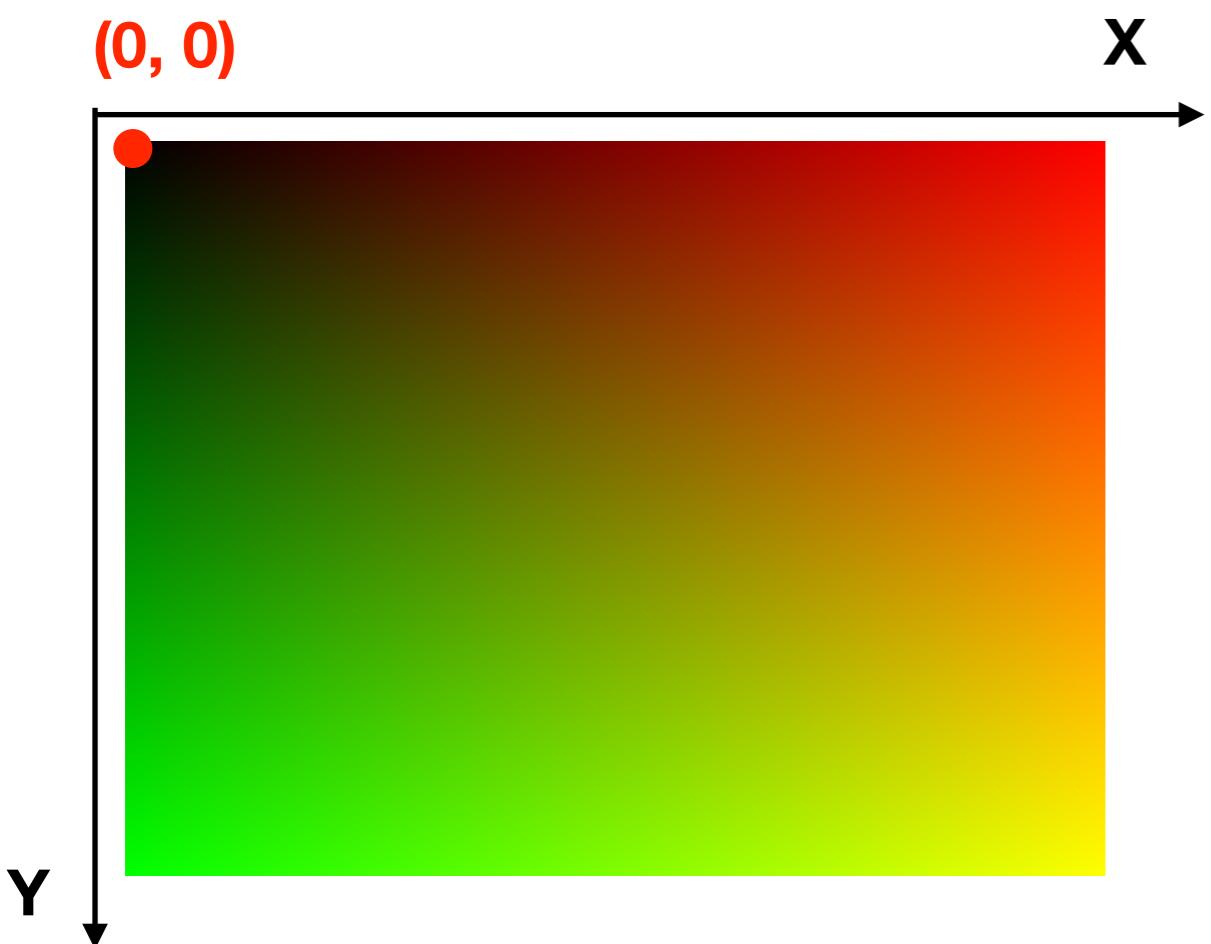
シェーダーでは2重ループ部分が
GPUで並列に処理されるので早い

シェーダーアートでは
この部分だけを書いている

座標の原点

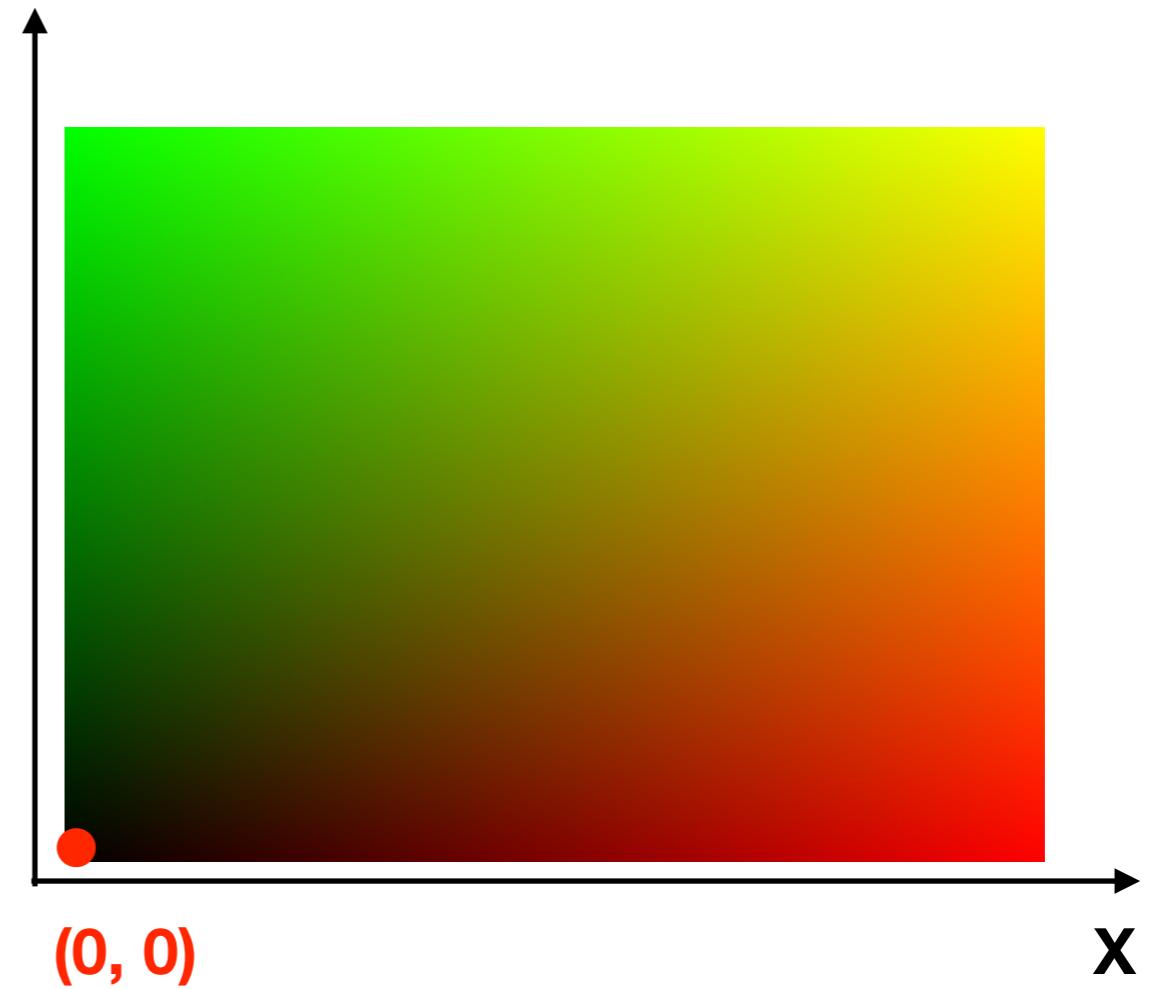
Processing / p5.js

左上が原点



GLSL

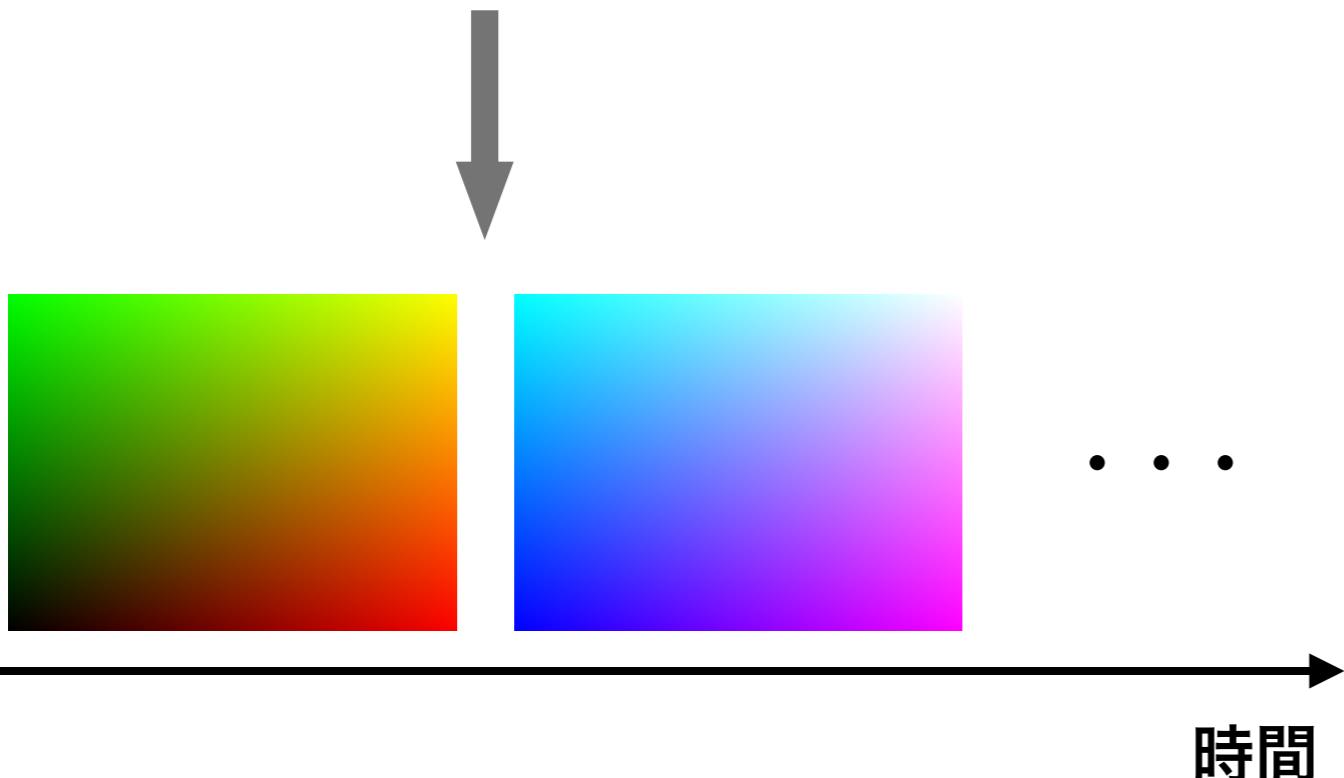
左下が原点



座標の原点が違うので注意！

時間の利用

```
precision highp float;  
  
uniform vec2 resolution;  
uniform float time;  
  
void main(void) {  
    vec2 pos = gl_FragCoord.xy / resolution;  
    gl_FragColor = vec4(pos, sin(time) * 0.5 + 0.5, 1.0);  
}
```



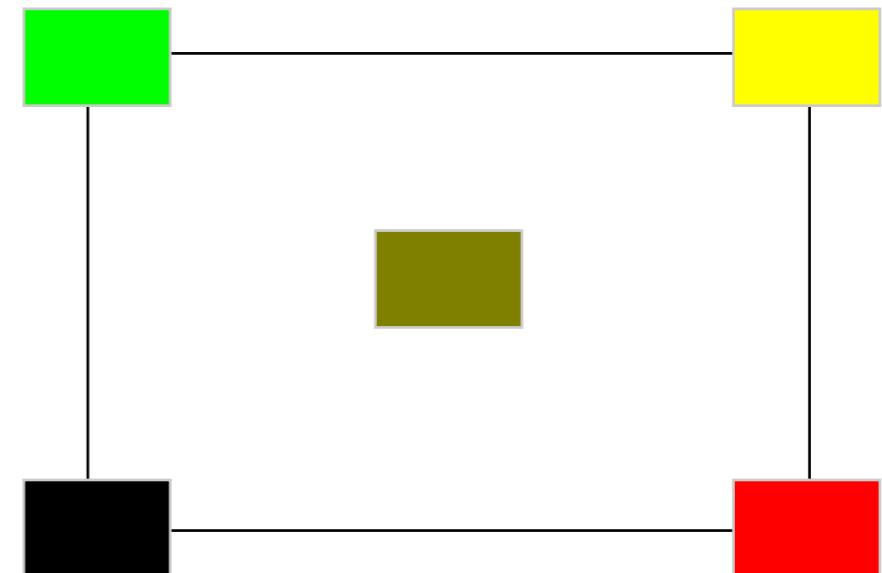
マウス位置の利用

```
precision highp float;  
  
uniform vec2 mouse;  
  
void main(void) {  
    gl_FragColor = vec4(mouse, 0.0, 1.0);  
}
```

uniform vec2 mouse; マウスの位置 (0~1)

使いどころ

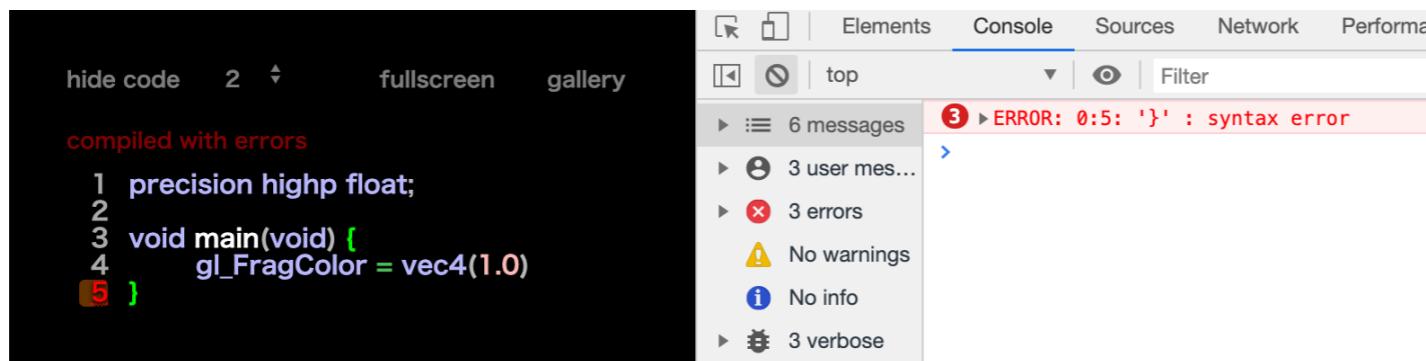
- ・ インタラクション
- ・ パラメータ確認



デバッグ

- ・ コンパイルエラーの見方
 - ・ 開発者ツールのコンソールで確認
 - ・ Chromeの場合
 - ・ メニュー / その他のツール / デベロッパー ツール / Console タブ

e.g. セミコロンがない場合...



- ・ デバッグ方法
 - ・ デバッガーや値を文字列で表示する方法(printデバッグ)はない
 - ・ gl_FragColor に調べたい値を入れて色として確認する

注意点

- ・ 「戻る」ボタンを押さない
 - ・ 押すと書きかけのコードが消える
- ・ 無限ループを作らない
 - ・ タブがクラッシュして書きかけのコードが消えたり、PCが不安定になる
 - ・ コーディング中の途中状態が自動コンパイルされて意図せず無限ループになる可能性がある

GLSLのビルトイン関数とProcessing / p5.jsの対応関係

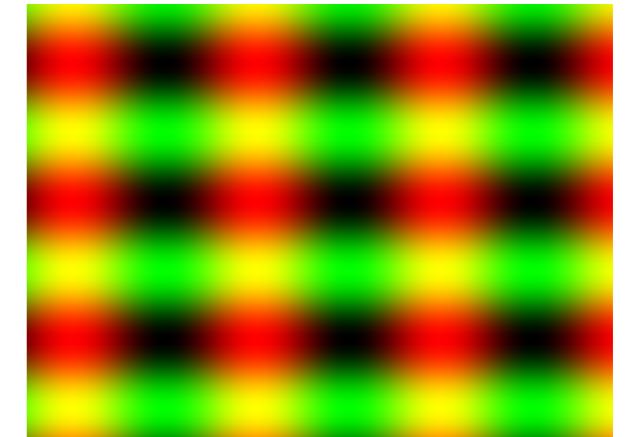
よく使う関数のみを抜粋

GLSL	Processing / p5.js	補足
sin, cos	sin, cos	
pow	pow	
exp	exp	
log	log	
sqrt	sqrt	
abs	abs	
floor	floor	
mod	—	余りを求める関数
mix	lerp	
step	—	この後に解説
smoothstep	—	この後に解説
length	mag	ベクトルの大きさ

引数の順番や型が異なることがあるので注意

GLSLの関数

```
void main(void) {
    vec2 pos = gl_FragCoord.xy / resolution;
    gl_FragColor = vec4(sin(20.0 * pos) * 0.5 + 0.5, 0.0, 1.0);
}
```



```
void main(void) {
    vec2 pos = gl_FragCoord.xy / resolution;
    vec3 col = mix(vec3(1.0, 0.0, 0.0), vec3(0.0, 0.0, 1.0), pos.x);
    gl_FragColor = vec4(col, 1.0);
}
```



GLSLのほとんどの関数はベクトルも引数に受け取れる
ベクトルの各要素ごとに値が計算される

/src/ch1-basic/sin.glsl
/src/ch1-basic/mix.glsl

step, smoothstep

step(edge, x)

xがedge未満のとき0、それ以外のとき1を返す

```
void main(void) {
    vec2 pos = gl_FragCoord.xy / resolution;
    vec3 col = vec3(step(0.5, pos.x));
    gl_FragColor = vec4(col, 1.0);
}
```



smoothstep(edge0, edge1, x)

