

フラグメントシェーダを使いこなそう

シェーダーアートの世界

前回のおさらい

前回は頂点シェーダをテーマに、主に「頂点を動かすこと」について掘り下げました。

3DCG やシェーダを使った制御のなかで、頂点そのものを変換し加工していくその過程はとても重要なプロセスです。数学的な話はやや難しい部分も多いのですが、意外にそれらは単純な計算で成り立っている場合も多いです。

“ 焦らずじっくりと取り組み、楽しい気持ちで継続することが大切です。 ”

そして今回は、フラグメントシェーダを用いた様々な処理について見ていきます。

前回扱ったテクスチャによる表現もフラグメントシェーダを用いた処理でしたが、今回はもう少し踏み込んだ、応用的な内容が中心になります。

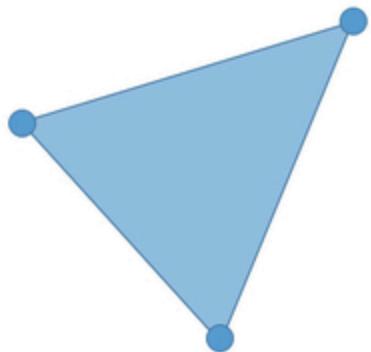
フラグメントシェーダの特性

さて、まず最初に簡単にですがフラグメントシェーダの特性についておさらいしておきましょう。

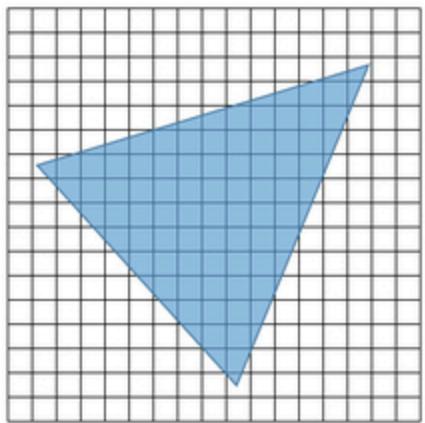
これは第一回で触れた内容になるかと思いますが、頂点シェーダとフラグメントシェーダのそれぞれの役割の違いを憶えているでしょうか。

フラグメントシェーダは、頂点シェーダが座標変換などを行ったあと、それがラスタライズされて塗りつぶされるピクセルが確定したあと呼び出されるのでした。

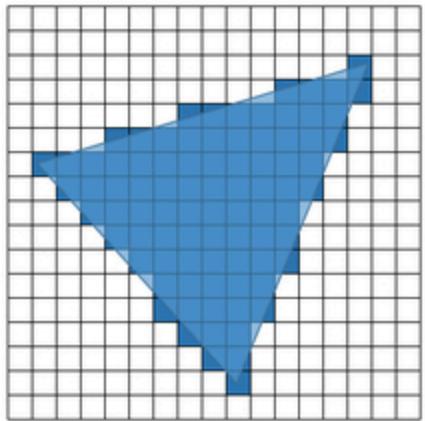
そして、色を塗ることになったピクセルでは、まったく同じフラグメントシェーダのコードが、すべての対象ピクセル上で実行されることになります。



三次元空間に置かれた
頂点やポリゴンが.....



ラスタライズされて
塗るべきピクセルが
決まり.....



対象となるピクセル
全てが同じフラグメント
シェーダで処理される

フラグメントシェーダに限りませんが、どうしてもシェーダは実行される回数が膨大なため、パフォーマンスに対してシビアに考えざるを得ないところがあります。そのような背景もあり世の中にある作例の多くは、シェーダを最適化して「独特なクセのあるコード」になっている場合が多いです。

シェーダに不慣れなうちからあまりにこれを強く意識してしまうと、筆が止まるというか、思い切ったコードが書けなくなり、結果行き詰まってしまうことがどうしても増えます。

最適化とは文字通り「既にあるもの・一度できあがったものを最適な状態に修正していく」ことですから、まずは自分の思った通りに動くこと、狙ったとおりの効果が得られることを重点に置きましょう。

結果、とんでもない重さのすさまじいシェーダができてしまったとしても、絵がなにも出でていないより遙かにマシです。そこから少しづつ、最適化などはあとから考えていいければいいのです。

シェーダーの世界

シェーダアート、と一口に言っても、明確にそういうアートのジャンルが定義されているかというと（教科書に載るような明確な）定義があるわけではありません。

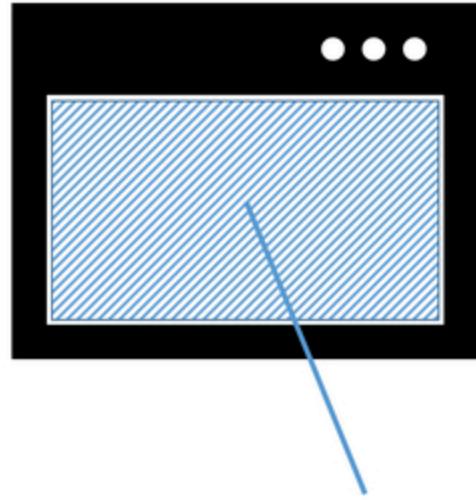
しかし近年、特に WebGL が登場したことによって、1つの表現の分野として徐々に浸透してきているような感じがします。

明確な定義こそありませんが、多くの場合、シェーダー・アートというと「フラグメントシェーダのみを利用して絵作りを行う」ことを指している場合が多いです。

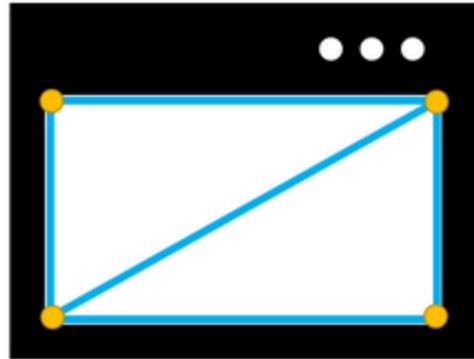
とは言えフラグメントシェーダだけで絵作り、と言われてもそれが何を指しているのかちょっとわかりにくいくらいかもしれません。

WebGL では頂点シェーダを使わずに描画することはできないので、頂点を頂点シェーダで処理するという今までの原則は変わりません。

画面いっぱいにピッタリと収まる板のようなポリゴンを描画することで、結果的に「画面上のすべてのピクセルに対してフラグメントシェーダを実行する」というのが基本的なアイデアです。



スクリーンの全体（すべてのピクセル）でまったく同じ
フラグメントシェーダを実行させるにはどうしたら？



四角形のポリゴンを、領域全体にぴったり重なるように
描画してやることで、結果的に、このポリゴンの描画に
利用したフラグメントシェーダが画面全体に適用される。

Shader of the Week



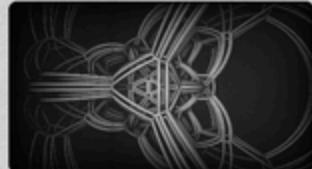
うろうろシェーダー 241201 by TheBoiledHotWater

1062 28

Featured Shaders



Electrocardiogram_Loewe by Loewi 7258 66



Polychora (4D) by Syntopia 8792 94



Elevator music by zelotochoy 4121 15



Globby Gyroscope by zackpudil 3761 21

Build and Share your best shaders with the world and get Inspired

 PayPal Donate

Become a patreon

Latest contributions: "Fractions '24" by zadgy5534 47 minutes ago, "Is this also a moire ?" by zadgy5534 55 minutes ago, "Shader Assignment 1.1.1.1" by cwalton20 2 hours ago, "hat polykite" by Boogalo 2 hours ago, "Phyllotactic spiral demo" by stegu 3 hours ago

Shadertoy BETA

最もハイエンドなシェーダーアートの最高峰

一般によく言われるシェーダーアートとは

- シェーダーアートではフラグメントシェーダのみを用いて絵作りを行う
- 近年では主にウェブを主戦場に多くの作例が生まれている
- 原理的には板ポリゴンを画面ぴったりに描画し.....
- それにより結果的に画面全体でフラグメントシェーダを実行している
- 画面に描いているのはただの板だけであるため、原理や考え方がいわゆるラスタライズ方式の CG とは異なる部分が多い

まずは下準備

シェーダーアートについてざっくりと概要がつかめたら、実際にそれを実装する方法についてさっそく考えてみましょう。

先程も書いたように、シェーダーアートでは大抵の場合「画面全体を覆う板状のポリゴンを1枚だけレンダリング」します。それにより、結果的に画面全体にフラグメントシェーダを実行するわけです。

今回一番最初のサンプルは「シェーダーartのための雛形」となっており、板ポリゴンを一枚だけレンダリングするシンプルな構成です。

まずはこのサンプルを通じて、全体の構造をザッと理解しましょう。

シェーダアート雛形の要点

- 描かれるのは4つの頂点からなる板が1枚のみ（2ポリゴン）
- 正規化デバイス座標空間上にぴったり収まる頂点座標を定義
- 正規化デバイス座標空間は縦横比は無関係に、常に -1.0 ~ 1.0 の範囲を取る三次元空間
- 頂点シェーダでは行列などは使わず、座標をそのまま出力
- テクスチャ座標に相当する値は、今回のようなケースではわざわざVBOにしなくてもシェーダ内で計算で求めることができる

シェーダアートのような表現方法・技法では、色を見た目の感覚的なニュアンスではなく数値として捉えられるようになってくると上達が早いです

012

- すべてのピクセルでまったく同じシェーダが実行されることを常に意識する（シェーダによって 1px の色が決まっているにすぎない）
- 処理対象のピクセルの座標は `gl_FragCoord` を参照することで知ることができる
- uniform 変数で `resolution` を送ることで `gl_FragCoord` を正規化する（0.0 ~ 1.0 に変換する）ことができる
- JavaScript の `console.log` のような機能は GLSL には存在しないが、計算結果を色として出力することでおおよその値を把握することができる（色デバッグ）

