

レンダリングの流れとシェーダの役割 を知ろう

WebGL 本来の API と使い方の基本

はじめに

今回から、いよいよ本来の WebGL の API を直接利用する実装を見ていきます。

経験的に、最初はかなり面食らうと言いますか..... かなり難しいと感じる場合が多いと思います。

それでもなお毎年 WebGL スクールでネイティブな WebGL を扱うのにはきちんと理由があります。

1 つには、three.js のようなラッパライブラリが使いやすさを考慮して覆い隠してくれている部分にこそ、3D プログラミングの本質的な理解を助ける概念がたくさん潜んでいるからです。

また、three.js を使っていてもどんどん表現の幅を広げていくと、いずれ必ず「ネイティブな WebGL の原理を知っていないと表現できない領域」に直面することになります。

一見遠回りに見えても、結局は本質を押さえておくことが最短で実力アップにつながります。

「
一步一步、確実に理解を進めていきましょう！
」

レンダリングパイプライン

まず最初は、OpenGL や DirectX、あるいは WebGL などの 3DCG 系の API がどのようにグラフィックスを描画しているのか、その仕組みから見ていきます。

ここはどちらかというと概念的な話なので、細かいところまで漏れ無く暗記する必要はありませんが、雰囲気をまずは掴みましょう。

みなさんはレンダリングパイプラインという言葉を聞いたことがあるでしょうか。そもそも「レンダリング」という言葉は、なにかしらのシステムチックなロジックによって映像やイメージを描き出すことを指す言葉です。

3DCG はまさに、そういう意味で「レンダリングされるもの」であると考えることができます。

そしてレンダリングパイプラインとは、この 3DCG が描き出されるまでのプロセス、処理の流れのことと言います。

パイプラインは本来、石油などを運ぶためのパイプの道のことです。CG が描画されるまでの一連の流れをパイプラインという単語を使って表現しているというわけです。（場合により、グラフィックスパイプラインと呼ばれたりすることもあります）



次々と連続で処理を行っていく様子をパイプラインになぞらえている

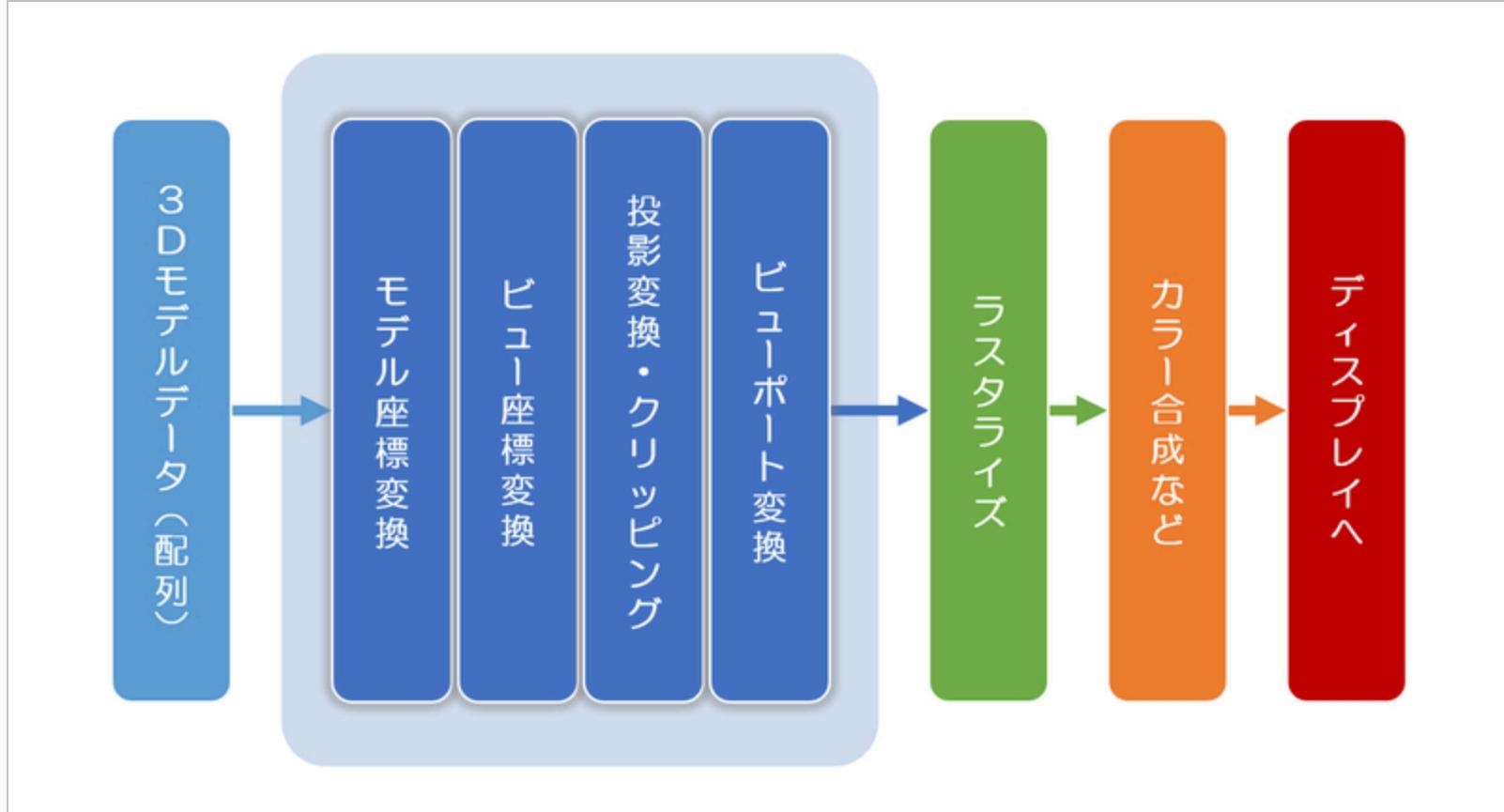
WebGL はブラウザ上で実行されるとは言っても、手続き的には OpenGL の API をほぼそのまま呼び出すことになります。

つまり、一般的な 3DCG のレンダリングパイプラインをそのまま利用した描画を行うことになるわけで、必然的にその処理の流れやコード記述の流れは C++ 等で記述した OpenGL 実装などと基本的には同じになります。

まずは、ものすごく簡略化したレンダリングパイプラインの図式を見てみましょう。

ポイントは、これらのパイプラインの処理は GPU 上で動作するもの であるということを念頭に置いて考えることです。

“ JavaScript が動作しているのは CPU の世界 ”



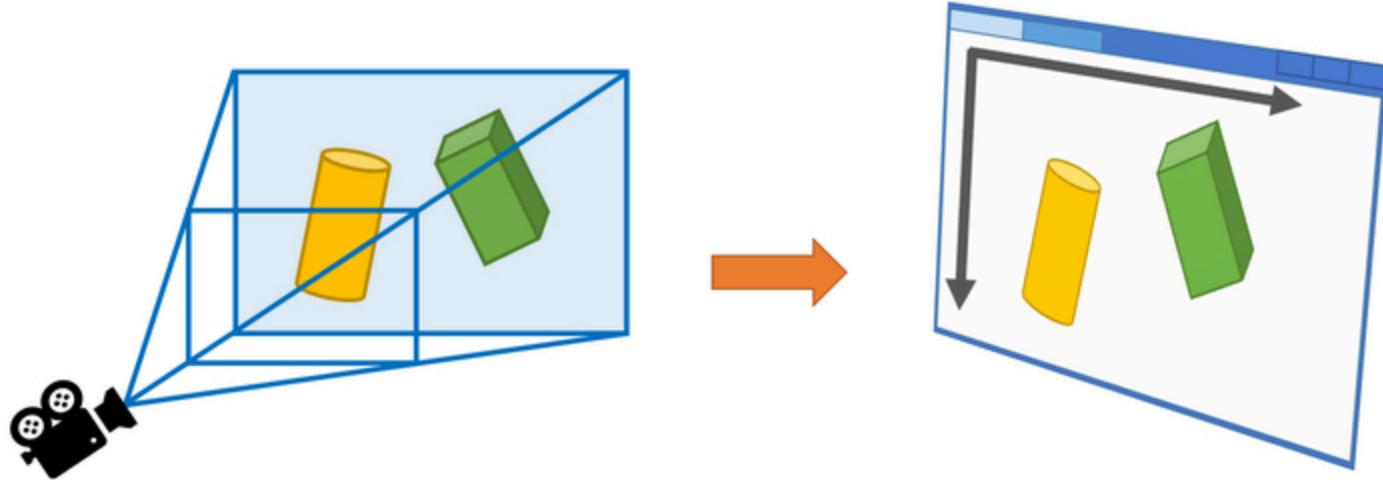
左から右へ、処理が流れていきます

まず最初（先程の図の左端）にあるのは、アプリケーションによるデータの入力です。

これは多くの場合、3D ジオメトリの頂点に関する情報の入力であり、3DCG をレンダリングするためのパラメータの入力であったりします。

次に、CPU から入力された情報を元にして GPU 内で頂点データが加工されます。

これには、複数の変換のステップがあり、○○座標変換、というような処理がいくつか行われます。なぜこのようなステップがあるのかというと、3D の三次元の情報を、スクリーンという二次元の世界にレンダリングするために 次元を越えるための変換作業 が必要になるためです。

