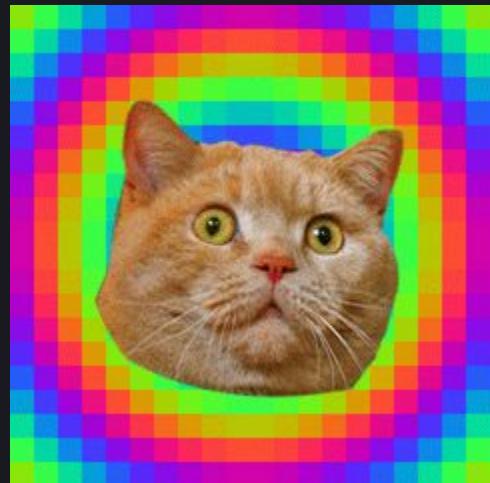


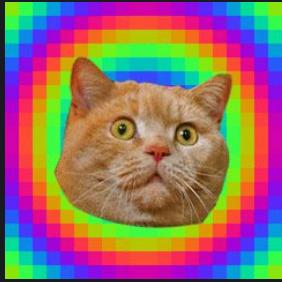


FMS\_Cat

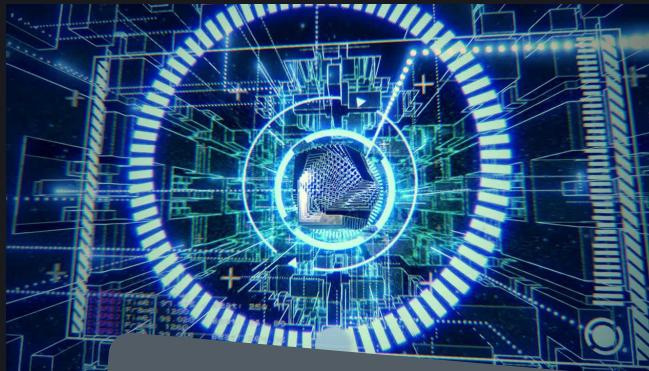
Hello to  
Processing Community!



FMS\_Cat



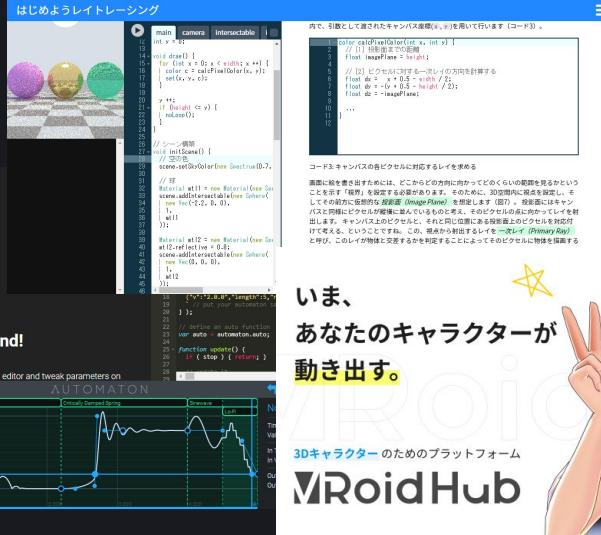
FMS\_Cat



# Webばっかやん！

# Yutaka Obuchi

# クリエイティブコーディング愛好家 Webフロント / CGエンジニア



いま、  
あなたのキャラクターが  
動き出す。





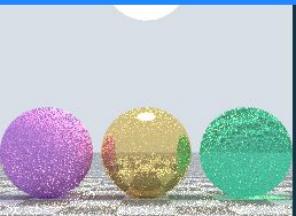
FMS



Wenはつかやん！

# Yutaka Obuchi

## はじめようレイトレーシング



```
main camera intersectable i
12 int y = 0;
13
14 void draw() {
15     for (int x = 0; x < width; x++) {
16         color c = calcPixelColor(x, y);
17         set(x, y, c);
18     }
19
20     y++;
21     if (height <= y) {
22         noLoop();
23     }
24 }
25
26 // シーン構築
27 void initScene() {
28     // 空の色
29     scene.setSkyColor(new Spectrum(0.7,
30
31     // 球
32     Material mt11 = new Material(new Spec
33     scene.addIntersectable(new Sphere(
34         new Vec(-2.2, 0, 0),
35         1,
36         mt11
37     ));
38
39     Material mt12 = new Material(new Spec
40     mt12.reflective = 0.8;
41     scene.addIntersectable(new Sphere(
42         new Vec(0, 0, 0),
43         1,
44         mt12
45     ));
46 }
```

内で、引数として渡されたキャンバス座標( $x$ ,  $y$ )を用いて行います（コード3）。

```
1 ~ color calcPixelColor(int x, int y) {
2     // [1] 投影面までの距離
3     float imagePlane = height;
4
5     // [2] ピクセルに対する一次レイの方向を計算する
6     float dx = x + 0.5 - width / 2;
7     float dy = -(y + 0.5 - height / 2);
8     float dz = -imagePlane;
9
10    ...
11 }
12 }
```

コード3: キャンバスの各ピクセルに対応するレイを求める

画面に絵を書き出すためには、どこからどの方向に向かってどのくらいの範囲を見るかということを示す「視界」を設定する必要があります。そのために、3D空間内に視点を設定し、そしてその前方に仮想的な 投影面 (Image Plane) を想定します（図7）。投影面にはキャンバスと同様にピクセルが縦横に並んでいるものと考え、そのピクセルの点に向かってレイを射出します。キャンバス上のピクセルと、それと同じ位置にある投影面上のピクセルを対応付けて考える、ということですね。この、視点から射出するレイを 一次レイ (Primary Ray)

と呼び、このレイが物体と交差するかを判定する機能を 交差検査 (Intersection Testing) といいます。この機能によって、レイが何の物体を描画するかを判断できます。

