フレームバッファとポスト処理

マルチパスで質感向上

前回のおさらい

前回はフラグメントシェーダをテーマにさまざまなテクニックを扱いました。

頂点そのものを直接操作する頂点シェーダとは異なり、ピクセルレベルで細かく制御を行っていくフラグメントシェーダでは見た目に対してダイナミックな変化が起こりやすいため、慣れてくるとシェーダコーディングが非常に楽しいものに変わってきます。

シェーダでは小さなテクニックの組み合わせ次第で、本当に様々なバリエーションの表現が行えます。

一見シンプルすぎて役に立つ場面が想像できないようなものであっても、組み合わせることで優れた効果を発揮することも頻繁にあります。少しずつで構わないので、どんどんシェーダを自分で書いて慣れていきましょう。

フレームバッファとポストプロセス

今回紹介する一連のサンプルでは、その多くがシェーダを複数利用する実装になっています。

これまでのサンプルでは main.vert と main.frag の 1 組だけのシェーダだけで表現を行っていましたが、今回の講義では複数のシェーダを同時に使ってより複雑な表現を行っています。

より具体的には、まず第一のシェーダによって「ベースとなるシーン」が描かれます。

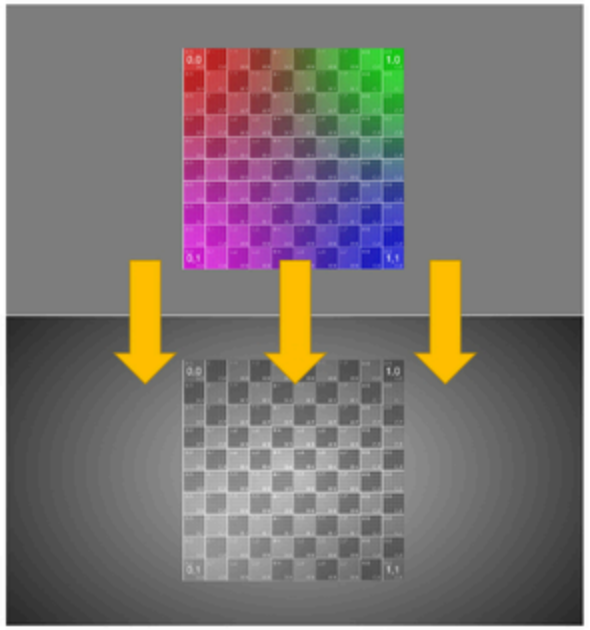
今までは、これがそのまま画面に出ていた最終的な出力結果でした。今回のサンプルでは、この「ベースとなるシーン」を元にして、そこにさらになにかしらの事後処理を加えたうえで「完成形のシーン」を作ります。

> つまり二度の描画によってシーンが完成する！

いったんできあがったシーンに対して事後的になにかしらの処理を加えるとき、その後から行う事後処理のことを一般に ポストプロセス と呼びます。

同じような意味の言葉としてはポストエフェクトと言ったりもしますし、ちょっとニュアンスは違いますが、マルチパスでのレンダリング、のように言うこともあります。

> 一枚の絵を出すプロセスを「パス」や「レンダリングパス」と呼び、それが複数回組み合わさるのでマルチパスと呼ばれる

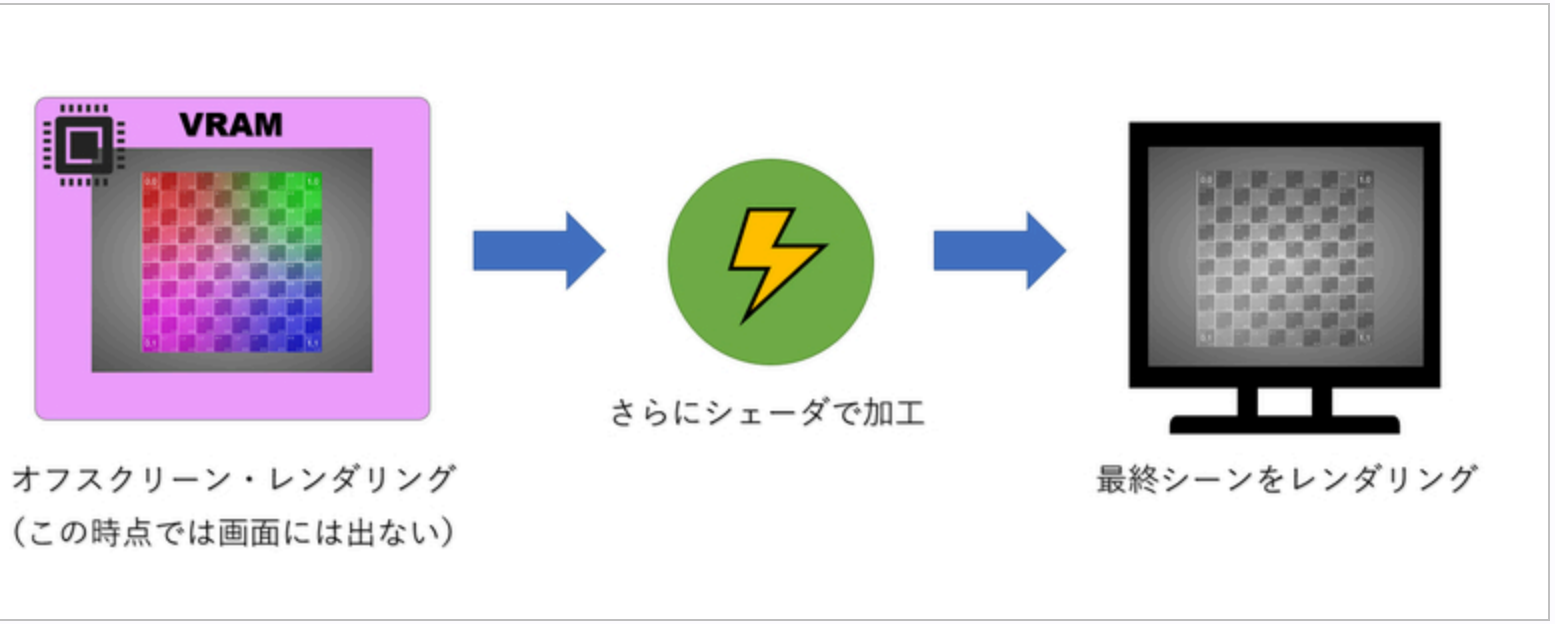


photoshop で加工するがごとく、一度メモリ空間上で（画面上は見えないが）レンダリングを行っておき、リアルタイムにこれを加工してから画面に最終的な結果だけを描画する。

上記はポストプロセスでグレイスケール化する場合の例

このようなポストプロセスを行う描画では、まず最初にスクリーンではなくメモリ上に描画を行って結果を保持しておきます。

従来は描画結果がそのままスクリーンに出ていましたが、ポストプロセスを加えたあとの完成形だけを画面に出せばいいわけですからメモリ上で描画を行った段階では画面にはなにも出力しません。このようなバックグラウンドでのメモリを対象とした描画処理は一般に オフスクリーンレンダリング と呼ばれています。



> VRAM というのは、GPU のメモリ空間のこと

今回のサンプルの多くが複数のシェーダを用いるのは、オフスクリーンレンダリングと、最終的にスクリーンに絵を出すためのレンダリングとを それぞれ別のシェーダで実行する ためです。

つまり、最初のシェーダでベースとなるシーンを描き、これは画面には出ない内部的な描画結果として保持しておきます。次に、ふたつ目のシェーダを使って今描いたばかりのベースとなるシーンを読み込み、そこに事後処理を加えながら最終的な描画結果として画面に出力します。