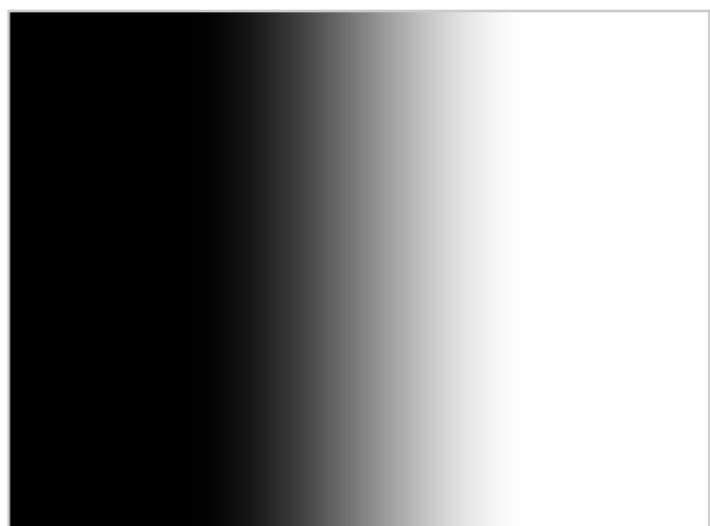
xがedge0以下のとき0、edge1以上のとき1、

それ以外のとき0から1の間の滑らかに補間した値を返す

```
precision highp float;

uniform vec2 resolution;

void main(void) {
    vec2 pos = gl_FragCoord.xy / resolution;
    vec3 col = vec3(smoothstep(0.25, 0.75, pos.x));
    gl_FragColor = vec4(col, 1.0);
}
```

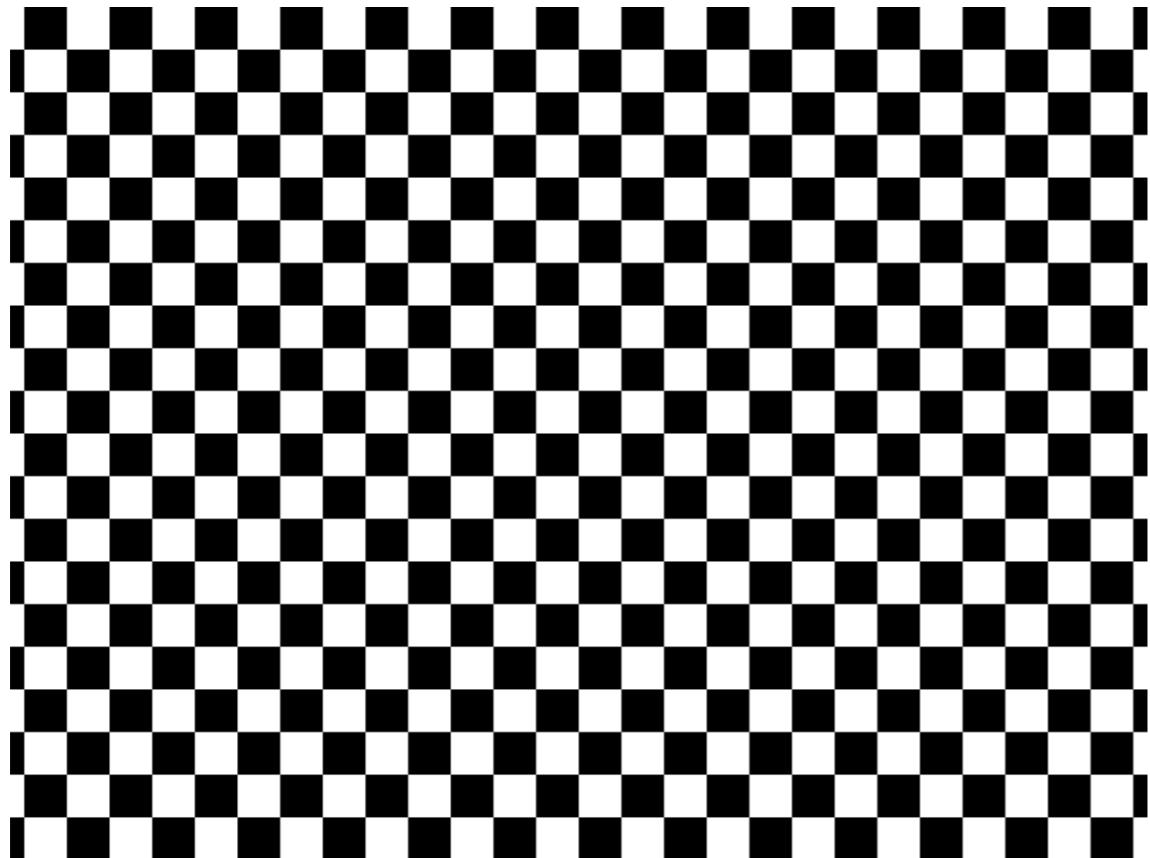


/src/ch1-basic/step.glsl

/src/ch1-basic/smoothstep.glsl

作例 (基礎編)

市松模様 (チェッカー)



リング



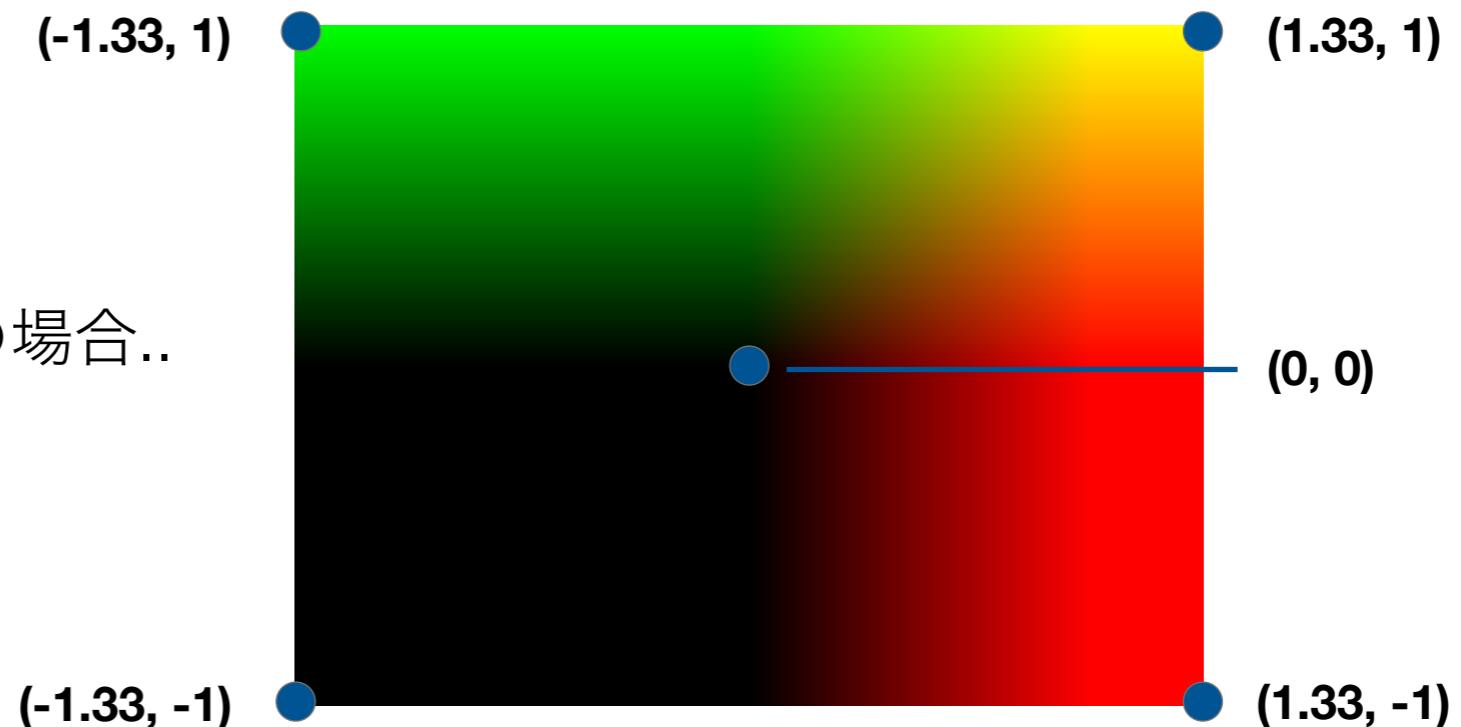
/src/ch1-basic/checker/step-4.glsl
/src/ch1-basic/ring/step-4.glsl

市松模様 1.

```
vec3 checker(vec2 pos, vec2 size) {
    return vec3(pos, 0.0);
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    vec3 c = checker(pos, vec2(1.0));
    gl_FragColor = vec4(c, 1.0);
}
```

resolutionがvec2(400, 300)の場合..



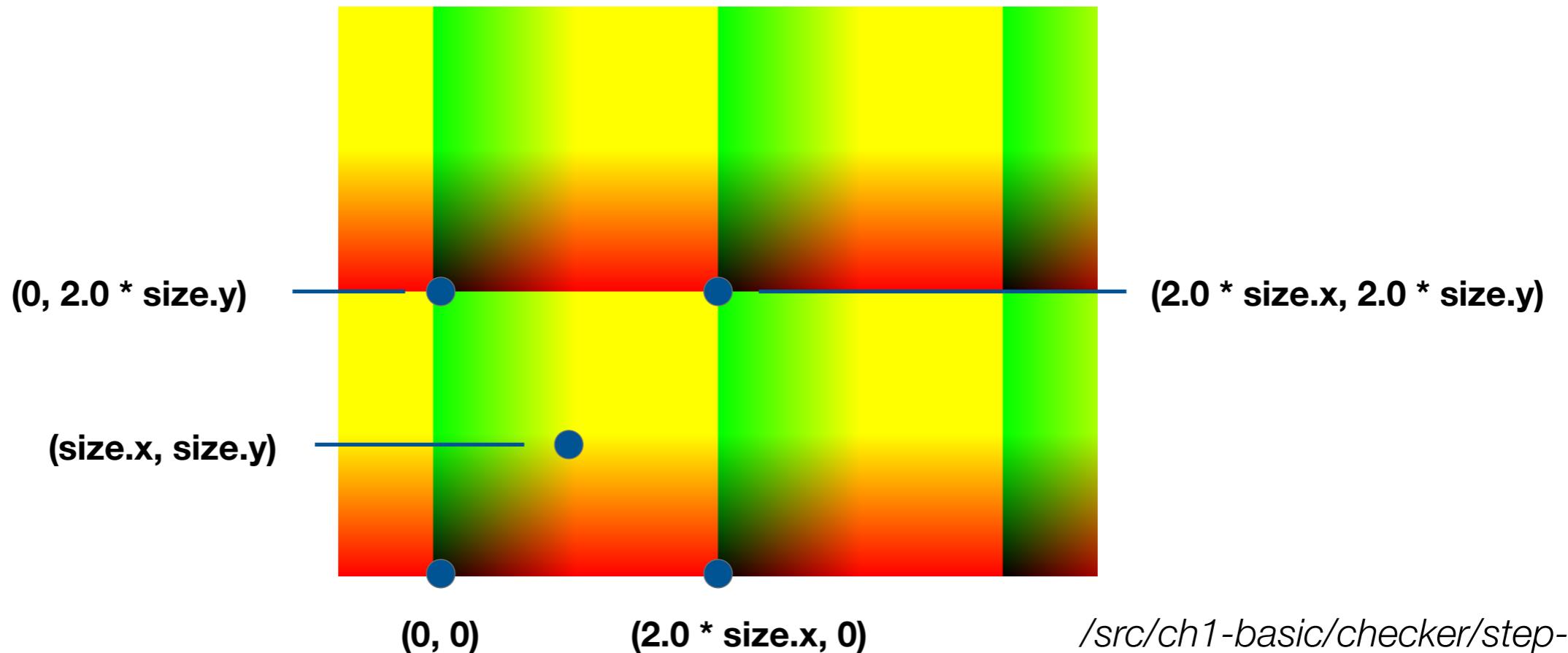
アスペクト比を維持したまま原点を中心にして座標を正規化

市松模様 2.

```
vec3 checker(vec2 pos, vec2 size) {
    vec2 m = mod(pos, 2.0 * size);
    return vec3(m, 0.0);
}

void main(void) {
    ...
    vec3 c = checker(10.0 * pos, vec2(1.0));
    ...
}
```

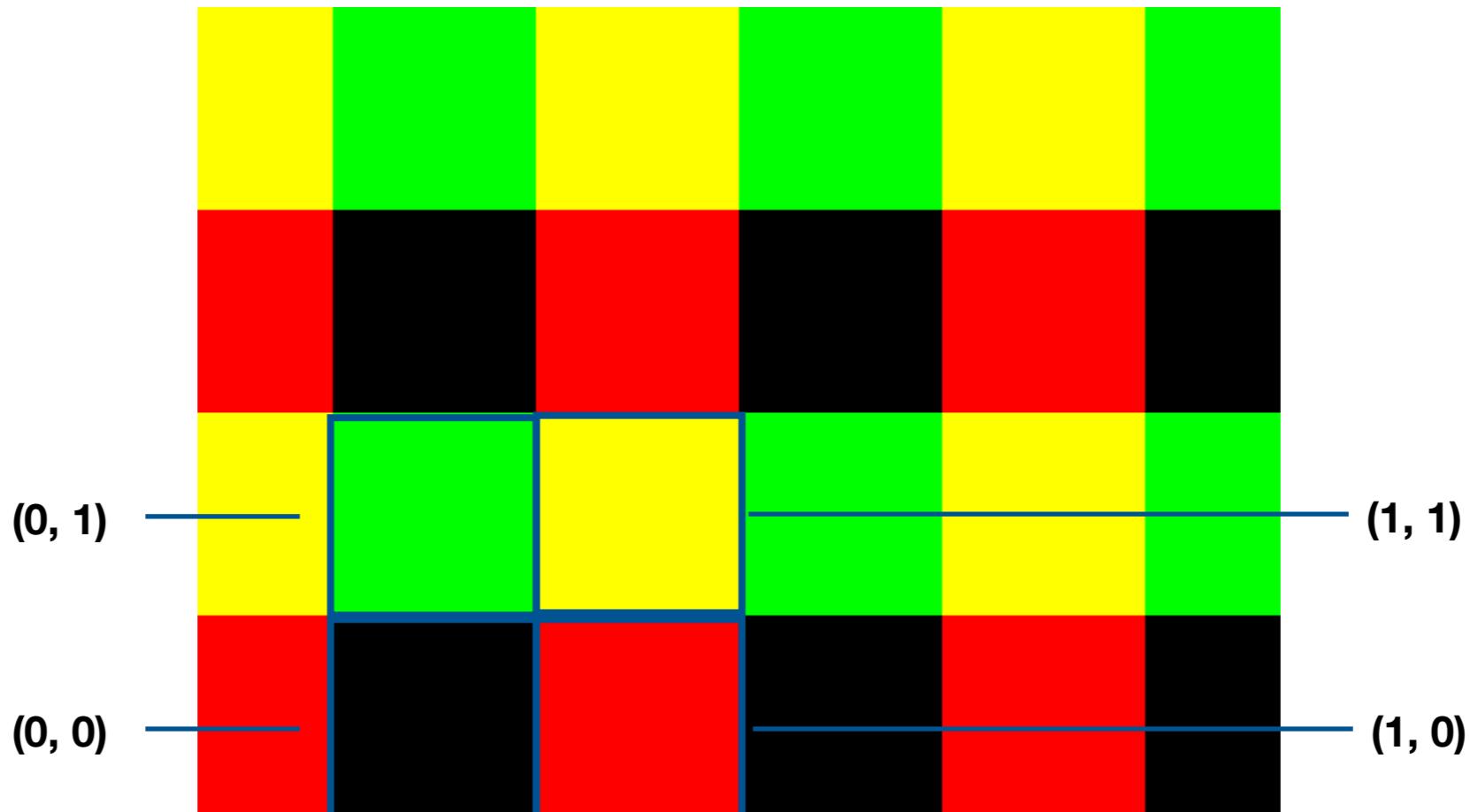
modで座標の繰り返しを作る



市松模様 3.