# Processing!!

MroidHub



FMS\_Cat

と Processing

# 学生時代

明治大学

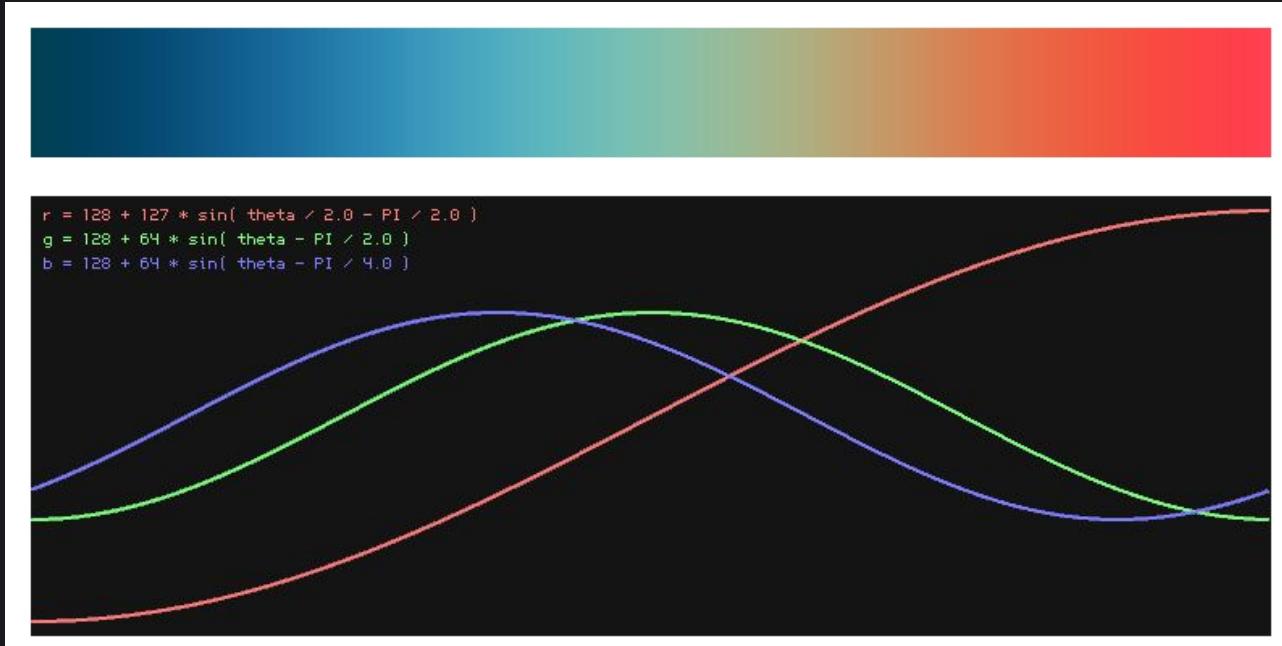
中野キャンパス

# 学生時代

# Processingで プログラミング教育を受ける



今でもちょっとしたプロトタイプは  
Processingが一番はやすく手が動きます



sketch\_190120a | Processing 3.4

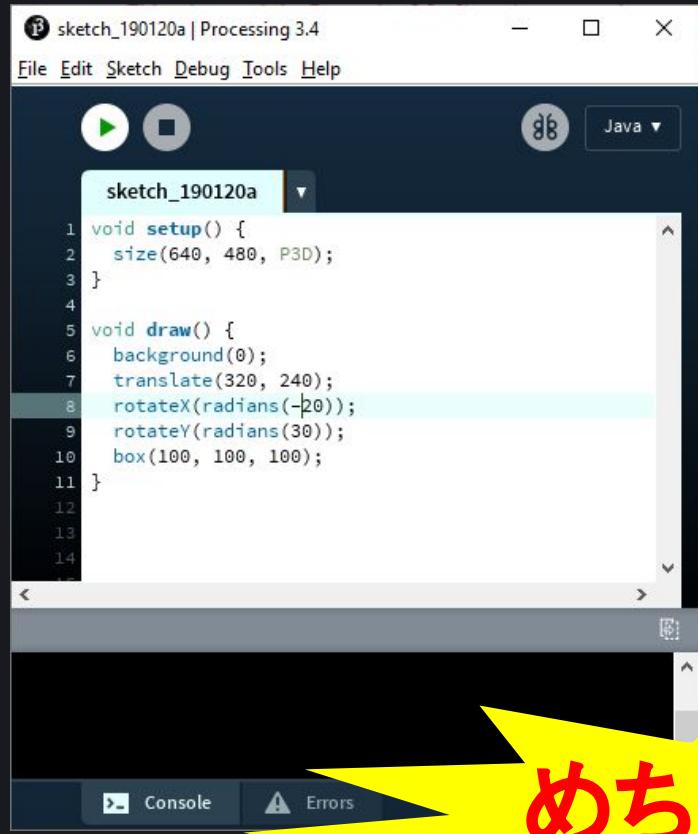
File Edit Sketch Debug Tools Help

Java ▾

sketch\_190120a

```
1 void setup() {  
2     size(640, 480, P3D);  
3 }  
4  
5 void draw() {  
6     background(0);  
7     translate(320, 240);  
8     rotateX(radians(-20));  
9     rotateY(radians(30));  
10    box(100, 100, 100);  
11 }  
12  
13  
14
```

Console Errors



めちゃカンタン

# Three.jsの場合

```
<html>
  <head>
    <title>My first three.js app</title>
    <style>
      body { margin: 0; }
      canvas { width: 100%; height: 100%; }
    </style>
  </head>
  <body>
    <script src="js/three.js"></script>
    <script>
      var scene = new THREE.Scene();
      var camera = new THREE.PerspectiveCamera( 75, window.innerWidth/window.innerHeight, 0.1, 1000 );

      var renderer = new THREE.WebGLRenderer();
      renderer.setSize( window.innerWidth, window.innerHeight );
      document.body.appendChild( renderer.domElement );

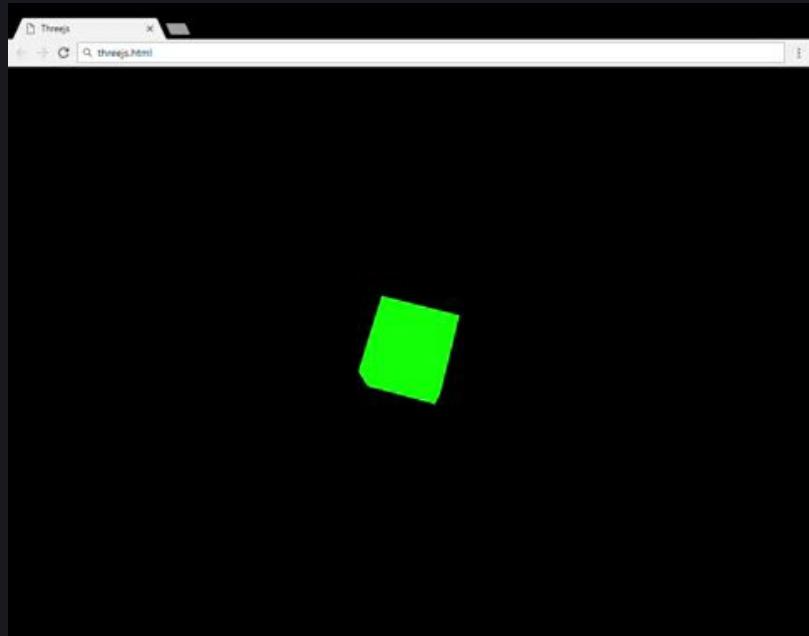
      var geometry = new THREE.BoxGeometry( 1, 1, 1 );
      var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
      var cube = new THREE.Mesh( geometry, material );
      scene.add( cube );

      camera.position.z = 5;

      var animate = function () {
        requestAnimationFrame( animate );

        cube.rotation.x += 0.01;
        cube.rotation.y += 0.01;

        renderer.render( scene, camera );
      };
      animate();
    </script>
  </body>
</html>
```



一方で

お手軽にかけるところで  
いろいろなチャレンジが発生する

- 絵が”Processingっぽい”
- インタラクションを組み込みづらい
  - めちゃ重い

お手軽にかけるところで  
いろいろなチャレンジが発生する

- めちゃ重い

大丈夫です  
Processingでも  
なんとかなります

頑張れば

PLAY WITH

Processing

THE HARD WAY

# おしながき

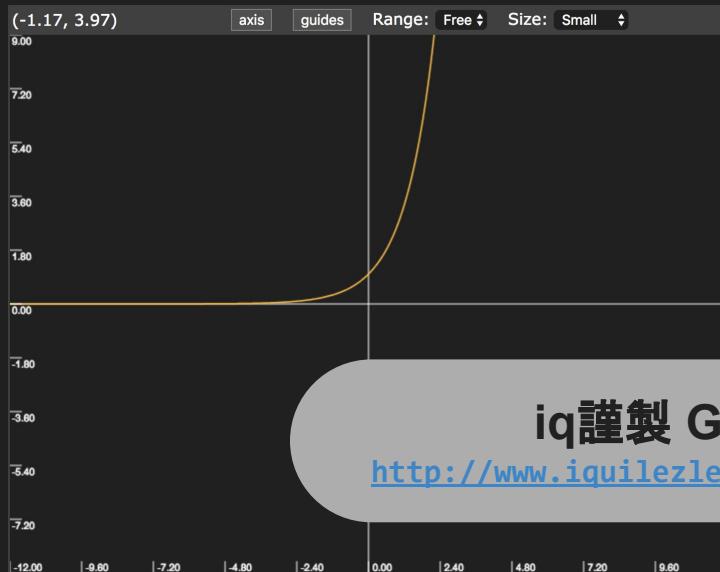


アニメーション

expを使え

# アニメーション

$\exp(x) = e^x = e(\text{自然対数の底})\text{のべき乗}$



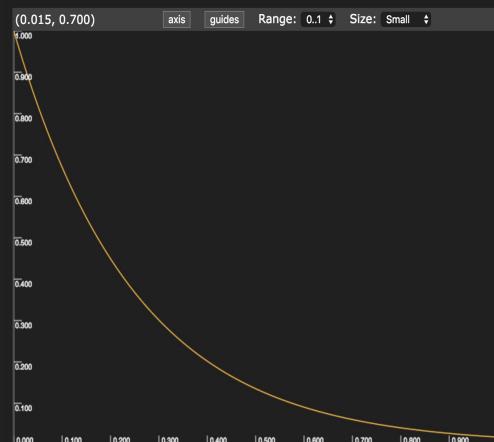
iq謹製 GraphToy

<http://www.iquilezles.org/apps/graphtoy/>

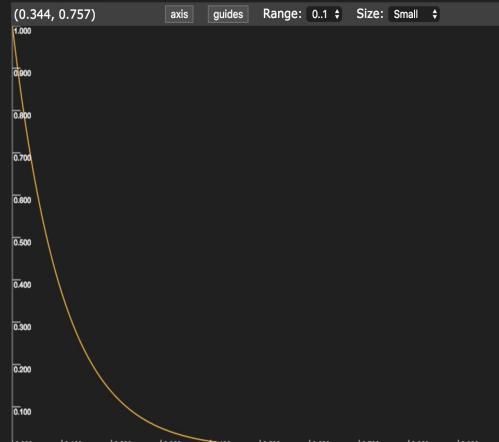
# アニメーション



$$\exp(-x)$$



$$\exp(-4*x)$$



$$\exp(-10*x)$$

## アニメーション

$\exp(-k*x)$ のここがスゴイ

- $x$ の増加に従い**0に漸近する**(0になることはない)
- **速度変化も同じ形** ( $d/dx e^{-kx} = -k e^{-kx}$ )
- **あらゆる自然現象でみられる曲線**(指数関数的減衰)