ここで出てきた「メモリ上の描画領域」は、一般に フレームバッファ と呼ばれています。

フレームバッファは WebGL でももちろん使うことができ、多くの場面で活用されています。

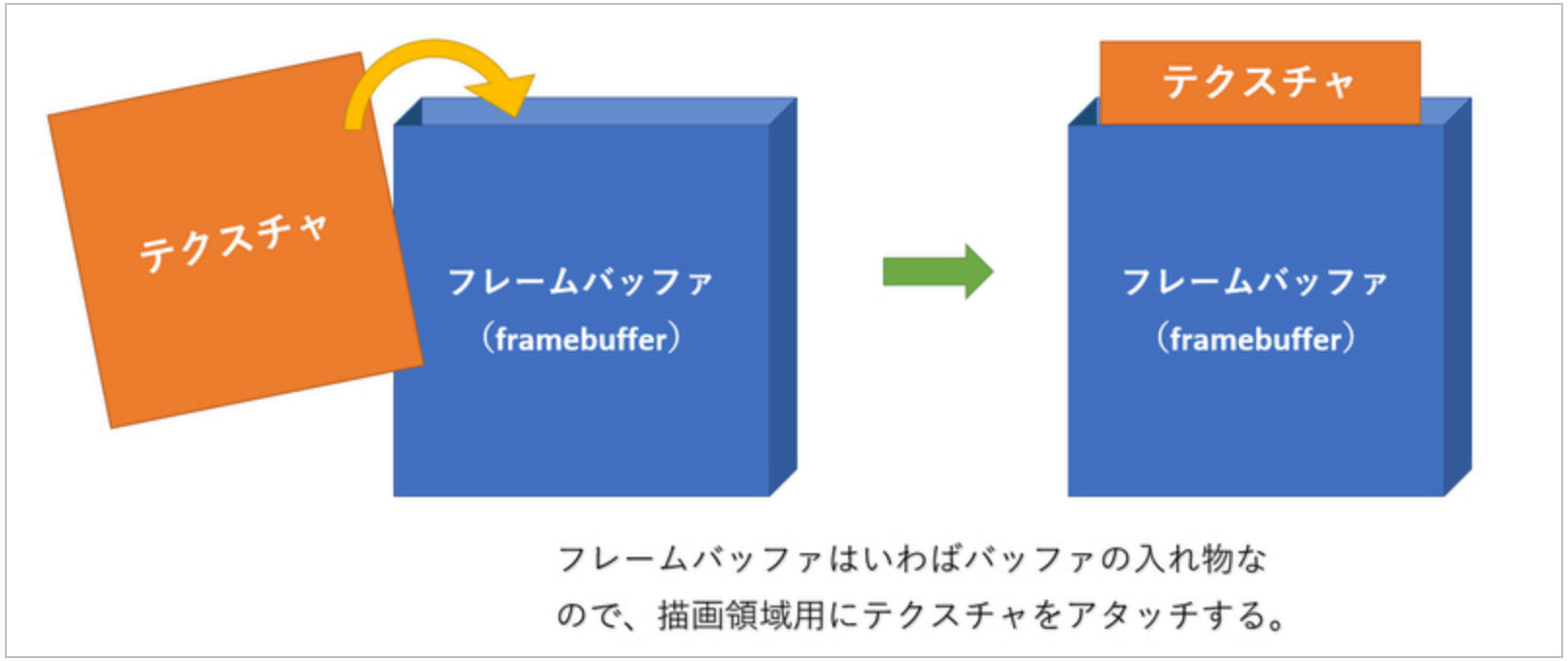
フレームバッファ

フレームバッファは、それ自身がデータを直接格納するというよりは、むしろ「データを格納するバッファ（単体、または複数）が入る箱」のようなものです。

ですからフレームバッファの初期化処理のタイミングでは、フレームバッファに格納するためのバッファ類を先に生成した上で、それをフレームバッファに対してアタッチするという作業を行います。

ただし多くの場合、フレームバッファを生成する際にフレームバッファに対して 中身が空のテクスチャを割り当て ることが多いです。（バッファの代わりにテクスチャを使う）

そうすることでフレームバッファに割り当てられたテクスチャが 描画結果を焼き付ける「一時的な描画領域」として使われ ます。



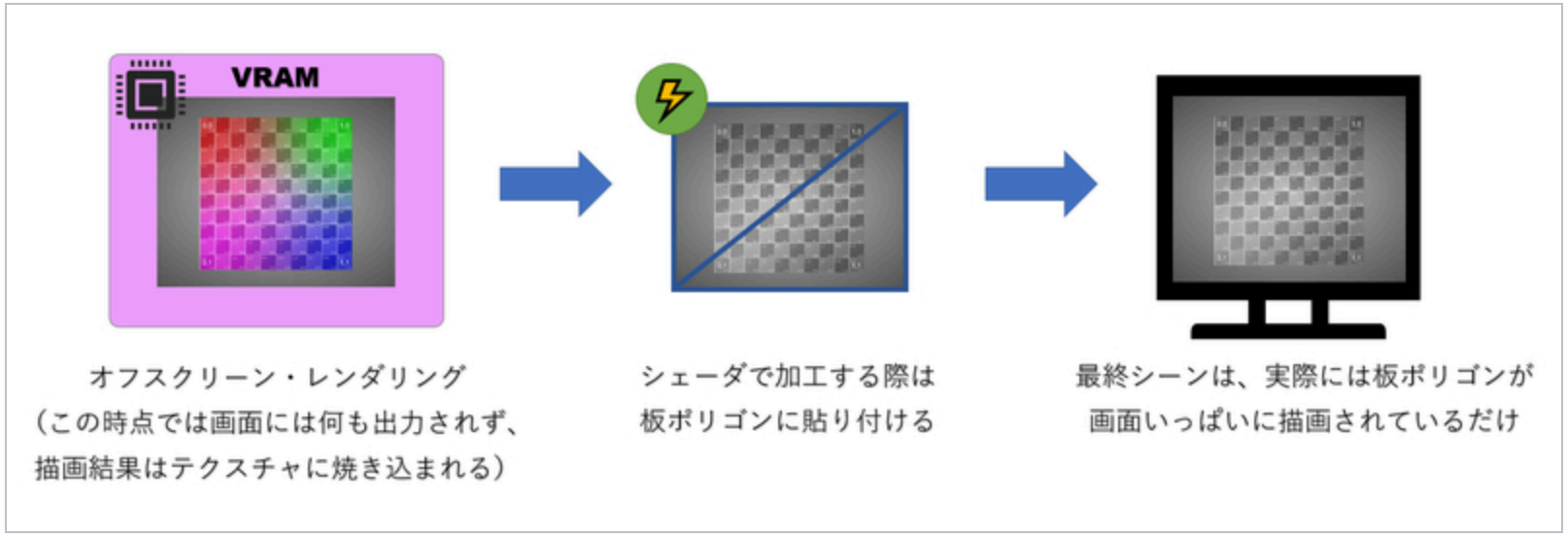
フレームバッファに対して描画を行うと、このテクスチャに結果が（つまりは色データが）焼き込まれるようなイメージ

ここで重要なのは、一度焼き込んだシーンは「要するにテクスチャ」なので、他のシェーダにこれを渡してやれば従来のテクスチャと同じようにそれをシェーダ内で利用することができる、ということです。

画像を読み込んでテクスチャとして使う場合とまったく同じように、描画結果をシェーダ内で `sampler2D` 型のデータとして読み込んで使うことができるのです。

あとは、この送り込まれてきたベースとなるシーン（第一のシェーダの描画結果）を、板ポリゴンにペタッと貼り付けて画面全体を覆うように配置すれば、これでポストプロセスを含む一連の描画プロセスを行うことができます。

フラグメントシェーダは 頂点やポリゴンが描画される全てのピクセルで実行される ので、画面全体を覆うような板ポリゴンを描画するということはすなわち、画面全体に第二のシェーダを実行した結果を出せるということになります。



> 一見すると直接 3D モデルなどが描画されたように見えるが、実際はそこにはポリゴンが一枚置かれているだけ

フレームバッファまとめ

- ポストプロセスでは複数のシェーダを使う場合がほとんど
- ベースとなるシーンを描く第一のシェーダと……
- ポストプロセスを行う第二のシェーダを用意する
- 第二のシェーダは画面全体を覆う板ポリゴンの描画に使う
- これにより画面全体（すべてのピクセル）を漏れなく第二のシェーダによって処理することができる