厳密で明確なルールは無い

シェーダアートの世界には、ルールなんてありません。自分で好きなように絵作りできます。考えてみれば紙とペンで落書きをするときだって、そこには「こう描かなくちゃならない」なんてルールはなく……各々が自由にペンを走らせていいわけです。

とは言え、最初は紙とペンだけを渡されても何を描いたらいいのかわからない場合も多いと思います。いくつかのサンプルを見ながらまずは比較的よくある表現に触れてみましょう。

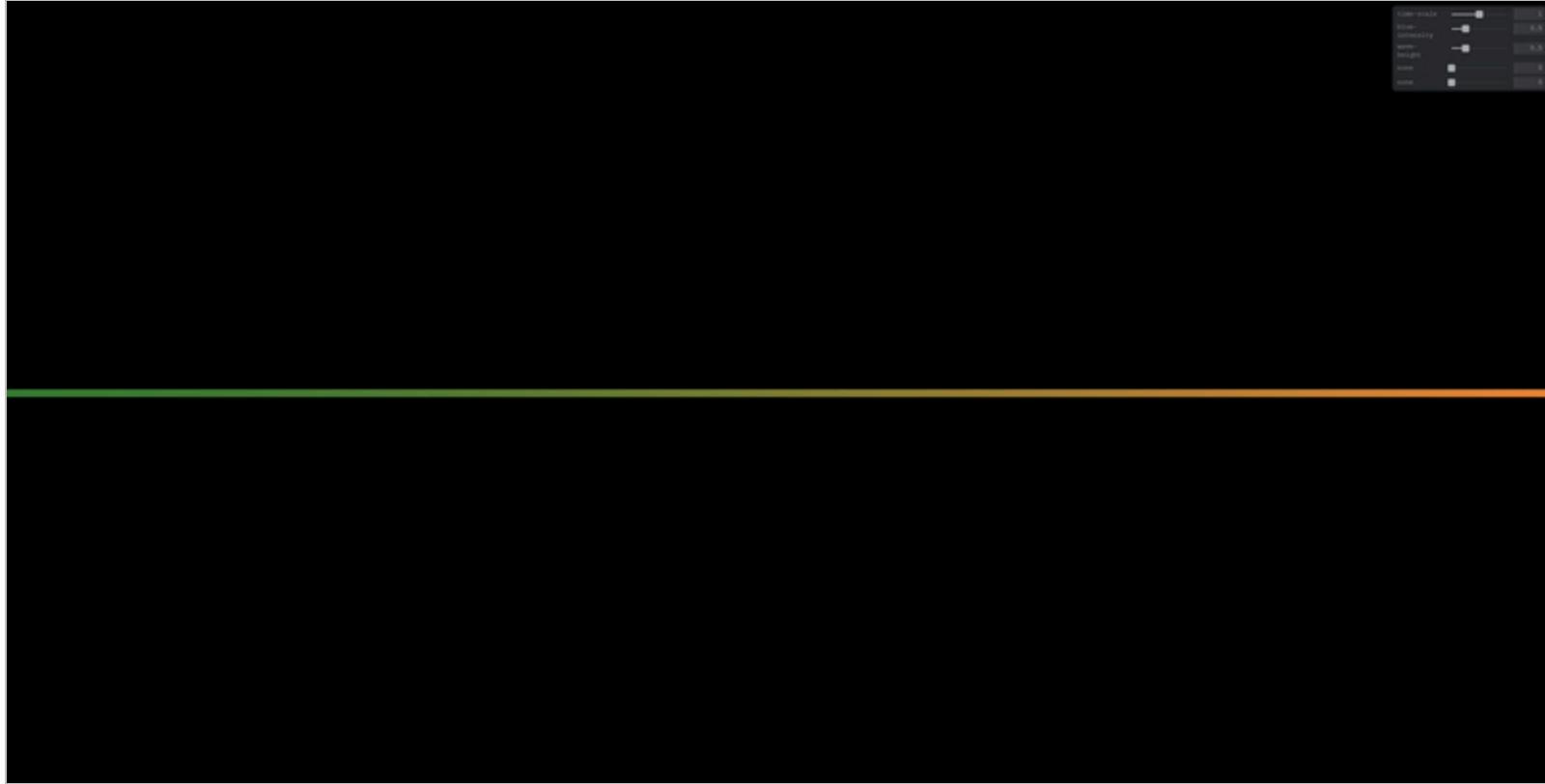
013

- フラグメントシェーダ内でコメントアウトされている箇所があるので、コメント解除しながら実行してみよう
- GLSL のビルトイン関数を使いこなそう
- `sign` は引数に応じて `0.0, 1.0, -1.0` を返す
- `abs` は引数を絶対値にして返す（符号を無視する）
- 時間の経過を上手に活用する

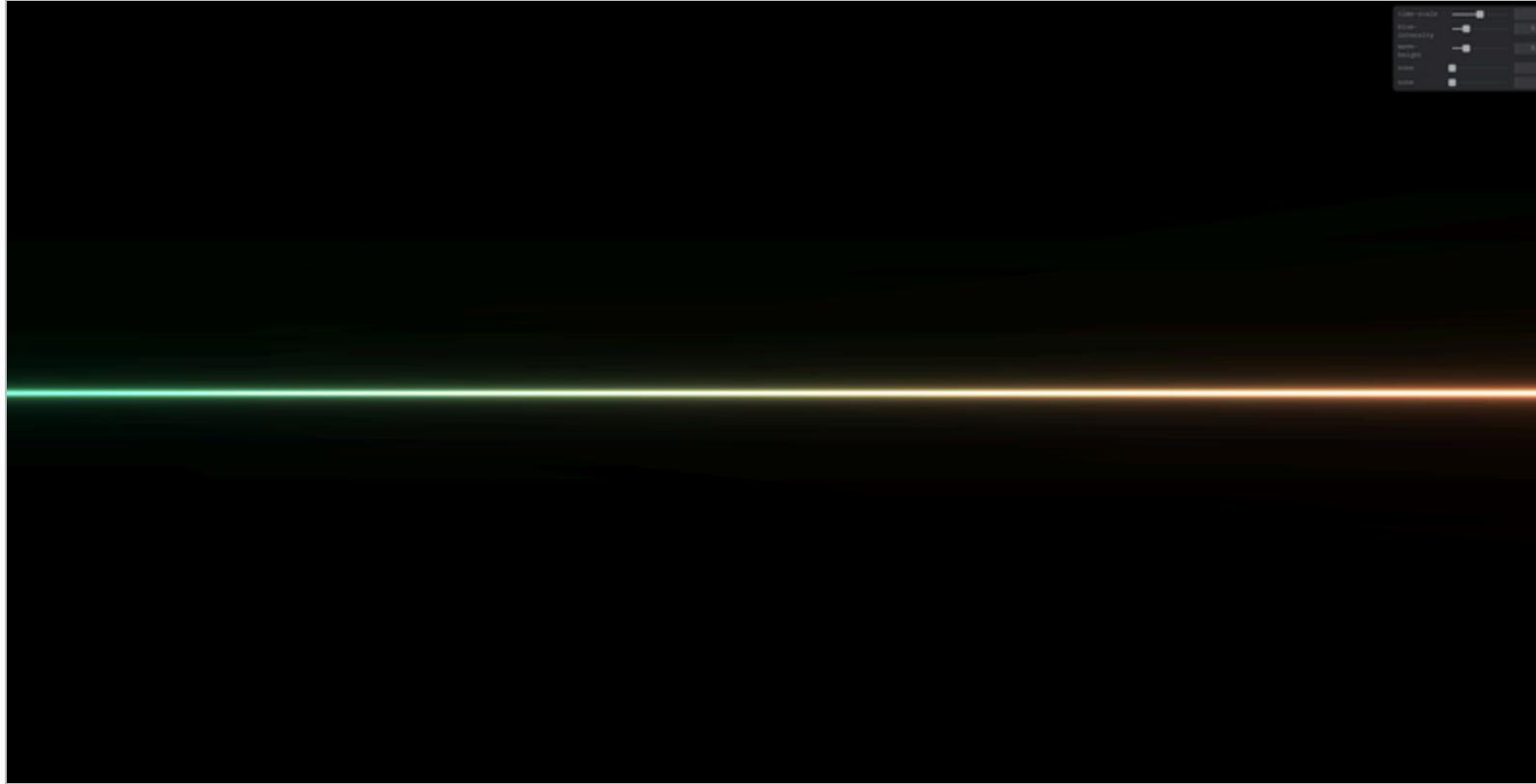
横に長くコードが書かれている場合は、括弧の中身を別の行に切り出すなどして複数行で記述すると、文脈がより理解しやすくなるのでおすすめです