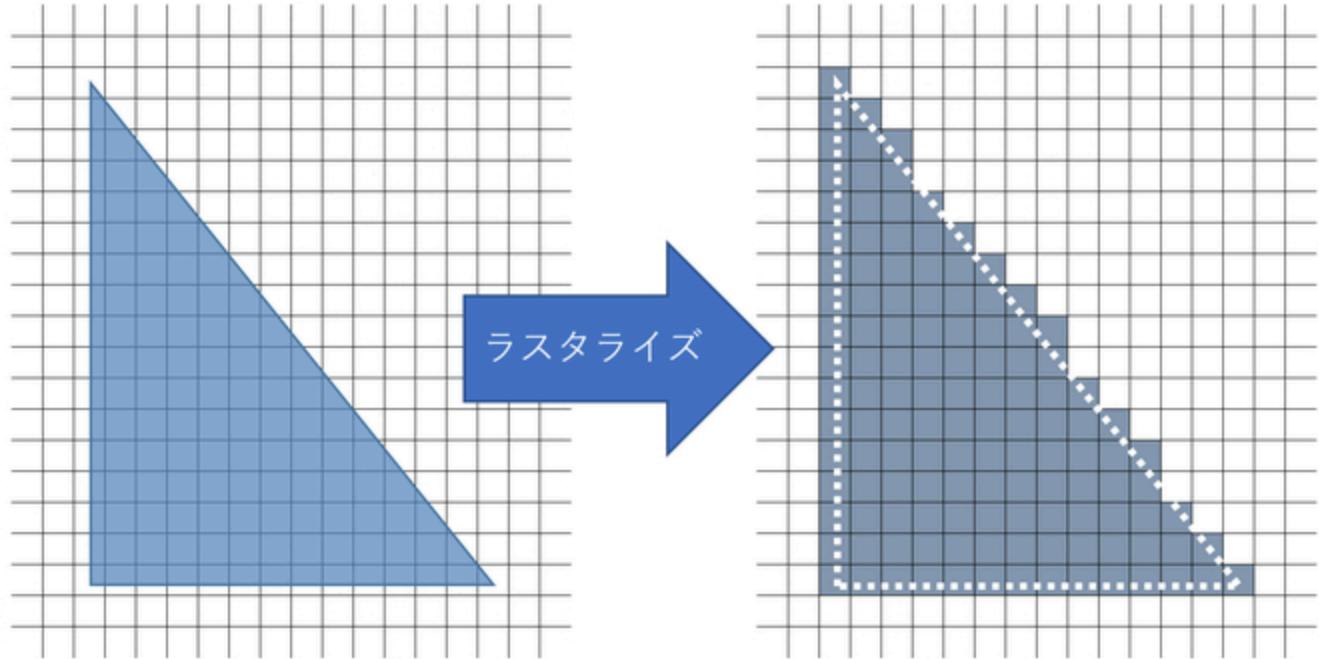


三次元的な広がりを持つデータ構造（ジオメトリ）から
二次元平面（スクリーン）上のピクセルへと変換する

様々な変換処理を経て、三次元的な情報が二次元的な情報へ

各種の座標変換が終わると、それに続いて行われるのがラスタライズです。

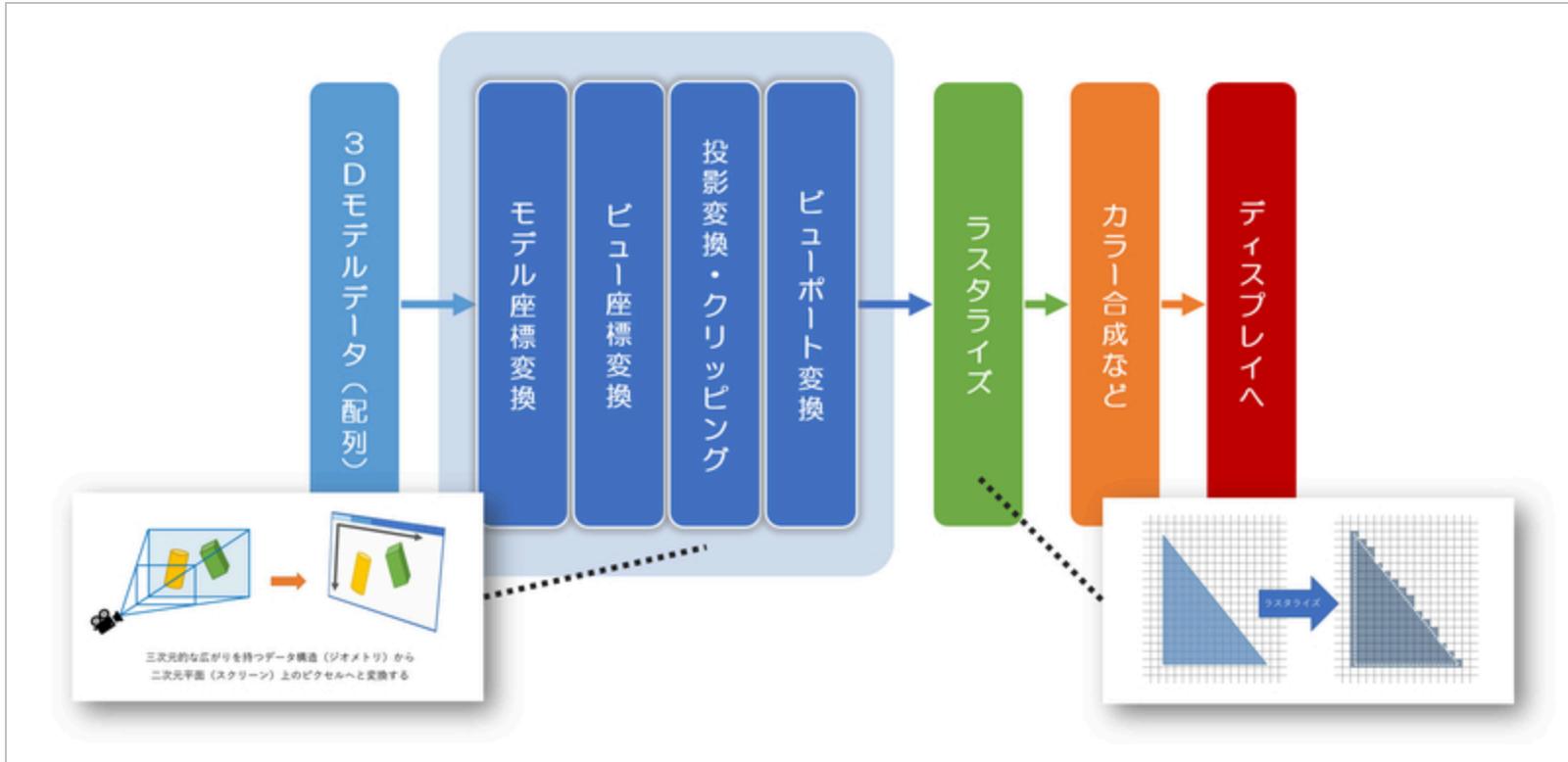
ラスタライズとはわかりやすく言えば、幾何学的な構造である形状データを最終的に画面に映し出される「ピクセルレベル」へと落とし込む作業です。ウェブの文脈で言うと、SVGなどのベクターデータをビットマップ（ラスタ画像）にする、というのと同じです。