フレームバッファ周りの処理を確認

さて、ポストプロセスによる複数回の描画の流れがざっくりと理解できたら、実際にコードを見ながら考えていきましょう。

フレームバッファを生成するプロセスは、コードの量だけを見ると非常に複雑に見えるかと思いますが…… 重要なポイントは、フレームバッファは要するにバッファの入れ物であり、これにレンダリング結果を焼き込むためのテクスチャをアタッチする、ということです。

```
createFramebuffer(width, height) {
   (中略)
  // フレームバッファを生成
  const frameBuffer = gl.createFramebuffer();
   (中略)
  // テクスチャを生成
  const fTexture = gl.createTexture();
   (中略)
  // フレームバッファに、カラーバッファとしてテクスチャをアタッチ
  gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, <長いので省略>);
```

> ここでアタッチしたテクスチャが描画領域として使われる

サンプル 019

- 複数のシェーダを初期化しておく
- まずベースのシーンを描画し……
- フレームバッファやシェーダを切り変えて……
- 最終シーンは板ポリゴンだけを画面いっぱいに描画！
- この板ポリゴンの描画時に、第二のシェーダが仕事をすることで、画面全体にエフェクトが掛かる

> インデックスバッファ（IBO）や、法線を変換するための行列の話など、ここでは主たるテーマではないので説明を省略していますが気になる方は Discord などで別途質問してください！

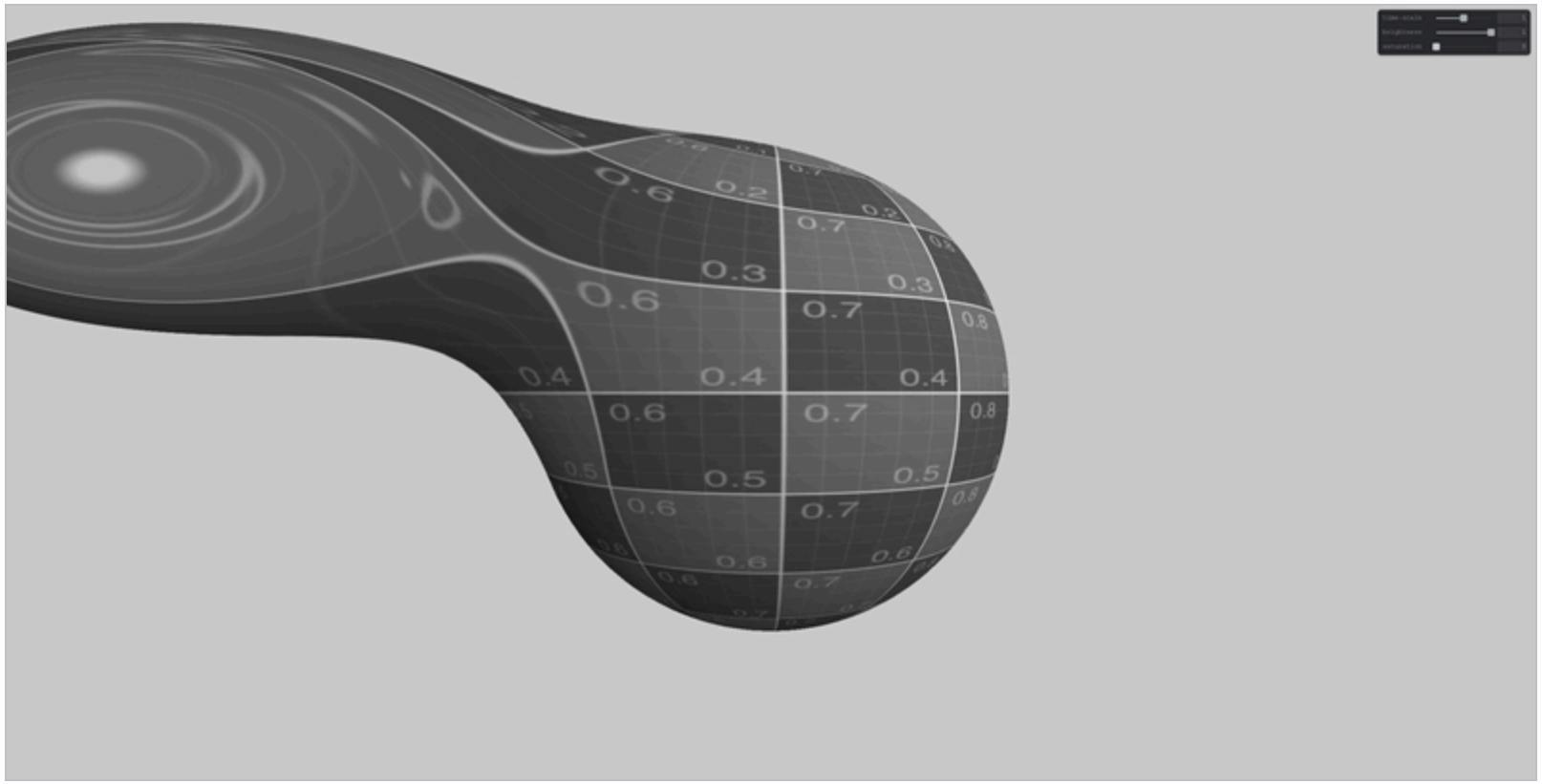
ディストーションとグレイスケール

さて、ここからはフレームバッファを活用したポストエフェクトの実装例をいくつか見ていきましょう。

まず最初はカーソルの位置に応じたインタラクティブなディストーション＋グレイスケールです。

マウスカーソルの位置に応じてディストーションが発生する、という演出はウェブの世界でもよく見かけます。

そのパターンは無数にあるのですべてを紹介することは難しいですが、ここではカーソルの位置に近ければ近いほどテクスチャ座標を収縮させています。



一見なにが起こっているのかわかりにくいが、画面の中心を原点とし、その原点に向かってカーソル位置のテクスチャ座標が収縮している

ここでは GLSL のビルトイン関数 `smoothstep` を使うことで、マウスカーソルからの距離に応じた変化が、より滑らかになるようにしています。

```
// 最小値 ～ 最大値の範囲に対して、値をエルミート補間
// 戻り値は 0.0 ～ 1.0 の範囲を返す
float result = smoothstep(最小値, 最大値, 値);
```

参考：The Book of Shaders: smoothstep

さらにこのサンプルではグレイスケールも併用しています。グレイスケールは、モノクロ化とも同義です。

もう少し踏み込んだ言い方をすると、RGB の 3 つのチャンネルによって表現されていた色情報を「明暗のみで表せる 1 チャンネルの情報へと変換する操作」というように考えても良いでしょう。

> つまり明度だけを可視化している、とも言える

GLSL でシェーダを書いていると、扱っている情報を単一のチャンネルに落とし込みたいケース（RGB のようなマルチチャンネルではなく単一のチャンネルだけで表せる状態にしたい）というのが比較的よくあります。

たとえば、カラー画像の色の差分を検出して線画のような質感にするポストエフェクトなどを実装しようと思ったら、グレイスケールと組み合わせたほうが都合が良い場合があったりします。

Yuichiroh Arai さんの以下の作例では、テクスチャから「色の差分が大きい箇所」を検出して、その部分にまるで線画のような黒いラインを描画しています。

このような実装では、色の差分情報を「単なる値の大小」で表すので、グレイスケールのようなロジックをシェーダ内で行ったほうがよい場合もあります。

Illustrized by Yuichiroh Arai | Experiments with Google

グレイスケールにはいくつかやり方があり、たかがグレイスケール、されどグレイスケール、実は結構奥が深いです。

グレイスケールの闇に触れてみたい方は以下の記事など目を通してみるといいかもしれません。

参考：グレースケール画像のうんちく - Qiita

今回のサンプルの場合は、物理的に正しいかどうかなどは一切無視して、非常に簡単なグレイスケール化を行っています。

以下のコードが、グレイスケール化を行っている部分になります。

```
// グレイスケール化した色をつくる（内積を用いた簡易的なグレイスケール化）
float gray = dot(samplerColor.rgb, vec3(1.0)) / 3.0;
```