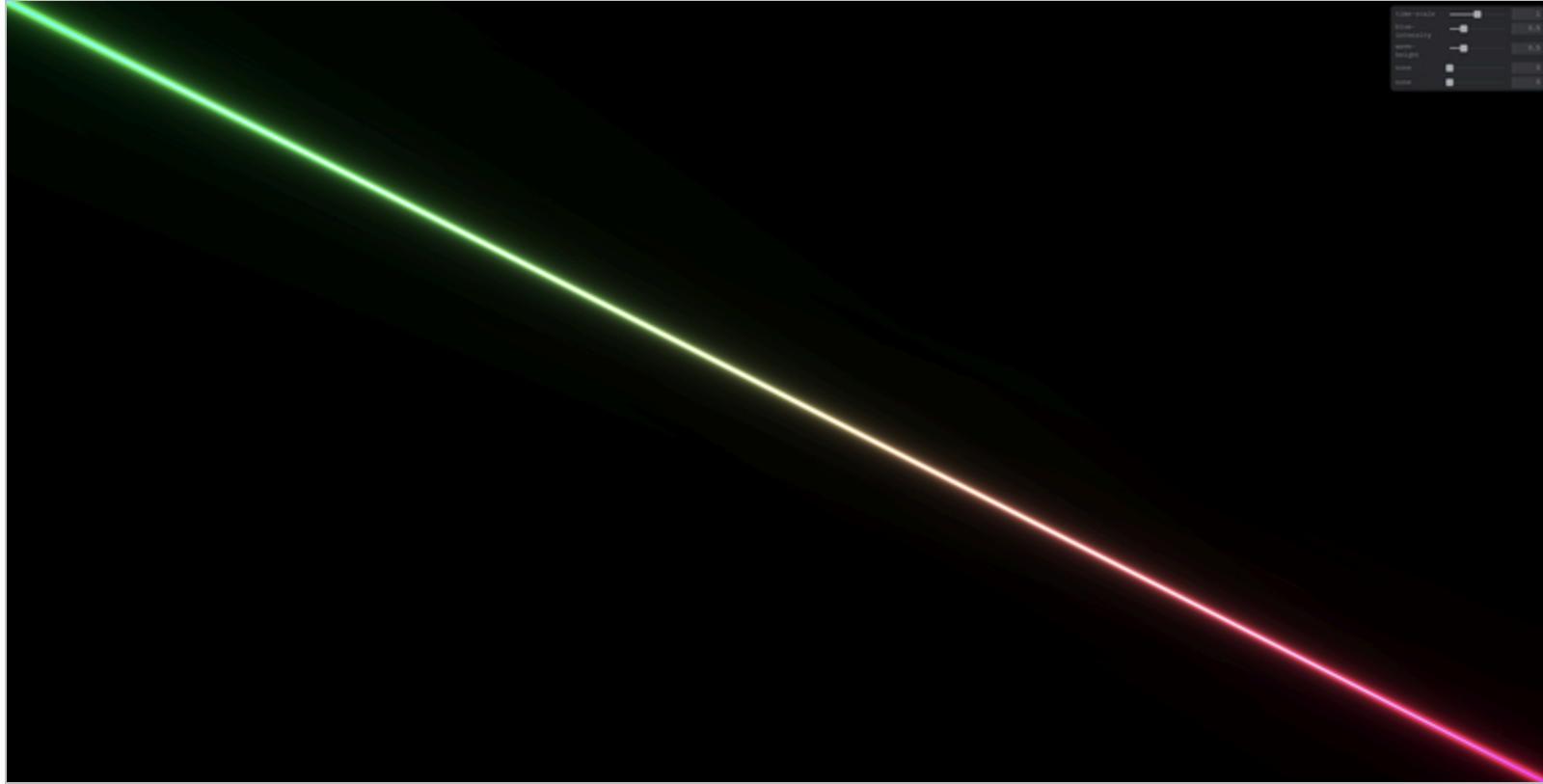
最初の状態では、座標がグラデーションとなり可視化されている



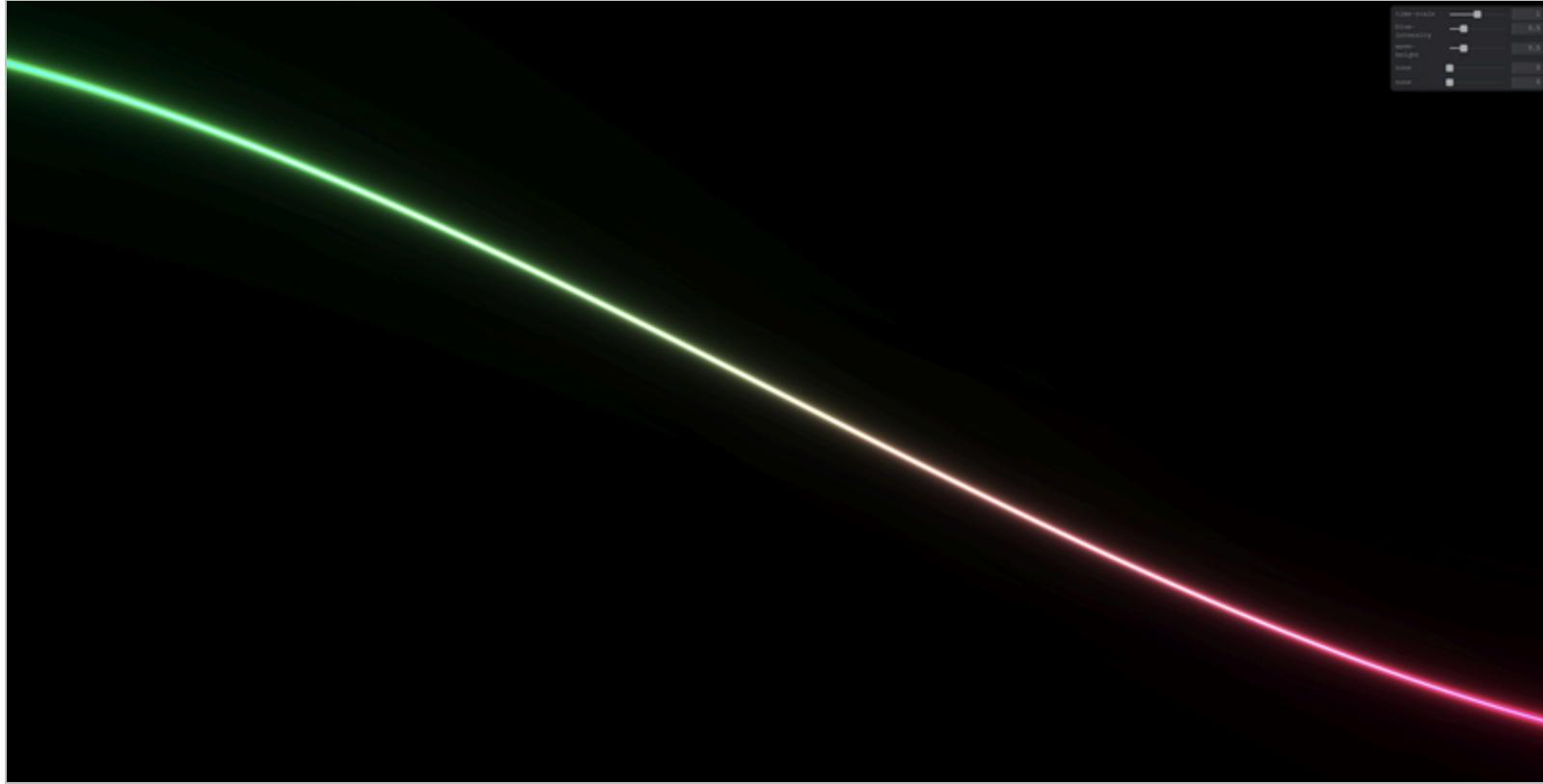
“if による条件分岐は、特定の状況であればビルトイン関数で代替できる



除算の結果が色として可視化されると光っているように見える



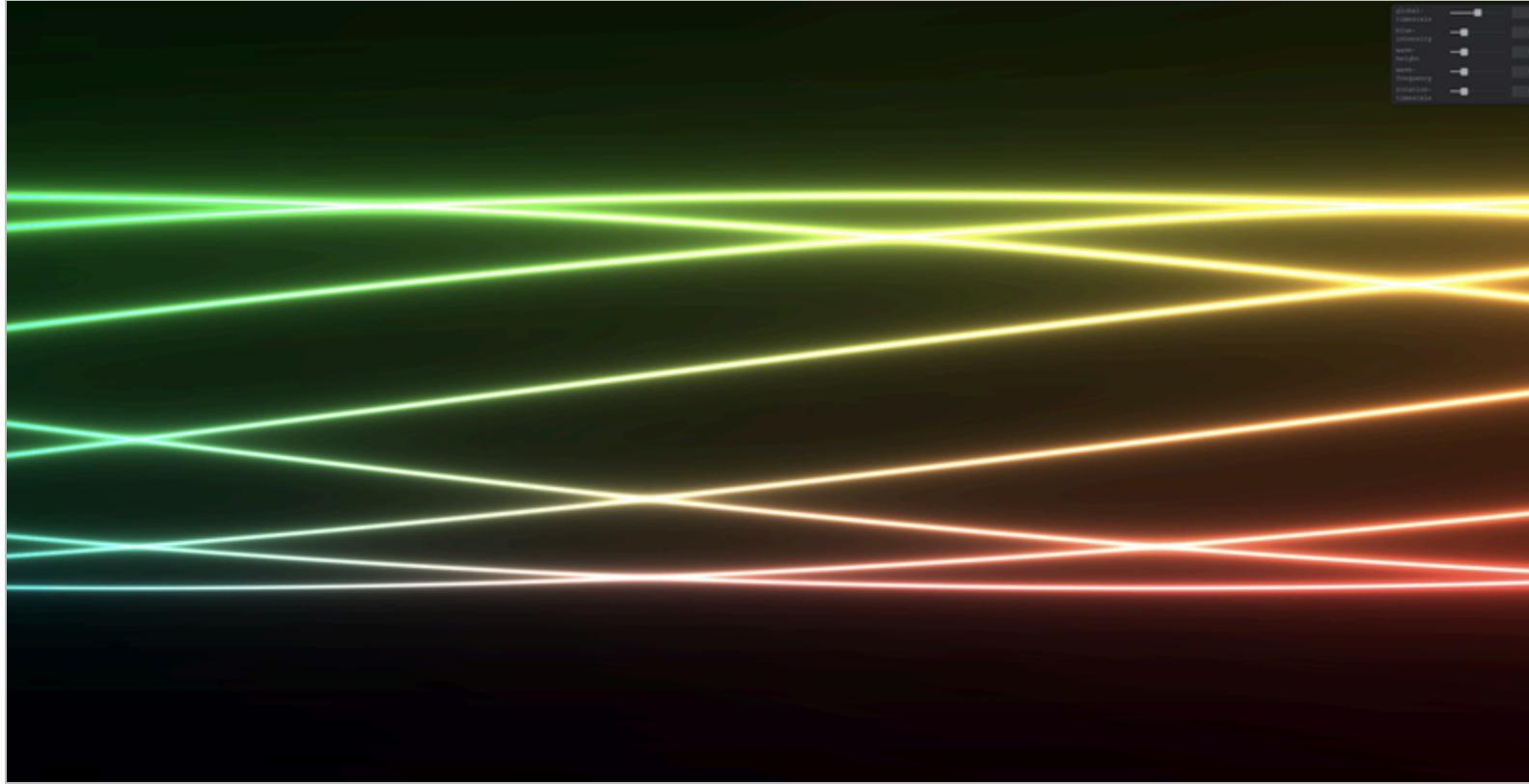