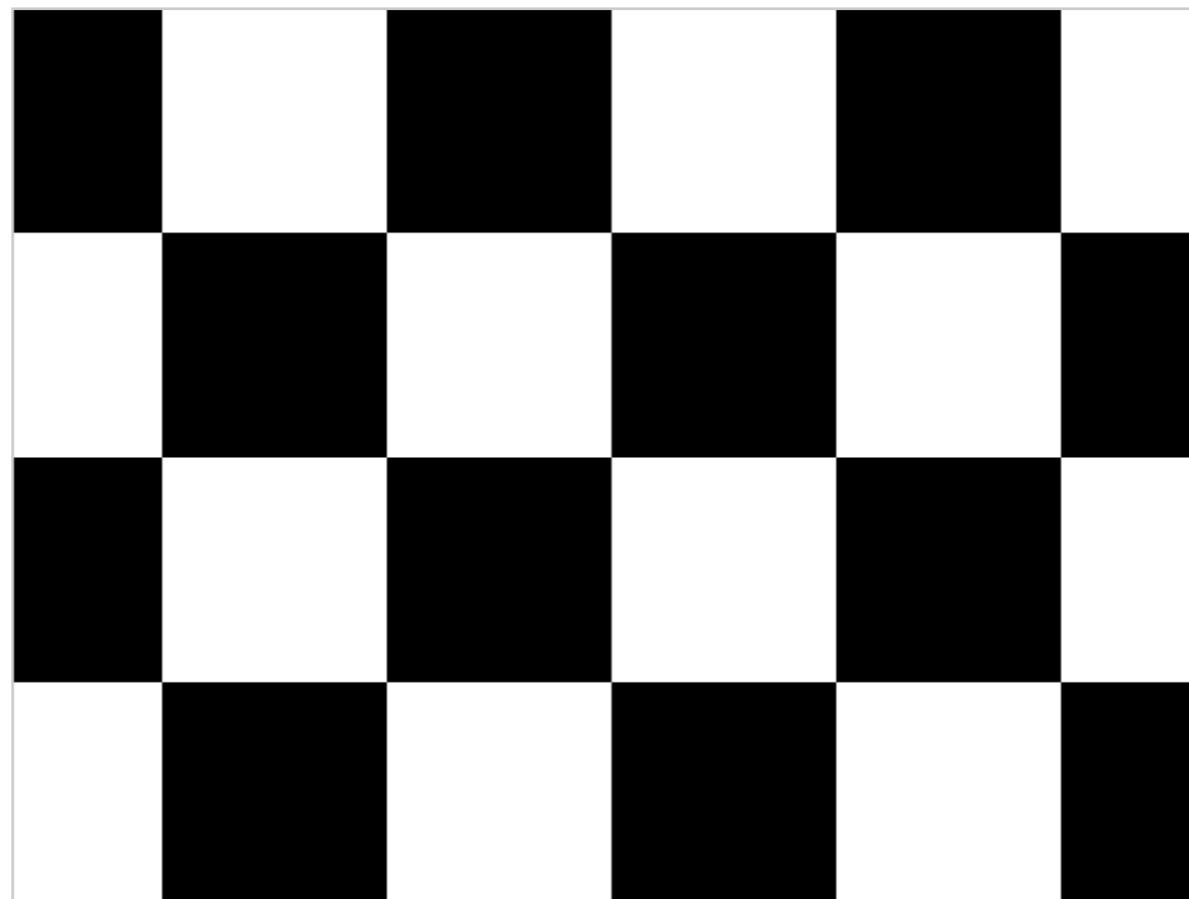
```
vec3 checker(vec2 pos, vec2 size) {
    vec2 m = mod(pos, 2.0 * size);
    vec2 s = step(size, m);
    return vec3(s, 0.0);
}
```

stepで同じ格子のピクセルは
同じ値になるようにする



市松模様 — 4. 完成 —

```
vec3 checker(vec2 pos, vec2 size) {
    vec2 m = mod(pos, 2.0 * size);
    vec2 s = step(size, m);
    return s.x == s.y ? vec3(0.0) : vec3(1.0);
}
```



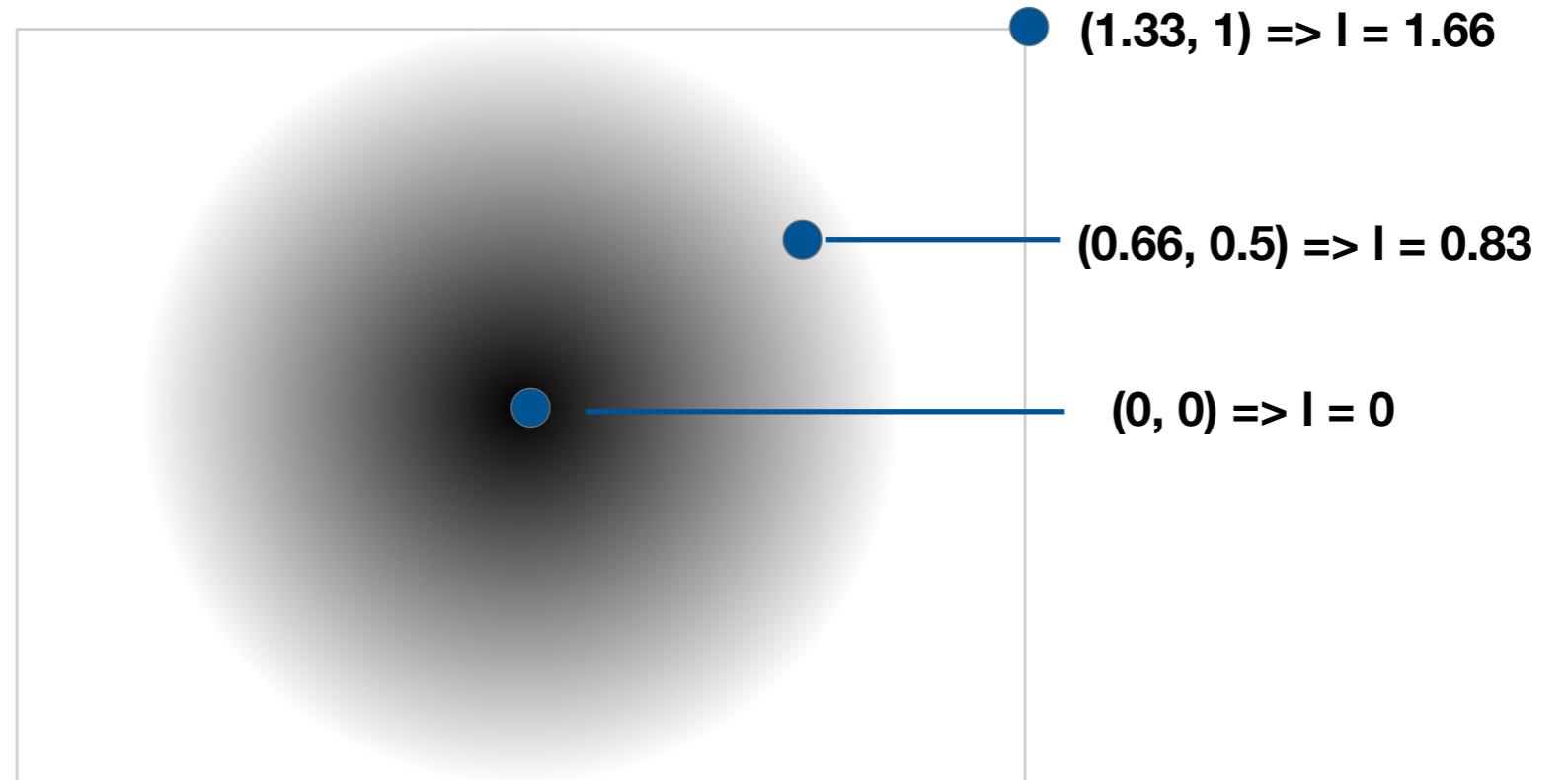
リング 1.

```
vec3 ring(vec2 pos) {
    float l = length(pos);
    return vec3(l);
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    vec3 col = ring(pos);
    gl_FragColor = vec4(col, 1.0);
}
```

resolutionがvec2(400, 300)の場合..

中心からの距離を求める



リング 2.

```
vec3 ring(vec2 pos) {
    float l = length(20.0 * pos);
    float s = sin(l - time) * 0.5 + 0.5;
    return vec3(s);
}
```

三角関数で周期パターンを作る

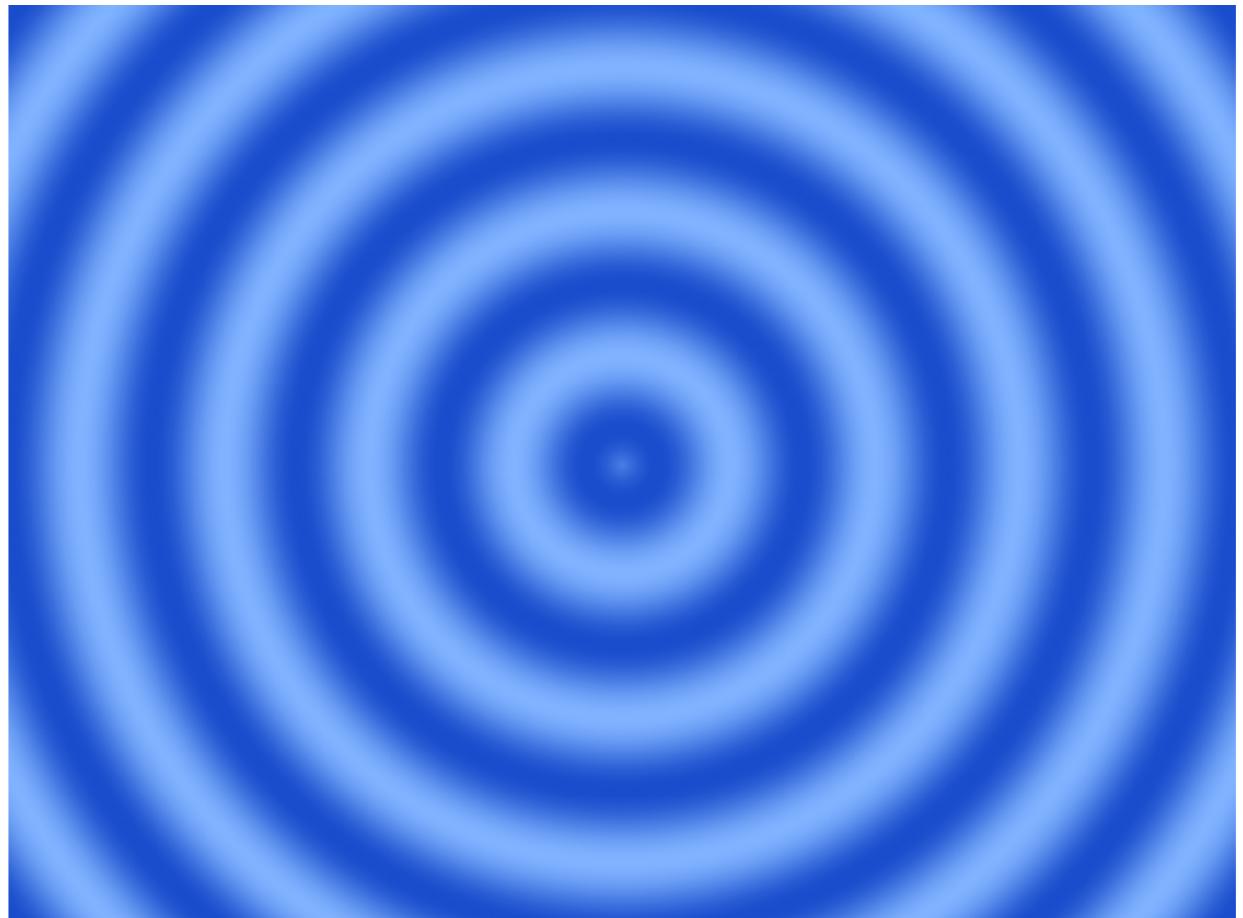


リング — 3. 色の線形補間 —

```
vec3 c1 = vec3(0.5, 0.7, 1.0);
vec3 c2 = vec3(0.1, 0.3, 0.8);

vec3 ring(vec2 pos) {
    float l = length(20.0 * pos);
    float s = sin(l - time) * 0.5 + 0.5;
    return mix(c1, c2, s);
}
```

mixで2つの色の
グラデーションを作る



リング — 4. 完成 —

```
vec3 c1 = vec3(0.5, 0.7, 1.0);
vec3 c2 = vec3(0.1, 0.3, 0.8);

vec3 ring(vec2 pos) {
    float l = length(20.0 * pos);
    float s = sin(l - time) * 0.5 + 0.5;
    return mix(c1, c2, smoothstep(0.8, 0.9, s));
}
```

smoothstepでイイ感じにする



2. ランダム

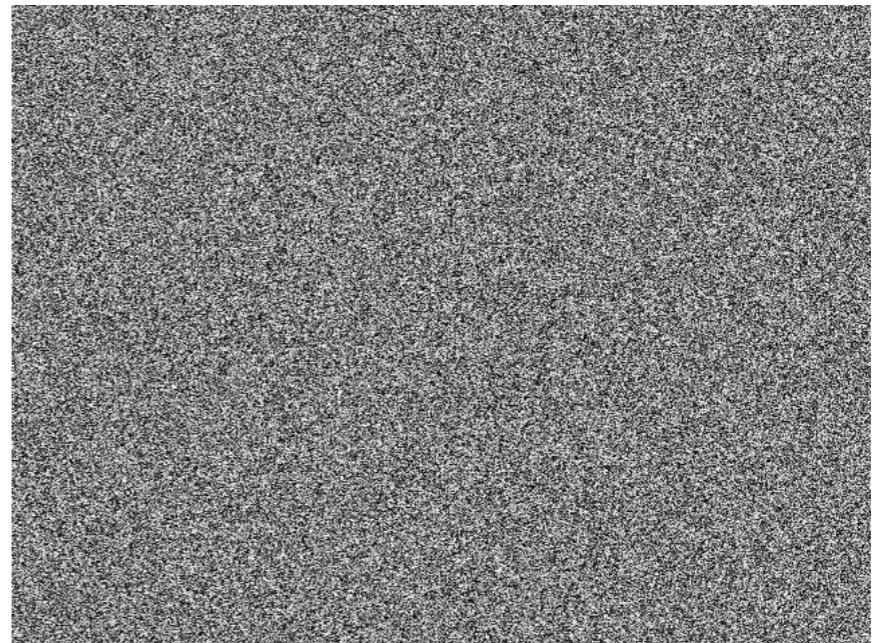
乱数・ノイズ

- Processing / p5.jsでいうところのrandom, noise
- GLSLには乱数を生成する組み込み関数はない
- なので、自分で実装する必要がある

一様乱数(ホワイトノイズ)

1次元引数の乱数

```
float random(float v) {
    return fract(sin(v * 12.9898) * 43758.5453);
}
```



2次元引数の乱数

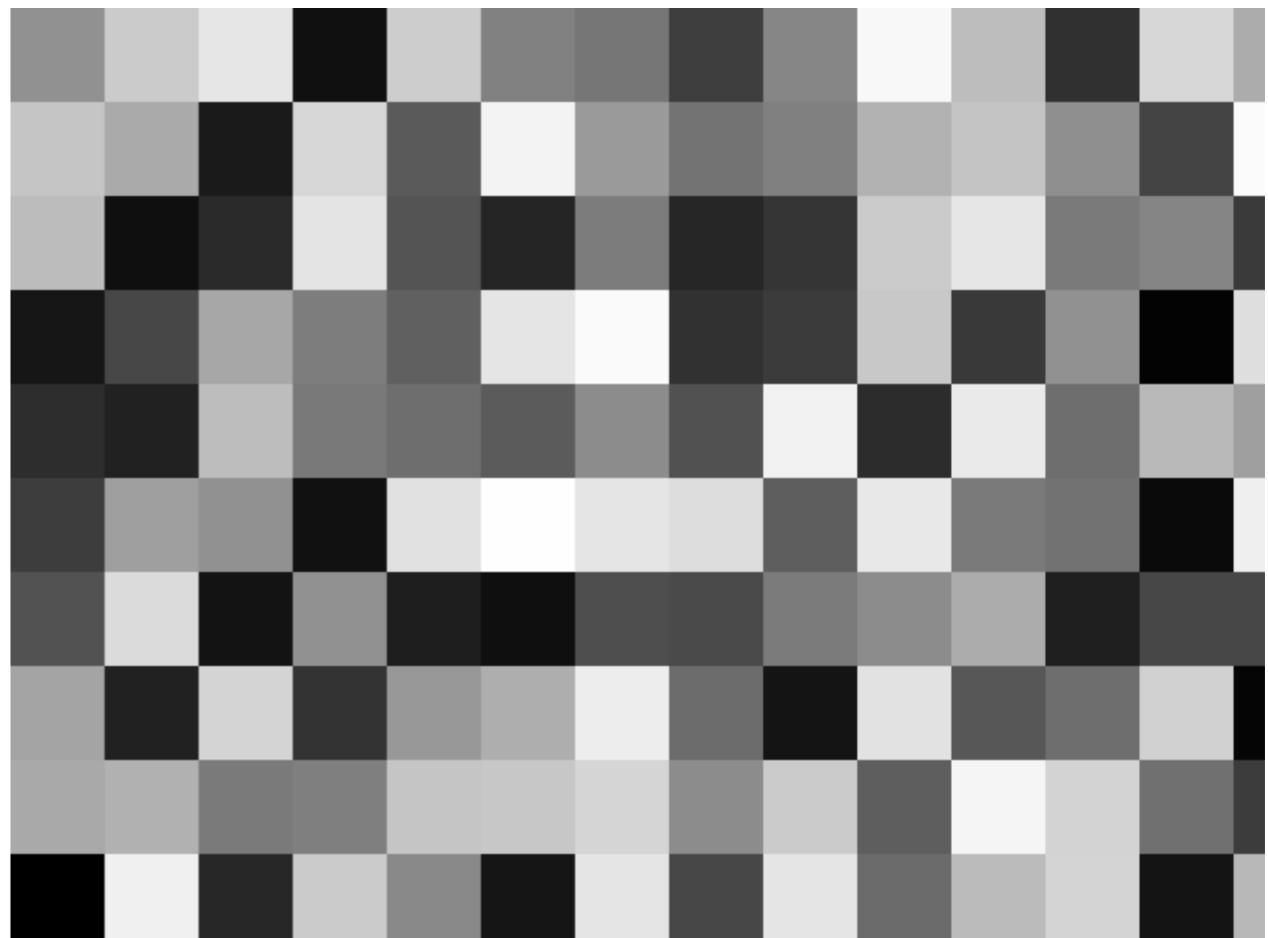
```
float random(vec2 v) {
    return fract(sin(dot(v, vec2(12.9898, 78.233))) * 43758.5453);
}
```

*/src/ch2-random/random-1d.glsL
/src/ch2-random/random-2d.glsL
/src/ch2-random/random-3d.glsL*

ブロックノイズ

```
float blockNoise(vec2 v) {  
    return random(floor(v));  
}
```

randomに渡す引数を
複数のピクセルで同じにすることで
ブロックを作る

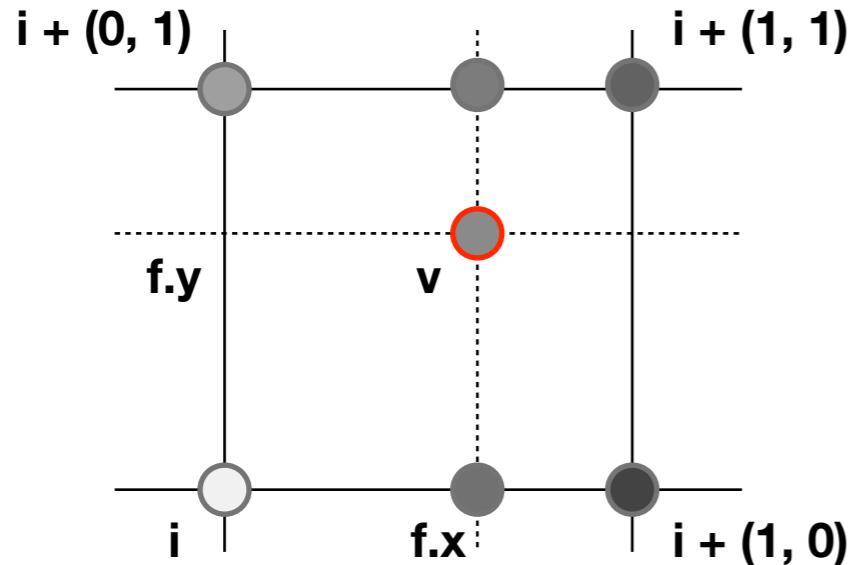


*/src/ch2-random/block-noise-1d.glsl
/src/ch2-random/block-noise-2d.glsl
/src/ch2-random/block-noise-3d.glsl*

