シェーダを活用したポスト処理

フラグメントシェーダを活用した画作り

振り返り

前回はテクスチャマッピングの基本とキューブ環境マッピングなど、テクスチャ関連の技術に加え、色を操る概念であるブレンドを扱いました。

テクスチャにしてもブレンドにしても覚えることがやや多くて大変なのですが、外見の変化が大きく表現の向上につながる技術になりますのでうまく取り入れていきましょう。

さて今回ですが、three.js で過去に扱った技術に「オフスクリーンレンダリング」がありましたが、これをネイティブな WebGL で実装してみます。

オフスクリーンレンダリングは現代の CG には欠かせない重要な技術です。WebGL でこれを実現するためにはフレームバッファと呼ばれるオブジェクトを利用します。まずはそのあたりから紐解いていきましょう。

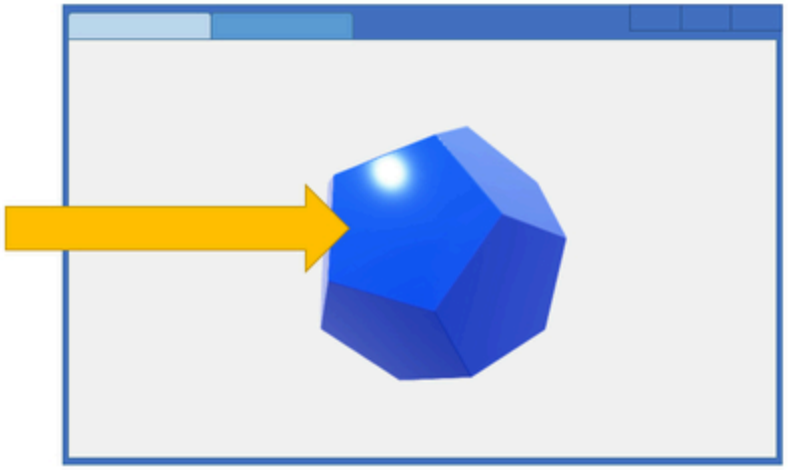
オフスクリーンレンダリング

（マルチパスでのレンダリング）

これまで、ネイティブな WebGL のあらゆるサンプルでは、レンダリングを「１パス」で描画していました。

レンダリングを行う対象は常にひとつのキャンバスであり、１つの描画結果を得る（レンダリングする）ために複数回の描画プロセスを経ることはありませんでした。イメージ的には、現実世界の写真のように１回だけシャッターを切って絵を完成させるような感じ、といえばいいでしょうか。

gl.drawArrays や gl.drawElements で
canvas 要素に対して直接描画結果を
レンダリングしていた



> 描画先は常に canvas そのものであり、一発で描画結果が決まる

フレームバッファを用いると、この「一度の描画ですべてのレンダリングを完了させる」という制限を超え、複数の描画結果を組み合わせて画作りをすることができるようになります。

これはちょっと違った言い方をすると Photoshop などのフォトレタッチソフトで、撮影した写真の画像にあとからフィルタを掛けたりするのと同じような処理、と捉えるとわかりやすいかもしれません。

別の言い方をすると、画面にレンダリング結果を直接出力するのではなく、メモリ上に一度仮のレンダリングを行っておき、それをさらに再利用して最終的な画作りを行うということです。

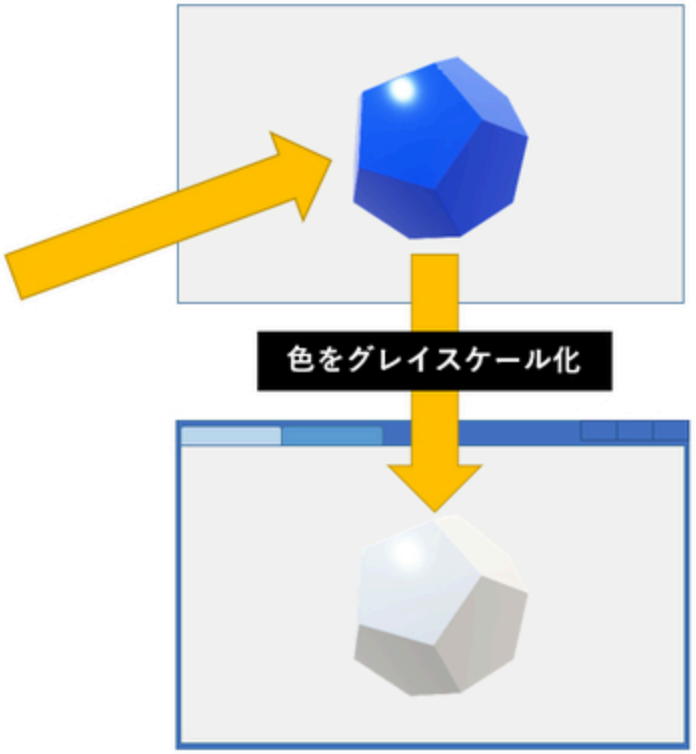
ここで言う「メモリ上でレンダリングを行う対象となる描画領域」を フレームバッファ と呼びます。

> three.js で言うと WebGLRenderTarget がフレームバッファを内包していると言えます

そして、この「フレームバッファに対してレンダリングしているため画面に現れない描画処理」のことを オフスクリーンレンダリング と呼びます。

オフスクリーンレンダリングを活用すると、A という描画結果をオフスクリーンで一度作ってから、それに対して B というシェーダを適用してさらに加工を行う、といった複雑なレンダリングを行うことができるようになります。

一度、別の描画領域にレンダリング
を行ったあと、これを別のプロセス
で加工してから画面に出力する



1パス目でオフスクリーンレンダリングを行い、2パス目で別のシェーダを使うなどして加工しつつ画面に出すといったことができる

フレームバッファ、という名前が付いていますが、その実態としては「カラーバッファや深度バッファなどを１つにまとめたバッファの集合体」のことを指します。

深度テストをしながらでないと 3D シーンは描画できない場合も多いので、カラーバッファや深度バッファはそれらをまとめて１つのオブジェクト（つまりフレームバッファ）として扱えるようになっているのです。

実はいままで一切触れてこなかったのですが、フレームバッファは WebGL コンテキストが初期化されると同時に無条件で 1 つ、既定のものが必ず自動的に生成されるようになっています。

これまで WebGL の実装でわざわざフレームバッファを生成するような手続きはしてきませんでしたが、既定のフレームバッファが自動的に生成されるおかげで、我々は深度テストを使ったり描画結果を見ることができたりしていたのです。