“X 座標を組み合わせると「横方向に変化が表れる状況」を作ることができる



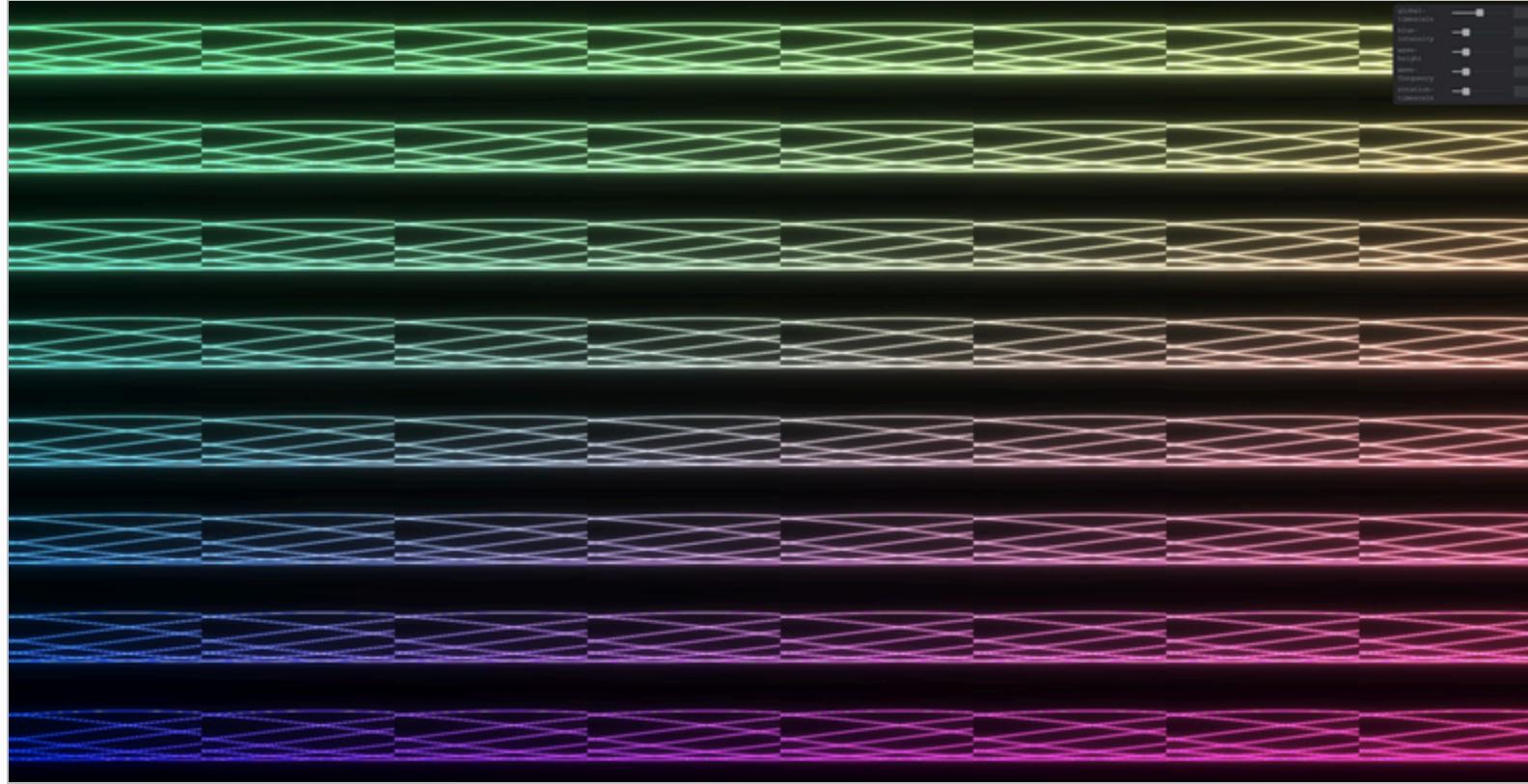
“ サインやコサインはあらゆる表現につながる基礎となる ”

014

- `mod` で剰余計算することで座標を複製できる
- 割った余りを使うシンプルな計算なので落ち着いて考えよう
- 座標を回転させる際は行列を使うのが便利
- 2D の回転行列はシンプルな構造なので暗記してしまうが吉



“for文を使って繰り返すと一気に情報量が増えて派手さがアップ”