アニメーション

expを使え

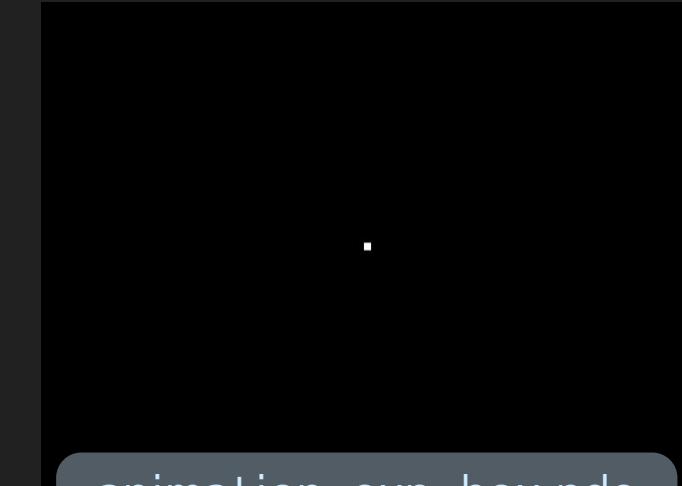
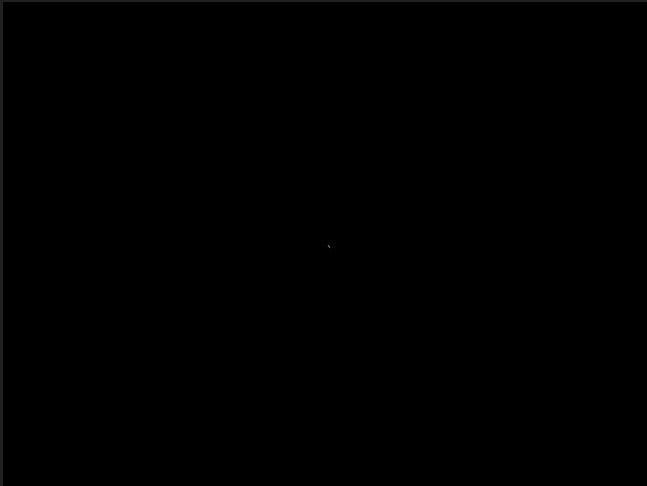
アニメーション

expを使え

どこで？

アニメーション

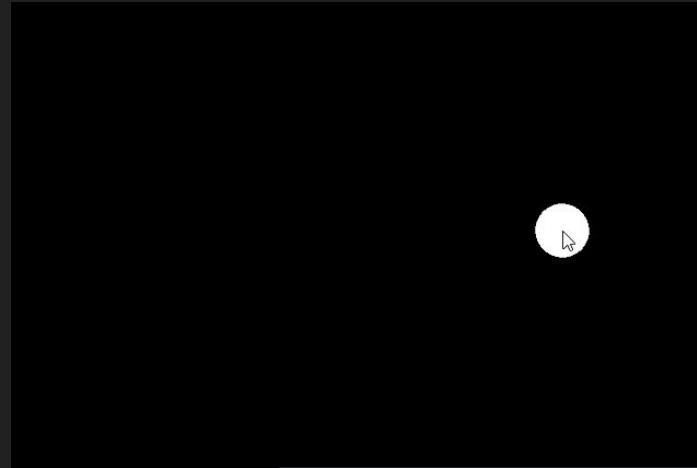
ふつうのアニメーション



animation\_exp\_box.pde

アニメーション

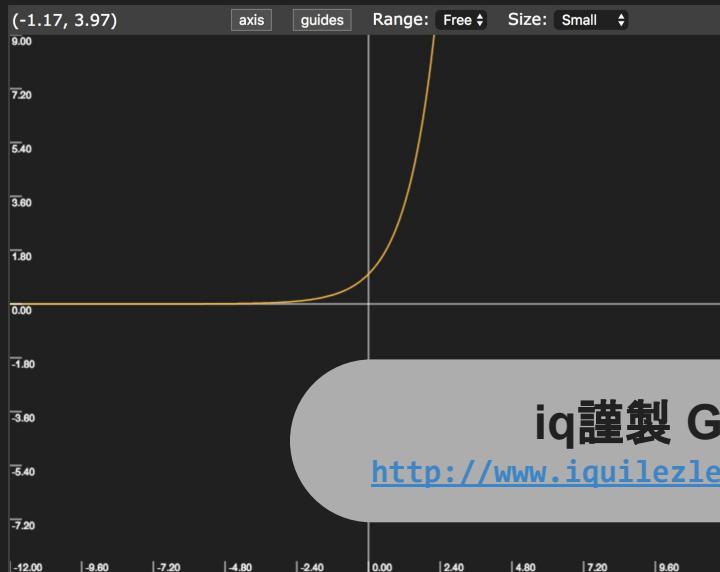
インタラクティブな動き



animation\_exp.pde

# アニメーション

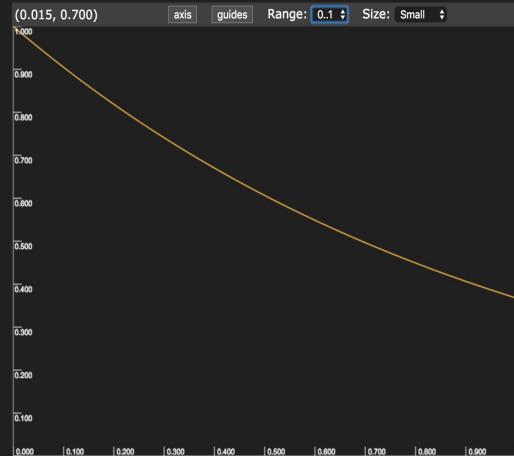
$\exp(x) = e^x = e(\text{自然対数の底})\text{のべき乗}$