一見すると、このコードでどうしてグレイスケール化できるのかがちょっとわかりにくいかもしれません。

ベクトルの内積の性質を利用して、RGB の各チャンネルの各値を 全部加算してから 3.0 で割る という処理を行っています。

ベクトルの内積は次のような計算を行います。これを見ると、実際には足してから３で割ってるだけなんだな～というのがわかると思います。

```
// 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 out = dot(v, w);

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

今回のグレイスケール化は、先程も書いたとおりで物理的に正しいかどうかなどは一切考慮せずに、単純に加算して平均を取っているだけのかなりシンプルなものです。

また、ベクトルの内積に少しでも慣れてもらいたい気持ちもあったので、ここではあえて内積を使っています。内積を使わないとグレイスケール化が行えないということではないので、その点は読み違えないように注意してください。

サンプル 020

- `smoothstep` を使うと手軽に補間結果を滑らかにできる
- `smoothstep` はイージングに例えると ease-inout 的な補間
- テクスチャ座標は 0.0 ～ 1.0 なので必要に応じて原点を中央に持ってくるような変換を活用する
- グレイスケールは RGB の情報（三次元）を明度という単次元の情報に変換する処理とも言える
- ここではベクトルの内積で簡易的に明度を求めている

RGB ディストーション

色収差エフェクト

続いてのサンプルでは、比較的手軽に利用できるテクニックである RGB ディストーションを行っています。

このようなエフェクトを加えると、レンズで光が屈折しているような雰囲気を演出するなど、グラフィックスとしての情報量を増やして印象を変化させることができます。

RGB ディストーションのアイデアはとてもシンプルで、RGB の各チャンネルごとに参照するテクスチャ座標を少しずつずらします。

たったそれだけのシンプルなアイデアですが、歪む量を中心から離れるほど強くする、などの効果と組み合わせることでより面白い効果が得られます。

サンプル 021

- 歪む範囲や、歪む量などを uniform 変数として送っている
- 歪み処理周りは単純な算数なので落ち着いて考えよう
- さらに RGB のそれぞれに固有のテクスチャ座標を用意
- 最終的に RGB を合成して出力する

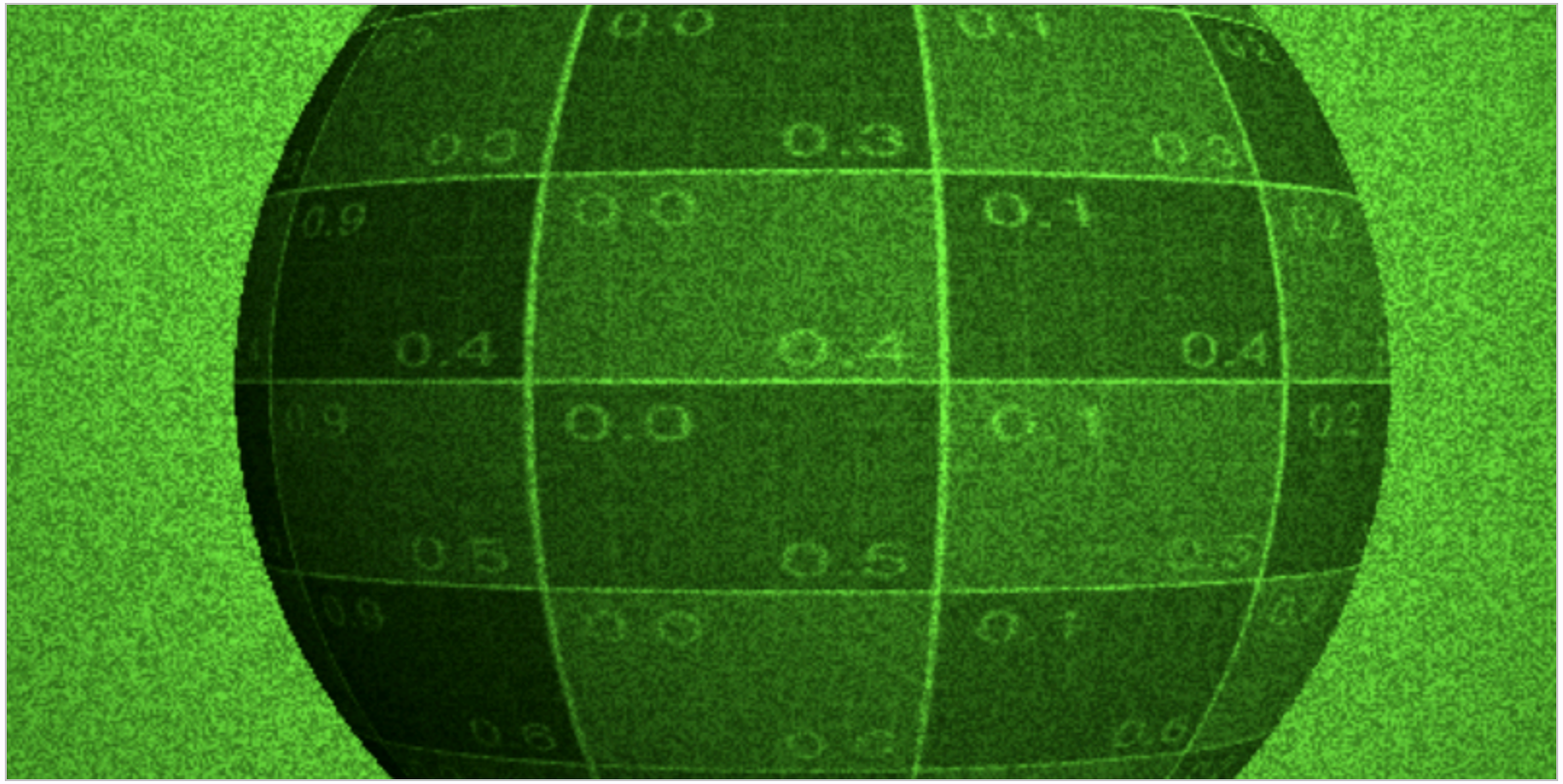
ホワイトノイズ

さて、続いてはシェーダを扱う上では欠かすことのできない 超重要な概念 のひとつ、ノイズ（Noise）を使ってみます。

ノイズは、乱数（ランダムな値）を元に作られます。乱数と言われると、そんなに「超」がつくほど役に立つの？ と思うかもしれませんが、これがとっても大事なのです。

まず一番最初に、ランダムな値を最もシンプルに描画結果に反映させるホワイトノイズを使ってみます。

ここでは、砂嵐のような描画結果が得られるのですが…… この描画結果のどこに乱数が関わっているのか、描画結果を見てわかるでしょうか？



この描画結果のどこにランダムが関係している！？

フラグメントシェーダが「ピクセルレベルで実行されるもの」ということがわかっていれば、描画結果をよく観察すると、どの部分にランダム性が生まれているのかはわかるのではないでしょうか。

答えを書いてしまうと、このサンプルでは乱数から得られたランダムな値を、色の明るさとしてそのまま使っています。

サンプルのポストプロセス用シェーダ（post.frag）を見ると、なにやら怪しげな関数が定義されているのがわかると思います。この関数が、ホワイトノイズを生成する元となる、ランダムな値を生成しています。

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

GLSL には（バージョンにもよるのですが）基本的にノイズ（乱数）を生成してくれるビルトイン関数はありません。

ビルトイン関数が無い、ということは自前で実装しないといけません。ここで出てくるマジックナンバーだらけの怪しげな関数は、GLSL でノイズを生成するときによく利用される関数で、コピペで伝承されている作者不詳の乱数生成器です。

そしてホワイトノイズは、ランダムに得られた数値を特に加工などはせずにそのまま用いている場合によく使われる呼称です。

ホワイトノイズは正確には、値が一定以上均等に分布していること、などの定義がある（らしい）ので、今回のシェーダで再現しているホワイトノイズは厳密にはその定義を満たしていません。人が目で見た感じでは、ほとんど完全にバラバラな値が取得できているように見えるんですが、実際は結構偏ってます。