自前で（既定のものとは別の）フレームバッファを生成することができれば、もちろんそれに対してもこれまでやってきたのと同様にレンダリングを行うことができます。

いままでは無条件に画面に描画結果として映すだけでしたが、別の描画領域を追加で準備し、そこに絵をレンダリングして保持しておくことができるようになるわけです。

ここまでのまとめ

- フレームバッファとはバッファの集合体
- フレームバッファにはカラーバッファや深度バッファが含まれる
- WebGL コンテキストを生成すると既定のフレームバッファが自動的に１つ生成され、それが使われる
- つまりフレームバッファとは「レンダリングの対象となる描画先」であり、これを自前でセットアップすれば、描画の対象となる領域を自由に拡張できる

フレームバッファとテクスチャ

フレームバッファの実際の初期化手順では「フレームバッファオブジェクト」を生成したあと、そこにさらに「用途を指定したバッファ」を格納していきます。色を書き込むためのカラーバッファや、深度を書き込むためのデプスバッファなど、種類・用途に応じたバッファを生成してそれらをフレームバッファに格納していくようなイメージです。

このような「個別に生成してフレームバッファに格納するバッファ」のことを広い意味でレンダーバッファと呼びます。

フレームバッファには、色を書き込むためのカラーバッファや、深度情報を書き込むための深度バッファのような、いくつかのバッファが格納された状態になっている

これらのバッファの１つ１つがレンダーバッファから作られる

レンダーバッファは様々な用途に利用できる汎用的なバッファで、生成後にどのようなフォーマットを指定されたのかによって性能が変化します。

代表的なところでは、`gl.RGBA4`、`gl.DEPTH_COMPONENT16`、などがあります。末尾の数字はビット数を表していて、なんとなく意味としては定数名を見ただけでも想像できますよね。

> RGBA4 なら、RGBA の各要素に対して 4bit のデータが割当てられる

さて、ここで唐突に出てきた `gl.RGBA4` ですが…… RGBA の各要素に 4 ビットのデータを割り当てるということは「色を表現するため」には明らかにデータ量が不足している、ということに気がついたでしょうか。

4 ビットでは、16 種類しかデータを表現できません。これは特殊なケースを除き、レンダリング結果の絵を描き込むバッファとしては精度が足りません。

1bit がスイッチ 1 つとして、合計 4bit では 16 までしか表現できないので、色を表現するのに十分な精度がないことがわかる

`gl.RGBA4` では精度が足りない、としたら `gl.RGBA8` のようなフォーマットを使えばいいのではないか、と思うかもしれません。

しかし実は `gl.RGBA8` というフォーマットは存在せず、その代わりに テクスチャオブジェクトをカラーバッファとして使う という方法を用いるのが一般的です。

テクスチャをカラーバッファとして割り当てると、描画した結果が「テクスチャに焼き付けられる」ということが起こります。

要は写真を撮るような感じで、テクスチャにレンダリング結果を描画して保持しておくことができるわけです。

この「描画結果が焼き込まれたテクスチャ」は要するに 扱いとしては要するに普通のテクスチャと同じもの なので、これまでのテクスチャのユースケースと同じように、なんらかのシェーダに渡して使うことだってもちろんできます。

つまり「レンダリング結果を焼き付けたテクスチャがほしい」というユースケースにおいては、フレームバッファが大活躍することになります。

ここまでのまとめ

- フレームバッファは複数のバッファが格納されたバッファの入れ物
- フレームバッファには、別途レンダーバッファを作って格納していく
- レンダーバッファにはいくつかフォーマットがあるが……
- `gl.RGBA4` などでは色を表すのに精度が不足しているので……
- 色を格納するバッファとしてはテクスチャを代替として使う

> もちろん、意図的に RGBA4 などを使うユースケースも世の中にはあると思いますが、わたしの経験では WebGL でテクスチャ以外をカラーバッファとして使ったことはありません

フレームバッファの初期化手順

フレームバッファを用いるサンプルでは、webgl.js 側にフレームバッファを生成するためのメソッドを新たに追加して利用しています。

一応最初なので、フレームバッファの初期化についてざっくりと確認しておきましょう。

> ここでもこれまでと同様にメソッドや引数をすべて暗記する必要はないので、まずは雰囲気を掴みましょう

ざっくりやっている事柄を書き出すと……

- フレームバッファの生成
- レンダーバッファの生成とフォーマット指定
- レンダーバッファをフレームバッファに関連付け
- テクスチャの生成とフォーマット指定
- テクスチャのフレームバッファへの関連付け
- 最後に念のためバインドは解除しておく
- 生成したオブジェクトをまとめて戻り値として返す