バリューノイズ

ブロックノイズを補間して滑らかにする

```
float valueNoise(vec2 v) {
    vec2 i = floor(v);
    vec2 f = smoothstep(0.0, 1.0, fract(v));
    return mix(
        mix(random(i), random(i + vec2(1.0, 0.0)), f.x),
        mix(random(i + vec2(0.0, 1.0)), random(i + vec2(1.0, 1.0)), f.x),
        f.y
    );
}
```



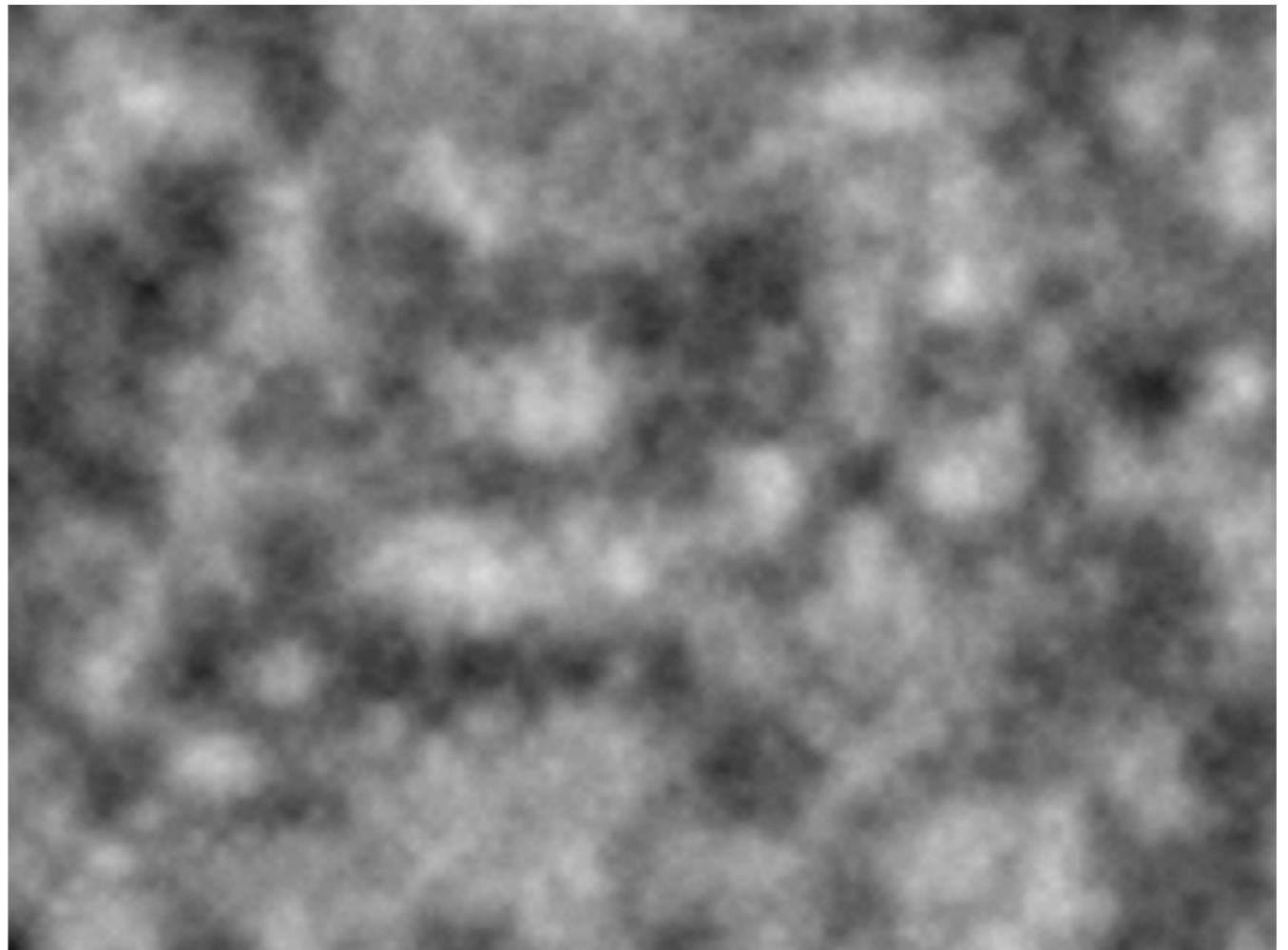
/src/ch2-random/value-noise-1d.glsl
/src/ch2-random/value-noise-2d.glsl
/src/ch2-random/value-noise-3d.glsl

連續だがブロック状の構造がうっすら見えている...

fbm (Fractal Brown Motion)

複数のノイズをスケールを変えながら重ねる

```
float fbm(vec2 x) {
    float n = 0.0;
    float a = 0.5;
    for (int i = 0; i < 5; i++) {
        n += a * valueNoise(x);
        x *= 2.0;
        a *= 0.5;
    }
    return n;
}
```



ブロック感が消えて自然なノイズが得られる

/src/ch2-random/fbm-1d.glsl
/src/ch2-random/fbm-2d.glsl
/src/ch2-random/fbm-3d.glsl

パーリンノイズ



Processingのノイズはパーリンノイズ
シェーダーアートでは余り使われない
(計算コストの問題??)

```
float perlinNoise(vec2 v) {
    vec2 i = floor(v);
    vec2 f = fract(v);

    vec2 v00 = f;
    vec2 v10 = f - vec2(1.0, 0.0);
    vec2 v01 = f - vec2(0.0, 1.0);
    vec2 v11 = f - vec2(1.0, 1.0);

    vec2 i00 = i;
    vec2 i10 = i + vec2(1.0, 0.0);
    vec2 i01 = i + vec2(0.0, 1.0);
    vec2 i11 = i + vec2(1.0, 1.0);

    vec2 g00 = normalize(random2(i00) * 2.0 - 1.0);
    vec2 g10 = normalize(random2(i10) * 2.0 - 1.0);
    vec2 g01 = normalize(random2(i01) * 2.0 - 1.0);
    vec2 g11 = normalize(random2(i11) * 2.0 - 1.0);

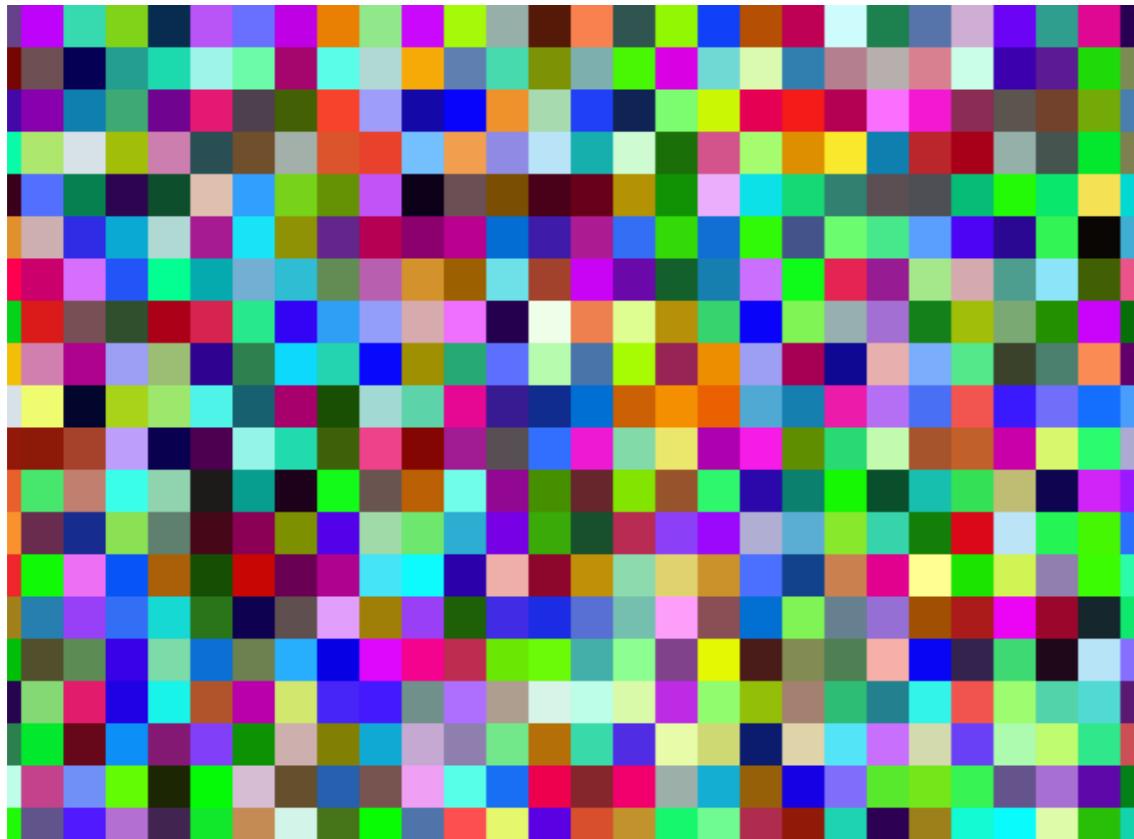
    float d00 = dot(v00, g00);
    float d10 = dot(v10, g10);
    float d01 = dot(v01, g01);
    float d11 = dot(v11, g11);

    vec2 u = smoothstep(0.0, 1.0, f);
    return mix(
        mix(d00, d10, u.x),
        mix(d01, d11, u.x),
        u.y
    );
}
```

/src/ch2-random/perlin-noise-2d.glsl
/src/ch2-random/perlin-noise-3d.glsl

作例 (ランダム編)

カラーブロック



RGBシフト



*/src/ch2-random/color-blocks/step-4.glsl
/src/ch2-random/rgb-shift/step-3.glsl*

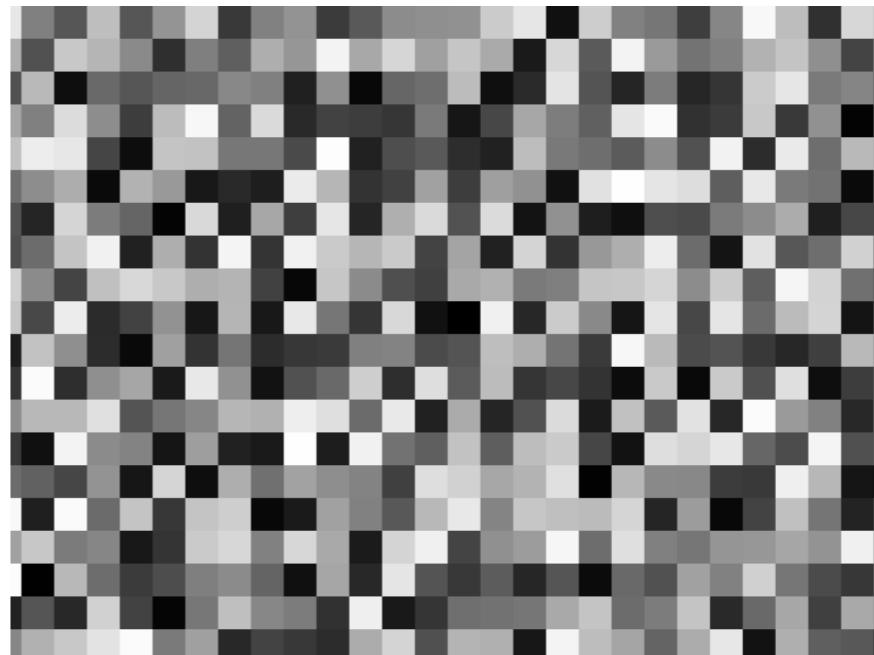
カラーブロック 1.

```
float random(vec2 v) {
    return fract(sin(dot(v, vec2(12.9898, 78.233))) * 43758.5453);
}

vec3 colorBlocks(vec2 pos) {
    vec2 posIdx = floor(10.0 * pos);
    return vec3(random(posIdx));
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    vec3 col = colorBlocks(pos);
    gl_FragColor = vec4(col, 1.0);
}
```

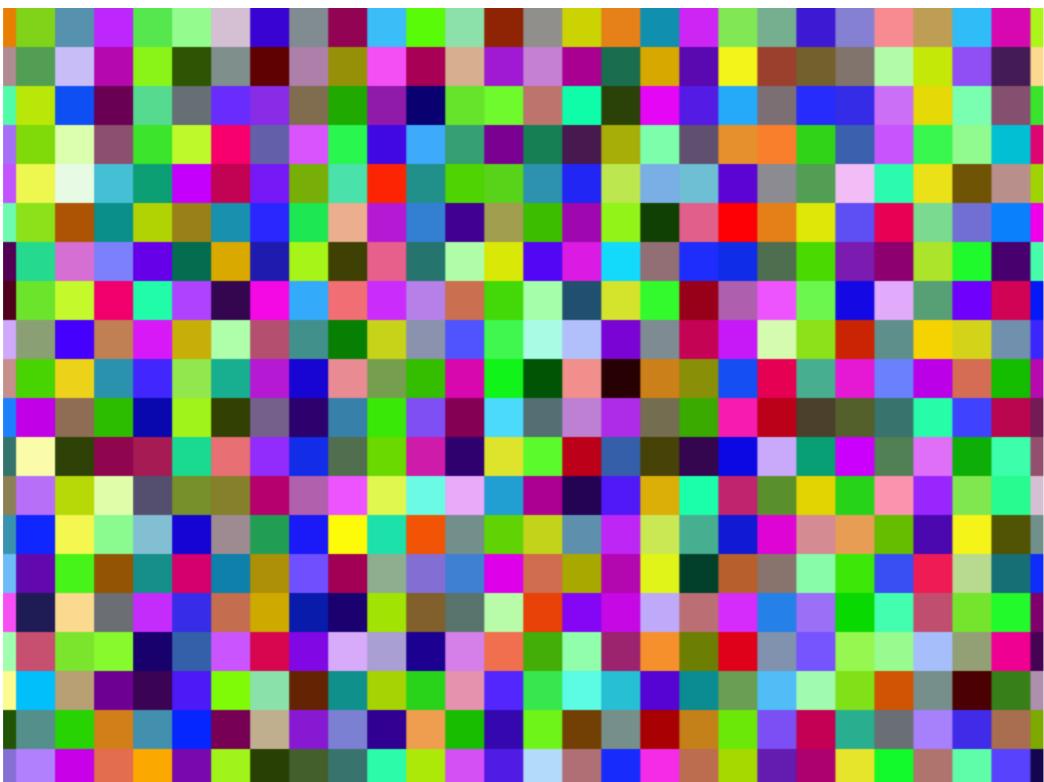
ここまでではブロックノイズと同じ



カラーブロック 2.

```
vec3 random3(vec2 v) {
    return vec3(random(v), random(v + 10000.0), random(v + 20000.0));
}

vec3 colorBlocks(vec2 pos) {
    vec2 posIdx = floor(10.0 * pos);
    return random3(posIdx);
}
```