ちなみに、先ほどのシェーダのコードにあった謎の関数で、どうして乱数が取得できるのかは以下のページを参考にするとわかると思います。

参考：The Book of Shaders: Random

> 波打つ模様が収縮して結果的に散漫に値が取得されている

このように、乱数を生成するロジックには、今回のシェーダで利用しているような実装も含めて非常に多くの種類があります。

とは言え、その乱数生成のロジックによって、品質の良い乱数や、すぐに繰り返し値が出てきてしまう品質のあまり良くない乱数というのがあり、そういう意味では今回の乱数はあまり品質がいいとは言えません。

コンピューターが扱えるビット数には物理的な限界があります。ですから本当に無限にランダムな数値を生み出すことはできないので、そういう意味では、どういった実装の乱数生成器であってもいつかは同じ値の繰り返しが出てきてしまいます。

品質の高い乱数を生成できる乱数生成器は大抵は計算の負荷が高くなるので、そこは用途に合わせて、適切な品質をチョイスして使っていくことになります。

> メルセンヌ・ツイスタなどが高品質な乱数生成器として有名です

つまるところノイズとは

- ものすごくざっくり言うと乱数のこと
- 乱数にも品質の善し悪しがある
- シェーダでは速度と品質のバランスを見て使い分け
- fract sin noise は結構偏った結果になる（仕組みがわかれば納得）
- 乱数やノイズの世界は沼……

> JavaScript の乱数は Xorshift というアルゴリズムで実装されていることが多いです（軽さと性能のバランスがよいためと思われる）

サンプル 022

- ポストプロセスでノイズを生成して合成している
- 乱数の生成にはいろいろな実装の種類がある
- 乱数の値をそのまま色に乗算しているので明暗が生まれる
- サイン波を使って明暗を変化させたり……
- 画面中央からの距離に応じて暗くしたりしている

> サンプル内の rnd と rnd2 は、整数部分の桁数の違いがあるため Mac 環境で多少マシな結果が得られるなど、若干の違いがあります

バリューノイズ

ホワイトノイズを利用したナイトスコープ風シェーダは、これはこれで非常に面白い効果ではあるのですが…… あくまでも、乱数をそのまま直接利用している感が強いエフェクトだと言えます。

CG の世界では、乱数をさらに活用し発展させ、複雑なノイズを生成することでより高い表現力を得ることを目指す場合が多いです。

ノイズの実装は大変奥が深いジャンルですが……そのなかでも、まだ簡単だなと個人的に思うのがバリューノイズと呼ばれるノイズの実装です。

これはかろうじて私でも解説できるので、今回はこれを簡単に説明します。

ちなみに、ノイズについては日本語で書かれたものだと以下の記事がとても参考になると思います。

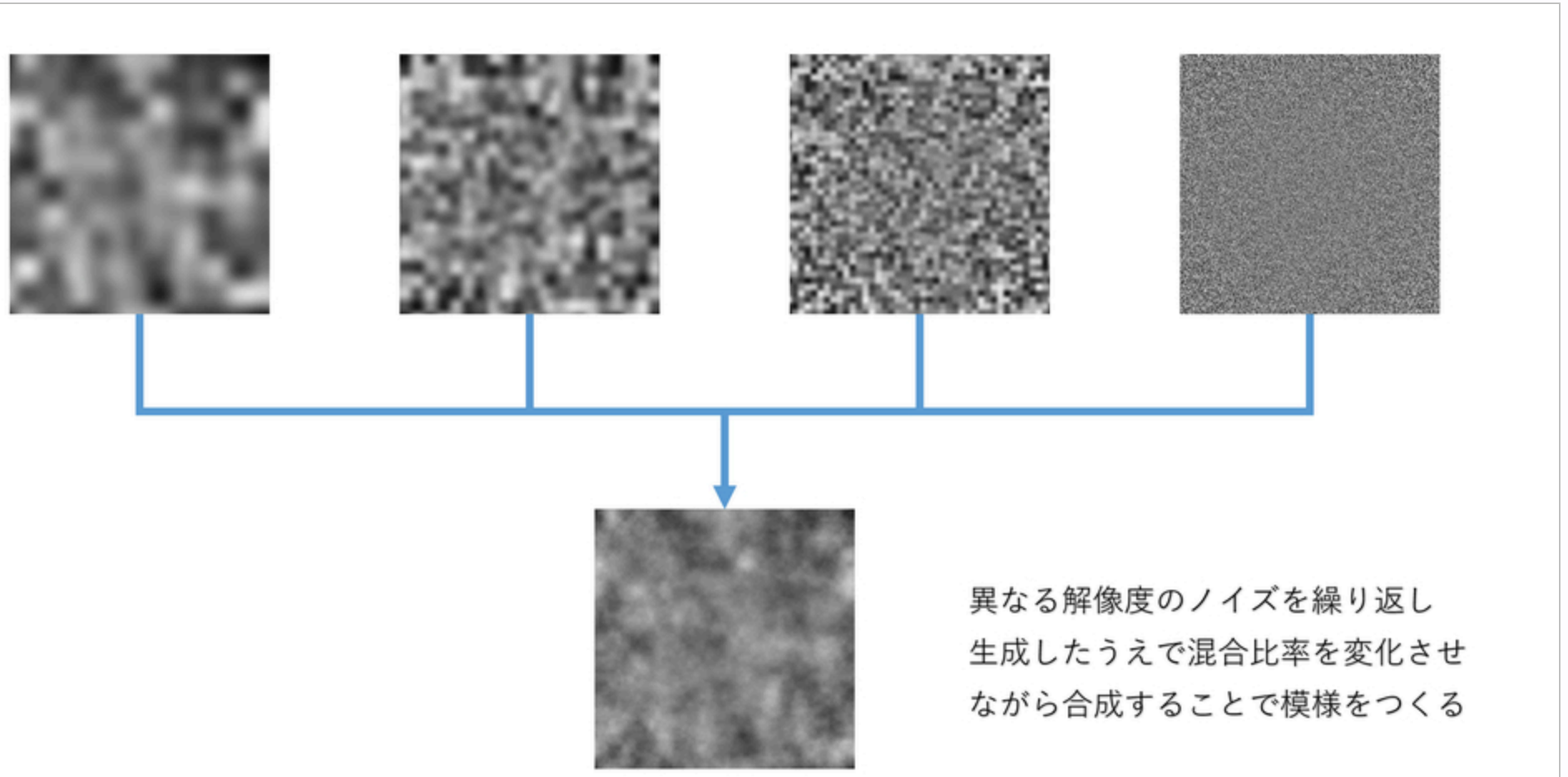
ノイズの話 - Pentanium Blog

他に、GLSL でのノイズ実装にどんなものがあるのか、とりあえず触れてみたければ以下のページも参考にしてみると良いかもしれません。

GLSL Noise Algorithms

先ほどの参考記事のなかでも触れられていますが、バリューノイズは補間などが加えられることになるので単なるホワイトノイズに比べてより複雑な表情のある模様を作ることができます。

原理的には、異なる解像度のノイズをシェーダ内で動的に生成してやり、それをうまく合成して模様を作っています。


異なる解像度のノイズを繰り返し
生成したうえで混合比率を変化させ
ながら合成することで模様をつくる

サンプル 023

- fract sin noise を使って乱数を生成
- それを補間して滑らかにつなげる
- 複数の解像度のノイズを生成して合成
- 最終的には雲や煙のような模様（値の分布）を生み出す
- トータルではかなりの回数の乱数生成が行われている
- 煙や炎の表現にも流用しやすい

> ノイズの動的生成は基本的に重い処理になるので、環境によってはあらかじめノイズが焼き込まれた状態のテクスチャを画像ベースで用意したほうがいい場合も

Simplex Noise

WebGL でのウェブ制作で、よく用いられる有名なノイズに Simplex Noise（シンプレックスノイズ）があります。

このノイズはバリューノイズよりもなだらかに値が変化するノイズで、テクスチャ座標を歪ませたり、複雑なグラデーションを生成したりと、かなりいろいろなところで利用されています。

ashima/webgl-noise

> 水面の波やグラデーションの描画に使いやすいなだらかな値の変化

Simplex Noise は本当に多彩なユースケースがあるので、徐々に見慣れてくると「あ、これ Simplex Noise だな〜」というのが結構すぐにわかるようになります。