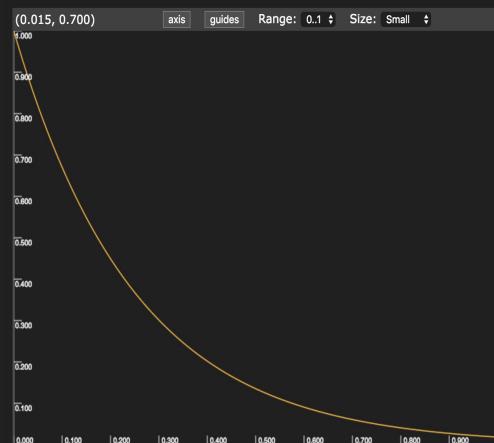
iq謹製 GraphToy

<http://www.iquilezles.org/apps/graphtoy/>

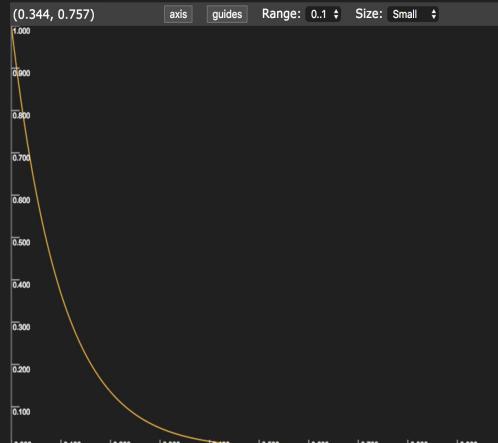
# アニメーション



$\exp(-x)$



$\exp(-4*x)$



$\exp(-10*x)$

## アニメーション

$\exp(-k*x)$ のここがスゴイ

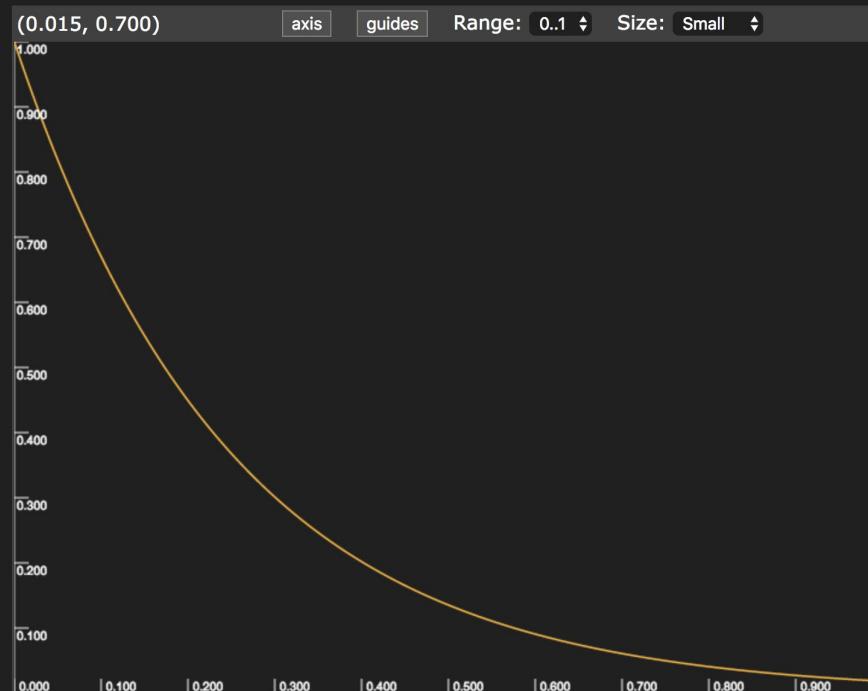
- $x$ の増加に従い**0に漸近する**(0になることはない)
- **速度変化も同じ形** ( $d/dx e^{-kx} = -k e^{-kx}$ )
- **あらゆる自然現象でみられる曲線**(指数関数的減衰)

アニメーション

expを使え

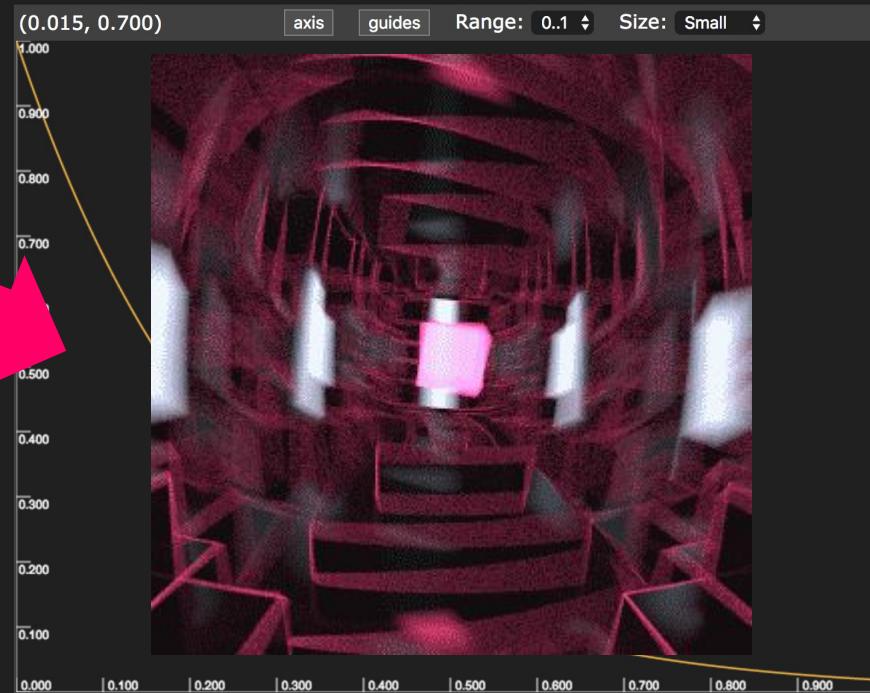
## アニメーション

速度変化が  
急！



# アニメーション

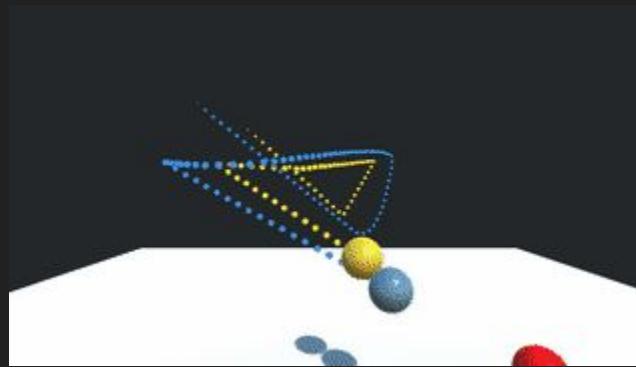
速度変化が



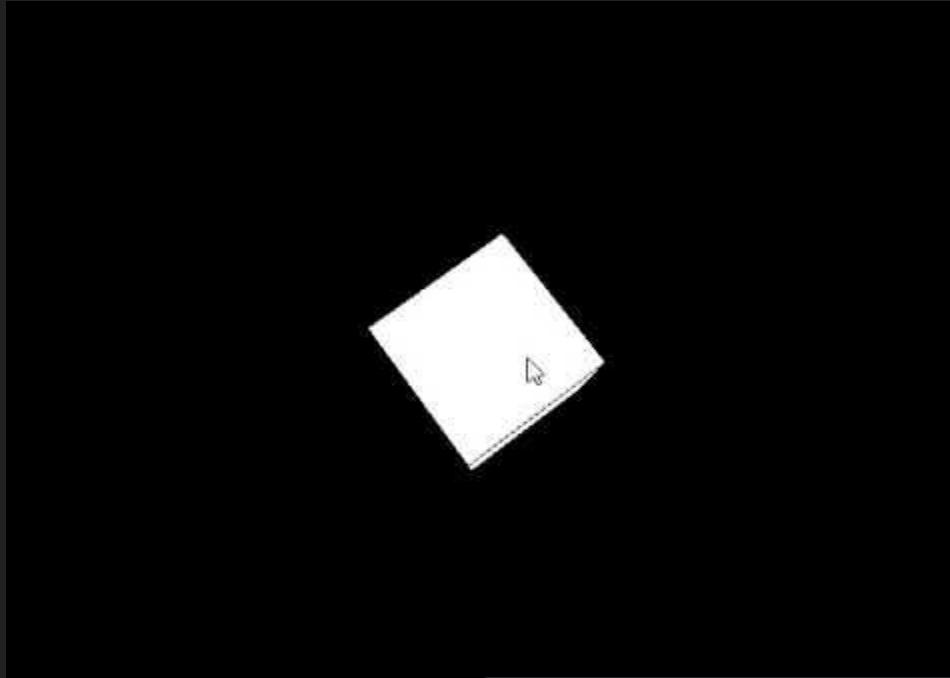
アニメーション

# Critically Damped Spring

<https://github.com/keijiro/SmoothingTest>



回転



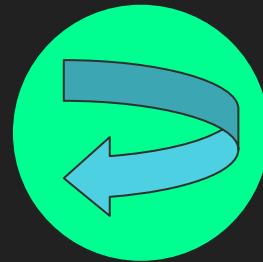
quaternion\_problem.pde

回転

使ってみよう！  
クオータニオンを実装しよう！

回転

クオータニオンなに



回転情報を表す

回転

クオータニオンなに



=



乗算

回転

以上

# 回転

```
// Convert this quaternion into Matrix.  
PMatrix3D toPMatrix3D() {  
    Vector3 x = new Vector3(1.0, 0.0, 0.0).rotate(this);  
    Vector3 y = new Vector3(0.0, 1.0, 0.0).rotate(this);  
    Vector3 z = new Vector3(0.0, 0.0, 1.0).rotate(this);  
  
    return new PMatrix3D(  
        x.x, x.y, x.z, 0.0,  
        y.x, y.y, y.z, 0.0,  
        z.x, z.y, z.z, 0.0,  
        0.0, 0.0, 0.0, 1.0  
    );  
}
```

applyMatrix()で使える

# 勝ち

```
// Create a new quaternion by given angle and axis.  
Quaternion quaternionByAngleAxis(float _t, Vector3 _a) {  
    return new Quaternion(  
        cos(_t / 2.0),  
        _a.scale(sin(_t / 2.0))  
    );
```