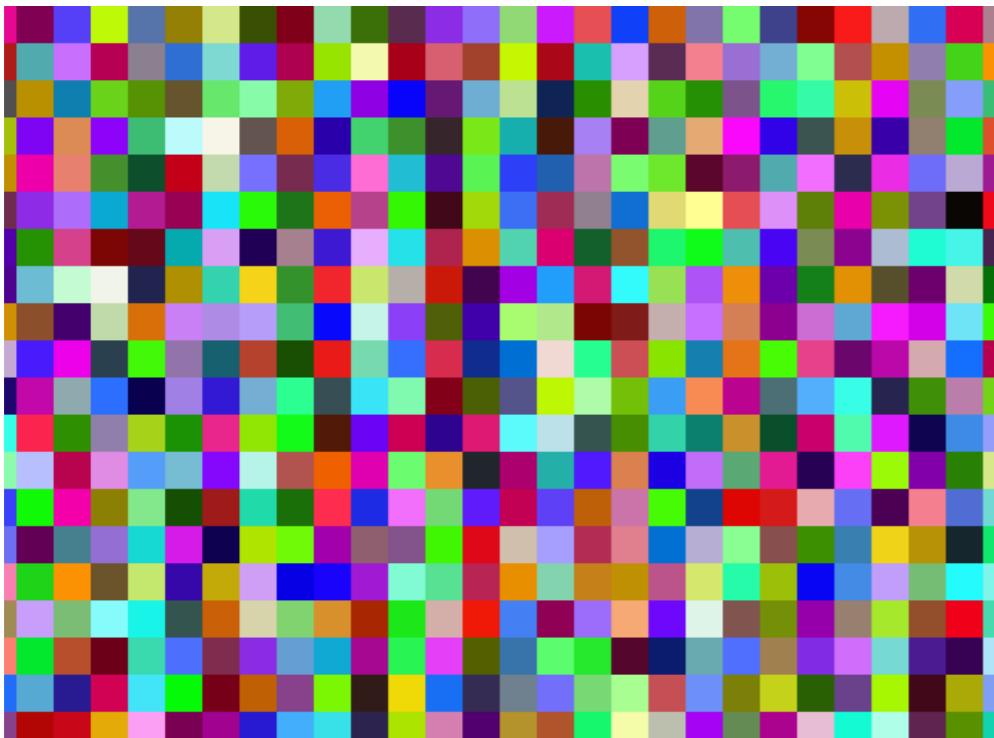
RGBの各要素にランダムな値を設定

カラーブロック 3.

```
float random(vec3 v) {
    return fract(sin(dot(v, vec3(12.9898, 78.233, 19.8321))) * 43758.5453);
}

vec3 random3(vec3 v) {
    return vec3(random(v), random(v + 10000.0), random(v + 20000.0));
}

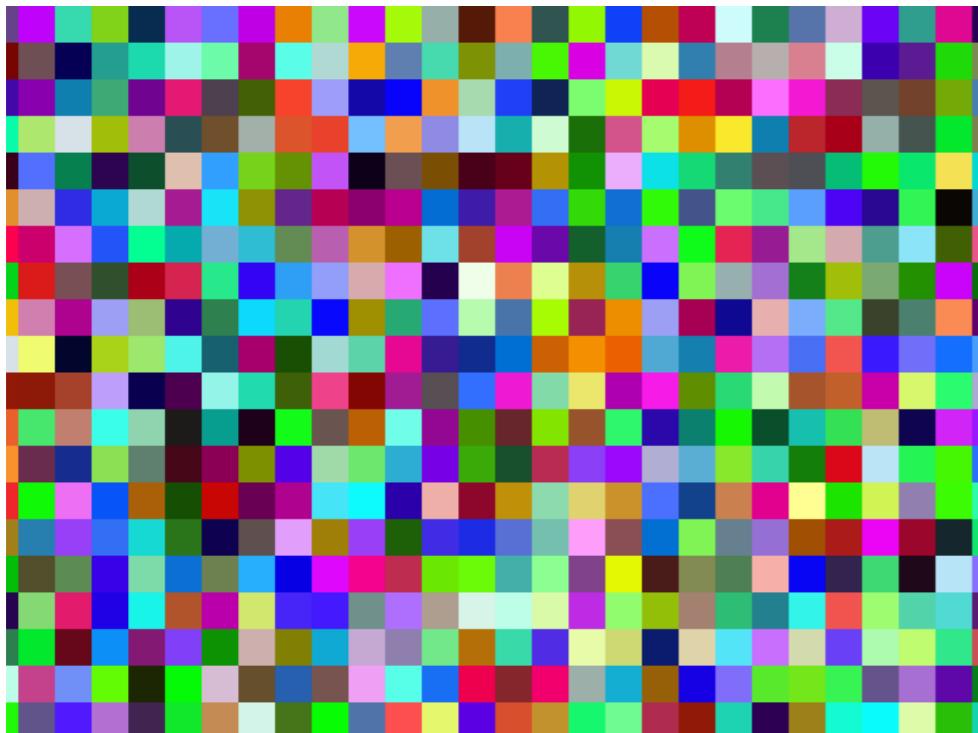
vec3 colorBlocks(vec2 pos) {
    vec2 posIdx = floor(10.0 * pos);
    float timeIdx = floor(10.0 * time);
    return random3(vec3(posIdx, timeIdx));
}
```



randomに時間も渡すこと
一定時間ごとに色が変わるようにする

カラーブロック 4. 完成

```
vec3 colorBlocks(vec2 pos) {
    vec2 posIdx = floor(10.0 * pos);
    float timeIdx = floor(10.0 * (time + random(posIdx)));
    return random3(vec3(posIdx, timeIdx));
}
```



timeIdxの値をブロックごとに変えることで
色が変わるタイミングをずらす

RGBシフト 1.



smoothstepでイイ感じにする

```
float random(vec2 v) {
    return fract(sin(dot(v, vec2(12.9898, 78.233))) * 43758.5453);
}

float valueNoise(vec2 v) {
    vec2 i = floor(v);
    vec2 f = smoothstep(0.0, 1.0, fract(v));
    return mix(
        mix(random(i), random(i + vec2(1.0, 0.0)), f.x),
        mix(random(i + vec2(0.0, 1.0)), random(i + vec2(1.0, 1.0)), f.y)
    );
}

float fbm(vec2 x) {
    float n = 0.0;
    float a = 0.5;
    for (int i = 0; i < 3; i++) {
        n += a * valueNoise(x);
        x *= 2.0;
        a *= 0.5;
    }
    return n;
}

vec3 rgbShift(vec2 pos) {
    float n = fbm(2.0 * pos);
    return vec3(smoothstep(0.5, 0.55, n));
}

void main(void) {
    vec2 pos = gl_FragCoord.xy / min(resolution.x, resolution.y);
    vec3 col = rgbShift(pos);
    gl_FragColor = vec4(col, 1.0);
}
```

RGBシフト 2.



ノイズを3次元にして
時間で動かす

```
float random(vec3 v) {
    return fract(sin(dot(v, vec3(12.9898, 78.233, 19.8321))) * 43758.5453);
}

float valueNoise(vec3 v) {
    vec3 i = floor(v);
    vec3 f = smoothstep(0.0, 1.0, fract(v));
    return mix(
        mix(
            mix(random(i), random(i + vec3(1.0, 0.0, 0.0)), f.x),
            mix(random(i + vec3(0.0, 1.0, 0.0)), random(i + vec3(1.0, 1.0, 0.0)), f.x),
            f.y
        ),
        mix(
            mix(random(i + vec3(0.0, 0.0, 1.0)), random(i + vec3(1.0, 0.0, 1.0)), f.x),
            mix(random(i + vec3(0.0, 1.0, 1.0)), random(i + vec3(1.0, 1.0, 1.0)), f.x),
            f.y
        ),
        f.z
    );
}

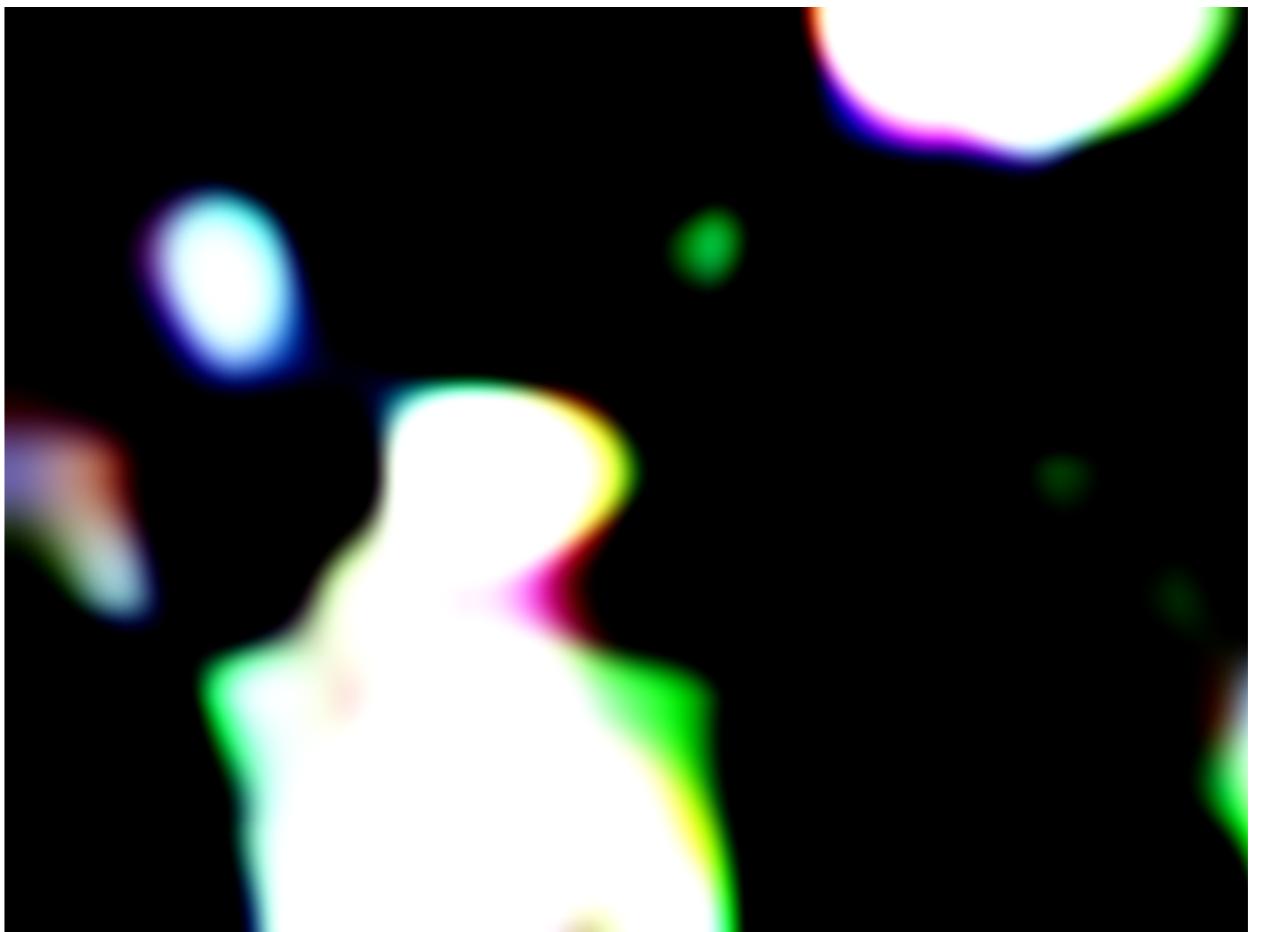
float fbm(vec3 x) {
    float n = 0.0;
    float a = 0.5;
    for (int i = 0; i < 3; i++) {
        n += a * valueNoise(x);
        x *= 2.0;
        a *= 0.5;
    }
    return n;
}

vec3 rgbShift(vec2 pos) {
    float n = fbm(vec3(2.0 * pos, 0.1 * time));
    return vec3(smoothstep(0.5, 0.55, n));
}
```

RGBシフト 3. 完成

```
vec3 rgbShift(vec2 pos) {
    vec3 n = vec3(
        fbm(vec3(2.0 * pos, 0.1 * (time - 1.0))),
        fbm(vec3(2.0 * pos, 0.1 * time)),
        fbm(vec3(2.0 * pos, 0.1 * (time + 1.0)))
    );
    return smoothstep(0.5, 0.55, n);
}
```

RGBの各要素ごとに
ノイズの時間をずらす



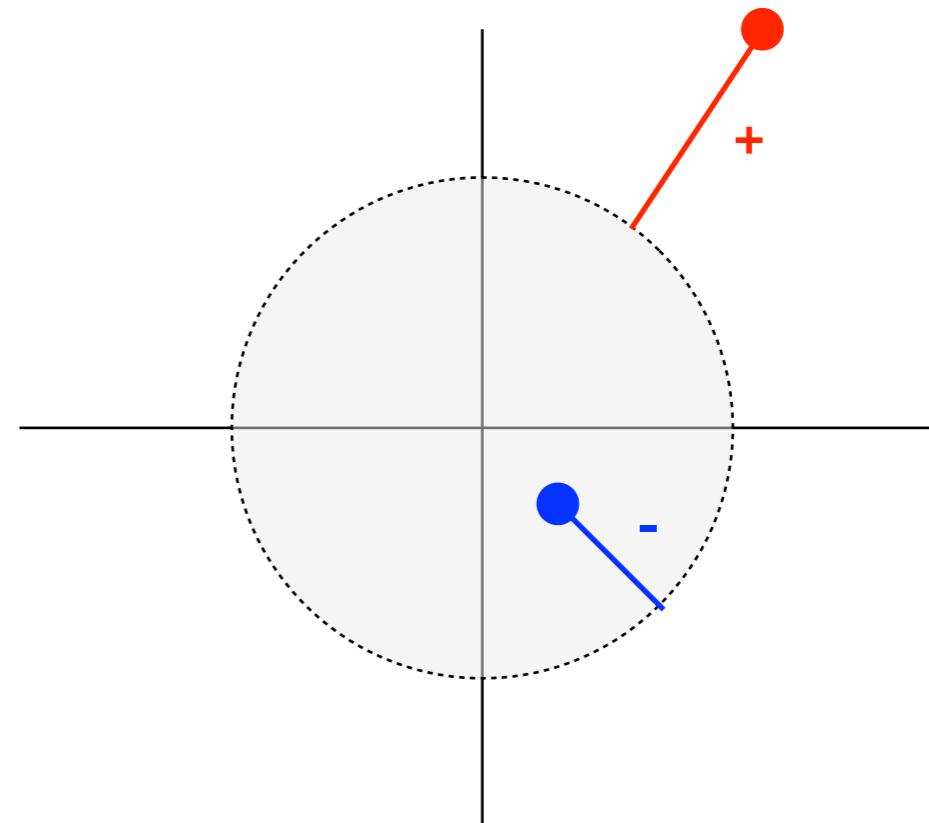
3. 図形

符号付き距離関数(Signed Distance Function; SDF)

- ある位置から図形の表面までの距離を返す関数
- 図形内部の場合は負の値を返す

円の距離関数

```
float sdCircle(vec2 pos, float radius) {  
    return length(pos) - radius;  
}
```

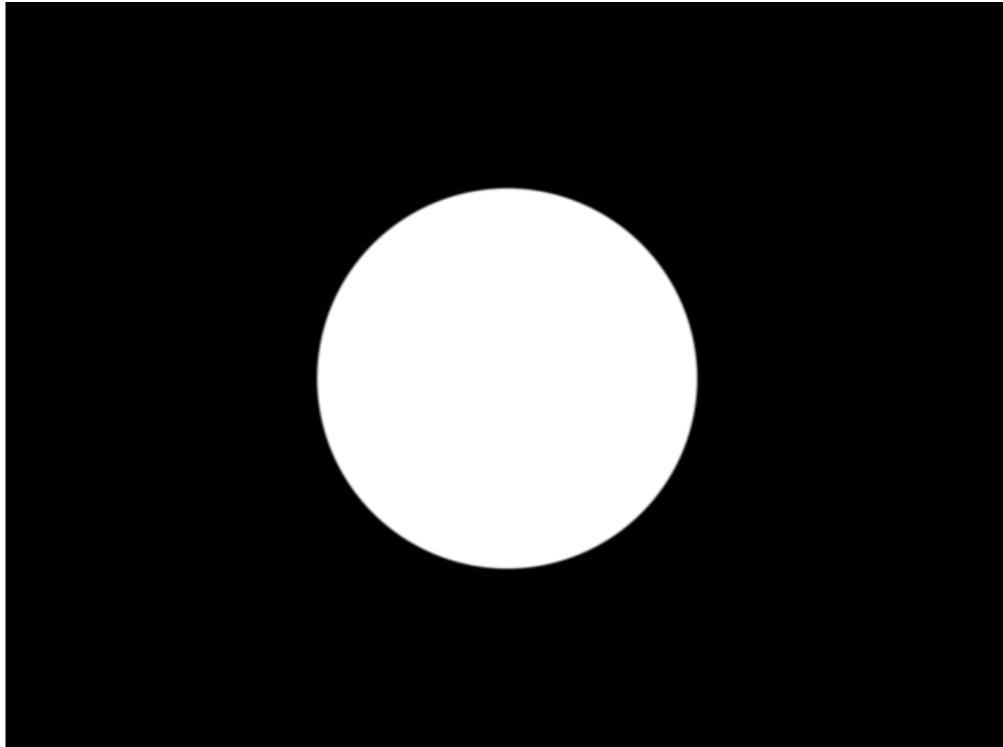


SDFによる図形の描画

```
float sdCircle(vec2 pos, float radius) {
    return length(pos) - radius;
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    float d = sdCircle(pos, 0.5);
    vec3 col = mix(vec3(1.0), vec3(0.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

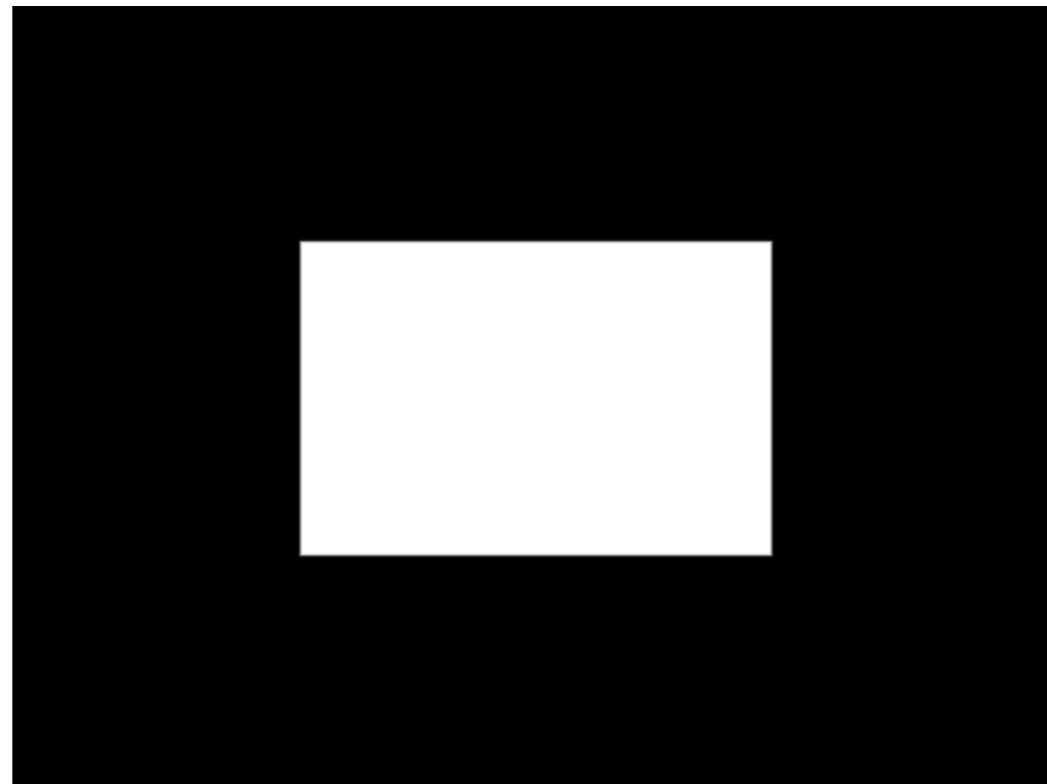
図形の内部は白に、外部は黒に



四角形

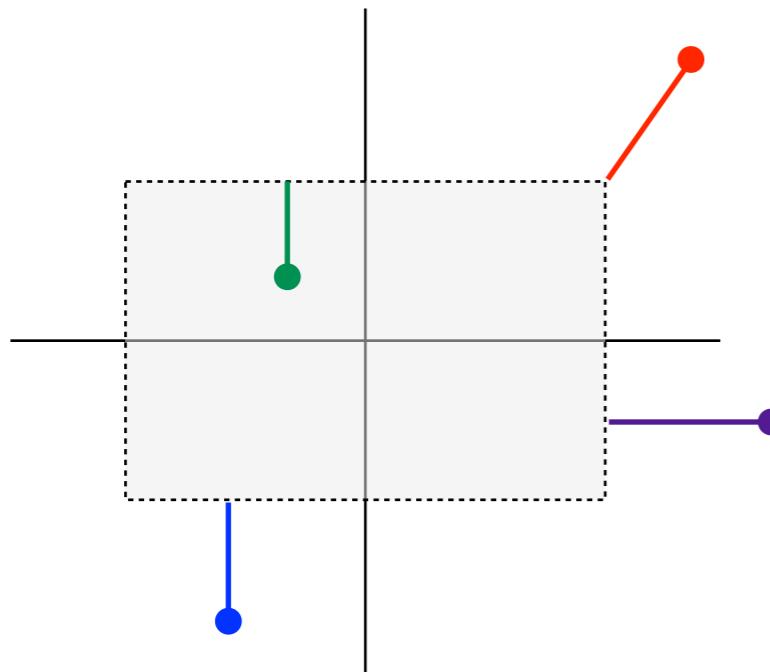
```
float sdRect(vec2 pos, vec2 size) {
    pos = abs(pos) - size;
    return length(max(pos, 0.0)) + min(max(pos.x, pos.y), 0.0);
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    float d = sdRect(pos, vec2(0.6, 0.4));
    vec3 col = mix(vec3(1.0), vec3(0.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