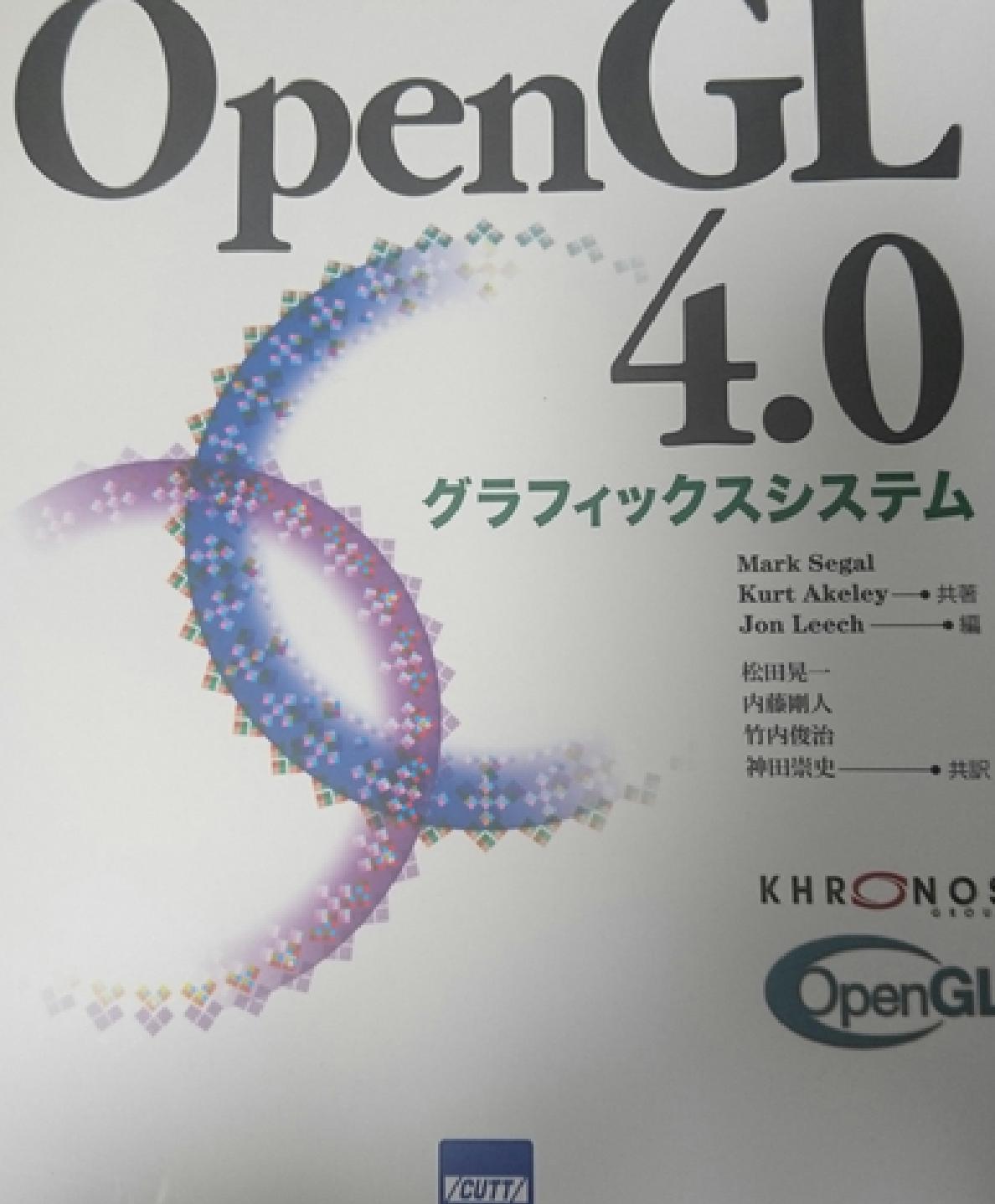
このときピクセルレベルで、そこに描くものがあるかそうでないかが決まる

二次元的な情報へとラスタライズされたデータは、最終的にカラー合成などのステップを経て、着色されたレンダリング結果になります。

ちょっと変な感じがするかもしれません、このステップより前の段階では、あくまでも頂点の状態がそれぞれにメモリ内で遷移している（つまり単純な数値の計算をしている）だけで、どんな色になるのかはまったく決まっていないのですね。