今回は、ノイズをうまく活用して左右で描画結果が異なるトランジションエフェクトをやってみましょう。

サンプル 024

- Simplex Noise は戻り値が -1.0 ～ 1.0 になる点に注意
- 思考の順番としては、まず単純にカーソルの位置に応じて描画結果を左右に分割することを考える
- 境界線の位置をノイズによってずらしてやる
- ただずらすだけでは境界が目立つので（そういう表現ならそれはそれでよいが、ここでは）光のラインを重ねている

> 比較的単純な計算をしているだけなので落ち着いて考えよう

おまけの高難易度シリーズ

さて GLSL スクールも、本講義のほうはいよいよ佳境です。

ここからは「高難易度シリーズ」としていくつかのサンプルを紹介します。

この高難易度サンプルは、本当に若干難易度が高いものなので、おそらくですが現段階で理解することはかなり難しい作例になるかと思います。

それでも、このサンプルをあえて紹介するのは、今はちょっと理解が難しくても「続けていけばこういうことができるようになるんだ！」ということをみなさんにお伝えしたいからです。（解説もサッと簡易的に行うに留めます）

> なので肩の力を抜いてまずは触って楽しんでみてください

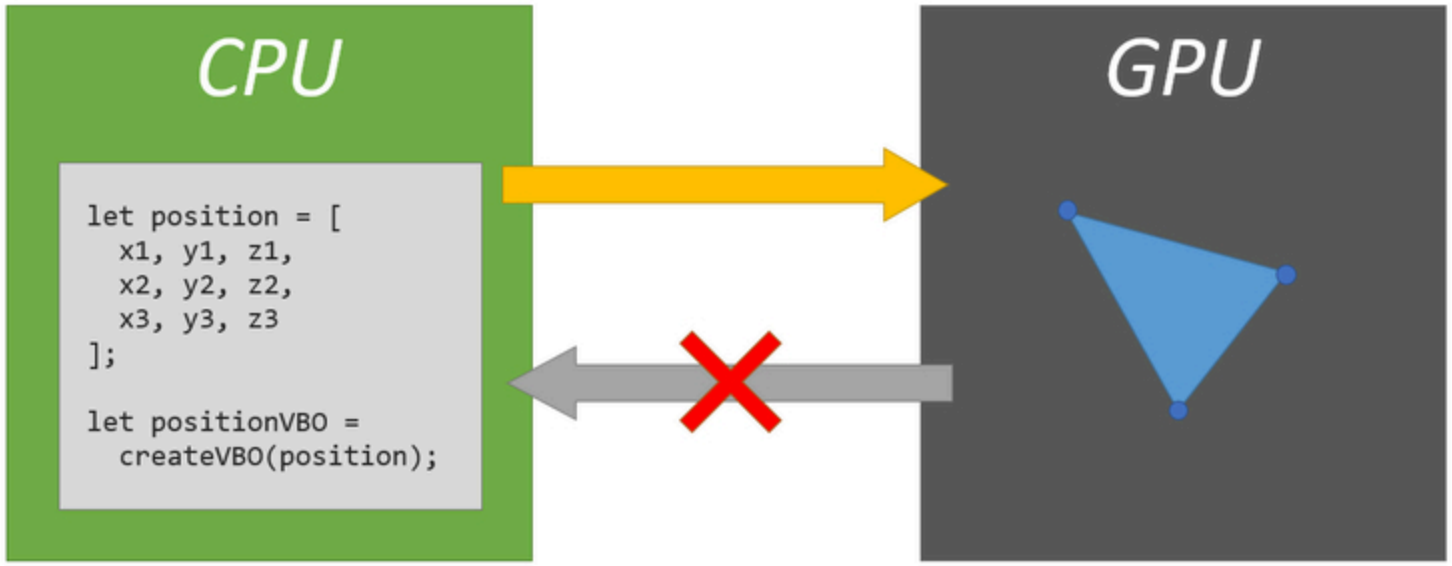
transform feedback

高難易度サンプル第一弾は、transform feedback と呼ばれる機能を使ったものです。

これまでのサンプルでは、頂点の最初の定義は常に JavaScript 側で一括して行っていました。つまり、VBO の生成（attribute 変数用のデータの生成）は基本的に CPU 側で行う作業だったわけです。

頂点シェーダは、CPU 側で定義された情報を attribute 変数を経由して受け取り、それを元に頂点を処理していました。

つまり構図としては「常に一方通行」なんですね。CPU で定義して GPU に送る、という以外のフローは通常のレンダリングでは発生せず、たとえば GPU 側で行った処理の結果を CPU 側で受け取ったり、あるいは別のシェーダに送りつけたりすることはありませんでした。



CPU 側で定義した頂点情報が VBO となり GPU に送られる

GPU で処理した結果を CPU 側に戻す仕組みは基本的に無かった

つまり前のフレームの結果を引き継いでいるのではなく、毎フレーム全部計算をやり直している、とも言える

この限界を突破することを可能にするのが transform feedback です。

ちなみに、transform feedback は WebGL の場合は WebGL 2.0 以降でないと使えません。

transform feedback を利用すると、GPU で処理された頂点の情報を、そのまま VBO に書き戻す ということが行えるようになります。

つまり、これまで CPU 側で一度最初の定義を行うと、その後一切触ることの無かった「頂点属性の情報」を、GLSL を使ってバリバリ書き換えていくことが可能になります。

CPU 側でゴリ押しで頂点情報を更新するのは、大量のループ処理などを行う必要があってなかなか難しいですが、GPU であれば超高速に頂点情報を書き換え続けることができます。

この機能を利用して、頂点の座標を更新し続けながら描画を行うのが transform feedback を利用したサンプルの実装になります。

サンプル 025

- transform feedback の準備は結構手数が多く面倒だが……
- GPU で VBO の中身を動的に書き換えることができる
- これは一般に GPGPU と呼ばれる技術に含まれる
- 頂点の座標をシェーダで更新して VBO に書き込む
- シェーダで更新した頂点情報で第二のシェーダを実行する
- 第二のシェーダでポイントスプライトで頂点を描画している

テクスチャを利用した GPGPU

先程も少し書いたとおり、transform feedback は特定の目的を達成するために大変便利なのですが、反面 WebGL 1.0 環境では利用できないなど制約もありました。

それに対して、WebGL 1.0 のような、transform feedback を利用することができない環境でも GPGPU を行う手法もきちんと存在します。

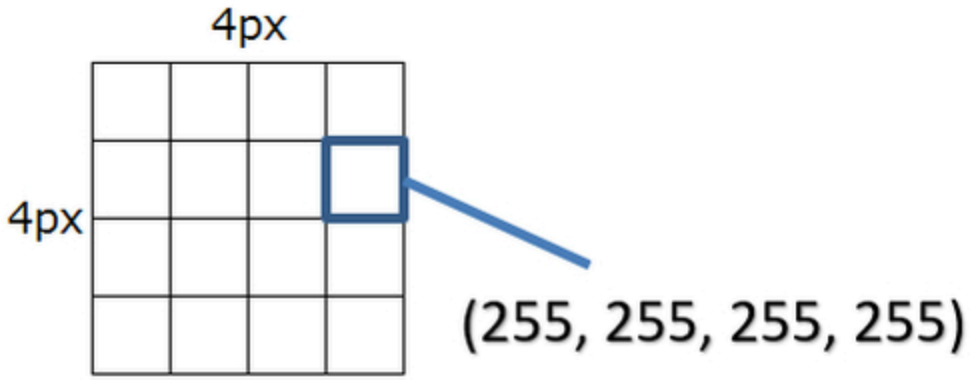
> きちんと存在、とは言いつつちょっと変化球的なアイデアではあります……

その手法とは テクスチャに対して色ではなく頂点の座標を焼き込む ことで VBO を介すのではなく、テクスチャを介して頂点座標の情報をやりとりするという方法です。