フレームバッファも、やはりこれまでに登場したバッファ類（たとえば VBO など）と同じようにバインドすることでそれが利用可能な状態になります。

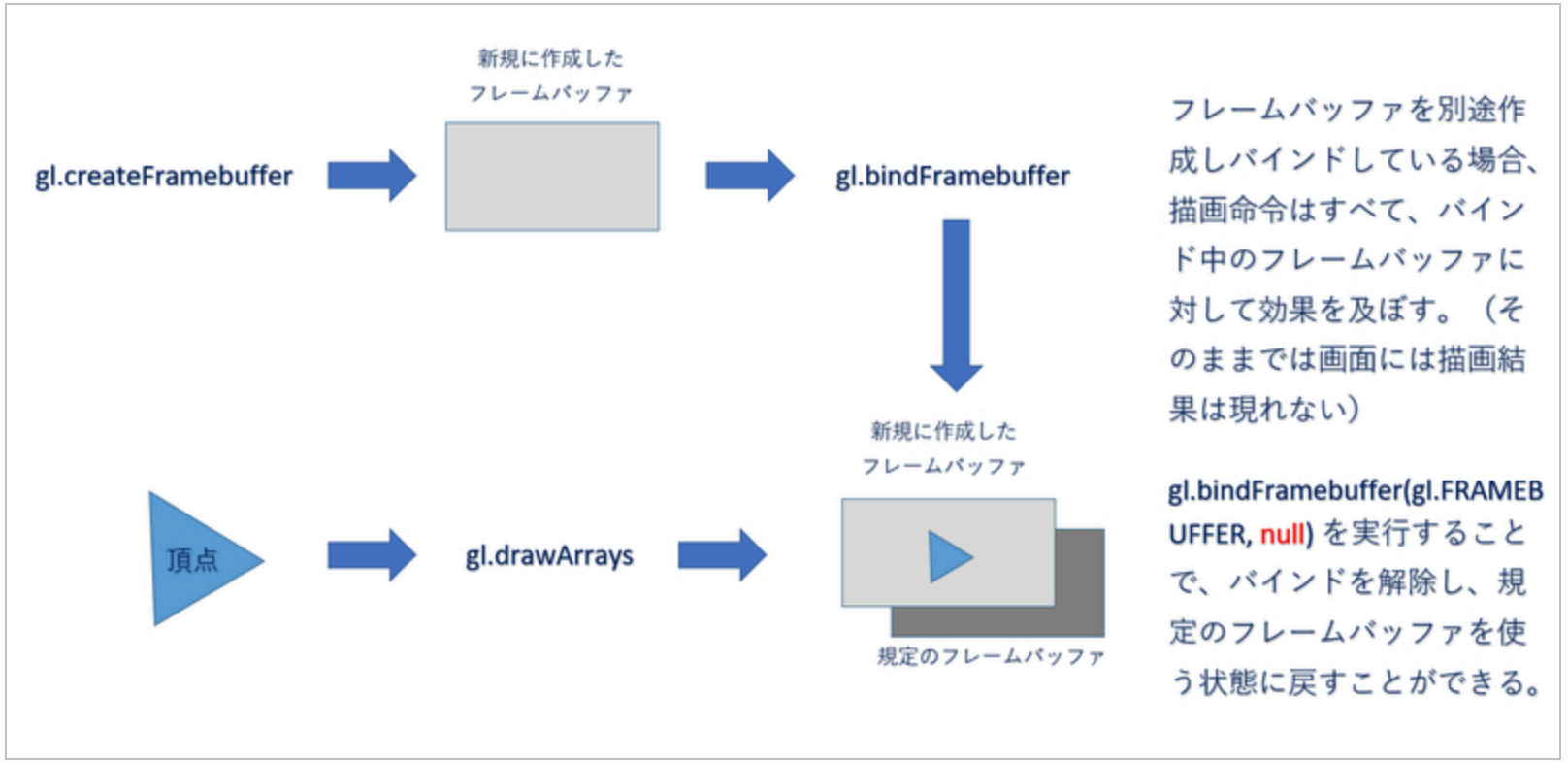
また、バインド時に `null` を指定することで、フレームバッファのバインドを解除して既定のフレームバッファが利用されるもともとの状態に戻す ことができます。

> 何もバインドされていない == 既定のフレームバッファが使われる == 画面（canvas 要素上）に絵が出る状態

フレームバッファを利用した描画を行う際は、フレームバッファオブジェクトをバインドするだけでよく、フレームバッファに関連付けたレンダーバッファやテクスチャを個別にバインドする必要はありません。

あらかじめアタッチされているレンダーバッファやテクスチャが、自動的に描画対象として使われるようになります。

特にフレームバッファをバインドしていない場合、規定のフレームバッファがそのまま使われる。



フレームバッファを別途作成しバインドしている場合、描画命令はすべて、バインド中のフレームバッファに対して効果を及ぼす。（そのままでは画面には描画結果は現れない）

gl.bindFramebuffer(gl.FRAMEBUFFER, null) を実行することで、バインドを解除し、規定のフレームバッファを使う状態に戻すことができる。

017

- フレームバッファとはバッファの集合体であり、レンダリングを行った際の描画対象として利用できる
- フレームバッファを新規に生成して利用することで、直接画面には出ないオフスクリーンでのレンダリングを行うことができる
- フレームバッファのバインドを解除すると既定のフレームバッファが使われる状態に自動的に戻る
- オフスクリーンでテクスチャに対して一度描画した結果は、これまでと同様に「テクスチャ」として普通に利用できる

> 描画プロセスが複雑になるので落ち着いて考えよう！

ポストエフェクト

フレームバッファの初期化手順などがざっくりと理解できたところで、比較的簡単なポストエフェクトを行ないながら少しずつ慣れていきましょう。

3DCG でポストエフェクトと言った場合には、これはオフスクリーンにレンダリングした描画結果に対して、事後的にシェーダで加工を行う ことを言います。

> three.js で EffectComposer を使って行ったエフェクト処理と同じ系統

ここで重要なのは、ポストエフェクトの実装では原則として「既に一度描画された結果に対してエフェクトを掛ける」ということです。

つまり、ポストエフェクトを用いるケースの多くは基本的に 一度レンダリングが完了した二次元ビットマップデータを加工していく のであり、三次元的な情報はすでに失われた二次元ビットマップ（要するに画像）として扱う、という点をしっかり意識することが大切です。

サンプル 018 では、ポストエフェクトで用いられることの多い、グレイスケールとヴィネットを実装しています。

ここで登場した２つのポストエフェクトは、汎用性が高くいろいろなシーンで使うことができます。特にグレイスケールはかなり多目的に使えるエフェクトになりますので覚えておくとよいでしょう。

018

- ポストエフェクトとは一度描画結果をテクスチャに焼いて事後処理を施すことを指す
- 一度レンダリングしてしまっているので二次元データになっていることを念頭に置く
- フォトレタッチソフトのように後から加工するイメージ
- グレイスケール化には内積を応用すると簡潔に書ける
- ヴィネットの処理ではベクトルの長さを活用する

3DCG とノイズ

さて、続いては 3DCG には欠かせない非常に重要な概念を扱ってみます。

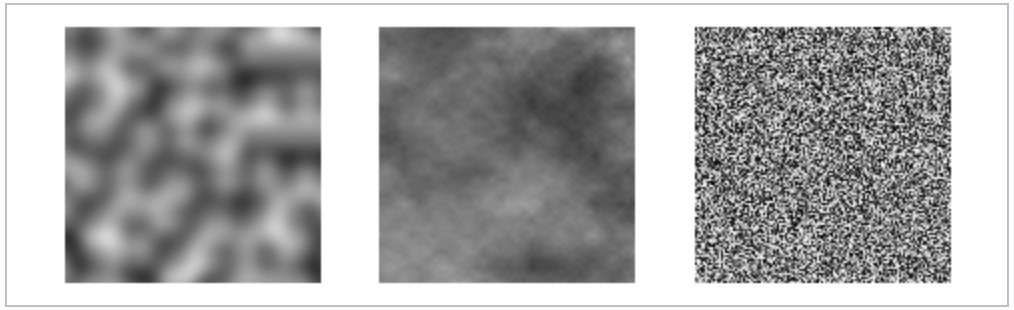
それは広義には ノイズ と呼ばれています。ノイズというと音のほうのノイズを想像してしまうかもしれませんが、CG の世界にもかなりポピュラーな存在としてノイズという概念があります。