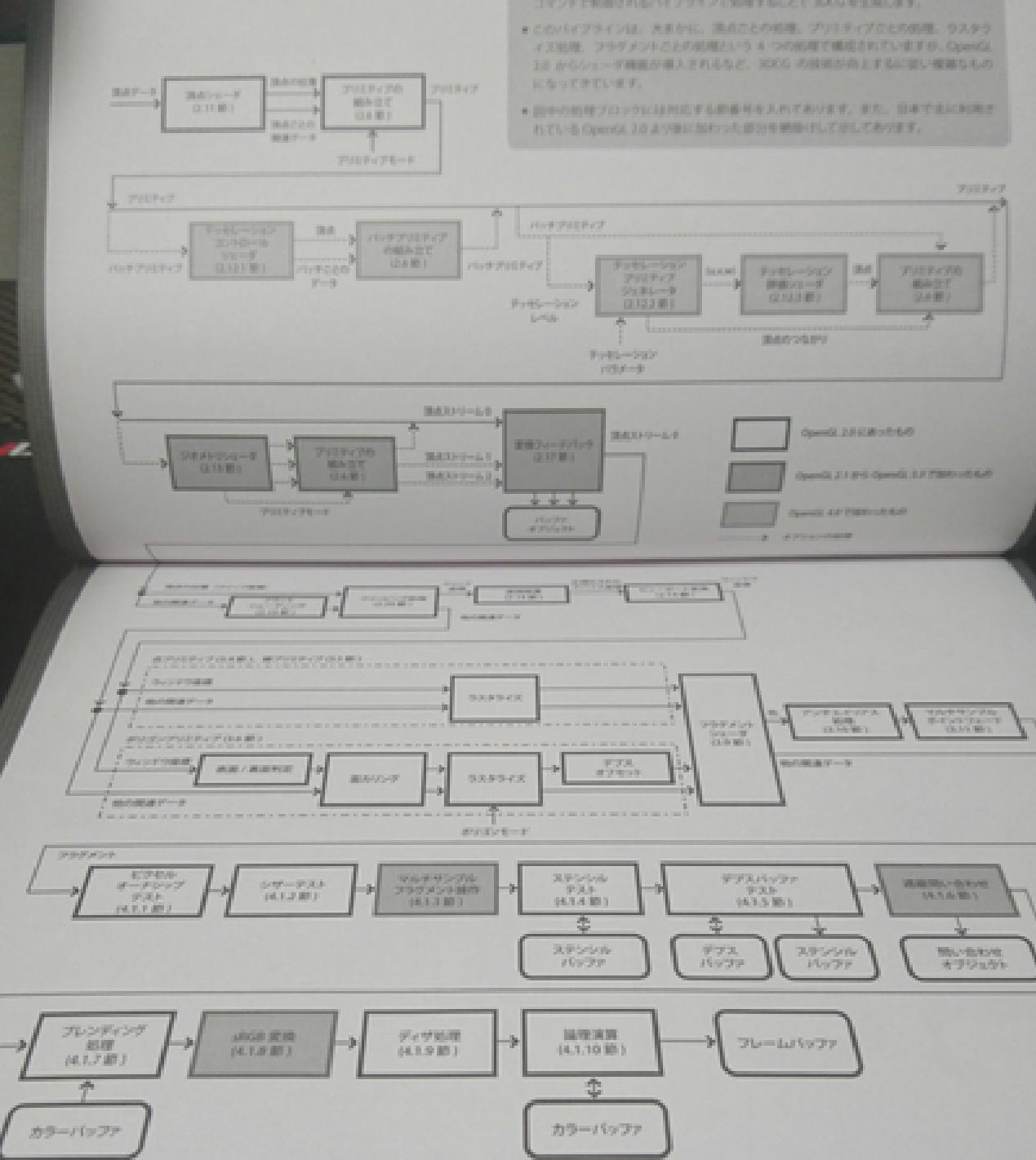


この概念図はかなり簡略化されています。簡略化せずにパイプラインの全ステージを可視化する例は、次のページに参考例があります。



参考：OpenGL 4 の場合の
パイプライン全図

※OpenGL 4.0 の書籍（カット
システムより出版）から引用



小さくてほぼ見えないですが、左上から処理がスタートし……

右下にある最後のブロックに「フレームバッファ」と書かれしており、これがディスプレイへの出力とほぼ同じ意味になります。

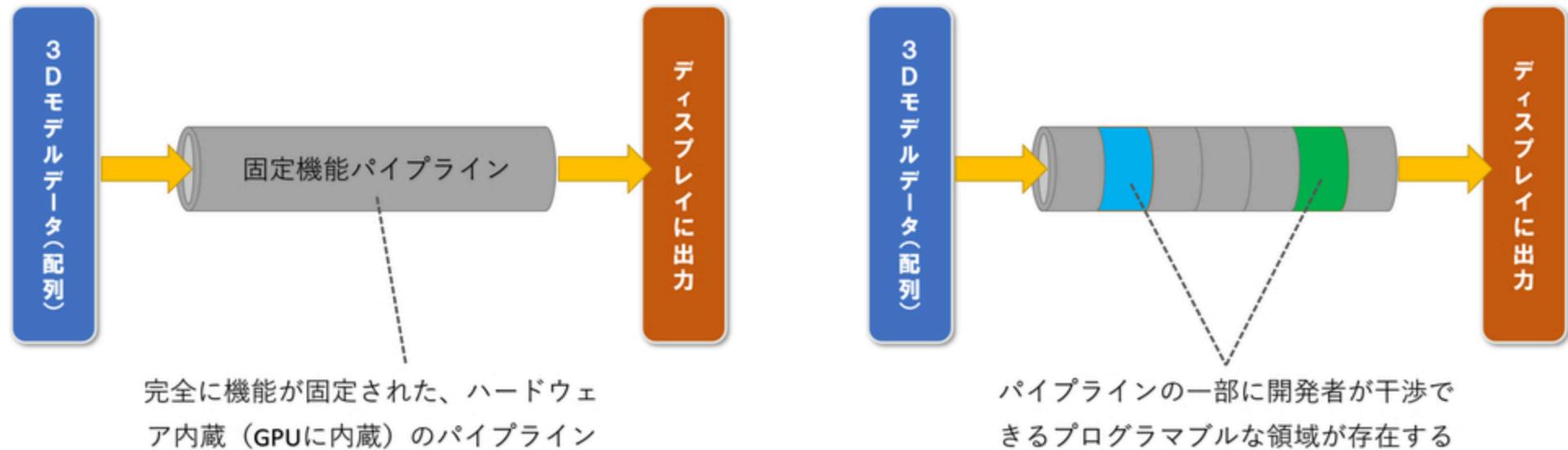
パイプラインの種類

さて、このようにレンダリングパイプラインには一見するととても冗長に見えるほど、とても多くのステップがあります。

最初の入力部分を除く全てのステップは、GPU のなかで高速に、かつ並列に処理されます。

そしてこのパイプラインの仕組みには、ざっくり言うと 2 つの種類があります。

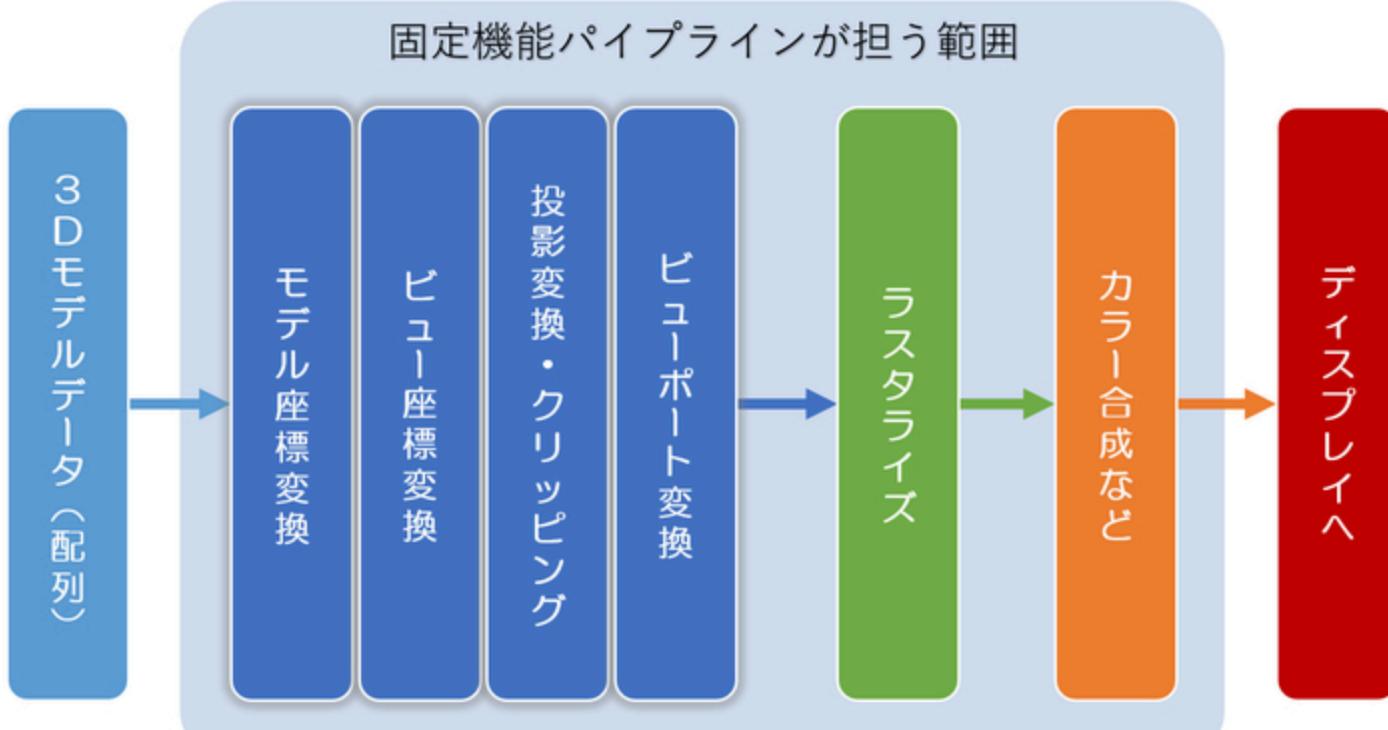
あらかじめ GPU に組み込まれている 固定機能パイプライン と、プログラマがある程度パイプラインに干渉・制御することができる プログラマブルなパイプライン です。



“ 固定機能パイプライン（左）は GPU 側にパイプラインの実装があらかじめ入っていて、干渉することはできない ”

固定機能パイプラインは、まさにその名の通り機能が固定されてしまっているのですが、そのかわりデータの入力さえ行ってあげればそこからさきの描画までのプロセスすべてを GPU に組み込まれた機能が自動的に処理してくれます。

したがって、固定機能パイプラインを用いた処理は総じて簡単であり、シンプルです。



固定機能パイプラインではその工程のほとんどが自動的に処理される

“パイプラインの中のほとんどの処理が勝手に行われるので簡単”

しかし、WebGL には、この簡単に使える固定機能パイプラインの仕組みは ありません！

これはどうしてかというと、WebGL のベースになっている OpenGL ES に秘密があります。

ES シリーズは、モバイル機器向けの軽量な OpenGL 実装です。そして軽量化が行われる過程で、なんと固定機能パイプラインの機能全てがごっそりと切り捨てられてしまったのです。

結果的に、それをそのままウェブに持ってきた WebGL の場合も固定機能パイプラインを使うことはできません。

WebGL を含む固定機能パイプラインを持たない API では、必然的に プログラマブルなパイプライン を用いたレンダリングを行ってやる必要があります。

ちょっと違った言い方をすると、自動的に固定で処理してくれる機能を使うことはできないので「自らプログラムを記述する」ことでパイプラインを構築・制御する必要があるのです。

ここで登場した「パイプライン上で各種制御を行うプログラム」こそが
シェーダ と呼ばれる概念です。

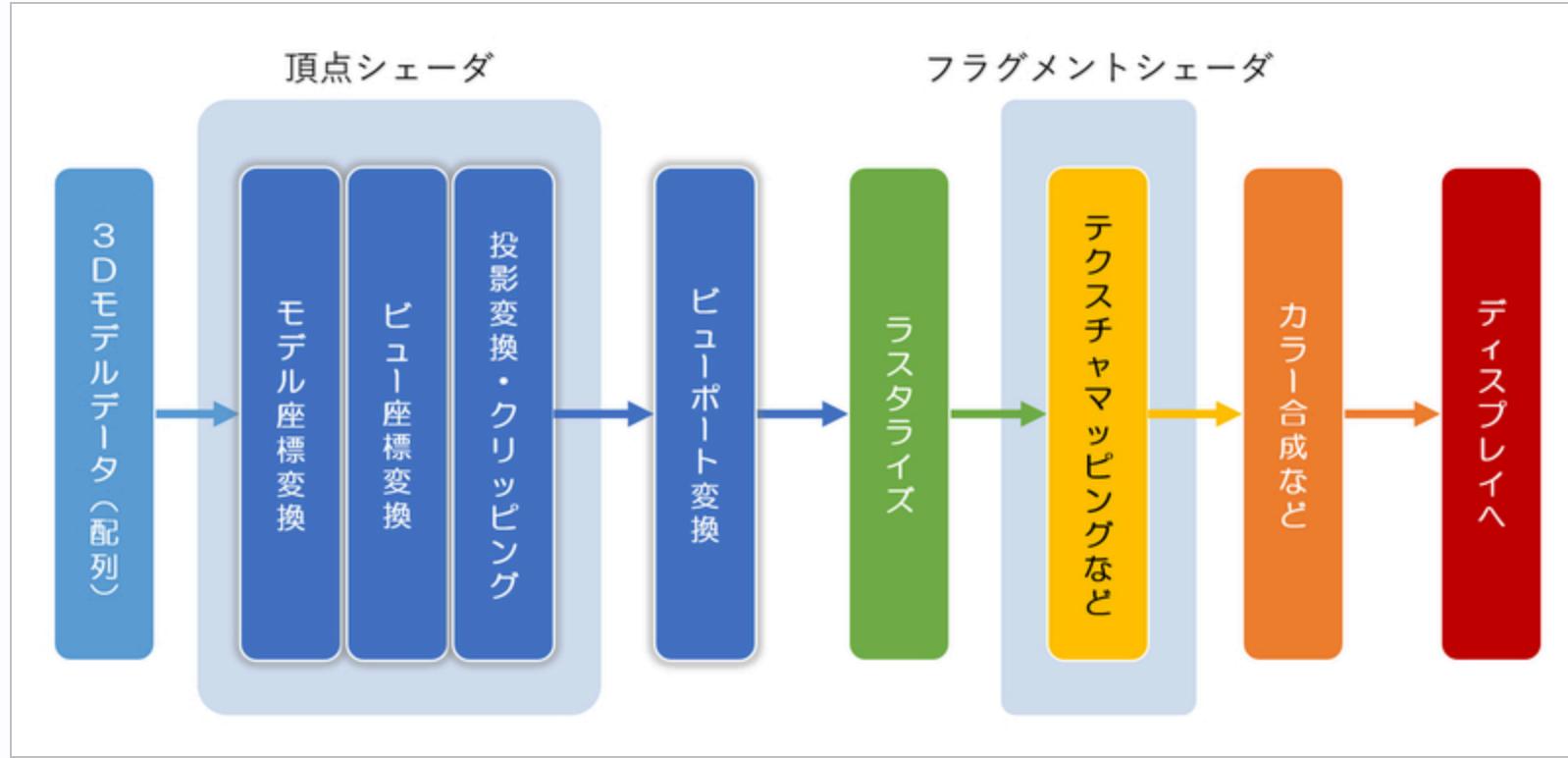
つまり WebGL では、シェーダを用いずにレンダリングを行うことはで
きませんし、どのような描画結果が得られるかは、シェーダの内容に大
きく左右されることになります。