quaternion\_interactive.pde

軸と角度から  
クオータニオン生成

## 回転

quaternion\_interactive.pde

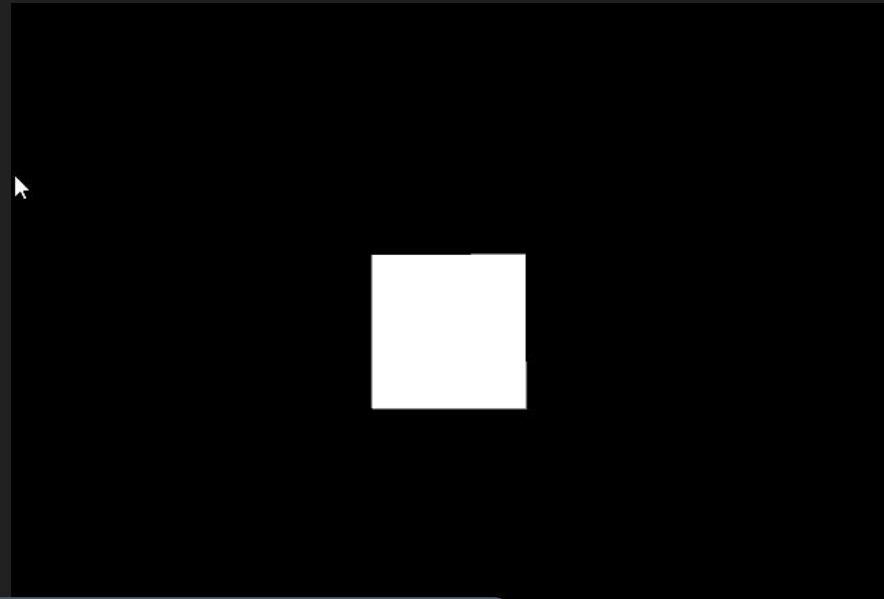


自由な回転が実現できた

## 回転

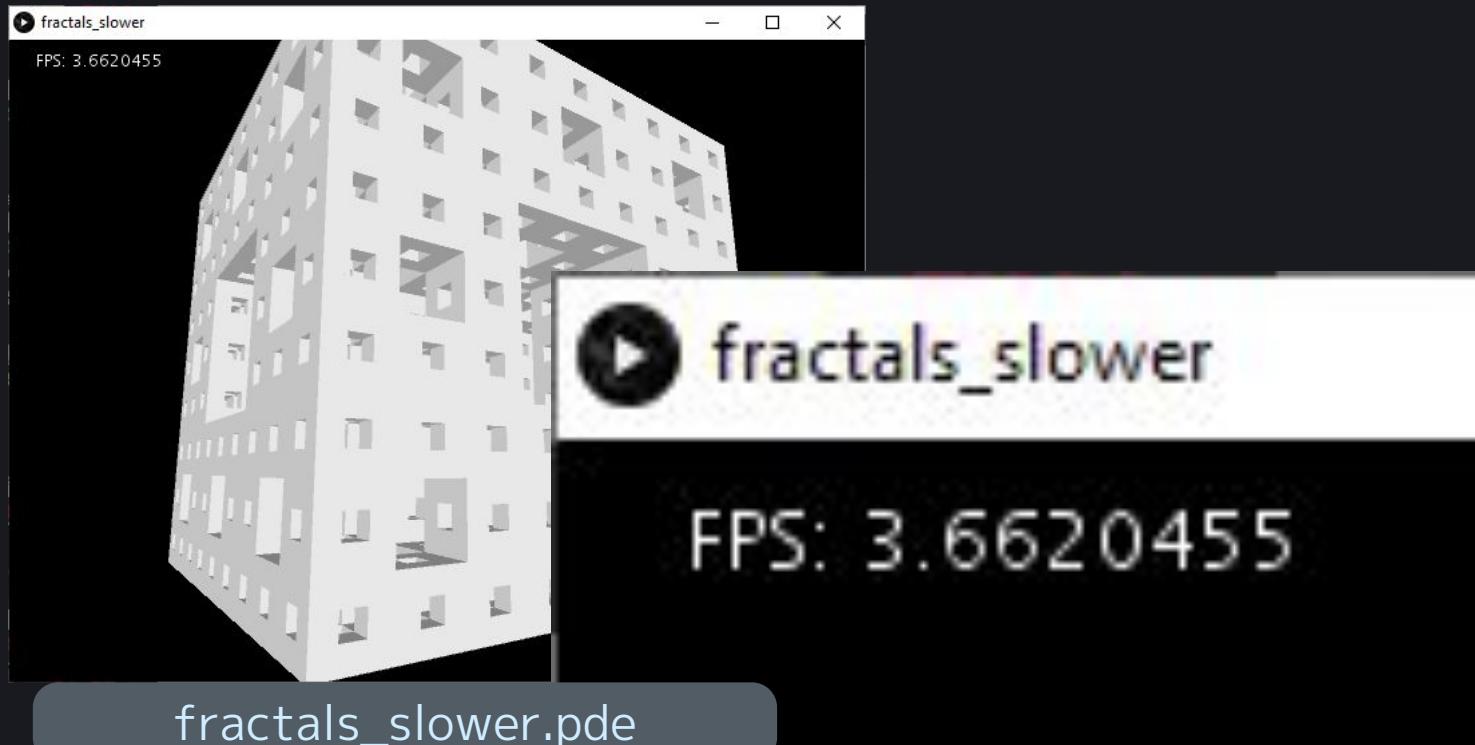
Further:  
Slerp関数

2つの回転状態間を  
線形に補間する関数



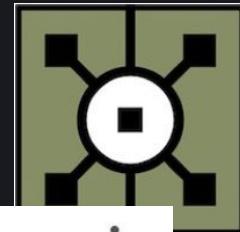
quaternion\_slerp.pde

## パフォーマンス



## パフォーマンス

This is



three.js

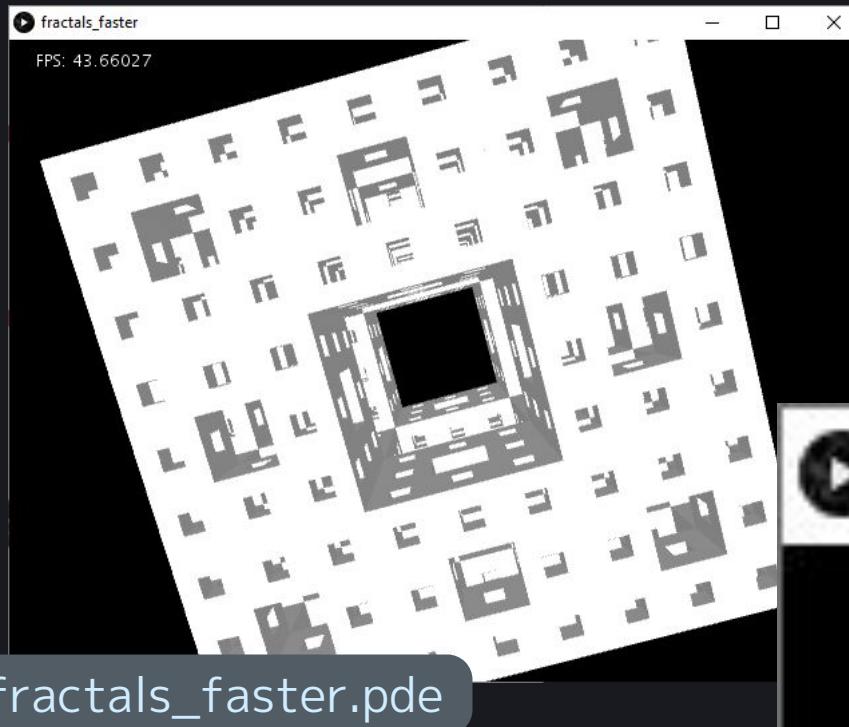


パフォーマンス

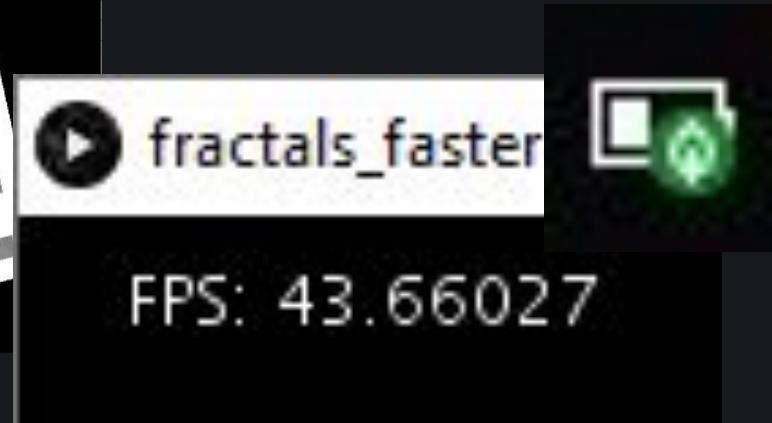


から逃げるな

## パフォーマンス



fractals\_faster.pde



パフォーマンス

## 基本方針

### GPUを意識しよう

- できるだけ事前に計算！
- ドロー コールを少なく！
- CPUに全部任せると！



## パフォーマンス

できるだけ事前計算！

```
41 void draw() {  
42     lights();  
43  
44     background(0);  
45  
46     pushMatrix();  
47     translate(width / 2, height / 2);  
48     rotateX(frameCount / 300.0);  
49     rotateY(frameCount / 100.0);  
50     drawFractals(0, 0, 0, 300, FRACTAL_ITER);  
51     popMatrix();  
52  
53     text("FPS: " + frameRate, 20, 20);  
54 }
```

fractals\_slower.pde

再帰的に呼び出され  
160,000個のboxが  
生成される

パフォーマンス

できるだけ事前計算！



## パフォーマンス

できるだけ事前計算！

PShapeにあらかじめ  
フラクタルを入れておこう

```
5 void setup() {  
6     size(640, 480, P3D);  
7  
8     myShape = createShape();  
9     myShape.beginShape(QUADS);  
10    myShape.fill(255);  
11    myShape.noStroke();  
12    addFractals(myShape, 0, 0, 0, 300, FRACTAL_ITER);  
13    myShape.endShape();  
14 }
```

fractals\_faster.pde

## パフォーマンス

できるだけ**事前計算**！