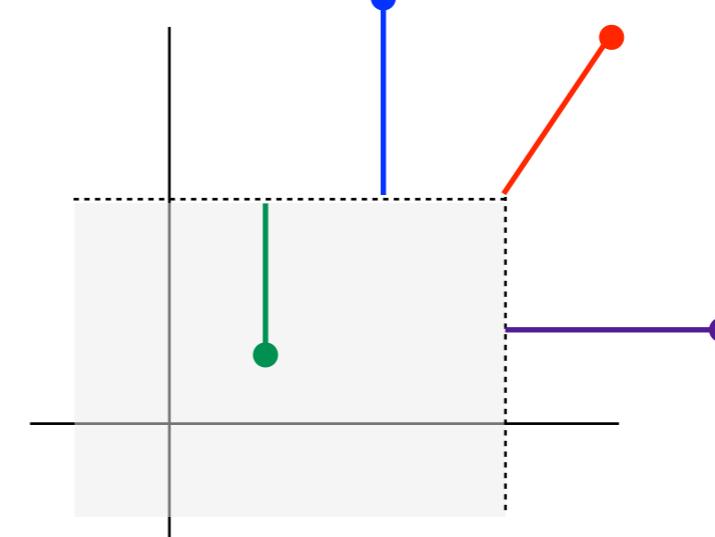


四角形の距離関数のイメージ

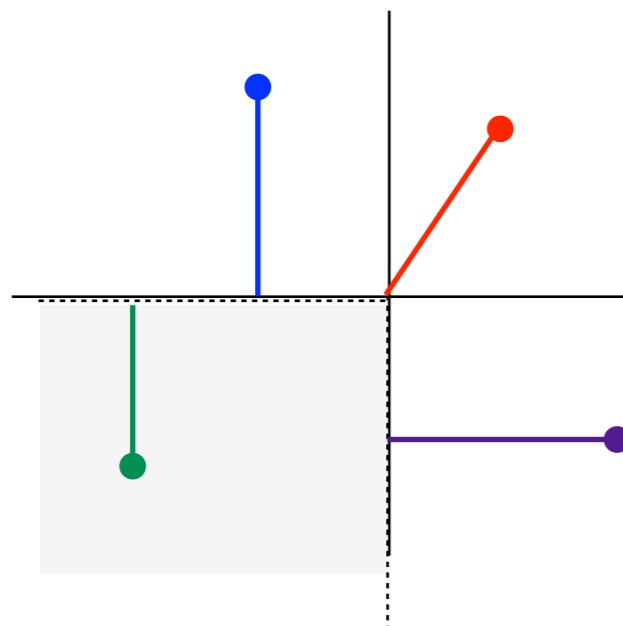
1. 初期位置 pos



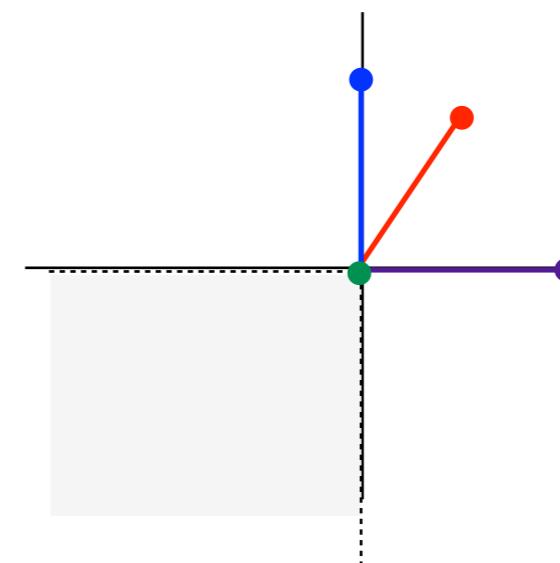
2. abs(pos)



3. abs(pos) - size



4. max(abs(pos) - size, 0.0)



点が距離を求める位置、線が最短距離を表している

その他の図形

<https://www.iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm>



Equilateral Triangle - exact (<https://www.shadertoy.com/view/Xl2yDW>)

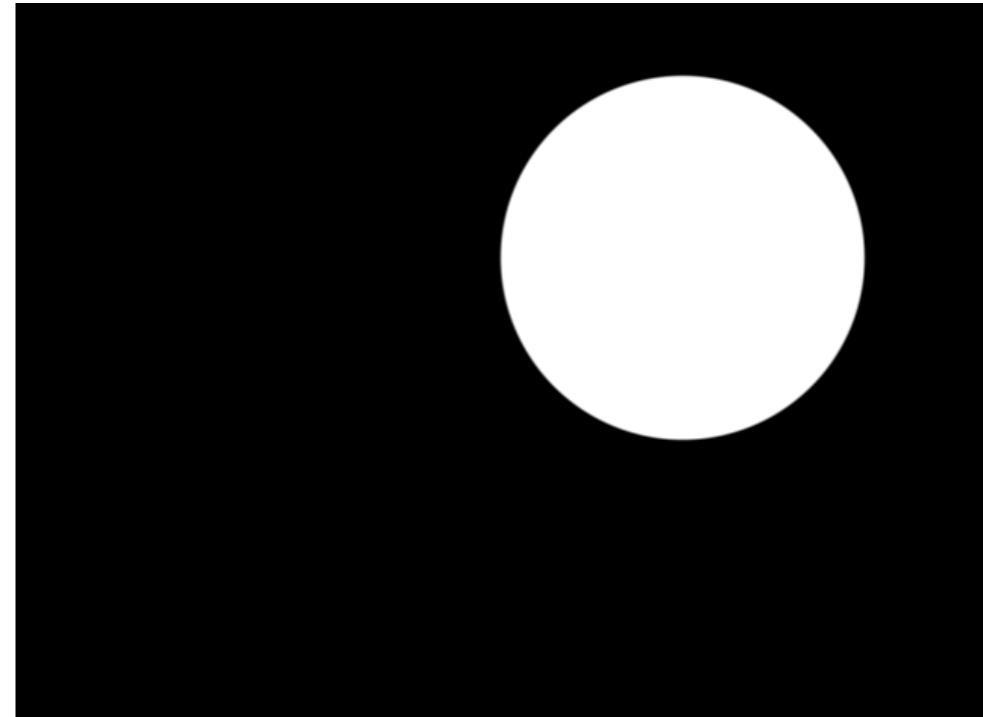
```
float sdEquilateralTriangle( in vec2 p )
{
    const float k = sqrt(3.0);
    p.x = abs(p.x) - 1.0;
    p.y = p.y + 1.0/k;
    if( p.x+k*p.y>0.0 ) p = vec2(p.x-k*p.y,-k*p.x-p.y)/2.0;
    p.x -= clamp( p.x, -2.0, 0.0 );
    return -length(p)*sign(p.y);
```

移動

```
vec2 translate(vec2 pos, vec2 offset) {
    return pos - offset;
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    pos = translate(pos, vec2(0.5, 0.3));
    float d = sdCircle(pos, 0.5);
    vec3 col = mix(vec3(1.0), vec3(0.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

図形を移動するためには
座標を反対方向に動かす



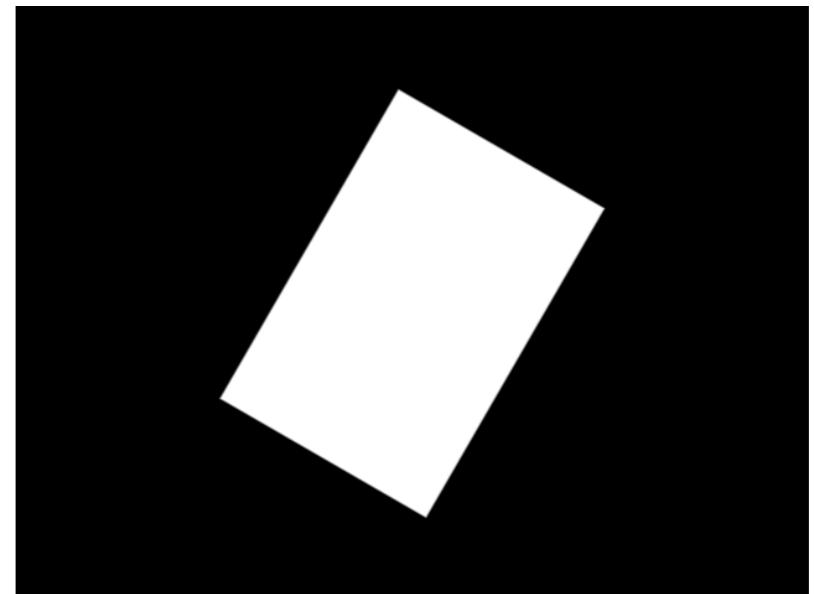
回転

```
#define PI 3.14159265359

vec2 rotate(vec2 pos, float radian) {
    float c = cos(-radian);
    float s = sin(-radian);
    return mat2(c, s, -s, c) * pos;
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    pos = rotate(pos, PI / 3.0);
    float d = sdRect(pos, vec2(0.6, 0.4));
    vec3 col = mix(vec3(1.0), vec3(0.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

図形を回転させるためには
回転行列で座標を反対方向に回転させる

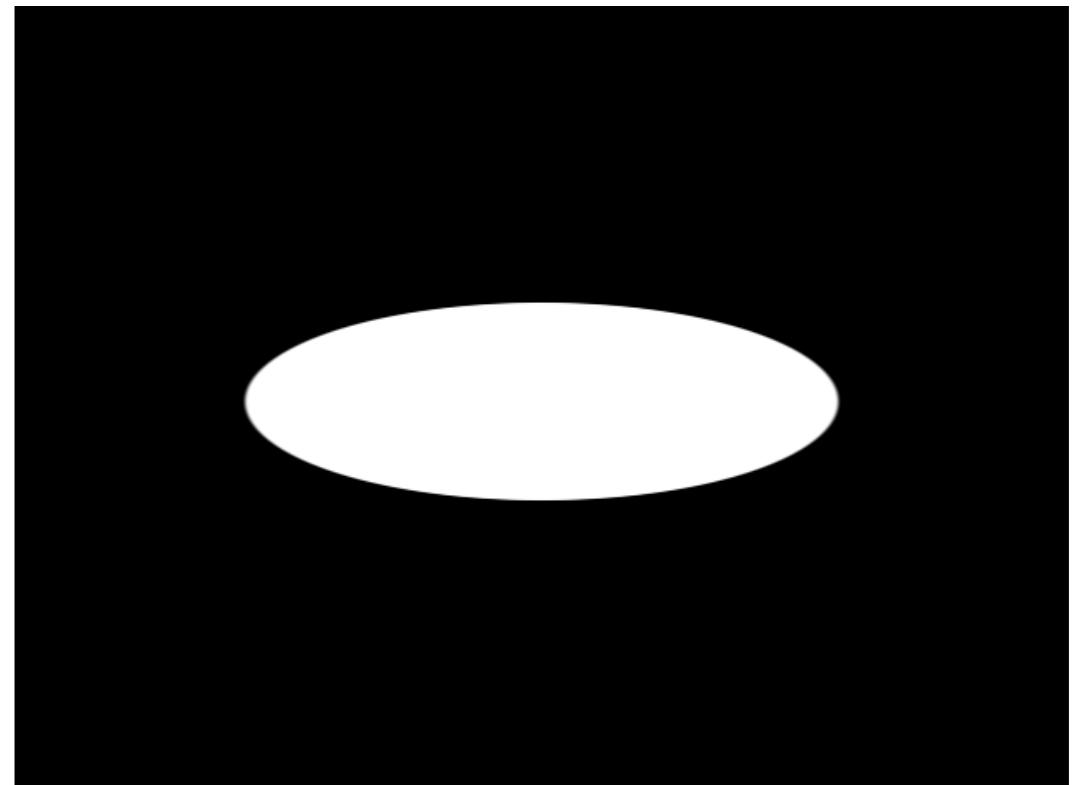


拡大縮小

```
vec2 scale(vec2 pos, vec2 rate) {
    return pos /rate;
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    pos = scale(pos, vec2(1.5, 0.5));
    float d = sdCircle(pos, 0.5);
    vec3 col = mix(vec3(1.0), vec3(0.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

図形を拡大(縮小)させるためには
座標を縮小(拡大)させる

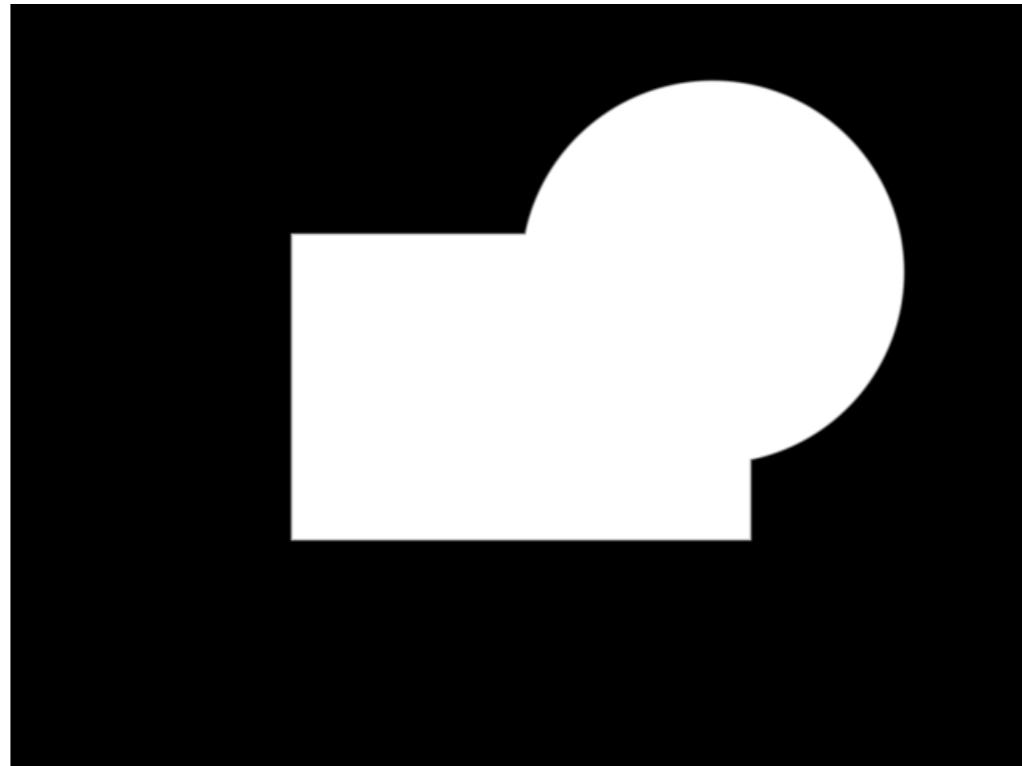


ブーリアン演算 - 和 -

```
float opUnion(float d1, float d2) {
    return min(d1, d2);
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    float rd = sdRect(pos, vec2(0.6, 0.4));
    float cd = sdCircle(translate(pos, vec2(0.5, 0.3)), 0.5);
    float d = opUnion(rd, cd);
    vec3 col = mix(vec3(1.0), vec3(0.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