これまでほとんど登場してませんでしたが three.js も当然内部ではシェーダ
を使ってます

プログラマブルシェーダ

シェーダを使ってレンダリングパイプラインを独自に実装する場合、固定機能パイプラインのように簡単にはいきません。

頂点を変換する 頂点シェーダ と、色に関する処理を行う フラグメントシェーダ のふたつを用意し、これを GPU に渡してやらなければパイプラインが完成しないからです。



頂点シェーダとフラグメントシェーダを記述してやることで、そこではじめてパイプラインを機能させることができる。

さて、簡単にレンダリングが行える固定機能パイプラインが存在しないと聞いて、ちょっとがっかりしたかもしれません。

しかし当然のことながら固定機能とプログラマブル、いずれのパイプラインにもメリットとデメリットがあります。

固定機能パイプラインは簡単に利用できる一方で、その名前からもわかるとおり機能は完全に固定されています。

つまり、何をどう頑張ろうが工夫しようが、同じような絵作りしかできません。

一方でプログラマブルなシェーダを用いた実装の場合、当然シェーダを自前で用意しなければならない分、手間と学習コストが増えます。

しかし、逆に考えてみれば レンダリングを自在に制御する ことができるわけですから、その表現力は無限大です。

“多少難易度が上がると言っても、恩恵のほうがはるかに大きい”

ウェブの話に例えるなら……たとえば HTML 要素のみを用いて CSS を一切使わなくても、ウェブページを作ることはできます。

しかし、その見栄えはけして多彩なものにはならず、誰が作っても、ある程度クオリティや表現には限界が出てくるでしょう。しかし、CSS を独自に記述したり、JavaScript を使って動的にページ内の処理を行うことができれば、その表現力は非常に高いものになります。

プログラマブルなレンダリングパイプライン（つまりシェーダ）を使う
というのは、それと似ています。

学習コストは上がりますが、その代わり、レンダリングパイplineを
独自に拡張するシェーダの存在により、非常に柔軟で多彩な表現が可能
になるのです。

ここまでまとめ

- 3DCG はレンダリングパイプラインによって描かれる
- レンダリングパイプラインには種類がある
- WebGL には固定機能パイプラインは存在しない
- パイプラインを独自に構築するためのプログラムがシェーダ
- WebGL では二種類のシェーダを使ってパイプラインを構築する

CPU と GPU

さて、それでは少し状況を整理しましょう。

まず最初にこれまでの通常のウェブサイトを例に、その構成を見てみると.....

- HTML → ウェブページの構成要素をマークアップ
 - CSS → ウェブページの外観に関するスタイルなどを決める
 - JavaScript → 動的な処理をウェブサイト上で実現する
- こんな感じでしょうか。

これが WebGL を含む実装になると、どんな構成になるんでしょう。

CPU 側の仕事

- HTML → ウェブページの構成要素をマークアップ
- CSS → ウェブページの外観に関するスタイルなどを決める
- JavaScript → 動的な処理 (WebGL API を含む)

GPU 側の処理

- 頂点シェーダ → 頂点を座標変換する
- フラグメントシェーダ → どんな色を出力するか決定する

このように、WebGL を本質的に理解するにはこれまでのウェブには無かった新しい勢力として「GPU 側で動くシェーダ」という概念が必要になります。

three.js の場合は、元からある程度柔軟に対応できるお手本シェーダが事前に組み込まれています。ですから実装者がシェーダ（や GPU 上での処理）について理解していないなくても、three.js のルールに沿って記述するだけで WebGL を利用できるのですね。

“three.js が直感的に使えるのは、GPU のことを気にしなくてよいため従来のウェブの実装に近い感覚で開発ができるからである”

ここで重要なことは、まず CPU の世界と GPU の世界という、2つの異なる領域があるという事実を把握しておくことです。

CPU と GPU は、いわばフロントエンドとバックエンドのように完全に切り離された世界です。それぞれの役割をごちゃまぜにしないように注意しながら理解していくことが大切です。

3D グラフィックスと頂点

ウェブの世界には、もともとリッチなグラフィックスを描くための Canvas2D API が存在します。

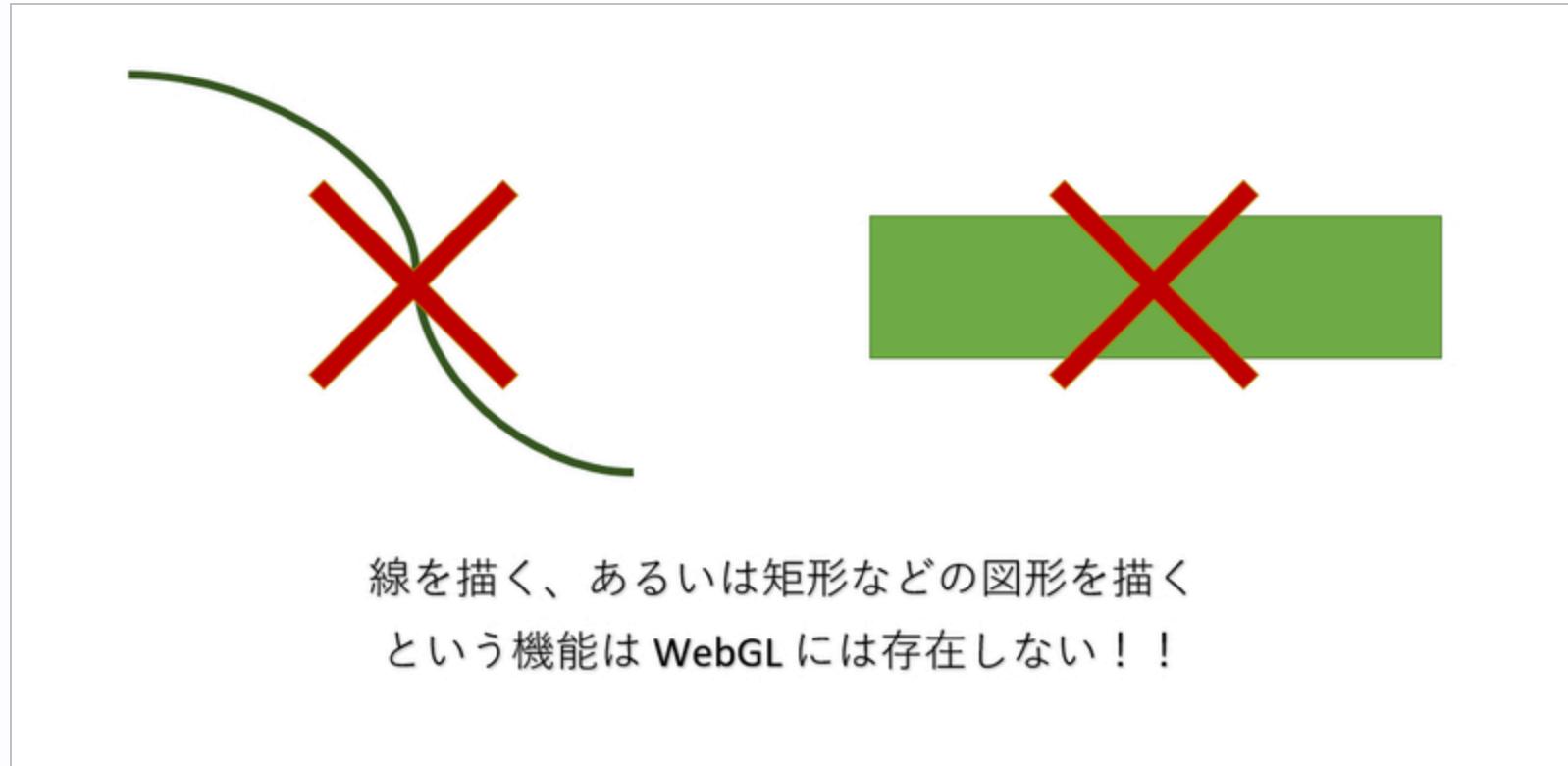
Canvas2D API を使うと、ウェブサイト上に様々な図形を描いたり、文字をビットマップとして描画したりといった処理が行なえます。

たとえば、Canvas2D API の `fillRect` メソッドとかを使えば、簡単に四角形を `canvas` 上に描画できます。

その他にも、扇形を描いたり、線を描いたり、画像を読み込んで描画したりといった機能があります。

参考：[CanvasRenderingContext2D - Web API | MDN](#)

一方で WebGL にはボックスや球体はおろか、三角形や四角形を描く命令も、線一本を引く命令すら一切存在しません。



線を描く、あるいは矩形などの図形を描く
という機能は WebGL には存在しない！！

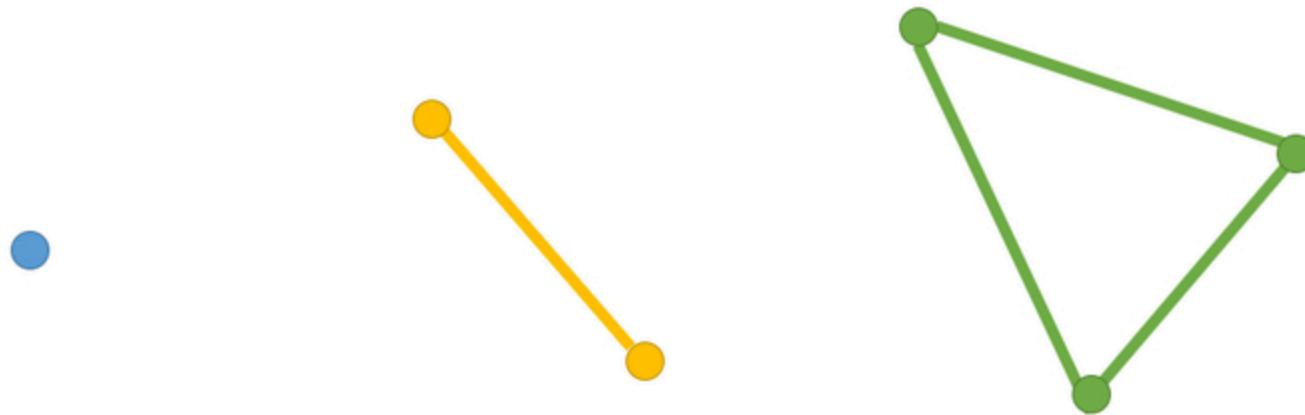
ではどうやって線や矩形などの図形を描くのかというと、WebGL の場合は **頂点** を利用してグラフィックスを描いていきます。