このアプローチであれば、WebGL 1.0 環境でも、もちろんテクスチャは使うことができるわけですから GPGPU を実現することができます。

テクスチャというと「画像やビットマップ」というイメージがどうしても強いかと思いますが…… テクスチャには RGBA の色情報が格納されているわけで、これは見方を変えてみると要するに「vec4 のデータが密集している配列のようなもの」とも言えるわけです。

ですからテクスチャに対して「色の RGBA ではなく、頂点の座標 XYZ などを焼き込む」というふうに思考を転換することができれば、テクスチャって要は超巨大な配列みたいなものなのだと考えることもできます。



たとえば 4x4 のテクスチャなら、長さ 16
の配列と同じようにみなすことができる。
しかもそれぞれのピクセルは長さ 4 の
配列と同じように vec4 になってる！

> 4x4 のテクスチャですら、配列として考えたら長さは 16 もある

一般的なテクスチャは、色を管理するために「8bit 精度で RGBA を表現」しています。CSS なんかでよく出てくる `#ffffff` みたいなのがありますが、これは 16 進数を使って 0 ～ 255 の範囲で RGB を表している記述法です。

しかし、頂点の座標を書き込むという用途では 8bit という精度は正直精度がまったく足りておらず、使い物になりません。

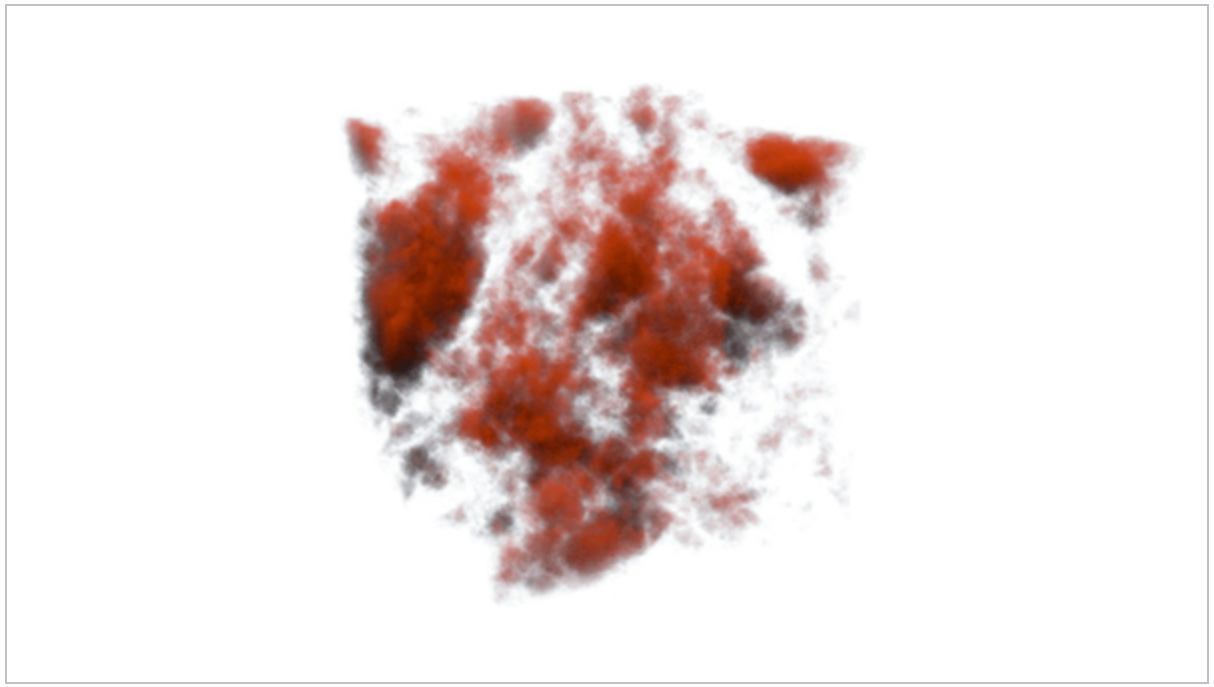
そこで、WebGL 1.0 の拡張機能である「浮動小数点テクスチャ」を利用して、座標の保持にも耐えうる精度の高いテクスチャ（Float texture）を生成して利用します。

iOS Safari などの一部の環境では浮動小数点テクスチャが使えず、半精度浮動小数点テクスチャで代用する場合もあります。

> WebGL や OpenGL では「あえて有効化しないと使えない機能」があり、浮動小数点テクスチャもそんな拡張機能のうちの１つ

ちなみにですが、サンプルでは Curl Noise と呼ばれる「ノイズを利用した流体っぽい座標の遷移」を行っています。

これは「頂点の座標をシード（種）にして 3D のノイズ空間から値を取り出し、それを進行方向として利用する」というもので、立体的なノイズを生成できる 3D ノイズと組み合わせて利用します。3D のノイズというとちょっとイメージが難しいかもしれませんが、次の図のような感じです。



> バリューノイズが平面上に分布した濃淡（高低）だとしたら、三次元以上のノイズは文字通り空間に分布した濃淡（高低）だと言える

このノイズで満たされた 3D 空間のなかを参照しながら頂点が進む方向を制御すると、まるで流体のような外観を実現できます。

事実、Curl Noise はそもそも、流体の計算が重すぎて実用に向かなかった時代、なんとか流体のような動きを再現することはできないだろうかというところから発想された手法でもあります。

サンプル 026

- Curl Noise という特殊なノイズを使って頂点座標を更新する
- 3D 空間を埋め尽くすノイズを使って進行方向をランダム化する
- サンプルでは Simplex noise を利用
- 更新した情報はテクスチャに対して出力してやり、別のシェーダで読み込んで使う
- 座標情報を書き込むテクスチャは浮動小数点テクスチャなどを利用する必要があるので注意

ラプラシアンフィルタを使った波

wave equation

波が伝搬する、あるいはバネが伸縮するといった運動を考えるときに用いられる波動方程式を活用し、水面に波紋が広がるような表現を行うことができます。

ここでは数学的なことまでは踏み込まずに、単純に実装方法だけを簡単に説明します。

フレームバッファに対して書き込むのは、平面の「高さ」の情報です。

初期状態では高さはゼロであり、それをポストプロセスで可視化した結果（スクリーン上の描画結果）も真っ黒です。

> 一切干渉していない場合は高さがゼロとなり、色で表すなら黒になる

カーソルでクリックしたときに、フレームバッファの該当する座標に高さの情報を加えます。

高さは自然と伝搬していくのですが、このときにラプラシアンフィルタを使って速度・加速度を求め、その影響で高さが常に変化し続けるようにします。

0	1	0
1	-4	1
0	1	0

ラプラシアンフィルタのカーネル

※カーネルは、画像処理などの分野で各ピクセルに対してどのような係数を割り当てるかを示したもの

レイマーチングで法線を求めた方法と考え方は似ていて、ここではラプラシアンフィルタを用いて隣接ピクセルを参照して求められる差分から、最終的に該当フレームの高さを求めている

サンプル 027

- フレームバッファに、高さと速度を書き込む
- 次のフレームで、前のフレームの情報を参照する
- 参照する際にラプラシアンフィルタを活用して加速度を算出