除算の剰余を活用することで座標系が複製される

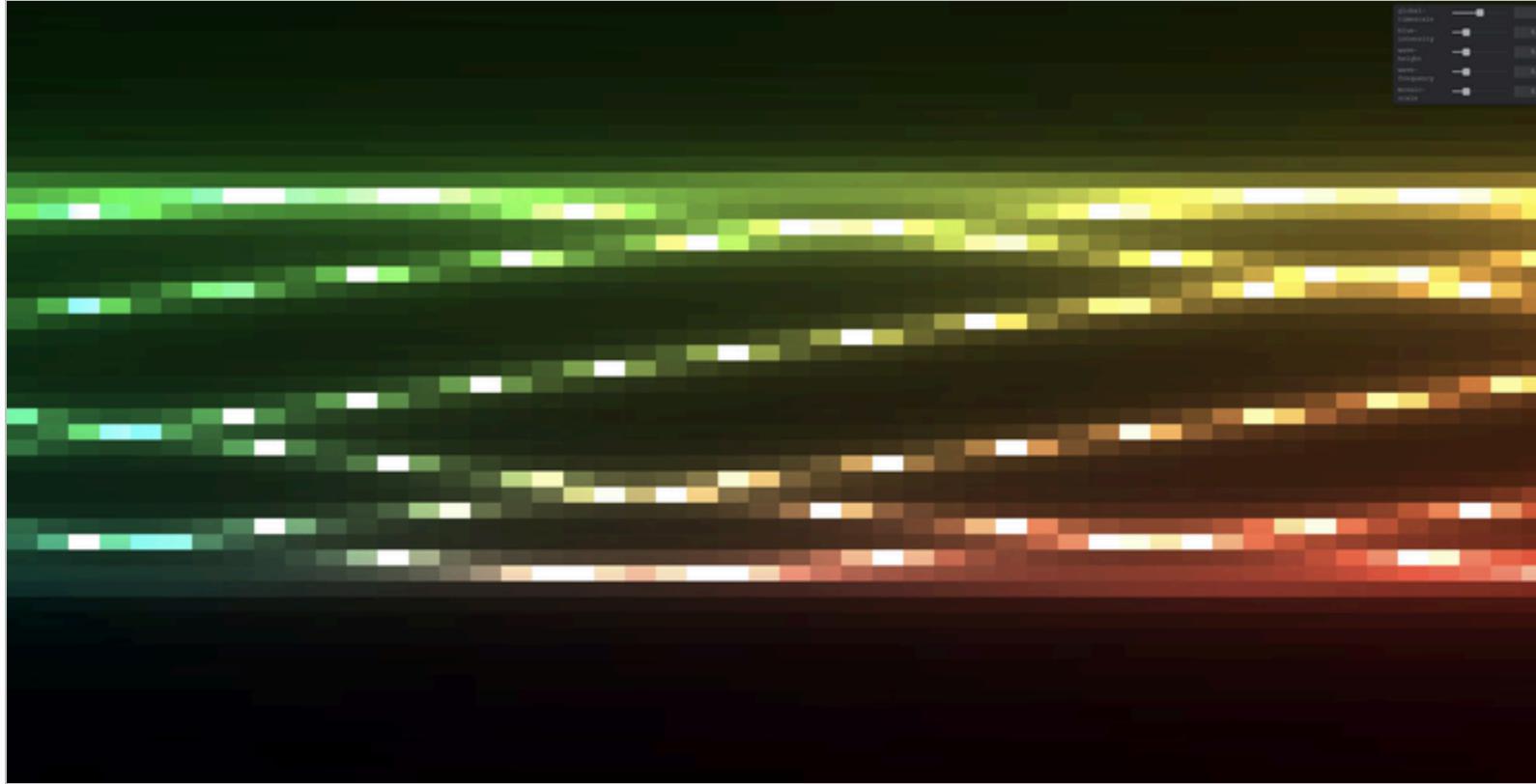


座標の回転には行列が便利

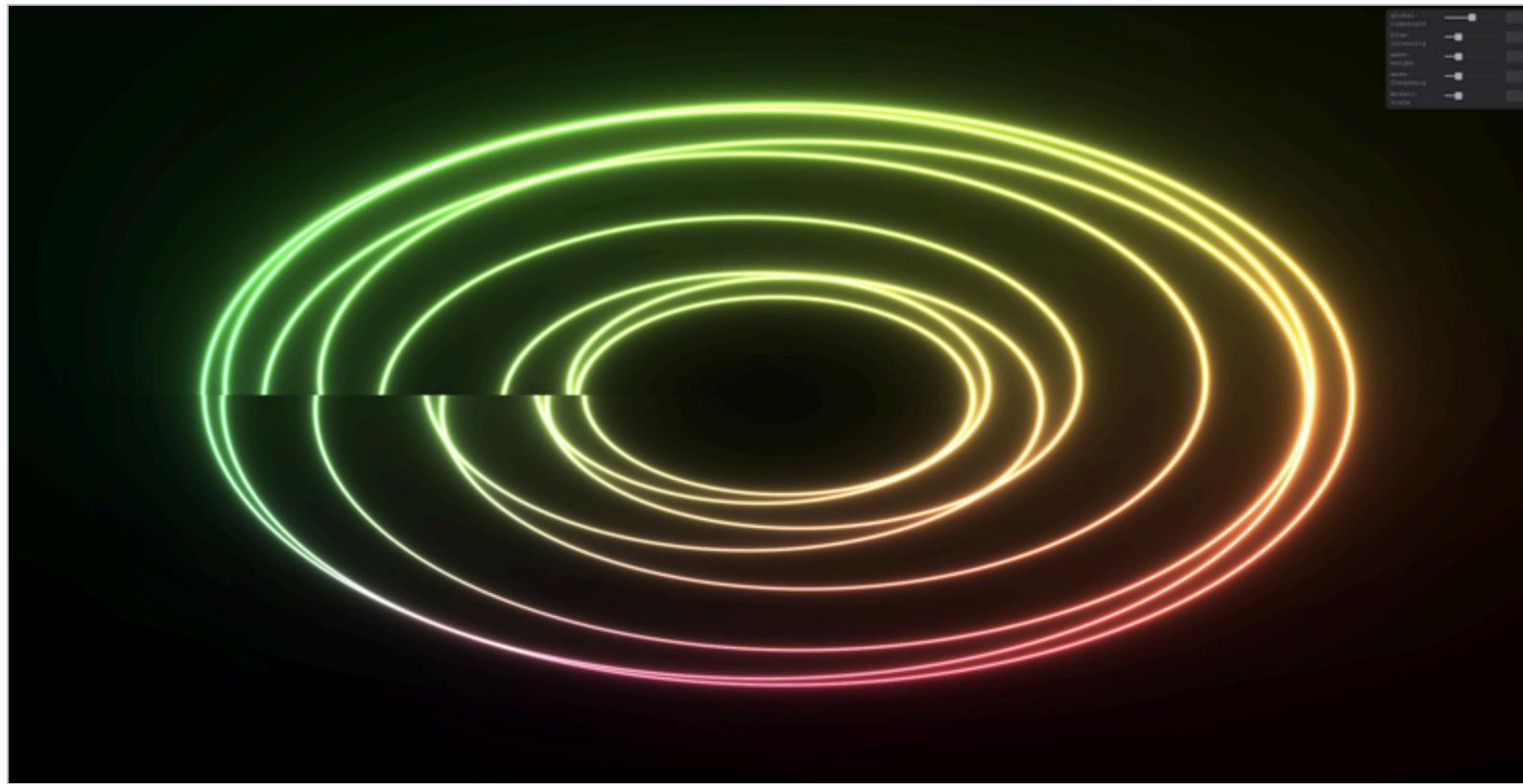
015

- 解像度をあえて落とすモザイク化
- 整数部分だけを抜き出すことで、一定の範囲のテクスチャ座標を固定している（単純計算なので落ち着いて考えよう）
- 極座標は XYZ ではなく角度と半径を使って座標を表す
- 今まで縦横垂直で表されていたものが、円を描くような見た目になる

どんなテクニックもあくまでも「引き出しの 1 つ」なので、パラメータを変化させたり、組み合わせたりして新しい表現を探ってみよう



“モザイク化の考え方は意図的に情報量を減らしたい場合に役立つ”



“極座標に変換することでまったく違った風景が広がる”

レイマーチング

シェーダアートの分野には、それこそルールなんてないので星の数ほどテクニックがあるのですが.....

その中でも、一般によく使われるテクニックに レイマーチング (ray marching) があります。

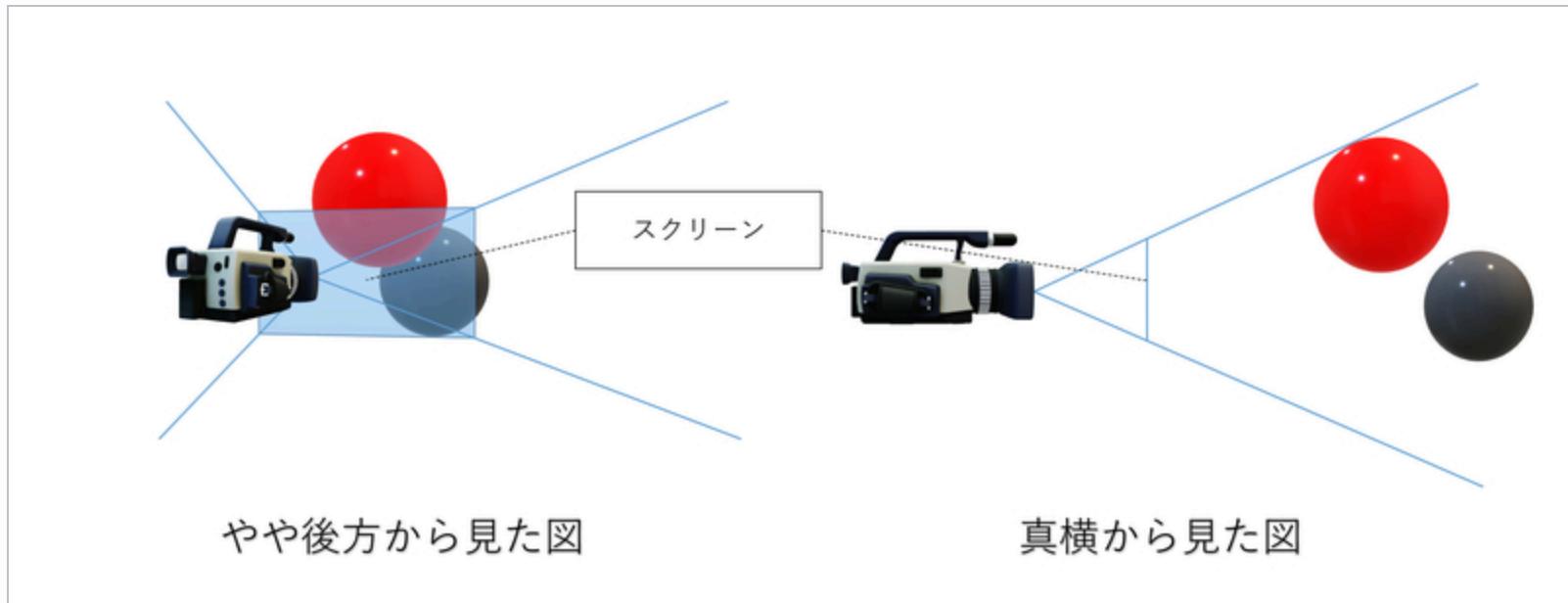
レイマーチングは、一般に「レイトレーシング（ray tracing）」と呼ばれている技術に含まれる技法の一種です。

また、レイマーチングと言ってもそのやり方（実装方法）は1つではなくいくつかのパターンがあり、GLSLでよく見られるレイマーチングの実装は正確には「スフィアトレーシング」と呼ばれる技法です。

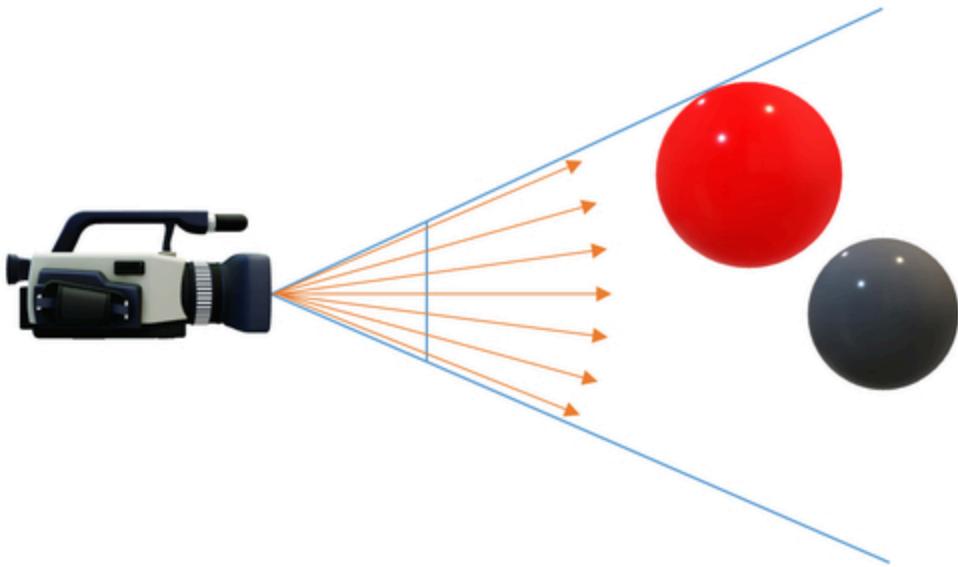
スフィアトレーシングの別名として、レイマーチングという呼称がよく使われている

それでは、スフィアトレーシング (sphere tracing) とはいっていいどんな技法なんでしょうか。

まずは概念を図解して説明します。

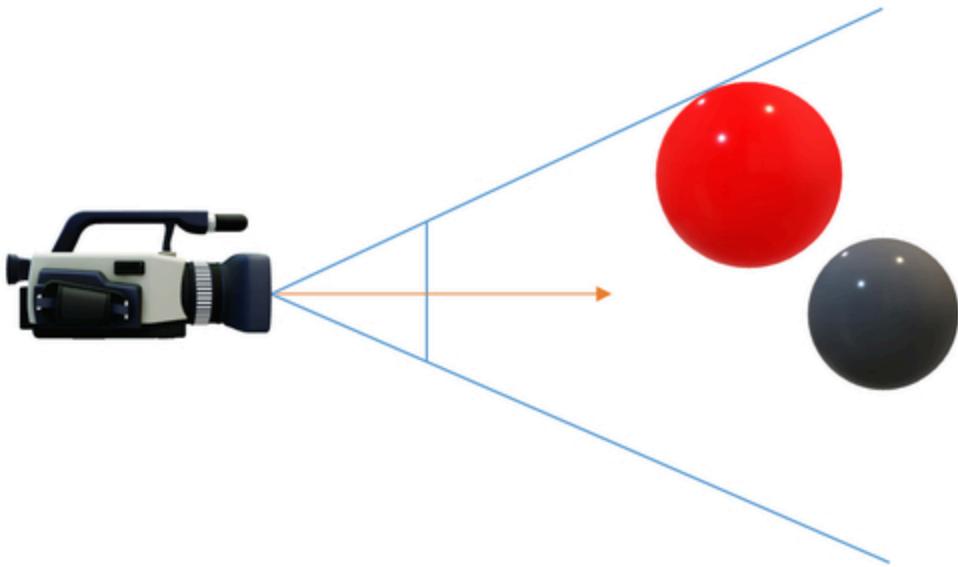


カメラの画角にスクリーンがピタッと収まっている状態をイメージします。このスクリーンは無数のピクセルの集合なので、そのピクセルの1つ1つにレイ（光線）を飛ばします。