違った言い方をすると「頂点を使ってジオメトリを組み立てて、それを描画する」以外のことは、基本的にはできないのです。

“すべては頂点であり、頂点以外のものは描くことができない”

頂点とは、WebGL における 描画の最小単位 です。

WebGL では、頂点を組み合わせることですべての形状を表現します。頂点が 2 つあれば、それを結ぶことで線を引くことができるようになりますし、頂点がさらに増えて 3 つになればそれらを結んでポリゴンを作ることができます。



頂点が1つでは単なる点でしかないが、2つで線を
3つあれば三角形ポリゴンも生成することができる

“頂点とそれを結んでできる図形のイメージ”

つまり WebGL などの 3D API では、なにを描くにも必ず頂点を使わなくてはなりません。頂点には、どこに描かれるのかという座標の情報はもちろんのこと、何色をしているのかといった様々な情報を付与することができます。

GPU はこれらの「頂点に紐付けられた情報」を、シェーダとして記述されたプログラム内で読み込み、加工し、それを用いて画面に CG のレンダリングを行います。

頂点まとめ

- OpenGL や WebGL では頂点以外のものは描画できない
- 頂点は WebGL における描画の最小構成単位
- 頂点を組み合わせることで線やポリゴンが描ける
- 頂点をどのように加工・利用するかはシェーダの内容によって決まる

頂點屬性

頂点は普通、座標情報を持っています。

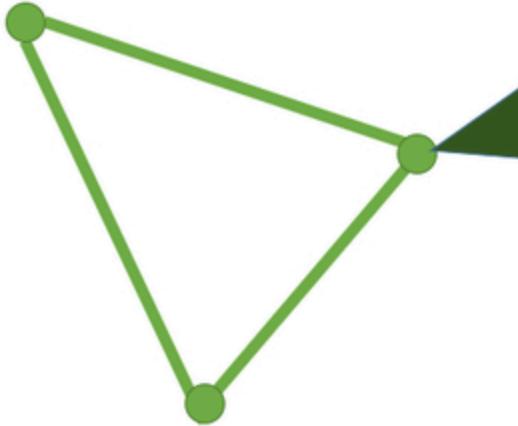
これは、形状を表すためには「その頂点がどこにあるのか」という情報が必要だからです。

仕様上は座標を持たせないということもできなくはない.....けど普通はあまりそういうのはやりません

このような、頂点に紐付けられた「座標」や、あるいは「色」や「法線」などの諸々の情報のことを 頂点属性 と呼びます。

英語で言うなら、vertex attribute（アтриビュート）という感じでしょうか。

この頂点属性という言葉は今後の講義でも頻繁に登場します



position: x, y, z
color: r, g, b, a
weight: w

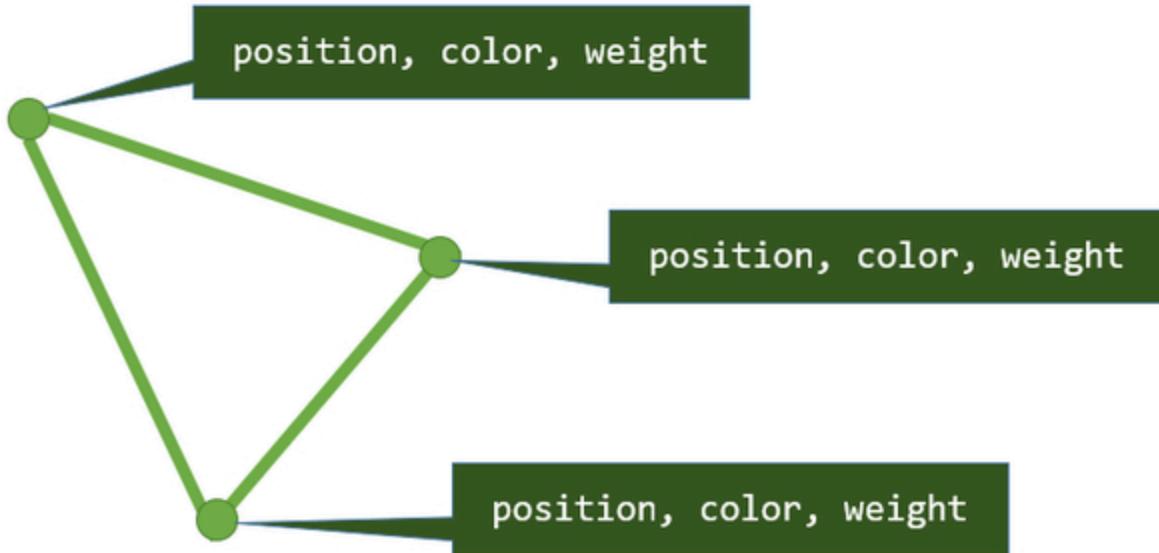
頂点が固有に持ち合わせている属性値を頂点属性と呼ぶ

※上記の例では position, color, weight がそれぞれ頂点属性

どのような頂点属性のセットが頂点に備わっているかは時と場合に（要はどう実装されているかに）ります。

頂点属性という言葉が講義のなかで出てきたときに 頂点属性 = 頂点 1 つ 1 つが個別に持っている固有の値 ということがイメージできることが大事です。

また、1つのジオメトリを構成する頂点は、すべての頂点が 同じ種類の 頂点属性を等しく備えている 状態である必要があります。



頂点属性は、それぞれの頂点が一様に同じものを備えている。
もちろん、中身の値はそれが個別に固有の値を保持できる

頂点属性まとめ

- 頂点属性とは、頂点 1 つ 1 つが持つ固有の属性値 (position や color) のこと
- 1 つのジオメトリを構成している各頂点は、同じ頂点属性を備えている必要がある（1 つの頂点だけ色がない、みたいなことはできない）
- どのような頂点属性を持たせるかは実装者が自由に決められる
- 最低でも座標を持たせないと世界のどこに置けばいいかが決まらないので、普通は座標は持たせる

頂点バッファ (VBO)

話がややこしくなってきたので、いったん振り返ります。

- WebGL は GPU 上に存在するレンダリングパイプラインを使い絵を描いており.....
- プログラマブルなパイplineで必要となるシェーダのプログラムを用意してやる必要があり.....
- WebGL では頂点しか描画することができないので頂点を定義する必要があり.....
- 頂点は、その頂点が持つ座標や色など「任意の数の頂点属性」を組み合わせることで 1 つの単位として定義される

具体的なやり方（コードでの記述）はあとであらためてしっかりやります

次に考えないといけないのは.....

「定義した頂点の情報を、どうやってシェーダ（GPU）に渡すのか」です。

頂点の定義そのものは、WebGL が動作する CPU 側、つまり JavaScript のほうで行います。

ここは仕組みとしては何も難しいことはなくて、単純に頂点属性を JavaScript の配列として記述していきます。

たとえば、頂点の座標を配列に与えるとしたらこんな感じ。

```
const position = [
  0.0, 0.5, 0.0, // ひとつ目の頂点の x, y, z 座標
  0.5, -0.5, 0.0, // ふたつ目の頂点の x, y, z 座標
  -0.5, -0.5, 0.0 // みつつの頂点の x, y, z 座標
];
```

これは、CPU 上で動作する JavaScript の世界で行います。基本的に多重配列にはせずに、1 つの配列に連續で値が並んだ状態で定義します。

こうして JavaScript で定義できた頂点の情報は最終的にシェーダ内で使われることになるので.....