2つの図形を結合する



ブーリアン演算 - 差 -

```
float opSubtraction(float d1, float d2) {
    return max(d1, -d2);
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    float rd = sdRect(pos, vec2(0.6, 0.4));
    float cd = sdCircle(translate(pos, vec2(0.5, 0.3)), 0.5);
    float d = opSubtraction(rd, cd);
    vec3 col = mix(vec3(1.0), vec3(0.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

ある図形から別の図形を引く

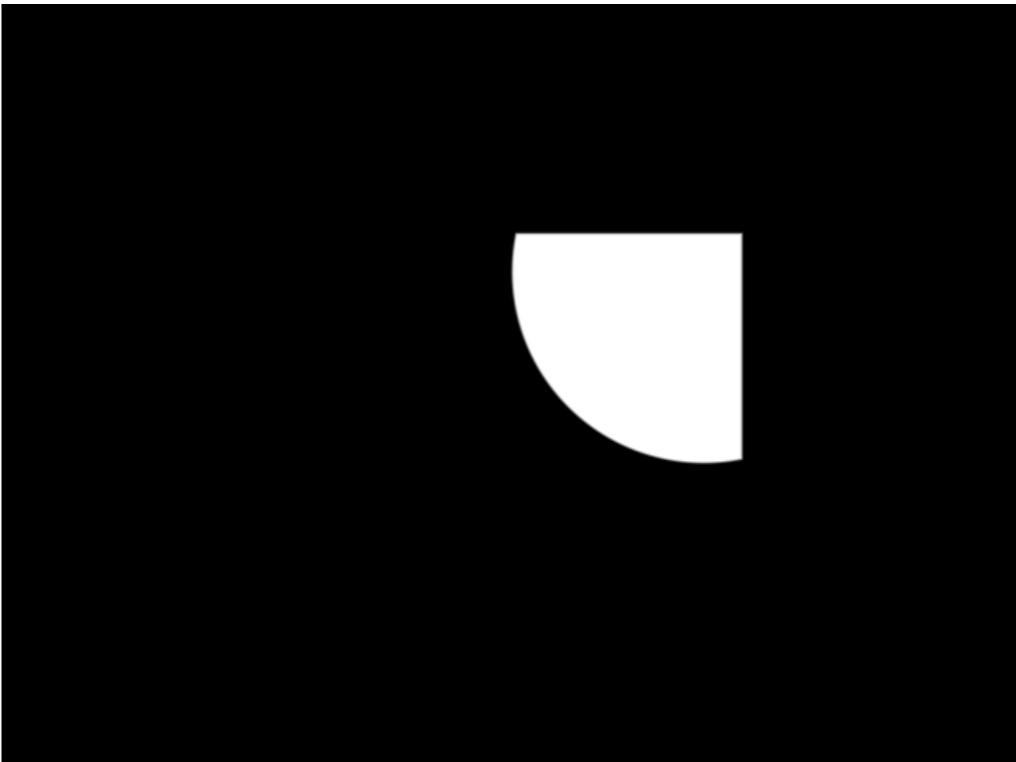


ブーリアン演算 - 積 -

```
float opIntersection(float d1, float d2) {
    return max(d1, d2);
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    float rd = sdRect(pos, vec2(0.6, 0.4));
    float cd = sdCircle(translate(pos, vec2(0.5, 0.3)), 0.5);
    float d = opIntersection(rd, cd);
    vec3 col = mix(vec3(1.0), vec3(0.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

2つの図形の重なっている
箇所のみを取り出す

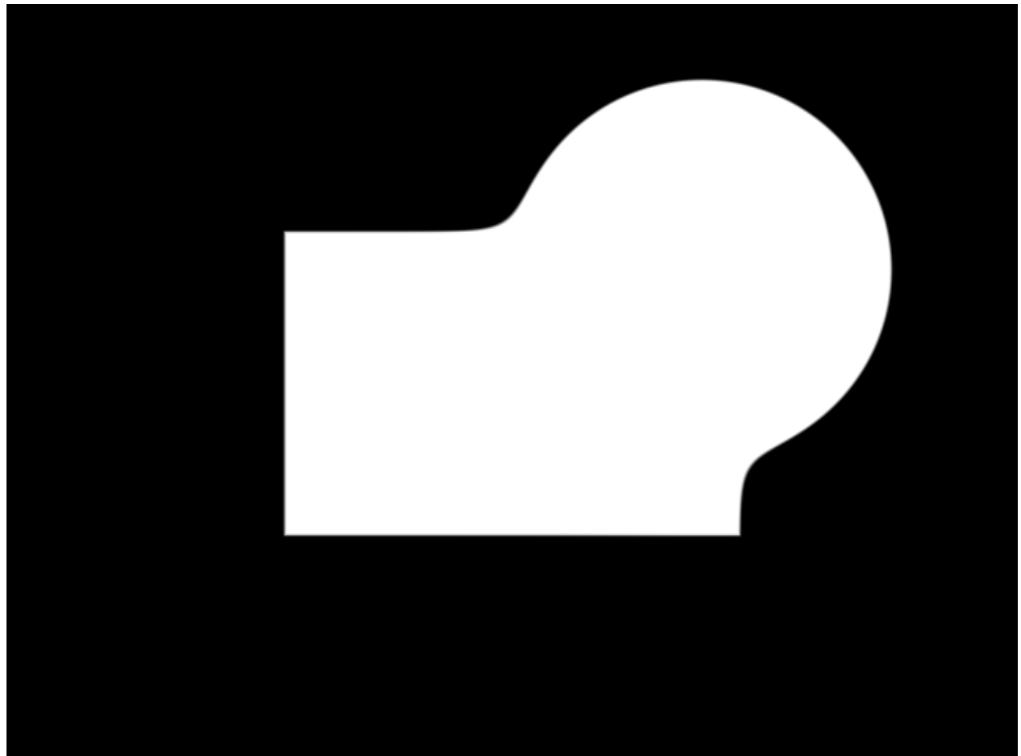


滑らかな結合

```
float opSmoothUnion(float d1, float d2, float k) {
    return -log2(exp2(-k * d1) + exp2(-k * d2)) / k;
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    float rd = sdRect(pos, vec2(0.6, 0.4));
    float cd = sdCircle(translate(pos, vec2(0.5, 0.3)), 0.5);
    float d = opSmoothUnion(rd, cd, 32.0);
    vec3 col = mix(vec3(1.0), vec3(0.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

2つの図形を滑らかに結合する

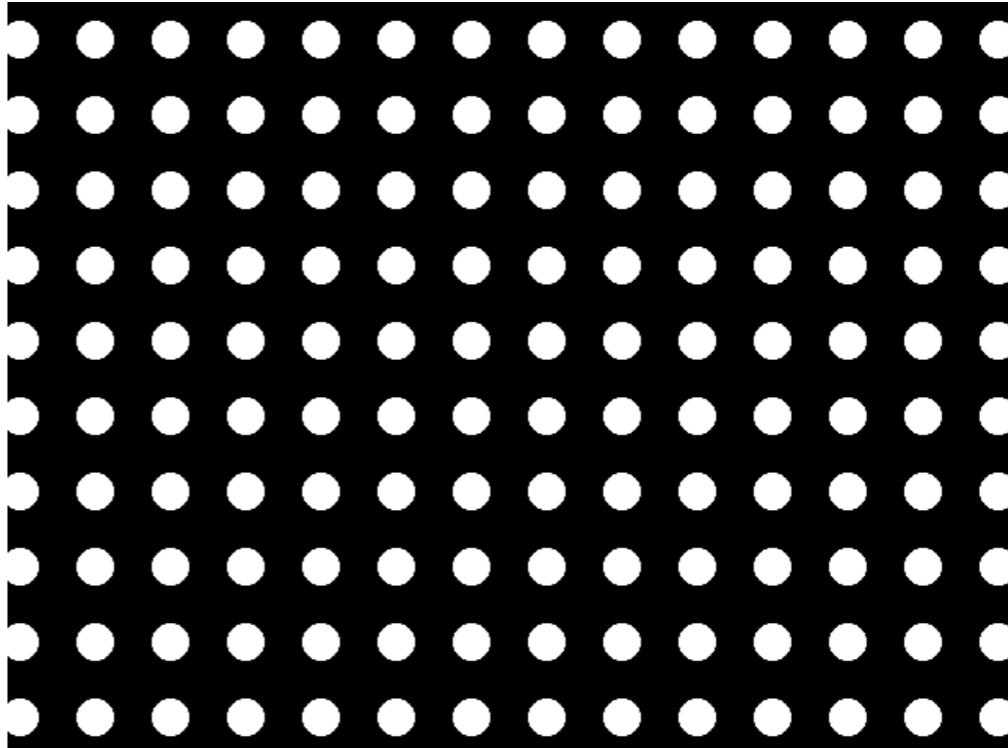


繰り返し

```
vec2 repeat(vec2 pos, vec2 size) {
    return mod(pos, 2.0 * size) - size;
}

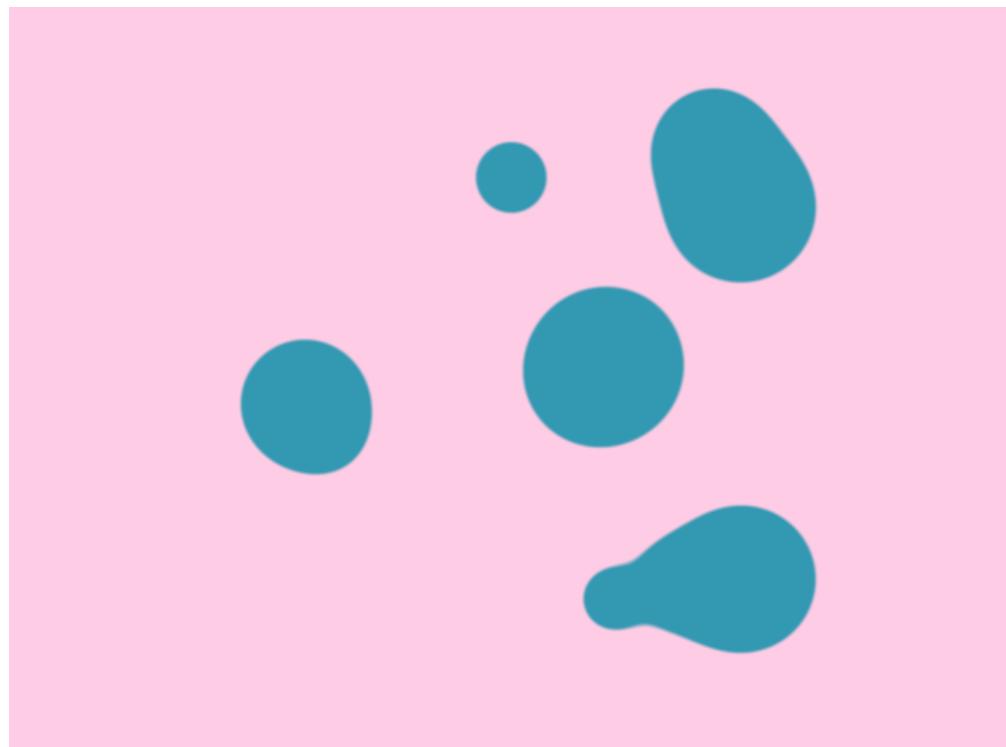
void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    pos = repeat(10.0 * pos, vec2(1.0));
    float d = sdCircle(pos, 0.5);
    vec3 col = mix(vec3(1.0), vec3(0.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

図形をタイリングする

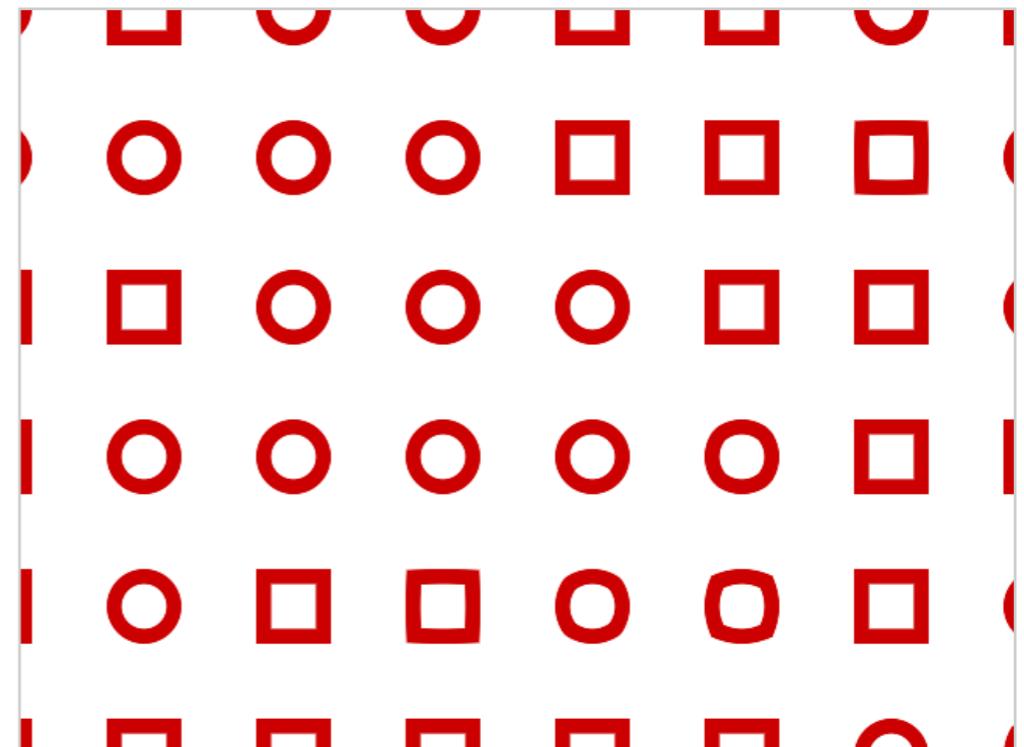


作例 (図形編)

メタボール



モーフィング



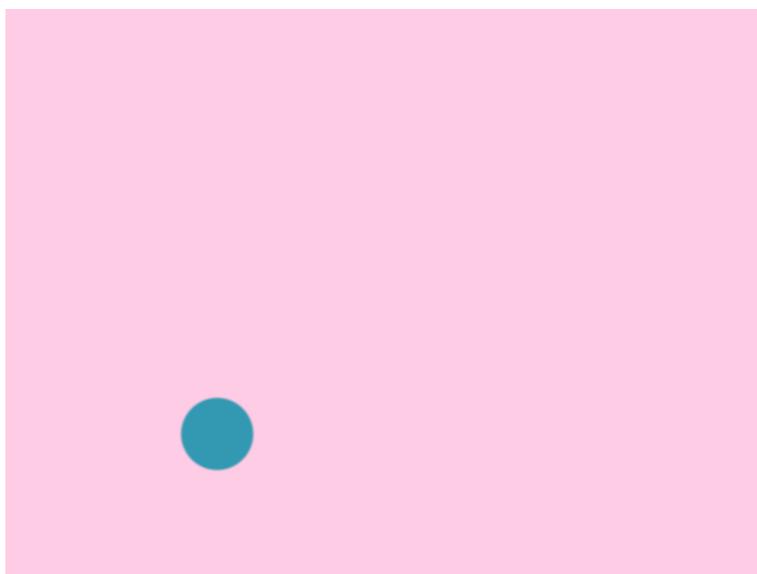
/src/ch3-shape/metaballs/step-3.gls/
/src/ch3-shape/morphing/step-4.gls/

メタボール 1.

```
float sdCircle(vec2 pos, float radius) {
    return length(pos) - radius;
}
vec2 translate(vec2 pos, vec2 offset) {
    return pos - offset;
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    pos *= 8.0;
    vec2 offset = 5.0 * sin(vec2(1.0, 0.8) * time);
    float radius = 1.0;
    float d = sdCircle(translate(pos, offset), radius);
    vec3 col = mix(vec3(0.2, 0.6, 0.7), vec3(1.0, 0.8, 0.9), smoothstep(-0.05, 0.05, d));
    gl_FragColor = vec4(col, 1.0);
}
```

円を一つ動かす



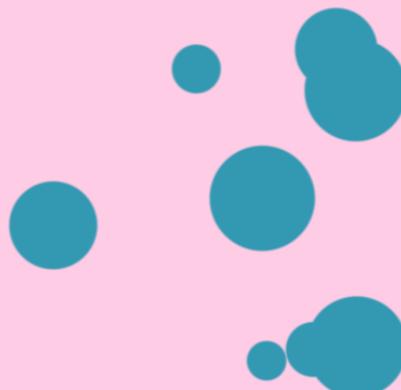
メタボール 2.

```
float random(float v) {
    return fract(sin(v * 12.9898) * 43758.5453);
}

float opUnion(float d1, float d2) {
    return min(d1, d2);
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    pos *= 8.0;
    float d = 1e6;
    for (float i = 1.0; i <= 10.0; i++) {
        vec2 offset = 5.0 * sin(2.0 * vec2(random(i), random(i * 10.0)) * time);
        float radius = mix(0.5, 2.0, random(i * 100.0));
        float cd = sdCircle(translate(pos, offset), radius);
        d = opUnion(d, cd);
    }
    vec3 col = mix(vec3(0.2, 0.6, 0.7), vec3(1.0, 0.8, 0.9), smoothstep(-0.05, 0.05, d));
    gl_FragColor = vec4(col, 1.0);
}
```

ループで複数の円を動かす



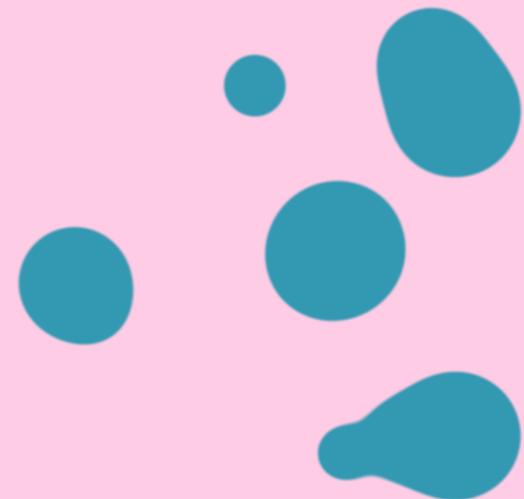
/src/ch3-shape/metaballs/step-2.glsl

メタボール 3. 完成

```
float opSmoothUnion(float d1, float d2, float k) {
    return -log2(exp2(-k * d1) + exp2(-k * d2)) / k;
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    pos *= 8.0;
    float d = 1e6;
    for (float i = 1.0; i <= 10.0; i++) {
        vec2 offset = 5.0 * sin(2.0 * vec2(random(i), random(i * 10.0)) * time);
        float radius = mix(0.5, 2.0, random(i * 100.0));
        float cd = sdCircle(translate(pos, offset), radius);
        d = opSmoothUnion(d, cd, 4.0);
    }
    vec3 col = mix(vec3(0.2, 0.6, 0.7), vec3(1.0, 0.8, 0.9), smoothstep(-0.05, 0.05, d));
    gl_FragColor = vec4(col, 1.0);
}
```

円を滑らかに結合する



ランダムな図形 1.