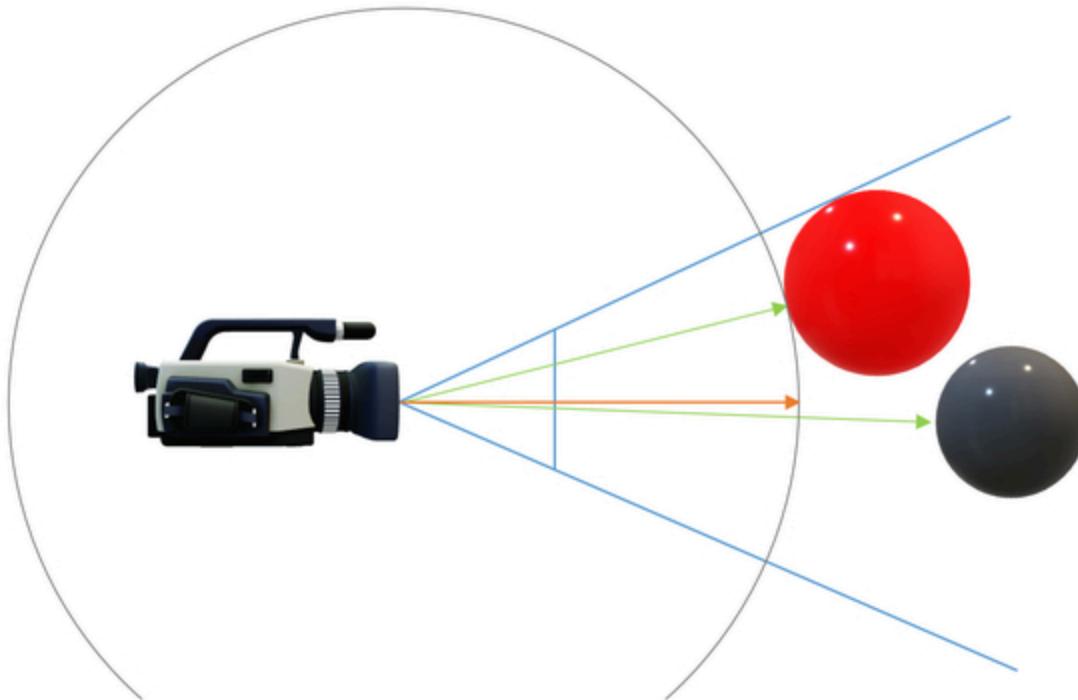


スクリーンのすべてのピクセルに対してレイを飛ばすイメージ図

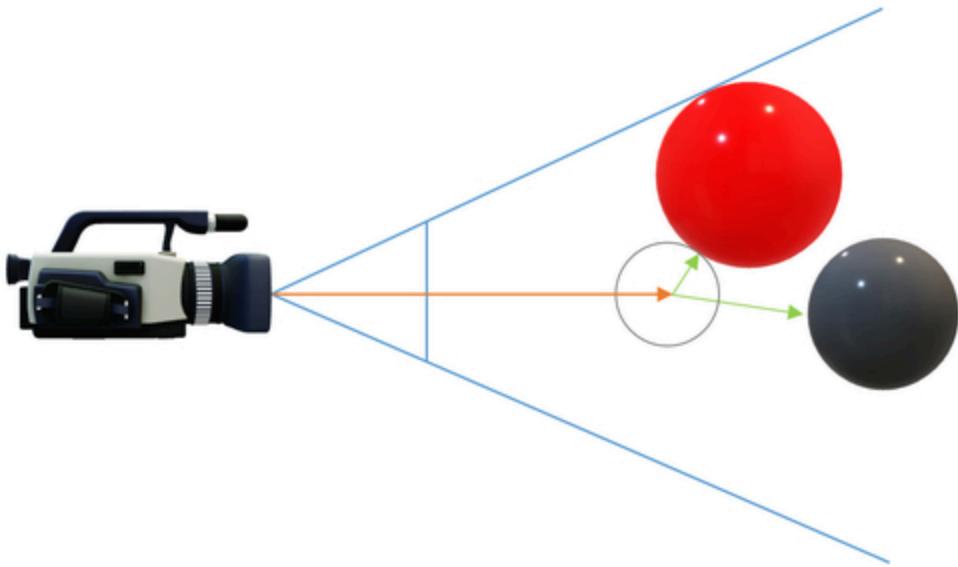
※フラグメントシェーダがすべてのピクセルで実行されるためこのようになる



ここではレイがまっすぐ直進する場合を考える



スフィアトレーシングでは、レイの先端から各オブジェクトまでの距離（緑矢印）を計測し、最も短い距離の分だけレイを進める



レイを進めただけでオブジェクトに衝突しない場合は、再度、
レイの先端から距離を測り、最短距離分を進むことを繰り返す

レイが最も近いオブジェクトまでの距離を頼りに進んでいく様子が可視化される参考デモ。

20221110_raycasting expl

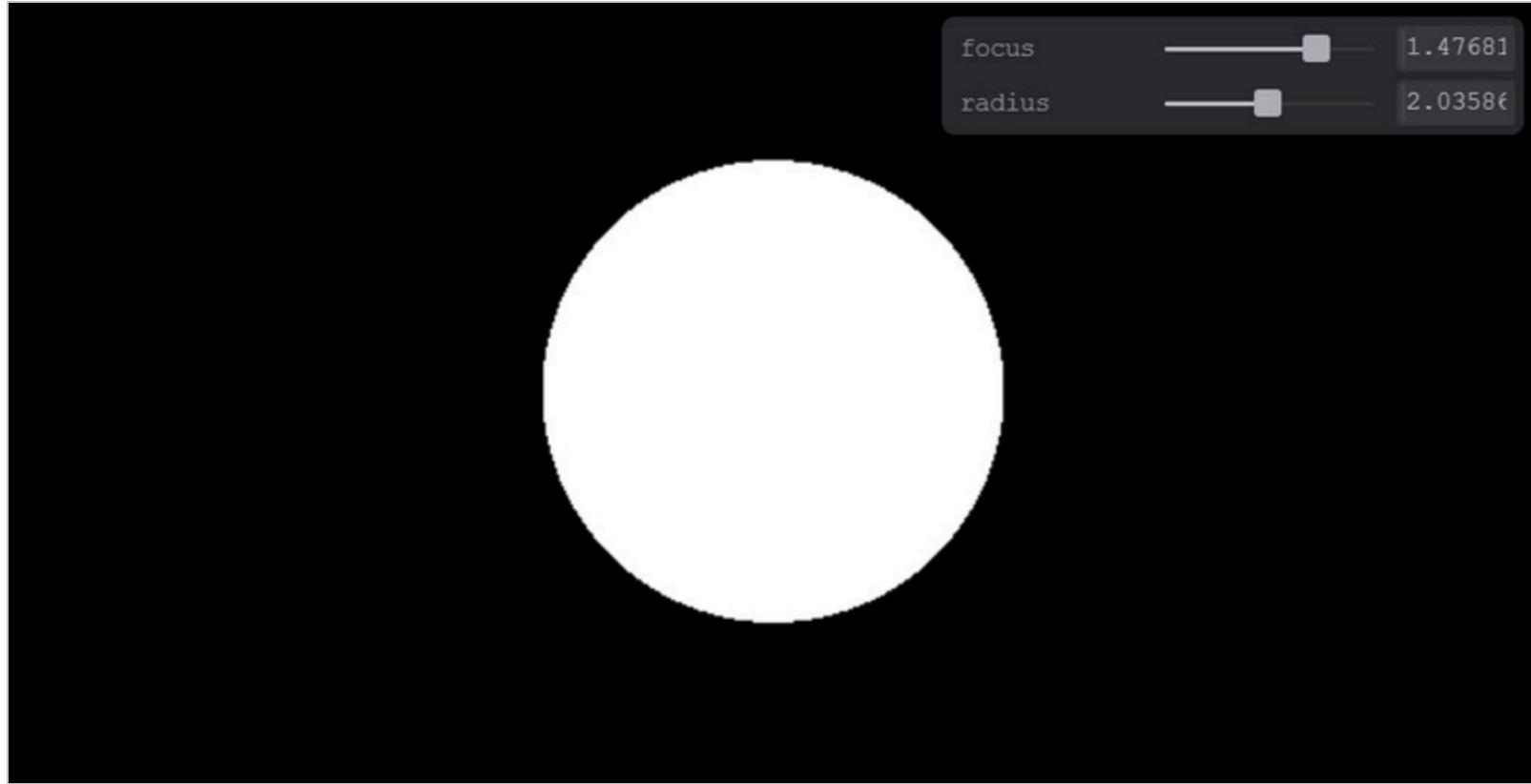
最短距離を進み、また距離を測り…… のプロセスが円で描かれている

言葉で簡単に説明をするとしたら、スフィアトレーシングでは「レイの先端からすべてのオブジェクトに対しての最短距離」を求めてやり、さらに「そのうちの最も短い距離」を取り出して、その分だけレイを前進させます。