音の場合のノイズ、というと、雑音とかそういった意味で使われることが多いと思います。

では 3D や CG におけるノイズとはいったいなんなのでしょうか。

3DCG におけるノイズは、いわゆる「乱数」を使って生成される値です。雑音とはちょっと違いますが、やはり、乱れたランダムな状態のことを指してノイズと表現するわけです。

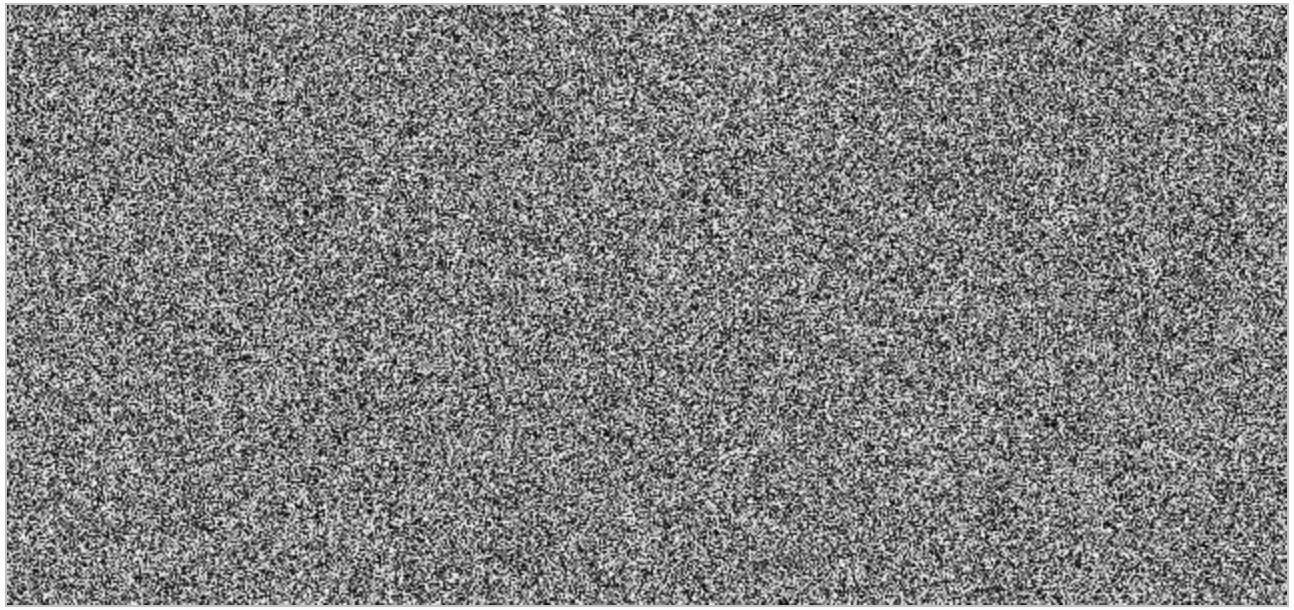
そしてこのノイズには、その生成手法などに応じて様々な種類があり、また同時に「ノイズの品質の違い」なども手法により様々です。



> ノイズを視覚的に捉える場合、値の大小を色の明暗として出力することが多いがその本質はあくまでもランダムな数値

品質が高いけど計算負荷も同時に高いノイズや、あるいは逆に負荷は小さいけど品質はそれほど良くないノイズ、といったようにいろんなノイズがあります。

今回はまず、一般にホワイトノイズと呼ばれているものから見てみましょう。



> ホワイトノイズはそのまま二次元に出力すると砂嵐のような印象に

JavaScript では、乱数を生成するための関数として `Math.random` があります。

ですから JavaScript では乱数が欲しくなったとしても困ることはありません。普通に `Math.random` を使えばいいだけだからですね。

それでは、GLSL の場合もそういった乱数を生成できる関数を使えばいいのかというと……

GLSL には 乱数を生成する命令はありません ので、自前でなんとかしなければなりません。

自前でなんとかする、といっても「乱数生成器を実装すればいいんですね！ わかりました！」という感じでサッと実装できる人はそうそういないと思いますので、乱数生成用の関数も実装したサンプルを用意してあります。

サンプル 019 のシェーダのソースコードを見てみましょう。

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

> 初見だと、なぜこのような処理で乱数が生成できるのかちょっと不思議に感じるであろう、マジックナンバーだらけの乱数生成器

先程の `rnd` 関数を使って生成した乱数を、そのまま画面に出力するとまるで砂嵐のような見た目になります。

サンプル 019 はまさにそれを行っており、特に乱数を（生成後に）事後的に加工したりはしていません。

余談ですが、先程の `fract` と `sin` を利用した一連の処理で、どうしてノイズが生成できるのかは The Book of Shaders の Random の項目に非常にわかりやすい説明とデモがあります。

The Book of Shaders: Random

019

- 小数点以下だけを取得する `fract` を活用
- 原理としてはサインの生成する値を上手に利用している
- あまり品質は良くないノイズだが手軽で高速
- ちょっとした演出には十分に実用
- 少しマジックナンバーを変更すると途端に絵が変わるため注意

余談：浮動小数点の精度とノイズ

余談ですが……

WebGL はウェブブラウザ上で動作することからプラットフォーム（動作しているハード）を選ばないことが大きなメリットですが、反面、どのような環境で動作しているのかがわからないために逆に問題が複雑になってしまうことがあります。

今回取り上げたノイズについても、この問題が起こりやすい分野だと言えます。

わかりやすい例を挙げると、今回のサンプルのうちノイズを使っている実装では Windows, Mac, モバイル端末など、それぞれで違う結果になってしまう可能性が高いです。

これは、GPU がどれくらいの精度で浮動小数点を扱っているのか、などが環境ごとに異なるために起こる問題です。

多くの場合「精度が不足している」ために、ノイズを利用した際に意図したような結果が得られないことがあり、それによって見た目上の違いが生まれます。

この問題は実は結構根深くて、同じコードなのに実行環境によって結果が変わってくるというのが困った点なので、コードの書き方で対応できる範囲にはどうしても限界があります。

一応この問題に対する１つの解決策として、精度が落ちないように工夫する、というアプローチがあります。

> 次のページにその例があります

```
float rnd2(vec2 n){
  float a = 0.129898;   // ← 整数部分の桁数を減らす
  float b = 0.78233;    // ← 整数部分の桁数を減らす
  float c = 437.585453; // ← 整数部分の桁数を減らす
  float dt= dot(n ,vec2(a, b));
  float sn= mod(dt, 3.14);
  return fract(sin(sn) * c);
}
```