注: この場合、フラクタルを  
あとから変形することはできない



## パフォーマンス

ドローコールを少なく！

ドローコール？

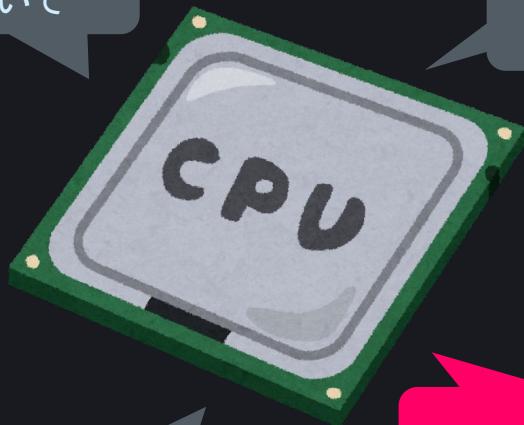


## パフォーマンス

ドローコールを少なく！

この変数  
覚えといで

頂点情報  
あげる



テクスチャ情報  
あげる

アレ描いて！

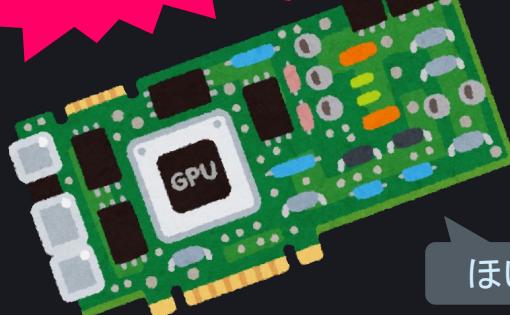
ドローコール

重労働

!

ほい

ほい



## パフォーマンス

ドローコールを少なく！

shapeは

**呼ばれるたびにドローコールが走る**

逆に box, sphere, ellipse 等は  
結構ガンガン使ってしまってOK  
(呼んだ段階では一旦蓄積されるぽい)

PGraphicsOpenGL.shape  
PGraphicsOpenGL.box

## パフォーマンス

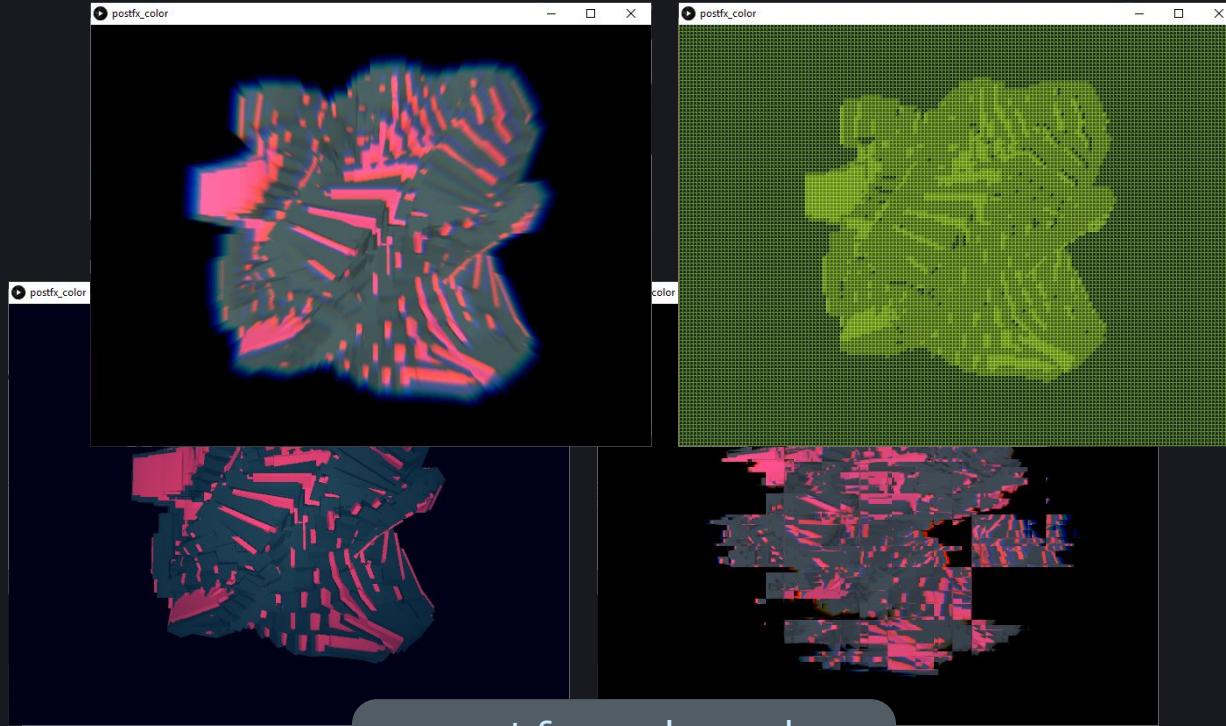
CPUに全部任せるな！

(later)

シェーダ

(most fun part of the workshop)

# シェーダ



postfx\_color.pde

シェーダ

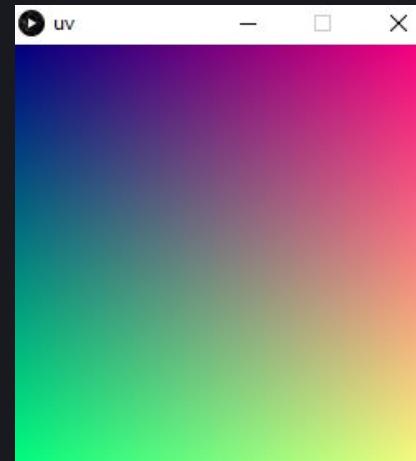
そもそも、なぜシェーダを扱うのか？

# シェーダ

シェーダでできることは大体  
Processingでもできる

```
uv
1 size(256, 256);
2
3 loadPixels();
4 for (int iy = 0; iy < height; iy++) {
5     for (int ix = 0; ix < width; ix++) {
6         float r = ix * 256.0 / width;
7         float g = iy * 256.0 / height;
8         float b = 127.0;
9         pixels[ix + iy * width] = color(r, g, b);
10    }
11 }
12 updatePixels();
```

uv.pde



シェーダ

ただし遅い

# シェーダ

```
uv  
1 size(256, 256);  
2  
3 loadPixels();  
4 for (int iy = 0; iy < height; iy++) {  
5     for (int ix = 0; ix < width; ix++) {  
6         float r = ix * 256.0 / width;  
7         float g = iy * 256.0 / height;  
8         float b = 127.0;  
9         pixels[ix + iy * width] = color(r, g, b);  
10    }  
11}  
12 updatePixels();
```