もちろん状況によりますが、大抵はレイが少しずつ、段階を追って徐々に進んでいくことになります。この様子を捉えてマーチング（行進）と呼んでいるわけですね。

また、レイとオブジェクトの距離が十分に短い場合、レイの先端はオブジェクトと接触しているにほぼ等しい状態とみなすことができます。レイマーチングではこれがそのまま衝突しているかどうかの判断基準となります。

衝突していないければ黒、衝突していたら白、というように状況によって出力される色を変化させてやることで、オブジェクトの形状が浮かび上がるという寸法です。



“白になったピクセルでは、レイがオブジェクトに極めて近い位置まで進むことができた、ということ

ここまでのお話を聞くとこれまでの講義で散々やってきた「頂点を座標変換・ラスタライズして描く CG」とは、まったく原理が異なることがわかるのではないかでしょうか。

つまりスフィアトレーシングにおいては、オブジェクトは「幾何学的に数式で表現できるもの」でなければなりません。ちょっと違った言い方をすると、スフィアトレーシングには頂点は一切登場せず、数学的な文脈のみを用いてグラフィックスを描いていくことになります。



Painting a Character with Maths - YouTube

“レイマーチングも極限まで極まってくるとこうなります…… 😂”

スフィアトレーシングが「幾何学的に数式で表現できるオブジェクト」を画面に描き出す手法の1つである、ということがわかったところで、まずはかなりシンプルな「スフィアトレーシングの実装例」としてサンプル016を見てみましょう。

このサンプルでは、オブジェクトにレイが衝突した場合はそのフラグメント（ピクセル）に白を出力します。また、ここではオブジェクトの形状として球体を採用しています。

016

- シェーダ内で 3D 空間を定義し.....
- レイ（光線）の原点と向き、フォーカスする深度を定義し.....
- レイの先端とすべてのオブジェクトとの距離を計測する
- この「距離を計測するための関数」を距離関数（Distance function）と呼ぶ
- 計測した距離のうち、最も近いものを見つける
- 最も近い距離は、別の言い方をすると「確実に進むことができる距離」でもある
- 計測した距離がある一定以下になったとき、衝突とみなす

法線の算出とライティング

頂点を定義し、これを頂点シェーダで座標変換したあとラスタライズ、さらにフラグメントシェーダで着色を行う……これまでずっと扱ってきたこのような 3DCG の描画方法は、一般にラスタライズ方式というふうに言われます。

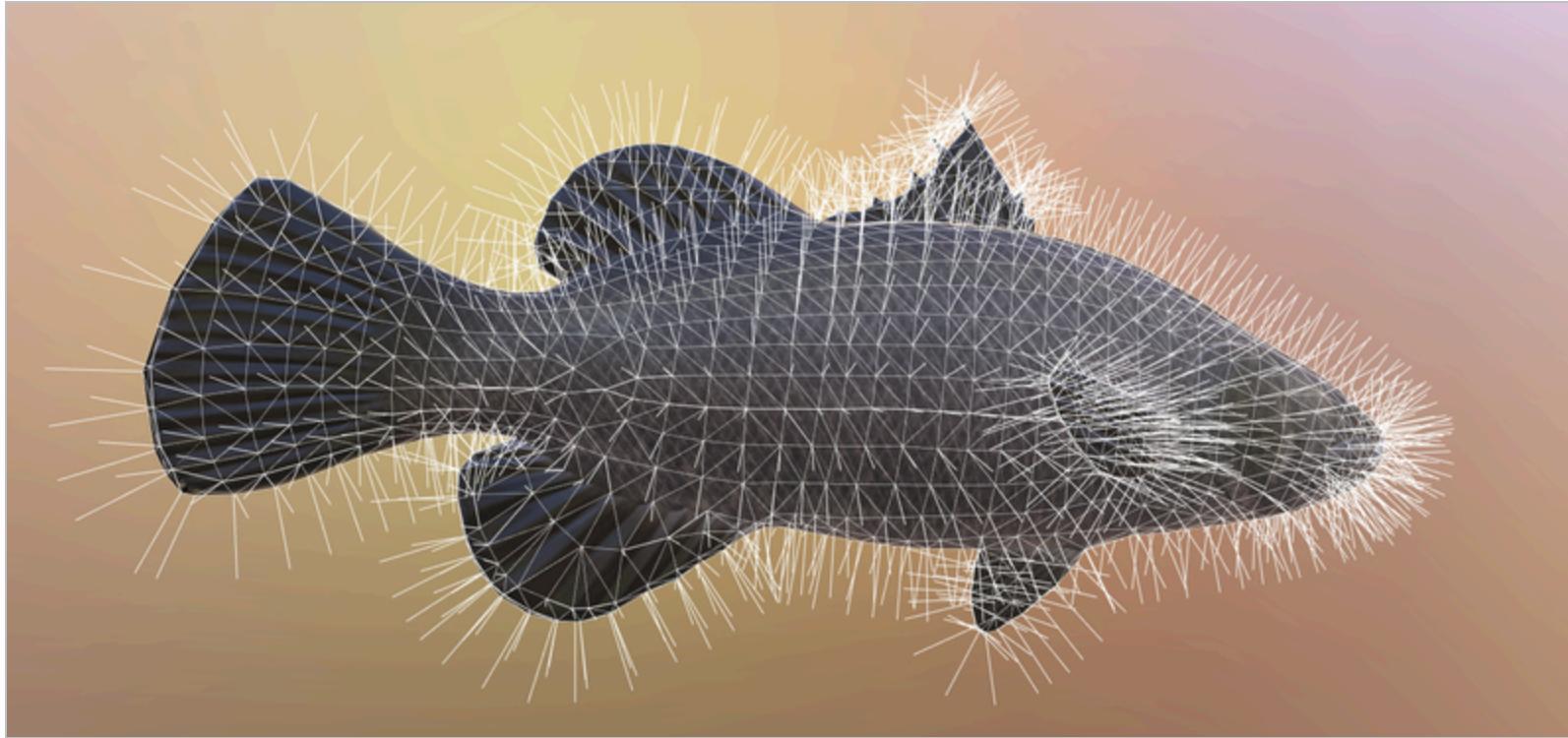
それに対して、一般にレイトレーシングと呼ばれる描画方法の場合、各ピクセルごとに幾何学的に計算を行った上で出力される色を決めていきます。今回登場したレイマーチングも、広義にはレイトレーシングに含まれる技法の 1 つであり、ラスタライズ方式とはまったく異なるアプローチでグラフィックスを描画します。

先程のサンプルでは、衝突しているかどうかだけを基準に色を出力しており、画面上に現れる色は白か黒かの2択になっていました。

大抵の場合、グラフィックスの描画では白黒の2択ではなくもっと複雑な、よりバリエーション豊かな色が出力されてほしいはずです。代表的なところでは例えば、光が当たったような陰影付けを行う「ライティング」があります。ここからは、光が当たったような効果について考えてみましょう。

一般的なラスタライズ方式の CG であれば、頂点に法線と呼ばれる「面や頂点の向きを表すベクトル」を持たせておき(`attribute vec3 normal`)それを元にライトの影響などを計算します。

しかし、レイマーチングには先程から書いているとおり「頂点という概念がそもそも無い」ので、法線も計算で動的に求めてやらなくてはなりません。



“ ラスタライズ方式なら頂点に法線を持たせる実装方法が圧倒的に多い ”

レイマーチングでライティングを行うアプローチには、大きく分けて2つの段階があります。

1つ目は、なによりもまず「シェーダ内で法線を動的に求める」ことが必要です。

レイマーチングで法線を求める方法は複数あります。

ここでは、昔からよく用いられてきた方法で法線を求める処理を記述しています。実際に続いてのサンプルでその「法線を求める処理」が使われているので、コードを見ながら考えてみましょう。