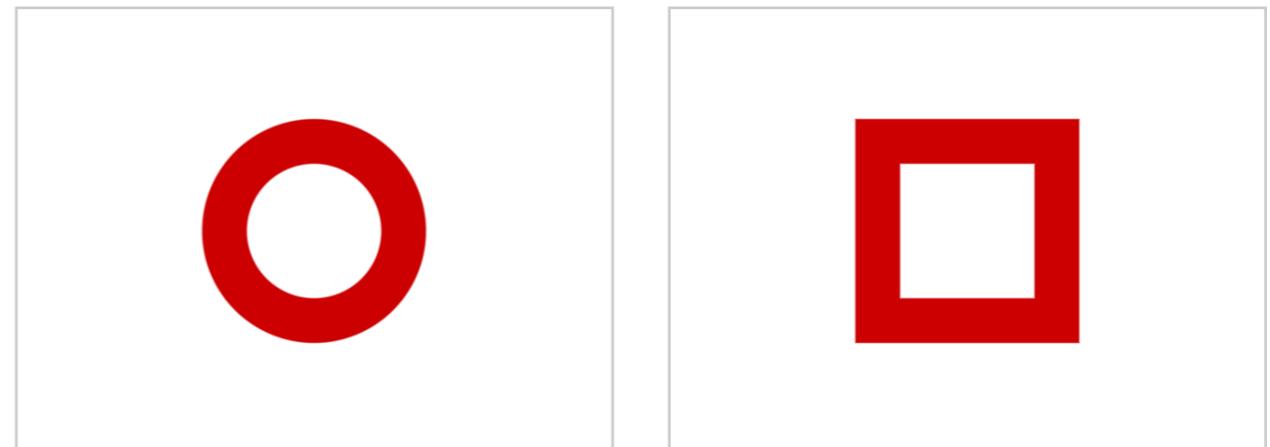
```
float opSubtraction(float d1, float d2) {
    return max(d1, -d2);
}

float sdOutlineCircle(vec2 pos, float radius, float thickness) {
    return opSubtraction(sdCircle(pos, radius), sdCircle(pos, radius - thickness));
}

float sdOutlineRect(vec2 pos, vec2 size, float thickness) {
    return opSubtraction(sdRect(pos, size), sdRect(pos, size - thickness));
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    float d = sdOutlineCircle(pos, 0.5, 0.2);
    // float d = sdOutlineRect(pos, vec2(0.5), 0.2);
    vec3 col = mix(vec3(0.8, 0.0, 0.0), vec3(1.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

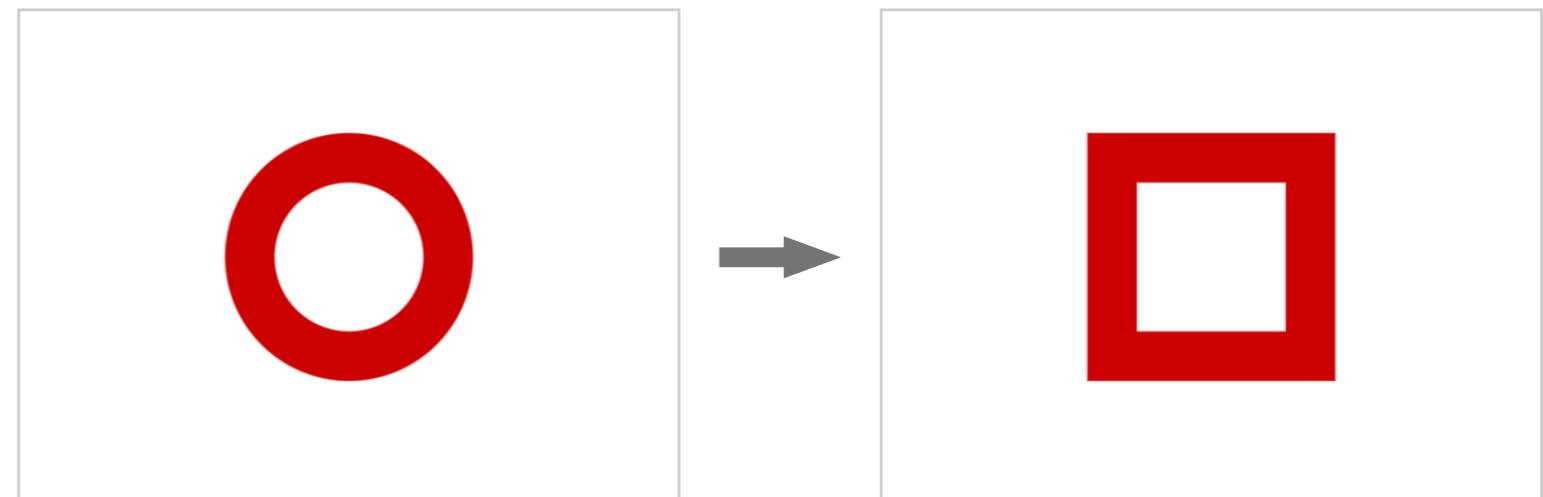
既存の距離関数と
ブーリアン演算で
新しい距離関数を作成する



ランダムな図形 2.

```
void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    float cd = sdOutlineCircle(pos, 0.5, 0.2);
    float rd = sdOutlineRect(pos, vec2(0.5), 0.2);
    float t = mod(0.8 * time, 2.0);
    float s = smoothstep(0.8, 1.0, fract(t));
    float d = mix(cd, rd, floor(t) == 0.0 ? s : (1.0 - s));
    vec3 col = mix(vec3(0.8, 0.0, 0.0), vec3(1.0), smoothstep(-0.005, 0.005, d));
    gl_FragColor = vec4(col, 1.0);
}
```

2つの距離関数を
mixすることで
図形をモーフィングする

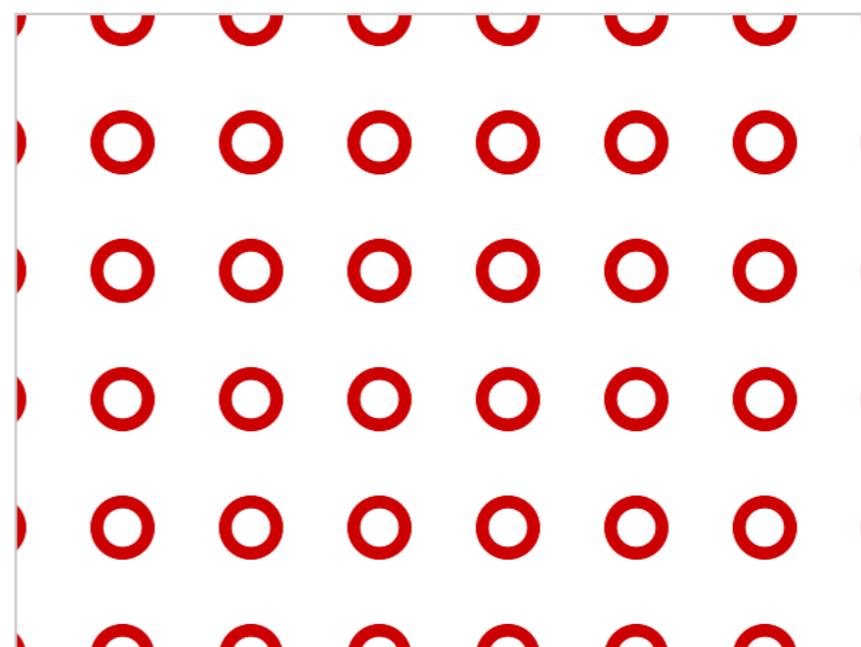


ランダムな図形 3.

```
vec2 repeat(vec2 pos, vec2 size) {
    return mod(pos, 2.0 * size) - size;
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    pos = repeat(5.0 * pos, vec2(1.0));
    float cd = sdOutlineCircle(pos, 0.5, 0.2);
    float rd = sdOutlineRect(pos, vec2(0.5), 0.2);
    float t = mod(0.8 * time, 2.0);
    float s = smoothstep(0.8, 1.0, fract(t));
    float d = mix(cd, rd, floor(t) == 0.0 ? s : (1.0 - s));
    vec3 col = mix(vec3(0.8, 0.0, 0.0), vec3(1.0), smoothstep(-0.01, 0.01, d));
    gl_FragColor = vec4(col, 1.0);
}
```

繰り返しで図形を敷き詰める



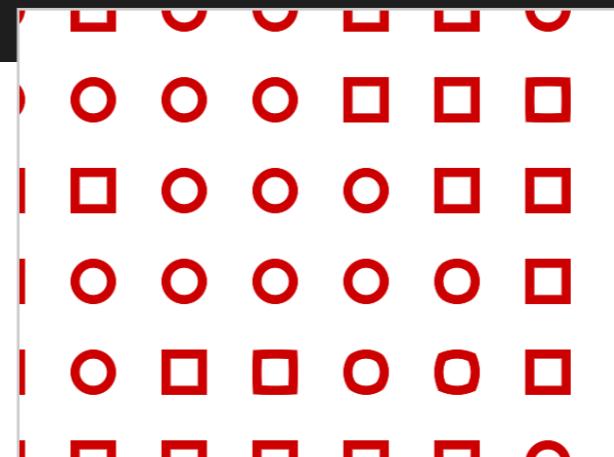
ランダムな図形 4. 完成

```
float random(vec2 v) {
    return fract(sin(dot(v, vec2(12.9898, 78.233))) * 43758.5453);
}

vec2 repeatIdx(vec2 pos, vec2 size) {
    return floor(pos / (2.0 * size));
}

void main(void) {
    vec2 pos = (2.0 * gl_FragCoord.xy - resolution) / min(resolution.x, resolution.y);
    vec2 repIdx = repeatIdx(5.0 * pos, vec2(1.0));
    pos = repeat(5.0 * pos, vec2(1.0));
    float cd = sdOutlineCircle(pos, 0.5, 0.2);
    float rd = sdOutlineRect(pos, vec2(0.5), 0.2);
    float t = mod(0.8 * (time + 10.0 * random(repIdx)), 2.0);
    float s = smoothstep(0.8, 1.0, fract(t));
    float d = mix(cd, rd, floor(t) == 0.0 ? s : (1.0 - s));
    vec3 col = mix(vec3(0.8, 0.0, 0.0), vec3(1.0), smoothstep(-0.01, 0.01, d));
    gl_FragColor = vec4(col, 1.0);
}
```

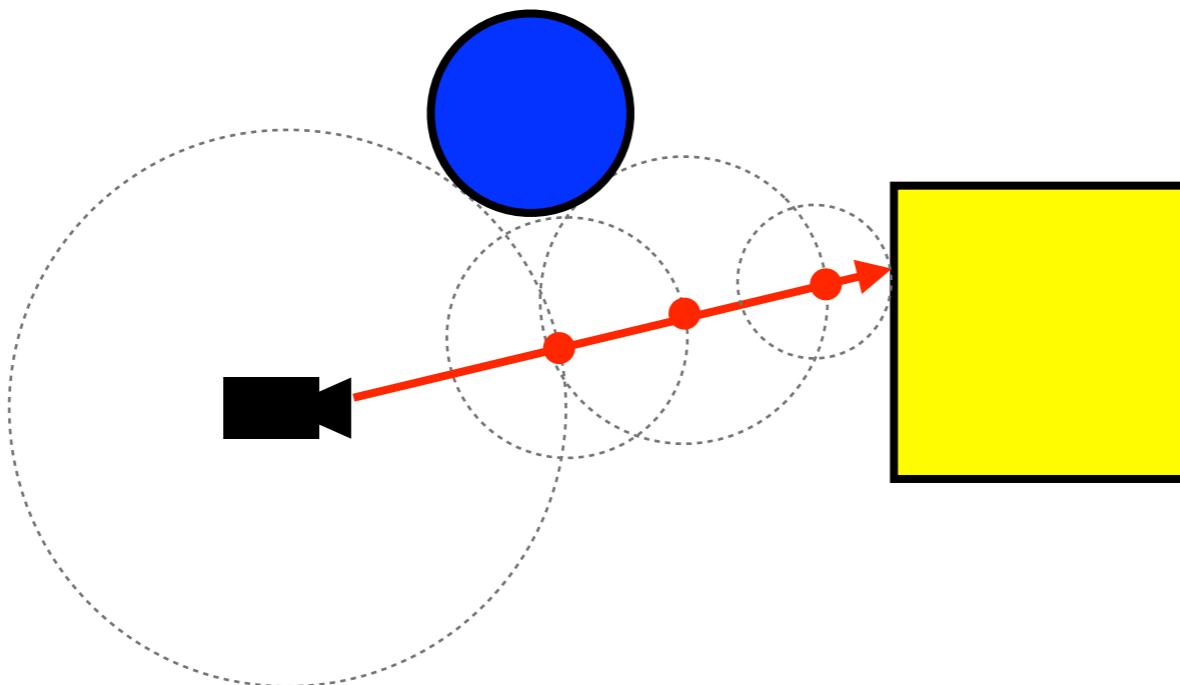
タイルごとにタイミングをずらす



4. 3D

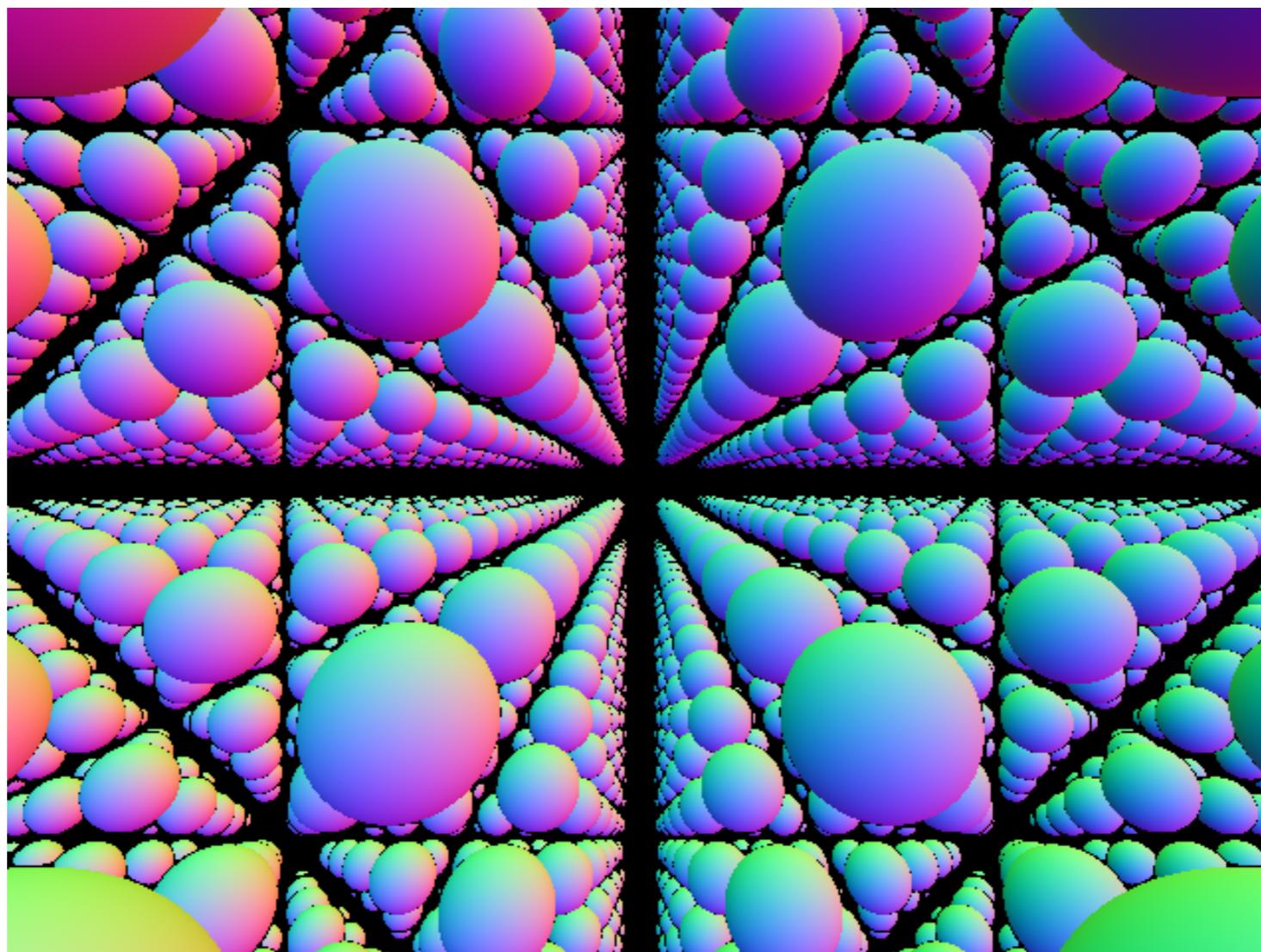
(スフィア)レイマーチング

- ・ シェーダーアートの3D表現で主に使われる手法
- ・ 符号付き距離関数でオブジェクトを表現する
- ・ カメラから飛ばしたレイをオブジェクトとぶつかるまで進める



- ・ 参考
 - ・ 魔法使いになりたい人のためのシェーダーライブコーディング入門 - Qiita
 - ・ <https://qiita.com/kaneta1992/items/21149c78159bd27e0860>
 - ・ シェーダだけで世界を創る！three.jsによるレイマーチング
 - ・ <https://www.slideshare.net/shohosoda9/threejs-58238484>

(スフィア)レイマーチング



終わりに

アドバイス

- ・ 座標と数値を常に意識しよう
- ・ 基本は要素の組み合わせ、引き出しを増やそう
 - ・ 他の人のソースコードを読む
 - ・ Shadertoy, GLSL
 - ・ 別分野の知識を持ち込む
 - ・ Processing / p5.js, 3D, グラフィックデザイン...

さらなる勉強のためのリンク集

- The Book of Shaders
 - シェーダーを基礎から解説
 - <https://thebookofshaders.com/?lan=jp>
- wgl.org GLSL contents
 - WebGLの勉強で誰もがお世話になるdoxas先生によるGLSLの解説
 - フラクタルからレイマーチングまで
 - <https://wgl.org/d/glsl/>
- GLSL-Noise.md
 - GLSLによるノイズ実装のまとめ
 - <https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83>
- Inigo Quilez :: fractals, computer graphics, mathematics, shaders, demoscene and more
 - 内容は難しみだが有益な情報が載っている、iq神
 - <https://www.iquilezles.org/index.html>