uv.pde

この処理は  
width \* height回  
行われる

# シェーダ

ここでシェーダのコードを見てみましょう

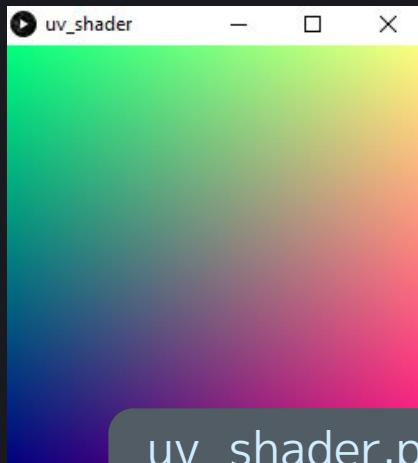
```
1 uniform vec2 resolution;  
2  
3 void main() {  
4     vec2 uv = gl_FragCoord.xy / resolution;  
5     gl_FragColor = vec4(uv, 0.5, 1.0);  
6 }  
7
```

for文がない！！！

uv\_shader.pde

# シェーダ

シェーダの世界では  
コードが並列に処理される

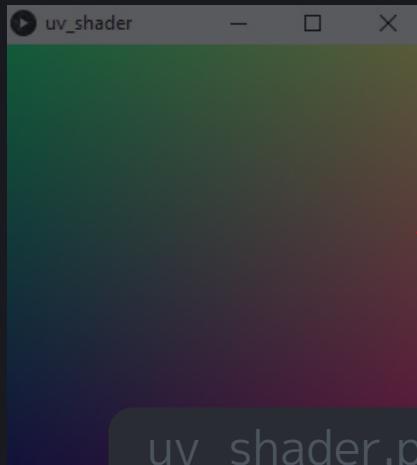


uv\_shader.pde

全ピクセルに対して  
**void main()** が実行される！

# シェーダ

シェーダの世界では  
コードが並列に処理される



uv\_shader.pde



## パフォーマンス

CPUに全部任せるな！

GPUで並列処理できる仕事は  
できるだけGPUに任せよう



vs.



シェーダ

まずはシェーダを書くことを考える前に  
シェーダを**道具**として捉えよう

# シェーダ

シェーダを**道具**として使う

postfx\_color drawExample

```
1 PShader colorShader;
2
3 void setup() {
4     size(640, 480, P3D);
5     colorShader = loadShader("color.frag");
6 }
7
8 void draw() {
9     background(0);
10    drawExample();
11
12    colorShader.set("resolution", float(width), float(height));
13    filter(colorShader);
14 }
15
```

postfx\_color.pde

使いたいシェーダを読み込む

画面全体に使う！

シェーダにパラメータを送る

シェーダ

ブラー

filter(**B**)  
mouseY);

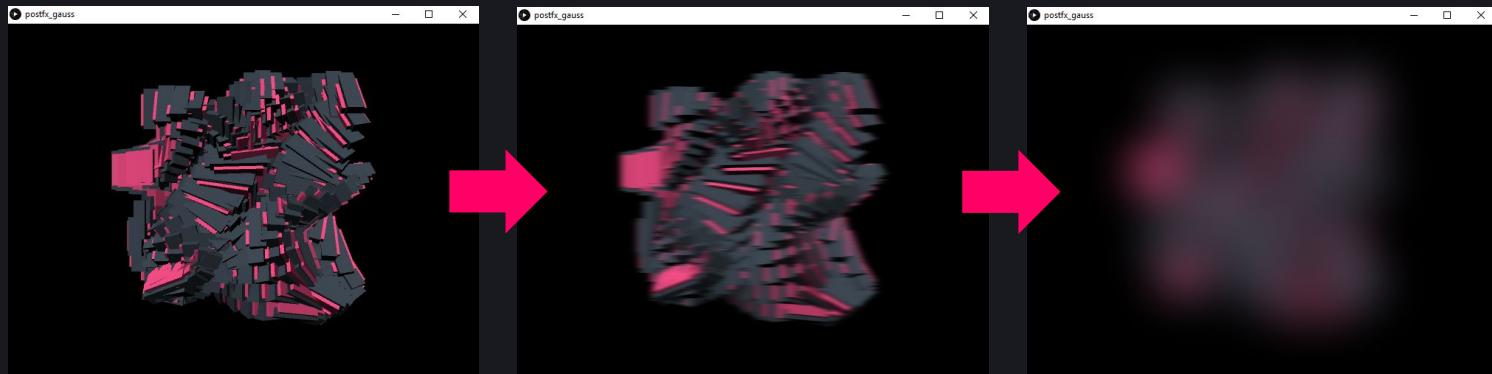
gauss\_slower.pde

シェーダ

ブラーもシェーダでめちゃ速くなります

# シェーダ

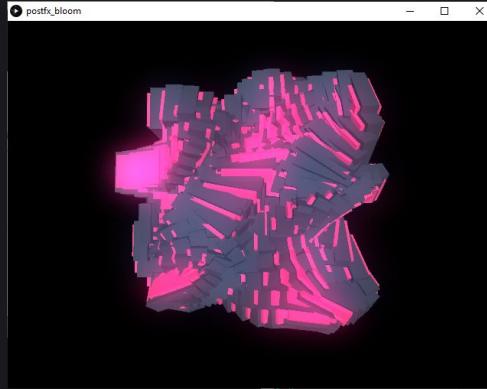
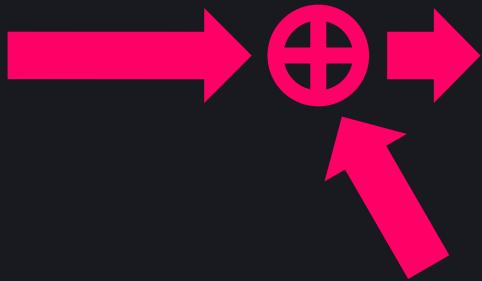
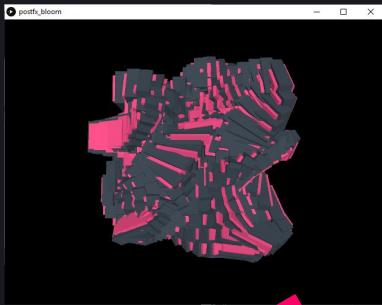
ガウスブラーを2パスで実現  
(2パス … シェーダの二度がけ)