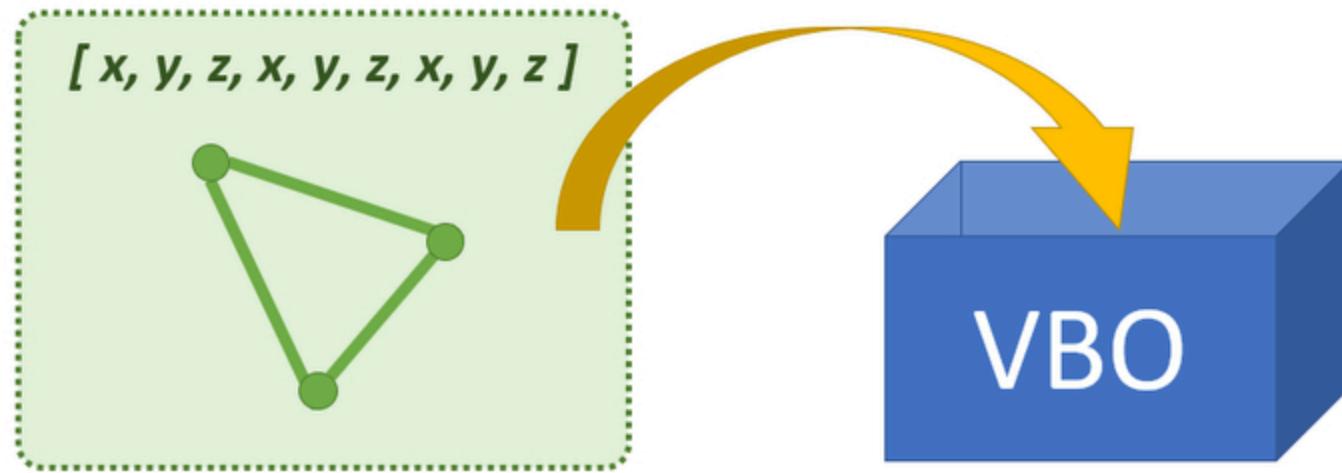
どうにかして GPU に渡してやらなくてはならない ということになります。

繰り返しになりますが、CPU と GPU は、フロントエンドとバックエンドのように違う世界に存在しているためですね

ここで登場するのが **VBO (Vertex Buffer Object)** です。

Vertex Buffer Object という名前からもわかるとおり、頂点のデータを格納するための専用のバッファ（いれもの）が VBO です。

“平たく言えば、頂点属性の専用のバッファが VBO です”



JavaScript の配列で定義した情報を、頂点データ専用の
バッファオブジェクト (Vertex Buffer Object) に詰め込む

VBO とは

- 頂点が持っている座標や色を頂点属性と呼び.....
- 頂点属性の定義は、CPU 側でまずは配列を使って行う
- 配列に詰め込まれた頂点属性の定義は、なんらかの方法で GPU に送る必要があり、そのための入れ物が VBO
- このバッファを生成するメソッドは WebGL コンテキストが持っている

ここまで理解できたら、一度サンプル内コードを見てみよう

001 (その1)

- WebGL 関連の処理は、そのまま書いていくと膨大になってしまうので `WebGLUtility` というクラスを定義して `webgl.js` にまとめています
- `WebGLUtility` というクラスは静的なメソッド（`static` なメソッド）のみで構成されています
- まずは WebGL のコンテキストの生成部分を確認（`App.init`）
- 次に配列から VBO を生成する部分を確認（`App.setupGeometry`）

“サンプルのコメントのほうがより詳細なので、そちらを中心にチェック”

シェーダと GLSL

ちょっと 3D API としては初歩的な話が続きますが、ここはある意味正念場なのでがんばってください。

続いては、シェーダ周りの初期化について詳しく見ていきます。

WebGL は先ほども書いたとおりで、プログラマブルシェーダが必須のパイプライン構造です。

ここで使われるシェーダには、頂点シェーダとフラグメントシェーダの二種類が存在します。両者の役割は明確に異なっています。この違いをしっかりと把握することがとても大切です。

頂点シェーダ

頂点シェーダはその名のとおり、頂点を変換するのが主な仕事です。

より具体的には 頂点の座標変換 こそが頂点シェーダの最大の役割です。頂点が回転したり移動したりできるのは、この頂点シェーダががんばって座標を変換してくれるおかげです。

これは地味に重要なポイントになりますが.....

頂点シェーダは 頂点1つ1つに対して個別に実行される というのがとても重要です。頂点が 100 個あれば、頂点シェーダは GPU 内で 100 回実行されます。

フラグメントシェーダ

フラグメントとは、直訳すると「欠片（カケラ）」といった意味になる言葉です。

フラグメントシェーダはディスプレイに無数にある 各ピクセルの1つ1つ に対して、何色を出力すればいいのかを決める役割を持ちます。

頂点シェーダが「頂点の個数に対応している」のに対して、フラグメントシェーダは「ピクセルの個数に対応」しています。

つまり、描画されるスクリーンが高解像度であれば、それだけフラグメントシェーダの負荷は大きくなります。

この頂点シェーダとフラグメントシェーダは、専用の「シェーダ記述言語」によるソースコードから生成されます。

この、シェーダを記述するための専用言語が **GLSL** です。GLSL は OpenGL 系の実装で利用される、GPU 側で動作する特殊なプログラムである「シェーダ」を記述するための専用言語です。

OpenGL や WebGL 同様、GLSL にも固有のバージョンがあります

GLSL は独自の文法を持つ独立したプログラミング言語で、文法は C 言語に似ています。（似ているだけで両者には関係性や互換性はありません）

C 言語にはコンパイラが必要ですが、それと同様に GLSL もコンパイルしてやることで初めて動作するシェーダオブジェクトになります。

そんなふうに言われると「コンパイラのインストールが必要なの？」と思ってしまうかもしれません、コンパイラは WebGL コンテキストに含まれていますので別途インストール等は不要です。

つまり、メソッドの呼び出しだけで、ブラウザ上でコンパイルを行うことができます。

シェーダについてここまでまとめ

- WebGL でレンダリングするには必ずシェーダが必要となる
- シェーダには二種類あり頂点シェーダとフラグメントシェーダ
- 頂点シェーダが頂点の座標変換を一手に引き受ける
- フラグメントシェーダが頂点によって描かれるすべてのピクセルに色を塗ることを担当する
- それらのシェーダはその場で（ブラウザ上で）コンパイルできる

シェーダプログラム

シェーダはソースコードから生成されますが、頂点シェーダとフラグメントシェーダのソースコードはそれぞれ個別に必要です。

当然ながら、シェーダのソースコードに文法間違いや齟齬があればシェーダのコンパイル等で失敗することもあり、きちんと手順を踏めばエラー内容を取得したりすることができます。

“`shader` フォルダの中にある `main.vert` と `main.frag` がシェーダのソースコードで、`*.vert` が頂点シェーダ、`*.frag` がフラグメントシェーダです”

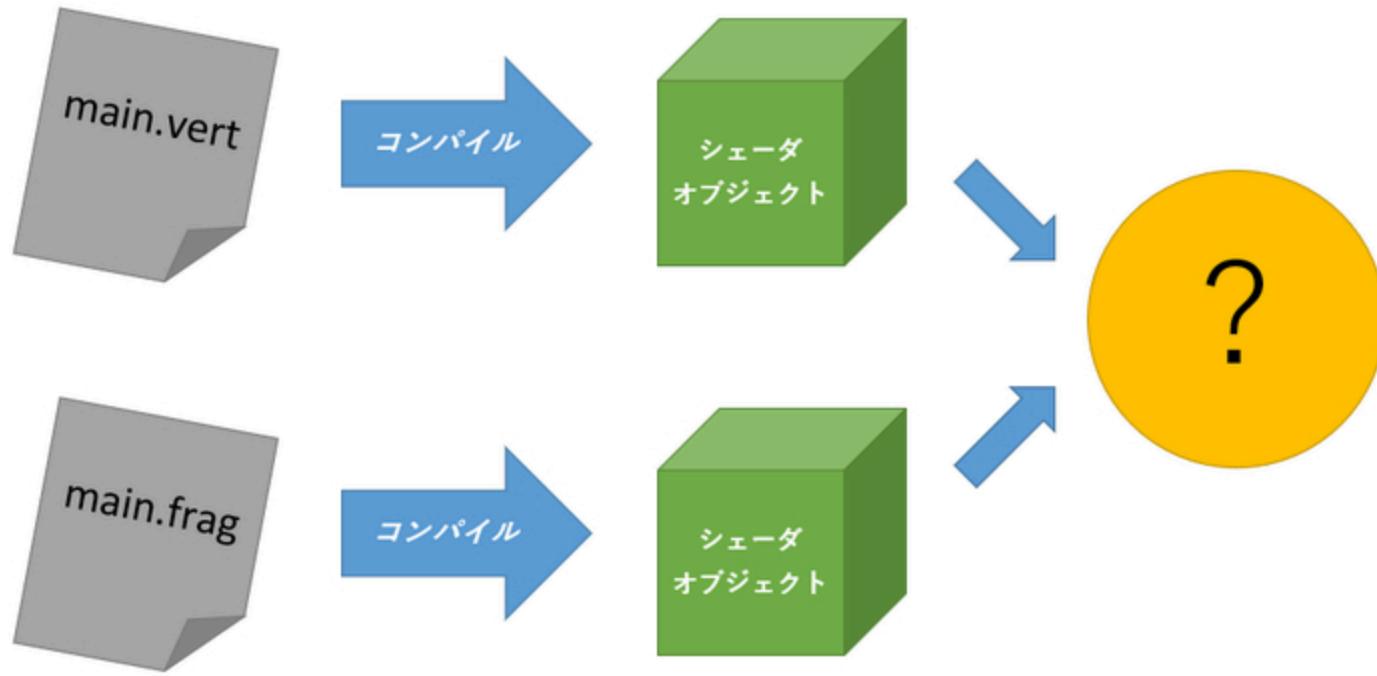
WebGL では、シェーダのソースコードは単なる文字列として定義します。ですからたとえば変数を宣言して、そこにシェーダのソースコードをそのまま代入したものを使うこともできなくはありません。