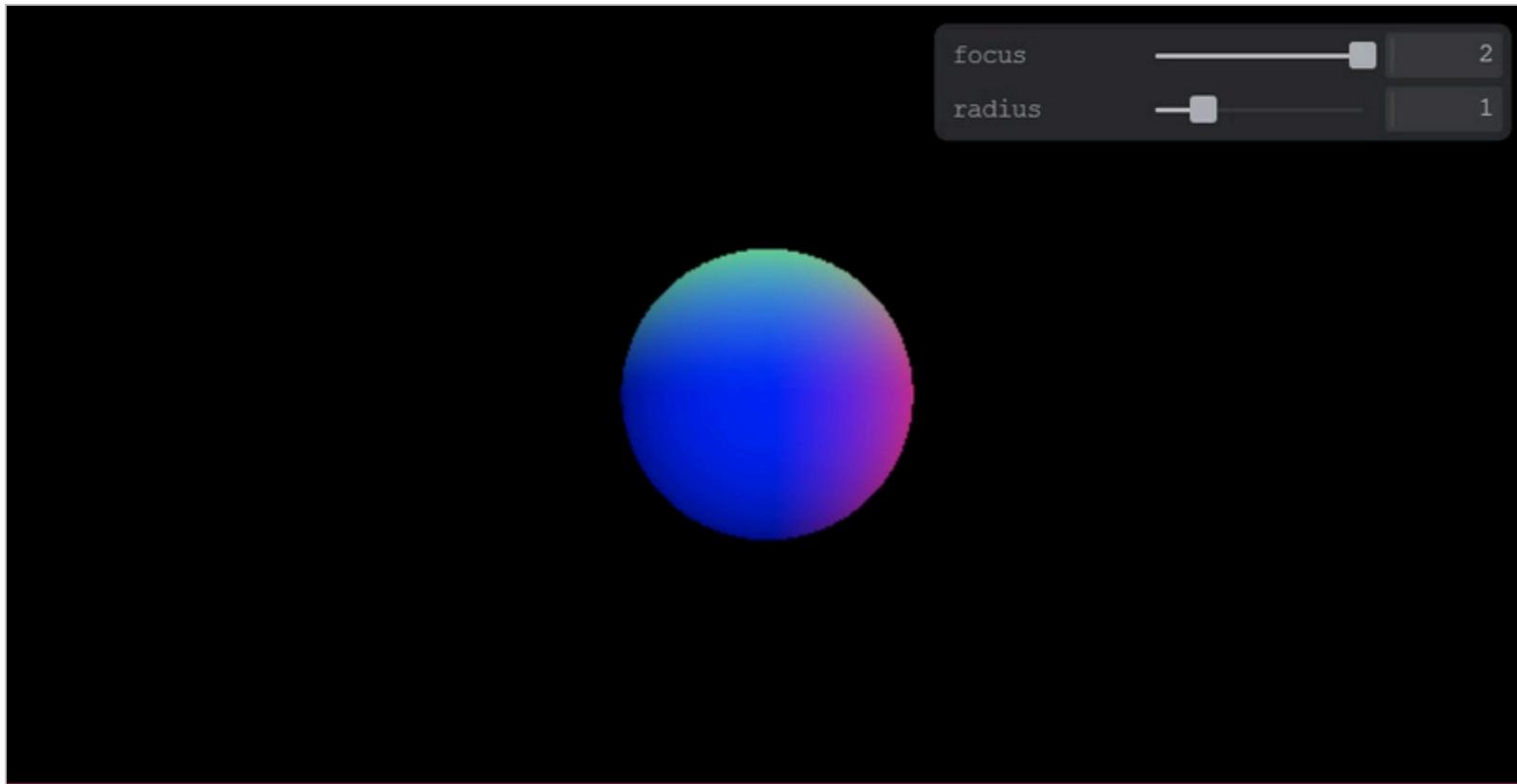
ほんの少しだけベクトルをずらした状態で複数回距離を計算し、勾配から法線を算出します。

```
vec3 generateNormal(vec3 p){  
    return normalize(vec3(  
        map(p + vec3(EPS, 0.0, 0.0)) - map(p + vec3(-EPS, 0.0, 0.0)),  
        map(p + vec3(0.0, EPS, 0.0)) - map(p + vec3(0.0, -EPS, 0.0)),  
        map(p + vec3(0.0, 0.0, EPS)) - map(p + vec3(0.0, 0.0, -EPS)))  
    ));  
}
```

算出された法線は、あくまでも「そのピクセルにおける衝突面の向き」を表すベクトルです。ですから必ず単位化してから利用します。

また、単位化した状態でそのまま色として出力すると、ある程度法線の値が正しく求められているかどうかを視覚的に確認することができます。

“色を目で見ておおよその値を予測する色デバッグ”



変数 destColor に法線を入れて出力すると目で確認できる

法線から陰影を求める

さて、第一の段階「法線を求める」ができましたので..... ライティングを行うためのもう1つの要素である「法線を用いて陰影をつける」をやってみます。

ここでは一般に「ランバート・ライティング」と呼ばれている超基本的なライティングの表現をやってみましょう。

ランバート・ライティングは、17世紀の數学者「ヨハン・ハインリヒ・ランベルト」さんにちなんだ名前です。

ランベルトさんの提唱した「ランバート反射」という「光の拡散反射モデル」をベースにした陰影付けの手法であることから、CGの世界ではランバート・ライティングや、ランバートの反射モデルなどと呼ばれることが多いです。

難しい数式とかが出てくるのではないかと身構えてしまった人も多いかもしれません。ランバート・ライティング自体はすごくシンプルな計算です。

どのくらいシンプルかというと、GLSL であれば次のように書くだけです。

```
float diffuse = max(dot(法線, ライトベクトル), 明るさの最小値);
```

この数式をよく見てみると、GLSL のビルトイン関数が 2 つ使われています。

`max` は文字通り、与えられた引数のうちの「大きいものだけを返す」という性質を持った関数です。そして `dot` は「ベクトルの内積」を求めるための関数です。

あらためて先程の数式を眺めてみると、`max` のほうは要するに「指定された最小値より小さくならないように値を補正」するために使われているだけで、ここには特に深い意味はありません。

ここでのポイントは `dot` 関数、つまりベクトルの内積の使い方とその意味です。

ベクトルの内積は、以下のような計算です。

```
// 三次元ベクトルの内積の例
vec3 v = vec3(x1, y1, z1);
vec3 w = vec3(x2, y2, z2);
float dp = dot(v, w);

// これは実際には以下のように計算している
float dp = x1 * x2 + y1 * y2 + z1 * z2;
```

掛けて、全部を足すだけの、実際にはかなり単純な計算！

そして、このベクトルの内積の計算方法を頭にイメージしながら「法線とライトベクトル」で内積を計算すると、実際にはどのようなことが起こっているのか考えてみましょう。

ここでの最大のポイントは「法線もライトベクトルも、いずれも必ず あらかじめ単位化しておく こと」です。単位化したベクトル同士の内積は、その性質上必ず $-1.0 \sim 1.0$ の範囲に収まります。

$$\begin{array}{c} \text{---} \\ | \\ (1, 0, 0) \end{array} \bullet \begin{array}{c} \text{---} \\ | \\ (1, 0, 0) \end{array} = 1.0$$

$$x * x + y * y + z * z == 1 * 1 + 0 * 0 + 0 * 0$$

まったく同じ向きの単位ベクトルの内積は 1.0

$$(1, 0, 0) \cdot (0, 1, 0) = 0.0$$

$$x * x + y * y + z * z == 1 * 0 + 0 * 1 + 0 * 0$$

“互いに垂直な単位ベクトルの内積は 0.0”

$$\begin{array}{c} (1, 0, 0) \quad \bullet \quad (-1, 0, 0) \\ \longrightarrow \qquad \qquad \qquad \longleftarrow \end{array} = -1.0$$

$$x * x + y * y + z * z == 1 * -1 + 0 * 0 + 0 * 0$$

完全に逆向きの単位ベクトルの内積は -1.0

つまり、単位化したベクトル同士の内積では、与えられた 2 つのベクトルが「同じ方向であるほど 1.0 に近い」結果が得られます。逆に、逆方向に近いほど、内積の結果は -1.0 に近くなります。

この性質をそのまま陰影付けに活用してしまおうというのが、ランバート・ライティングの考え方です。

頂点に法線を持たせるラスタライズ方式で CG を描く場合も、基本的な考え方はまったく同じです。

レイマーチングの場合はそもそも頂点という概念がなく、ピクセル単位でそこにオブジェクトが存在するかどうかを判定していくため、動的に法線を算出した上でランバート・ライティングなどで陰影付けを行う形になります。

017

- レイマーチングの法線算出にはいくつか方法がある
- ここでは、ほんの少しずらしたレイで距離関数の結果を計算し.....
- それらの差分から法線を求めている
- 法線が求まつたら、色として出力することで検証ができる
- ランバートの反射モデルでライティングを行う場合はベクトル内積を活用し、ベクトルが単位化されているかに気を配る

物量感を演出する座標系の複製

さて、本日の最後のサンプルでは、レイマーチングで物量感を演出する際に頻繁に用いられるテクニックを紹介します。

とは言え実際には、今回の講義すでに登場しているテクニックを三次元に応用するだけです。

サンプルのフラグメントシェーダを見てみると、新しく repetition という名前の関数が定義されています。

これが、座標系を複製することができる関数です。

```
vec3 repetition(vec3 p, vec3 width){  
    return mod(p, width) - width * 0.5;  
}
```

“ repetition は直訳すると繰り返しという意味です ”

実際にサンプルを動作させて描画結果を見てみると `mod` 関数を使った
ちょっとした計算を追加するだけでどうしてこれほど劇的に描画結果が
変化するのか不思議に感じるんじゃないかなと思います。

でも実際は、シェーダアートのサンプルにあった座標系の複製をレイマ
ーチングと組み合わせているだけなので、落ち着いて仕組みを考えてみ
ましょう。

たとえば 3D 空間の座標を 10 の幅 (width) で割った余りを考えてみると.....

10 で割った余りなので、割られる数がなんであれ、必ずその結果は 0.0 ~ 9.99999... という範囲を取ることがわかります。そこからさらに width の半分を減算すれば、-5.0 ~ 4.99999... の範囲に限定された結果が得られます。

ですから実際に起こっていることは「一定の範囲に区切られた座標空間が連續して並んだ状態」を作っているようなイメージ、という感じです。

とても少ないコード量で見た目が大きく変化するのは、レイマーチングやシェーダアートが持つ特徴の 1 つだと思いますが、これもその一種と言えるでしょう。

018

- 除算の剰余を計算する `mod` を活用して座標系を複製する
- 割った余りだけを使うことで座標系の範囲を限定させ、さらにそれがループするように繰り返されている
- たくさんのグリッドが並んだような空間をレイがまっすぐに進んでいる状態をイメージし.....
- たまたまレイの先端がいるグリッドで、グリッド内座標を使って計算を行って描画結果を作っている、のような感じ
- 図解するなどして落ち着いて考えよう

まとめ

今回はフラグメントシェーダをテーマに、様々な表現を紹介しました。

見た目がどんなに派手であっても、実際にはそれを実現する方法や考え方はとてもシンプルである場合が多く、いろんな意味で刺激的な内容だったのではないかなど思います。

ただし、忘れてはならないのは、シェーダーアートのたぐいは表現力を底上げしたり表現の幅を広げたりしてくれますが、どちらかというと変化球的な技術だということです。

特にレイマーチングなどはなかなかウェブ制作の文脈では用いないトリッキーな表現ですから、そこに学習コストを割くべきかどうかはご自身の実現したいことや習得したい内容とよく照らし合わせて考えるようにしてください。

さて今回の課題ですが、せっかくなのでシェーダーアート的に「フラグメントシェーダだけを用いた表現」に挑戦してみましょう。

とは言えそれだと課題として選択肢が広すぎるとと思うので.....「for 文による繰り返しを含む」という縛りを追加したいと思います。ループ処理をなんらかの用途として盛り込みつつ表現してみましょう。