> 整数部分を極力小数点以下に収めるようにする

ここでやっていることは、整数部分の桁数を極力減らす というだけの、一見精度とはなんの関係も無さそうな手法なのですが、これが結構効果が大きいです。

浮動小数点とは、その名前のとおり、小数点の位置が変化することを考慮した数値の表現方法です。ですが、小数点の位置が変化したからといって、その数値を表現できるビット数まで変化するわけではありません。

> つまりメモリの消費量は常に同じ

ですから、整数部分の桁数を減らすと結果的に「小数点以下の部分に使われるビット数が増え精度が上がる」ことになるので、それが一定の効果を生む場合があるのですね。

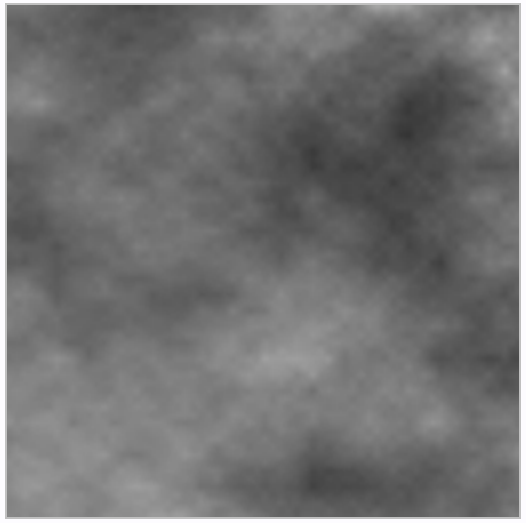
もし、GLSL や WebGL の実装で、どうも精度が足りないのか安定しないなあ……みたいなことがあったときは、このような浮動小数点の精度が関係しているかもしれない、ということを思い浮かべられるようになっておくと場合によっては有効な改善方法を思いつくこともあるかもしれません。

バリューノイズ

ノイズには様々な種類や生成手法があるという話をしましたが、ここからは、割と簡単なノイズの実装のひとつ「バリューノイズ」をやってみましょう。

このバリューノイズは、ホワイトノイズとはかなり見た目が異なります。

バリューノイズの実装例、その平面への出力結果。



> 平面にプロットすると雲や煙のような外観になる

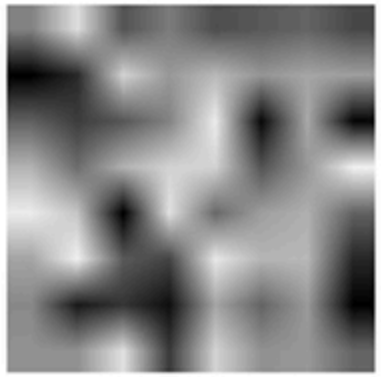
このバリューノイズのような、雲や煙のような外見のノイズは CG に活用できるシーンが非常に多いです。

雲などの表現はもちろんのこと、炎や、あるいは岩の凹凸なんかも、このようなノイズがひとつあれば容易に実装できます。

バリューノイズの原理はこうです。

まず、解像度の低い状態でノイズを生成してやり、それを補間して引き伸ばします。

解像度が低い状態なので……もちろん見た目はかなりぼんやりとした感じになります。

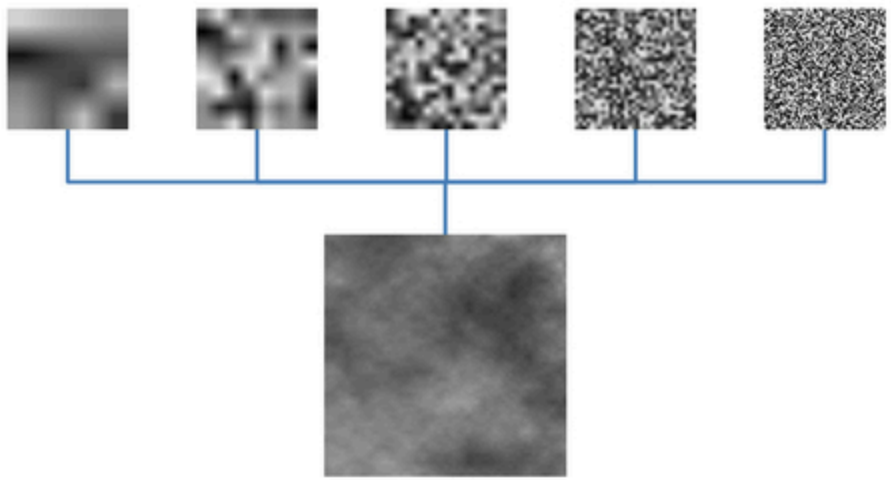


まずはランダムな値をピクセルごとに求め
てやるが、これをそのまま使うのではなく、
隣り合うピクセルを補間してから使う

> かなりぼんやりふんわりしたグラデーションに

次に、解像度を倍にして、同じようにノイズを生成します。次はさらに解像度を倍にして生成、次もさらに倍、次もさらに倍……というように、徐々に解像度を高くしていきます。

最終的に、必要な解像度の状態になるまでこれを繰り返していき、それらの結果を全て合算します。



複数回、異なる解像度で乱数の生成と補間を行ってやり……それらを重み付けしながら合成してやることで複雑な陰影を持つバリューノイズが生成される

複数の解像度のノイズを補間して合成する

このように、バリューノイズの生成においては基本となる乱数生成を繰り返し繰り返し、何度も行うことになります。

ですから、乱数生成関数自体が高負荷なものを使ってしまうと、なかなかリアルタイムに処理するのは難しくなります。今回は fract sin ノイズ関数を使っているので高速に処理できるため多少はマシです。

> それでも若干重いですが……

もし、高品質なノイズが使いたければ、まず最初に一発でテクスチャにレンダリングだけしておき、以降はそれを参照するだけにすると負荷はかなり抑えられます。

あるいは、もういっそのこと画像編集ソフトなどを使ってあらかじめノイズ用の画像を作ってしまい、普通にテクスチャのサンプリングで画像から値を取り出して利用する、などの方法を用いるのもよいでしょう。

> リアルタイムにノイズを生成するか、テクスチャに焼いておき参照するだけにするかの違い

020

- GLSL では C 言語などと同じで関数を呼び出すより前に、関数の定義が先になされている必要がある
- 繰り返しノイズを生成して合成してひとつの結果を生み出す
- 今回の実装例の場合 1px あたりの出力結果を得るために 128 回も乱数を生成している
- シームレスなノイズが `snoise` 関数
- シームレスである必要がなければ `noise` 関数のほうで十分