postfx\_gauss.pde

# シェーダ

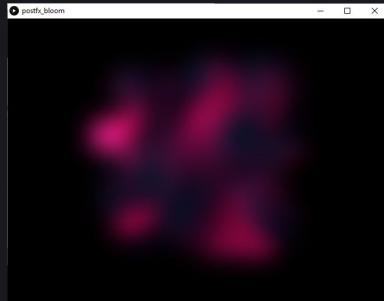
postfx\_bloom.pde



明るい部分を抽出



→  
ガウス  
ブラー



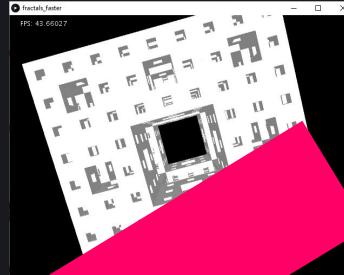
## シェーダ

ここまでがFragment Shaderのおはなし

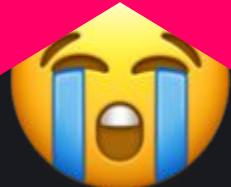
シェーダ

## パフォーマンス

できるだけ事前計算！



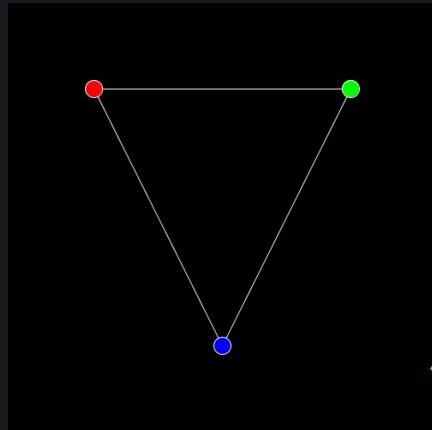
注: ごくまれに、フレームレートを  
あとから変えることはできない



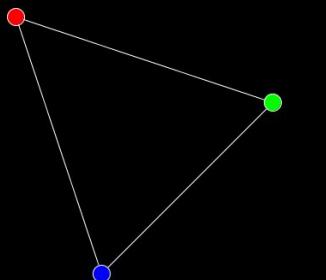
シェーダ

Introducing Vertex Shader

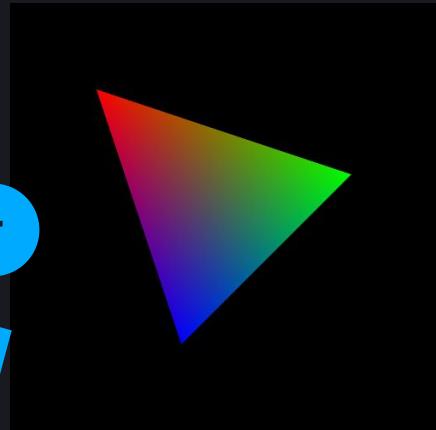
# シェーダ



Fragment Shader

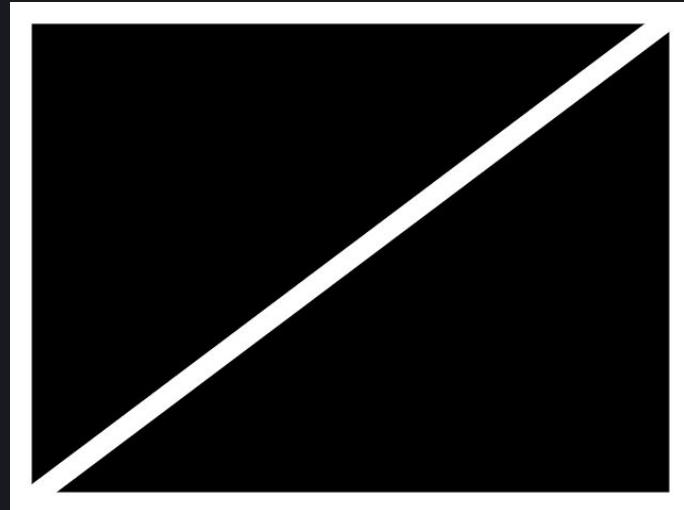


Vertex Shader



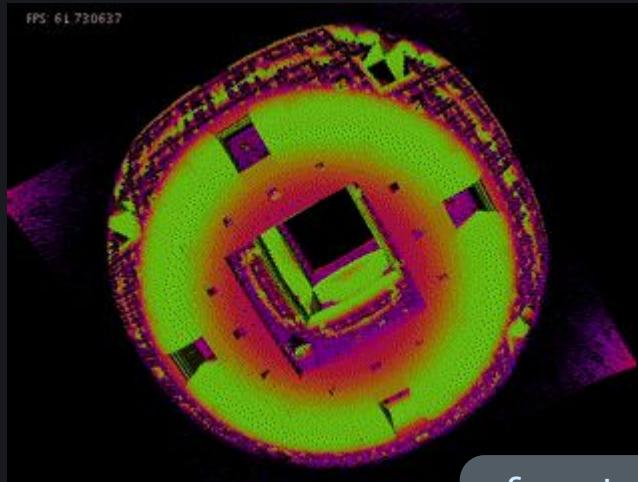
## シェーダ

さっきまでの filter は  
このような図形に対して絵を書いていただけ



## シェーダ

### 作例1 wave.vert

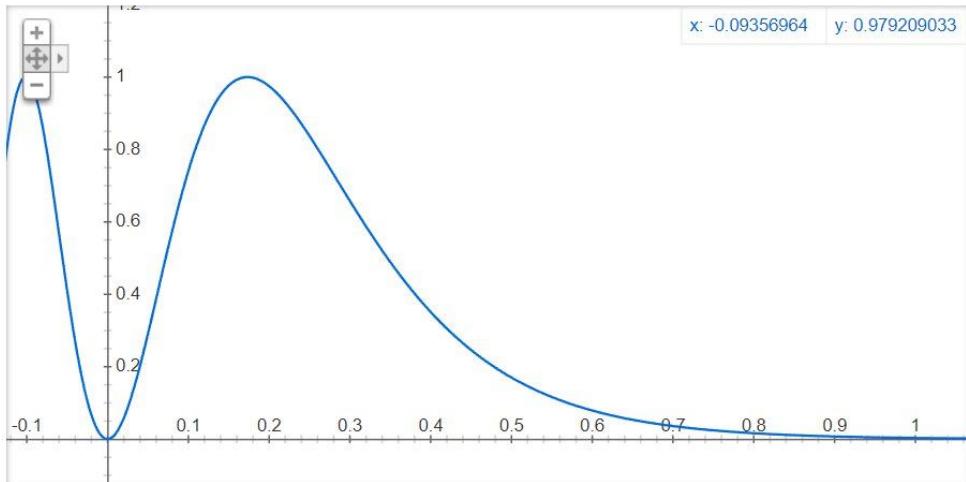


- 中心からの距離を用いてアニメーションを行う
  - 大きさが変わる
  - 色も変わる

fractals\_shader.pde

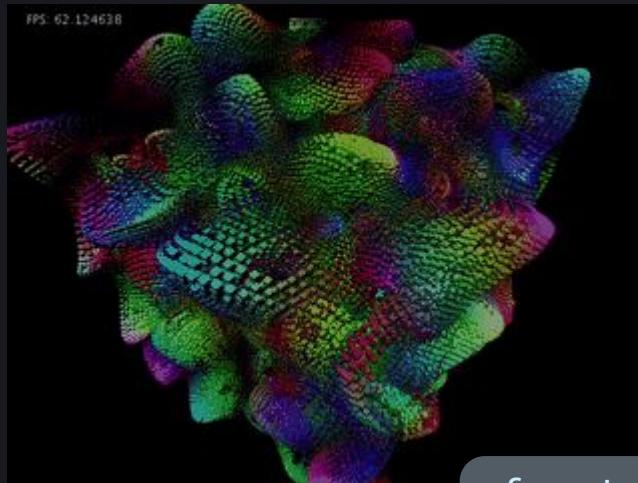
# シェーダ

グラフ:  $0.5 - 0.5 \cos(2\pi \exp(-4)x)$



## シェーダ

### 作例2 noise.vert



- ノイズ関数を用いてアニメーションを行う
  - 位置が変わる
  - 大きさも変わる
  - 色も変わる

fractals\_shader.pde

# シェーダ



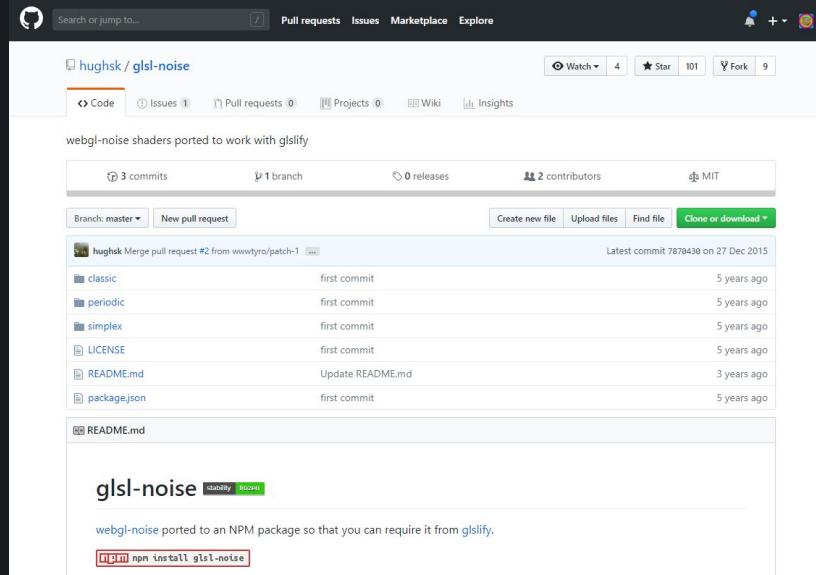
```
noise
size(100, 100);

2
3 loadPixels();
4 for (int iy = 0; iy < height; iy++) {
5   for (int ix = 0; ix < width; ix++) {
6     float v = 256.0 * noise(0.1 * ix, 0.1 * iy);
7     pixels[ix + iy * width] = color(v, v, v);
8   }
9 }
10 updatePixels();
11
12
13
14
```

noise.pde

# シェーダ

## 今回はあるものを使おう



まとめ

ニヤーン