- 波が伝搬していく様子を近似することができる
- 負荷軽減のため縮小バッファを使っている

Stable fluids

GLSL を用いた実装のなかでも、特に見た目のインパクトが強い実装に「流体表現」があります。

今までにも多くの実装例があり、どちらかというと枯れた技術なのですが…… しかし難易度がちょっと高いので、きちんと理解して実装するのはなかなか大変です。

詳しい内容は、論文を直接当たるのが一番です。以下の資料は Stable fluids をゲーム用に軽量化した実装に関するもの。（PDF 結構重いので注意）

Real-Time Fluid Dynamics for Games

Real-Time Fluid Dynamics for Games をより噛み砕いたプレゼンテーション

> 正直なところ、数学的な理解は私自身も若干怪しいです……

今回のサンプルは、できるだけ愚直に、論文のとおりに実装したものであまり大きな最適化などは行っていません。

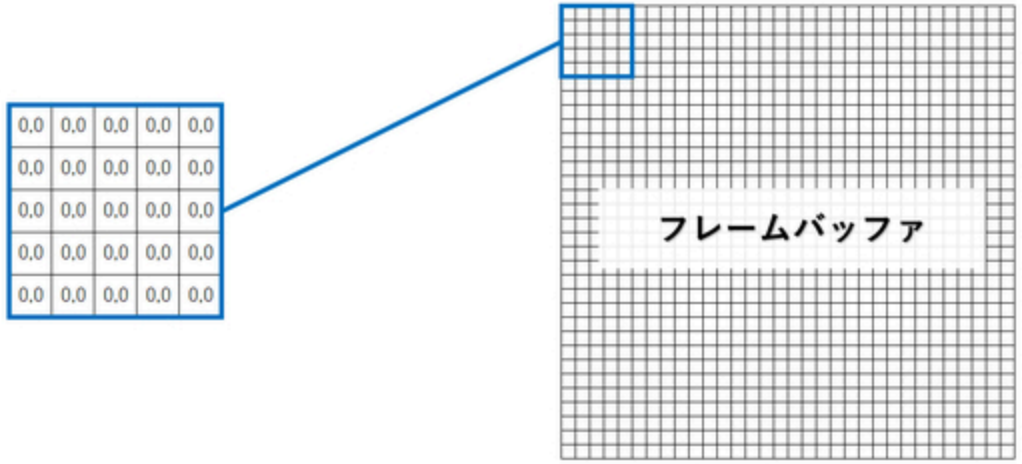
一応、実行環境の Device Pixel Ratio を参照して実行するようにはしてますが、それでもやっぱりちょっと重いのでそのまま案件とかに流用するのはちょっとむずかしいかもしれません。

論文や実装中に出てくる基本的な用語・概念のうち特に重要なものの意味は以下のとおり。

- Velocity（進行方向と速度を意味するベクトル）
- Density（密度・濃度を意味する浮動小数点）
- Diffuse（拡散の意）
- Advect（移流・流入の意）

基本的な考え方は、まず Velocity を確定させ、その Velocity に応じて濃度を変化させる、というアプローチです。このことから、フレームバッファは Velocity 用と Density 用にそれぞれ別々で用意します。

Velocity とは先程も書いたとおり「進行方向」と「速度」を意味するベクトルとして表されます。この Velocity は、フレームバッファ（浮動小数点テクスチャを含む）に焼き込まれた状態になっていて、初期状態ではゼロベクトルです。



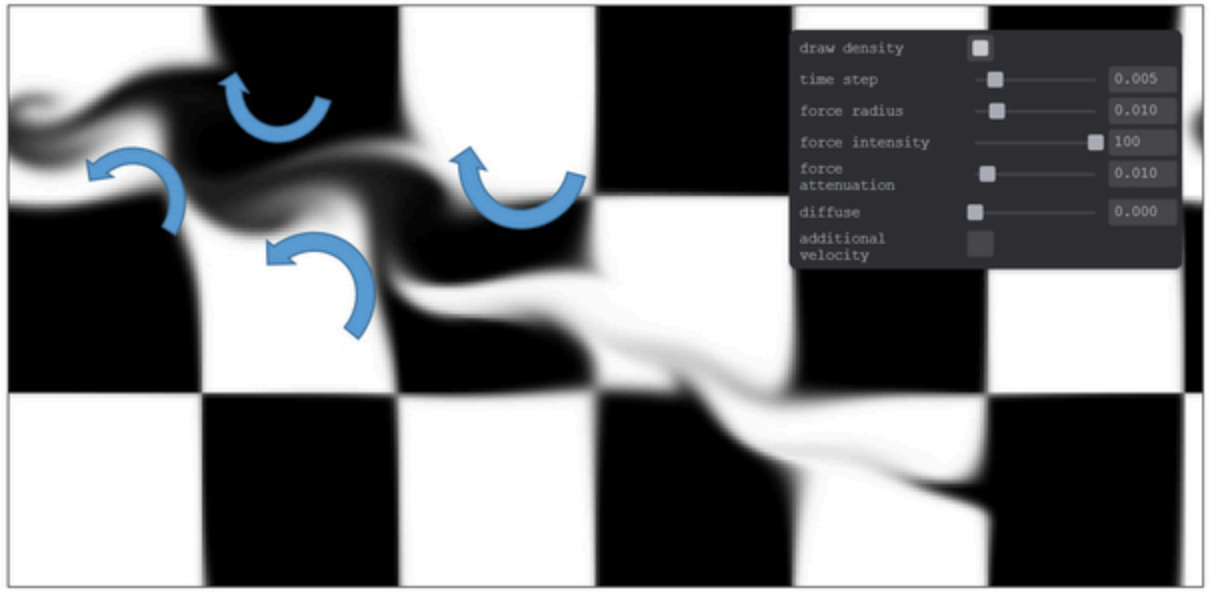
大量のピクセル（テクセル）からなるテクスチャの
そのグリッドの１つ１つがベクトルを保持している

“こういうのを「ベクトル場」と呼んだりしますね”

マウスカーソルが動き、スワイプのような操作が加えられた場合は、このベクトル場に対して速度が足し込まれます。

速度が計算の過程で飽和したり、逆に無意味に減衰したりしないように、周辺ピクセルとの関係性を考慮した計算を何度か繰り返し行います。この処理が JavaScript のコードで言うと `updateProject` というメソッドで処理している部分に該当します。

> 計算を収束させるために何度も繰り返し処理する



流れがある場所の周囲から流れ込むような
移流の動きがより顕著に現れるようになる

定数 `PROJECT_ITERATION` の値を大きくすると、結果的に「移流・流入」の効果が強く現れるようになります。

サンプル 028

- 使われているシェーダが大量……
- 使われているフレームバッファも大量……
- １フレーム描画するために、何パスもレンダリングを行う
- Velocity を更新し Project の計算までをまず行い……
- Velocity に応じて Density を変化させる
- パラメーターをいろいろ変化させて、まずは遊んでみよう！
- サンプルをベースに改造した実装例

最後に

GLSL は、WebGL が登場したことでだいぶ身近になりました。

とは言え、自力で、独学のみで体系的にそれを学習するにはやっぱり多少難しいジャンルではあると思います。今回のスクールでは基礎から応用まで幅広く取り入れたつもりですが、まだまだ現時点では手に馴染んでいないというか、自分で描画結果をイメージしながらシェーダを書くというのは難しい場合が多いと思います。

大事なことは、講義のなかでも何度も言ってきましたがとにかく「自分で手を動かし、それを継続すること」だと思います。

そのためにも、まずは楽しくシェーダを書くということをぜひ重視してみてください。少しずつ慣れ親しんでいくなかで、数学などの理解も徐々に深まっていくと思いますので……

スクール修了後も、引き続き Discord やメールで質問等は受付しています。

なにか困ったことがあったら、いつでも声をかけてください！

今回は課題は用意していません。次回開催予定の作品発表会に向けて、どんな簡単なものでもいいので作品を作ってみてください。

アウトプットすることで本当の実力が身につきます。臆せず挑戦してみてください！

> 作品提出などについてはあらためてアナウンスします 🙏