> ノイズの生成は基本的に重いので用途と負荷をよく検討して実装するのが吉

ノイズによるディストーション

さて、バリューノイズが生成できるようになったので、これを活用したテクニックを１つ紹介します。

ノイズは本当に汎用性の高い概念なので、柔軟に発想できるかどうかが大事です。

もしかしたら先にサンプルを実行して、既に結果をご覧になっている方もいるかもしれませんが……

バリューノイズの原理が分かった今なら、サンプル 021 のゆらゆらエフェクトをどうやって実現すればいいのかなんとなく想像がつくのではないでしょうか。

そもそも、ポストエフェクトとは「一度フレームバッファにレンダリングした結果」を「テクスチャを参照して読み出し、同時に加工する」ことでしたよね。

つまり テクスチャを参照する工程に一手間加える ことで、いろいろな効果を実現できます。

今回のディストーションエフェクトでは、オフスクリーンでレンダリングしたシーン（のテクスチャ）を参照する際に、ノイズの値を使ってランダムにテクスチャ座標をずらしています。

結果的に、ノイズの濃淡に応じてシーンが歪んで表示されます。

021

- 時間の経過と共にノイズがずれるようにしておく
- ずれていくノイズの値を元にテクスチャ座標を歪ませる
- 歪んだ結果をそのまま出すとなんだか水中にいるみたいな見た目に
- テクスチャ座標だけでなく色などにノイズを使っても面白い

ガウシアンブラー

さて、いよいよスクールのカリキュラムも仕上げです。フレームバッファを複数同時に利用する実装例として ガウシアンブラーシェーダ について見ていきます。

様々な処理に応用できるとても汎用的で強力なポストエフェクトの１つ、と言っていいと思うのですが、ちょっとだけ、処理フローが複雑になります。

> モバイル系 OS のインターフェースにもブラーが結構使われていますし、結構気軽に使える効果だと思われていることが多いブラー処理ですが、実装するのは結構たいへん

そもそも、画像などのイメージを「ぼかす」というのは、どういう処理のことを言うのか、考えたことはあるでしょうか。

画像処理系のプログラムを組んだことがある方なら察しがつくかもしれません。

プログラム的にぼかしを表現するには、本来のピクセルの色を周囲の色と混ぜあわせることで、鮮明な色を意図的にあいまいにしてやります。

WebGL や GLSL でこれを再現する方法はいくつか考えられますが、最も単純に行うとすれば、テクスチャを参照する際に本来の色の周囲を参照しつつ、周辺の色と混ぜあわせるような処理をすればいいですね。

> 1 つのピクセルの色を決めるために、周辺の色をすべて採集して、合成する

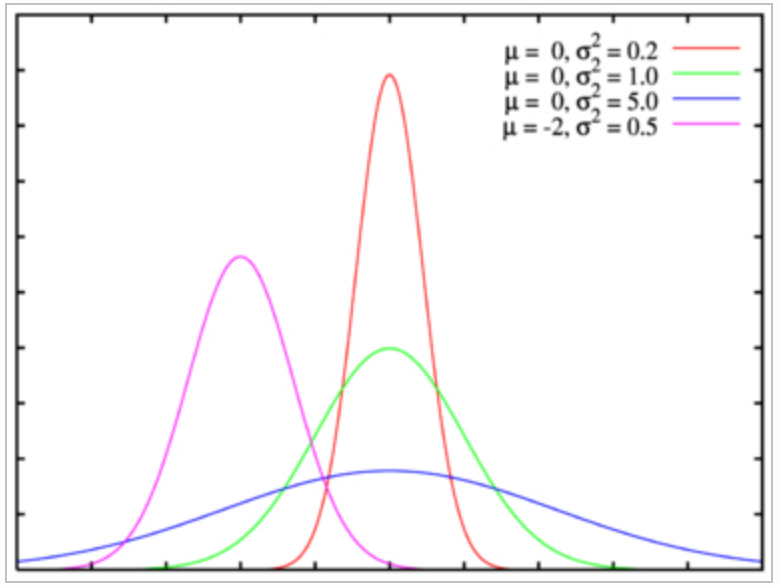
この時、周囲の色を どのように参照 し、どのように混ぜ合わせるか こそがぼかしのキモになります。

たとえば、本来の色を 60% にして周囲 4px から 10% ずつ色を取得して混ぜあわせたり……といったように、どのようなルールで混ぜあわせるのかをよくよく考える必要があるわけです。

> この混ぜ合わせ方のルールひとつで見た目も変わってくる

ガウシアンブラーはその名前の通り、この合成する処理にガウス関数を用います。

ガウス関数は、二次元のグラフ上にマッピングしてやると釣鐘型になることが知られています。この釣鐘のような値の上下を、周囲から取り込む色の割合として利用するのです。



中心に近いところほど割合が大きくなる釣鐘構造

釣鐘のような形となる値のマッピングを、そのままテクスチャから取得した色の混ざり具合いに利用します。

ガウス関数は係数をちょっと変化させるだけで、この釣鐘の形状を簡単に変化させることができ、縦長の釣鐘型や横長の釣鐘型の値の分布を生成できます。つまり、極端にぼかすことも、少しだけぼかすことも、係数ひとつで簡単に行えるわけです。

さて、ガウス関数がどのようなものかはざっくり理解できたかと思いますが、ぼかしを実現するためには、もうひとつ考えないといけないことがあります。

それは、ぼかすために色を参照する範囲を「どらくらいの広さ」に設定して処理するか、という問題です。

たとえば、単純に本来のピクセルから均等に 2px 分の周辺テクセルを参照するとしたら、どのようになるかというと……

注目するピクセルの周囲 1px なら合計 9px、周囲 2px なら合計 25px を同時に参照しなければならない

10%	10%	10%
10%	20%	10%
10%	10%	10%

2%	2%	2%	2%	2%
2%	4%	4%	4%	2%
2%	4%	36%	4%	2%
2%	4%	4%	4%	2%
2%	2%	2%	2%	2%

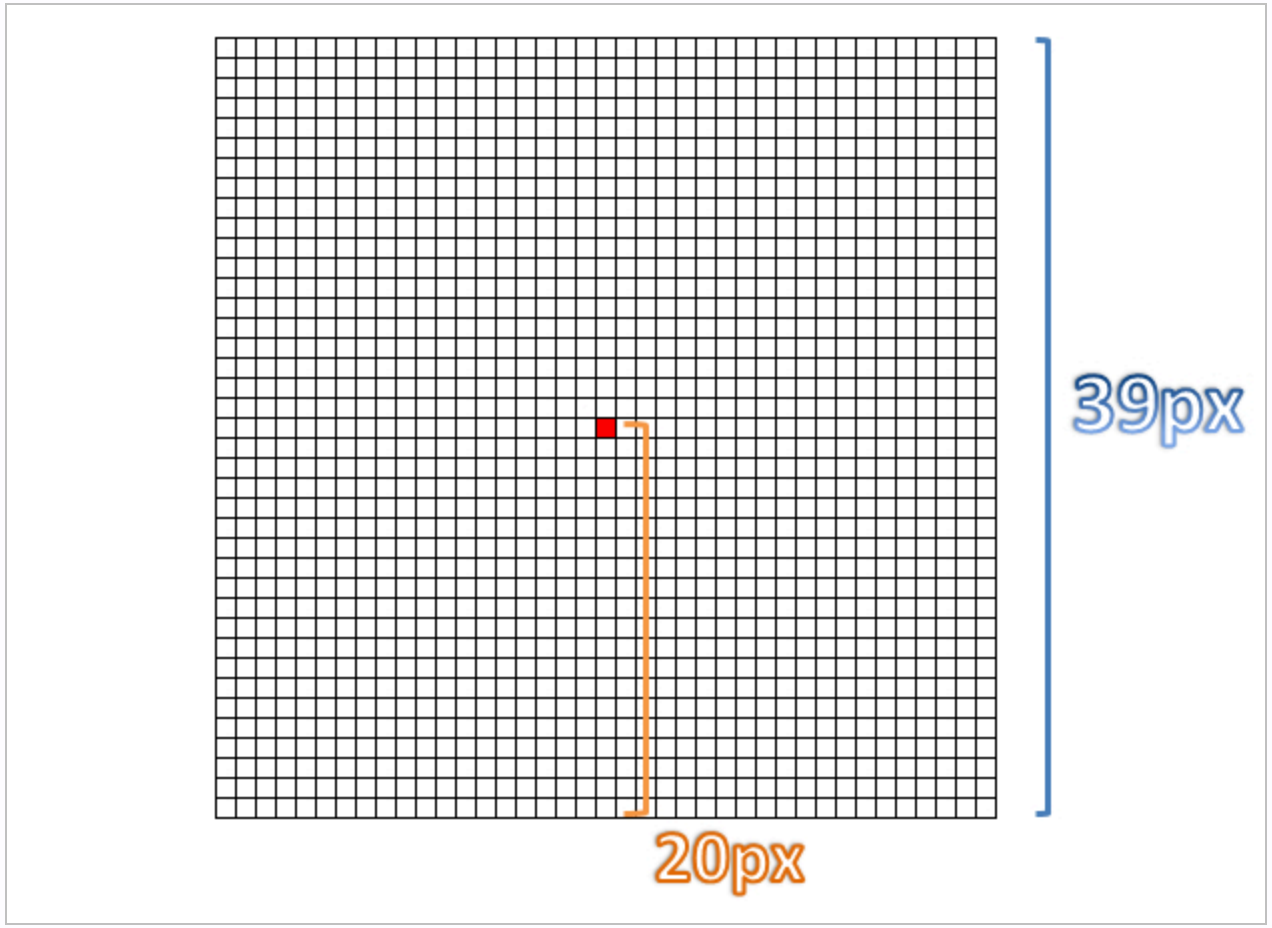
たったの周囲 2px を参照するだけで、なんと 25 回ものテクスチャの参照が必要になります。

しかも周囲 2px を参照したくらいだと、見た感じは全然ぼけていないくらいの効果しか得られません。（要は苦労の割に効果が薄い……）

> かすかに滲む程度で、ぼやけてるとはいい難い感じになる

ちなみに、今回のサンプルではガウシアンブラーを実際に使っていますが、参照する範囲は上下左右の方向にそれぞれ 19px ずつにしています。実際、それくらいの広い範囲を参照しないと十分なぼかしの効果が得られないからです。

でも先程の話からすると、2px 周辺でも 25 回参照する必要があったのに、19px 周辺の参照だと、いったいどれくらいのテクスチャ参照が発生するんでしょうか。



> 39 x 39 = 1521 !!!

さて、このように 19px 周辺を素直にテクスチャを参照してしまうと、1px の出力結果を得るために必要なテクスチャの参照回数が恐ろしいことになってしまうのが想像できます。

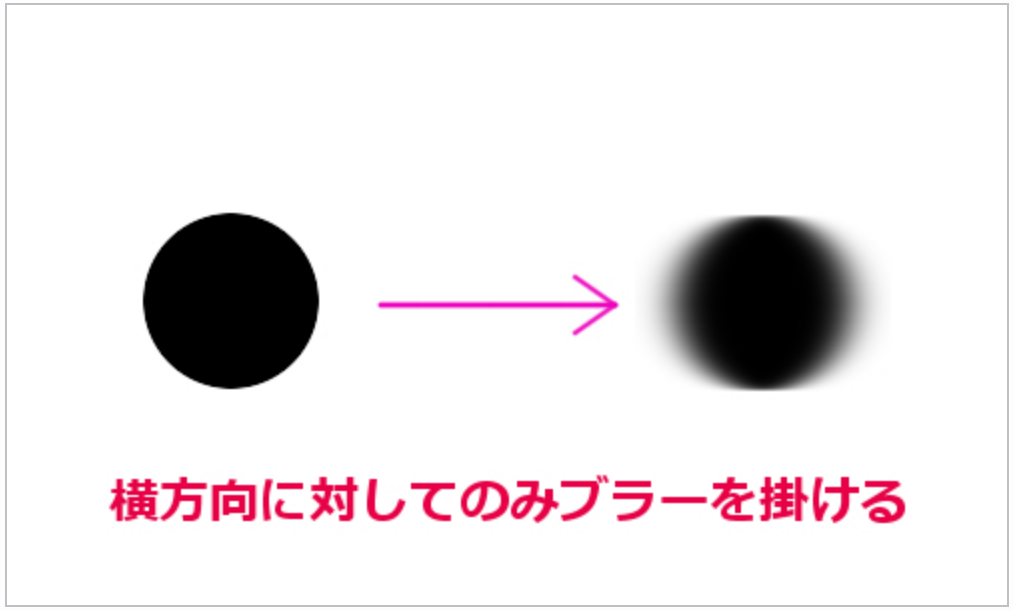
たった 1px の色を決めるために 1,521 回ものテクスチャ参照を行ってしまうと、いかに高速な GLSL を使っているとは言ってもさすがに無茶です。

そこで登場するのが必殺の **2 パスぼかし** です。

必殺 2 パスぼかし

2 パスぼかしとは、別に一般的な名称としてこのような技法があるわけではありませんが、言葉の解釈としてはそのままの意味でここでは使っています。

要するに、ぼかす手順を 2 パスに分解します。より具体的には、横方向と縦方向、それぞれを個別にぼかしてレンダリングを行います。



まずは横（あるいは縦）にだけぼかす

二度目は垂直に、異なる方向にぼかす

この方法は描画命令を 2 回発行しなければならない（つまりフレームバッファへのレンダリングを 2 回行う）という手間はあるものの、上下と左右をそれぞれ別々に処理することでテクセルの参照回数を大幅に削減することができます。

今回のサンプルのケースでも、39 回のテクスチャ参照が 2 回行われるだけで済むため、1,521 回分のテクスチャ参照に相当する処理がわずか 78 回で済むことになります。

> 効果は同じなのにトータルのテクスチャ参照回数は激減する

ただし、減ったとは言っても 78 回のテクスチャ参照ははけして軽い処理ではありません。

つまり、ぼかしという処理は、総じて高負荷であるということは忘れないようにしましょう。

ここまでのまとめ

- ガウス関数を用いることで釣鐘型に値を参照できる
- そうすることで均等に色を混ぜるよりも自然な結果になる
- ただし広い範囲の影響を考慮するとテクスチャ参照回数が膨大な回数になってしまう
- そこで描画自体を 2 パスに分離して負荷を軽減する
- それでもけして軽いわけではないので過信は禁物

ガウス係数の算出

さて、それでは実際にどのように実装すればいいのか考えていきます。

まずは、シェーダ内で色を混ぜ合わせるために必要となるガウス係数から算出します。この係数は、常に同じ内容を使い回すことができるものなので、シェーダ内で動的に求めるよりも JavaScript 側で計算して uniform 変数としてシェーダに送ります。

ちょっと行数が多いのでスライドには引用しませんが、
`App.calcGaussWeight` という関数がガウス係数を計算している部分です。

引数からは強さをどのように指定するかを設定します。強さ、というとちょっとわかりにくいかもしれませんが、大きな値を入れるほど釣鐘が扁平になりより見た目上のぼかされ具合いが強くなります。

ガウス係数が算出できたら、これをシェーダに uniform 変数として送ります。

今回の場合、シェーダへ 浮動小数点の配列 を送りたいわけですが、シェーダ内部ではこれを `float` 型として参照することになるので uniform 変数の送信に利用するメソッドは `gl.uniform1fv` を利用します。

> 1 つの float の配列を送りたいので、1、f、v となります

GLSL 側で配列を受け取るには、今回のサンプルのようにカギ括弧を利用します。

ガウス係数の算出に用いた範囲の数とイコールになるように、宣言を間違えずに行っておきます。

```
uniform float weight[20]; // 20 段階のガウス係数を受け取る
```

ガウシアンブラーでは、隣り合うピクセルを順に参照していくことになるので「ピクセル１つあたりの幅」を求めてやります。

ピクセル１つあたりの大きさを算出したら、これをずらす単位として考えてループで一気に色を取得して合算していきます。

> ピクセルと似たような言葉に「テクセル」がありますが、これはテクスチャの１画素を意味する言葉です。なのでここでピクセルと書いている部分は、実は正しくはテクセルです。

取得した色には、ガウス係数を掛け合わせてやり、それぞれの参照した色がどの程度の割合で混ざり合うのかを変化させます。

GLSL でループを記述する場合は、構文としては JavaScript などとほとんど同じですが、いくつか注意すべきポイントがありますので、次のページにまとめてあります。

GLSL のループ処理

- `float` でも記述できるが、原則 `int` を使う
- 配列のインデックスに `float` は使えないので注意
- 型をキャストする書き方が C とは違うので注意
- 不定の値をループの条件式に使えない（変数など）
- ループの終了条件における不等号の向きに注意
- 展開して書けるなら無理にループにしないこと

左右（水平）に処理する場合の例。

`vec2` の X 要素がループ毎に変化する。

```
// 20 回の繰り返し処理でぼかしていく
// ※ 繰り返しの回数は JavaScript 側の gaussWeight の引数と連動しています
// ※ GLSL 側では繰り返し回数を定数で指定する必要があるので注意！
for (int i = 1; i < 20; ++i) {
  texCoord = (fc + vec2(float(i), 0.0)) * fSize;  // 左にシフト
  blurColor += vec4(texture2D(textureUnit, texCoord).rgb * weight[i], 1.0);
  texCoord = (fc + vec2(-float(i), 0.0)) * fSize; // 右にシフト
  blurColor += vec4(texture2D(textureUnit, texCoord).rgb * weight[i], 1.0);
}
```

022

- 2 パスで処理することによって負荷を軽減する
- ぼかしは基本的に負荷が高いことに注意する
- ガウス係数は JavaScript 側で求めておき uniform で送る
- GLSL でループを書くときはちょっと癖があるので注意
- 正しくテクスチャを参照するためにテクセル 1 つ分の幅を求める

さいごに

さて、全八回にも及ぶ長期間の講義、お疲れ様でした。

これで、本講義の内容は全て終了となります。

最初は three.js を使って 3D そのものに慣れるところからスタートした本講義でしたが、ネイティブな WebGL が出てきた途端に急に難易度が上がって、正直戸惑ってしまった方も多かったかもしれません。

それだけ three.js はかなり手厚い WebGL のラッパーである、ということでもありますから、無理せず最初は three.js でどんどん作品を作っていき、慣れてきたら徐々にオリジナルのシェーダを記述するなどネイティブな WebGL に近い部分に挑戦していけばいいと思います。

講義のなかで本当にしつこく言ってきましたが、楽しくなくなったらマジでつらいだけになるのが 3D プログラミングだと個人的には思っています。実際難解なので、その状態に陥りやすいんですよね。

自分が得意とする部分や、楽しいと思える部分を見つけて、少しずつで構いませんから継続することが大切だと思います。

スクールの期間が終わったあとも、質問等は今後もいつでも受け付けます。

なにかわからないことや、相談したいことがあればいつでも気軽に声をかけてください。

一応最後に、第八回としての課題についても考えてみたのですが……

せっかくポストエフェクトの用法を扱ったので、マウスカーソルの位置に応じて変化するポストエフェクト、など実装してみると面白いと思います。

たとえば、マウスカーソルを動かしたら RGB がずれたように表示される、さらにそこにノイズを組み合わせたりしてみても面白いかもしれません。

提出期限などは設けませんので、楽な気持ちで取り組んでみてください。