とは言えファイルにしてソースコードを管理したほうが、Git で管理できたり、テキストエディタのシンタックスハイライト機能が有効になるなどメリットが多いです。

定義されたシェーダのコードは、コンパイルされることで頂点シェーダとフラグメントシェーダのそれぞれが「シェーダオブジェクト」になります。

この時点では、まだ頂点シェーダとフラグメントシェーダは別々のオブジェクトで、両者が相互に関連付けられた状態ではありません。最終的には、任意のシェーダオブジェクトを2つ組み合わせ、1セットの状態にすることで初めてレンダリングに利用できる状態になります。

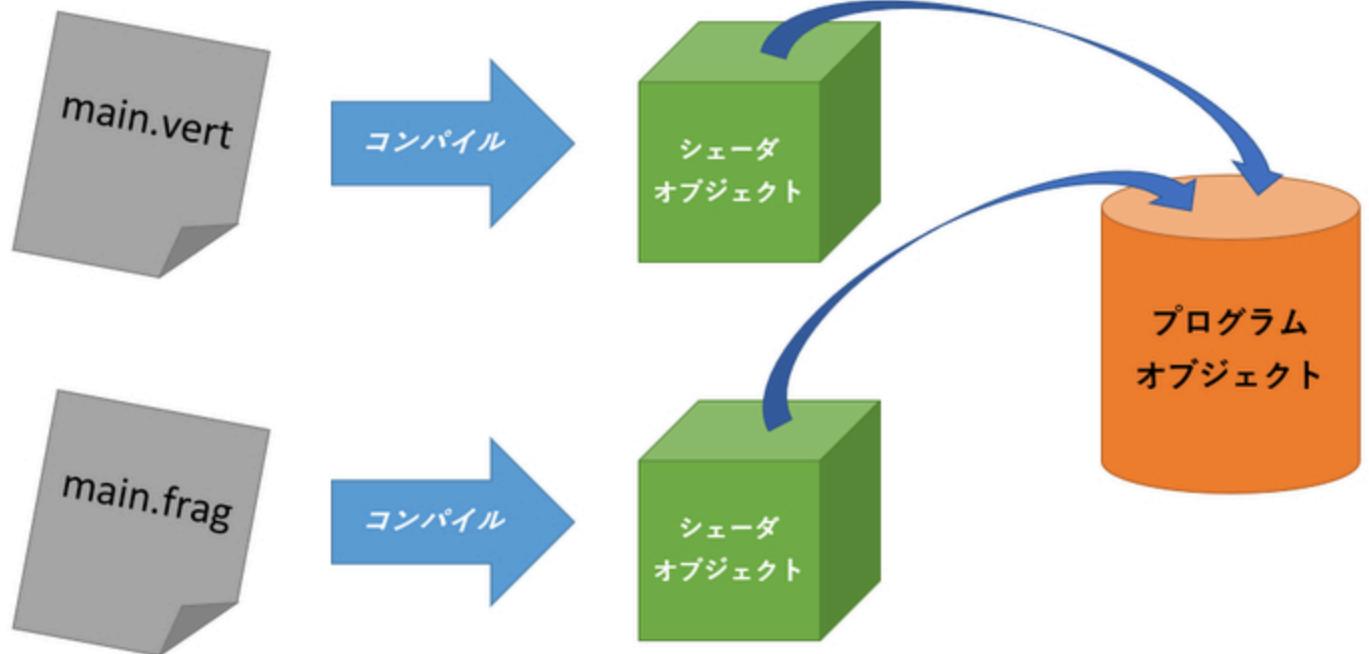
パイプラインを正しく処理するために、2つのシェーダの内容に齟齬がないかを確認する必要がある



レンダリングパイプラインの処理の流れ上
どうにかして両者を関連付ける必要がある

ここで出てくるのが プログラムオブジェクト です。

名前に「プログラム」と入っているので紛らわしいのですが、2つのシェーダを紐付け、両者間のデータのやり取りなどを行なってくれる一種の司令塔のような WebGL API の固有のオブジェクトです。



シェーダオブジェクトをプログラムオブジェクトに関連付け
したあとは、もうシェーダオブジェクトは破棄してもよい

もし、同時に複数のシェーダを用意して、それらを切り替えながら処理を行いたい.....

といった場合は、プログラムオブジェクトを切り替えることでこれを実現します。まず A というプログラムオブジェクトで描画して、次に B というプログラムオブジェクトで描画して.....というように、プログラムオブジェクトを切り替えること = シェーダを切り替えること、という意味になります。

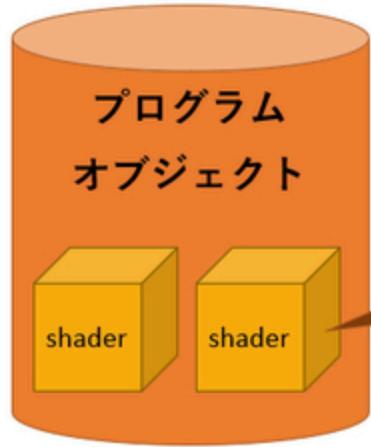
ここまでまとめ

- コンパイルしたシェーダは別々のシェーダオブジェクトになる
- これを紐付けてひとつにリンクしてやる必要がある
- シェーダをリンクするのがプログラムオブジェクト
- リンクに成功するとやっとシェーダが（プログラムオブジェクトを介することで）利用可能な状態になる
- リンク成功後は、プログラムオブジェクトを介して処理を行う
- 複数のプログラムオブジェクトを順番に切り替えながら使うことも

CPU から GPU へデータを送るための ロケーションという概念

プログラムオブジェクトが無事に生成できたら.....

続いて、このプログラムオブジェクトから「シェーダ内で使われている変数の情報」を取得します。たとえば、頂点属性としてどのような変数が宣言されているか、などの情報を取得するのが目的です。



どんな頂点属性が必要?
どんな変数を使っている?

シェーダを関連付けてリンクしたプログラムオブジェクトは、
シェーダ内で使われている変数の情報を取得するのに利用できる

GLSL では、変数を宣言する際に「その変数の意味」を付与する 修飾子 を一緒に用います。

どのような修飾子が付いているかを見れば、その変数が「どういう意図・意味を持つか」がわかります。実際の様子を確認するために `001/main.vert` を開いてシェーダのソースコードを見てみましょう。

冒頭に `attribute vec3 position;` と書かれている箇所があるはずです。

この `attribute` の部分が「変数に付与された修飾子」であり、ここでは「この変数は attribute（頂点属性）の意味を持つ」ということを示しています。

つまり「然るべき手順を経て VBO のデータを GPU に送る」ことで、「シェーダで定義された attribute 修飾子付きの変数」に、その値が割り当てられます。

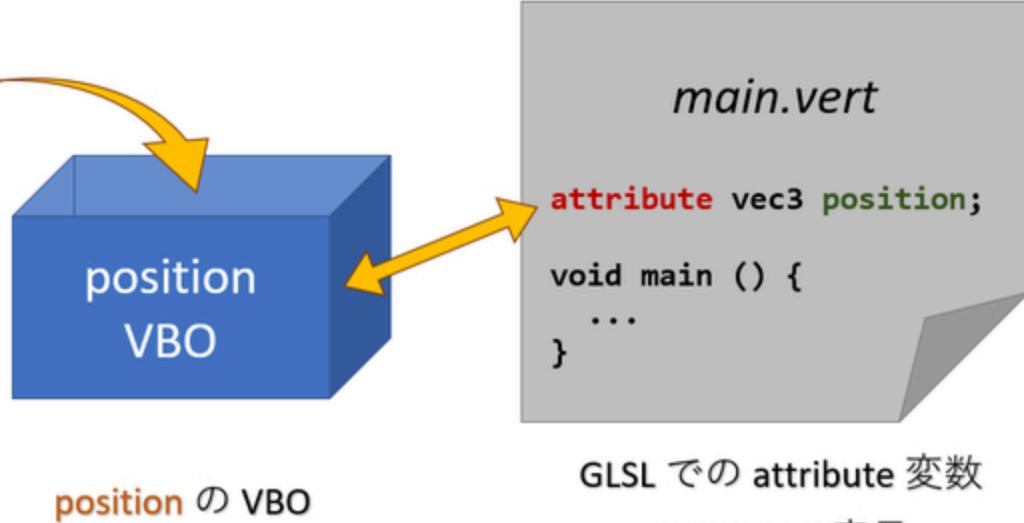
CPU 側で作った VBO が、GPU 側の世界と関連付けされるようなイメージです。

逆に言えば、attribute 変数にデータを送りたければ VBO を作らなければならない、ということ

```
const position = [  
    x, y, z,  
    x, y, z,  
    x, y, z  
];
```



JavaScript 上での
position の定義



```
main.vert  
attribute vec3 position;  
void main () {  
    ...  
}
```

この関連性がイメージできるかどうかがキモです

GLSL に書かれている `attribute vec3 position` と VBO を関連付けするためには必要となるのが ロケーション です。

これは一種のラベル、あるいはアドレスのようなもので「position という名前の変数にデータを関連付けするため」に必要となります。

001 のサンプルでは、`App.setupLocation` というメソッド内で、シェーダ側で宣言された `attribute` 変数のロケーションを取得しています。ここで変数名を 1 文字でも間違えてしまったりするとロケーションが正しく取得できませんので注意しましょう。

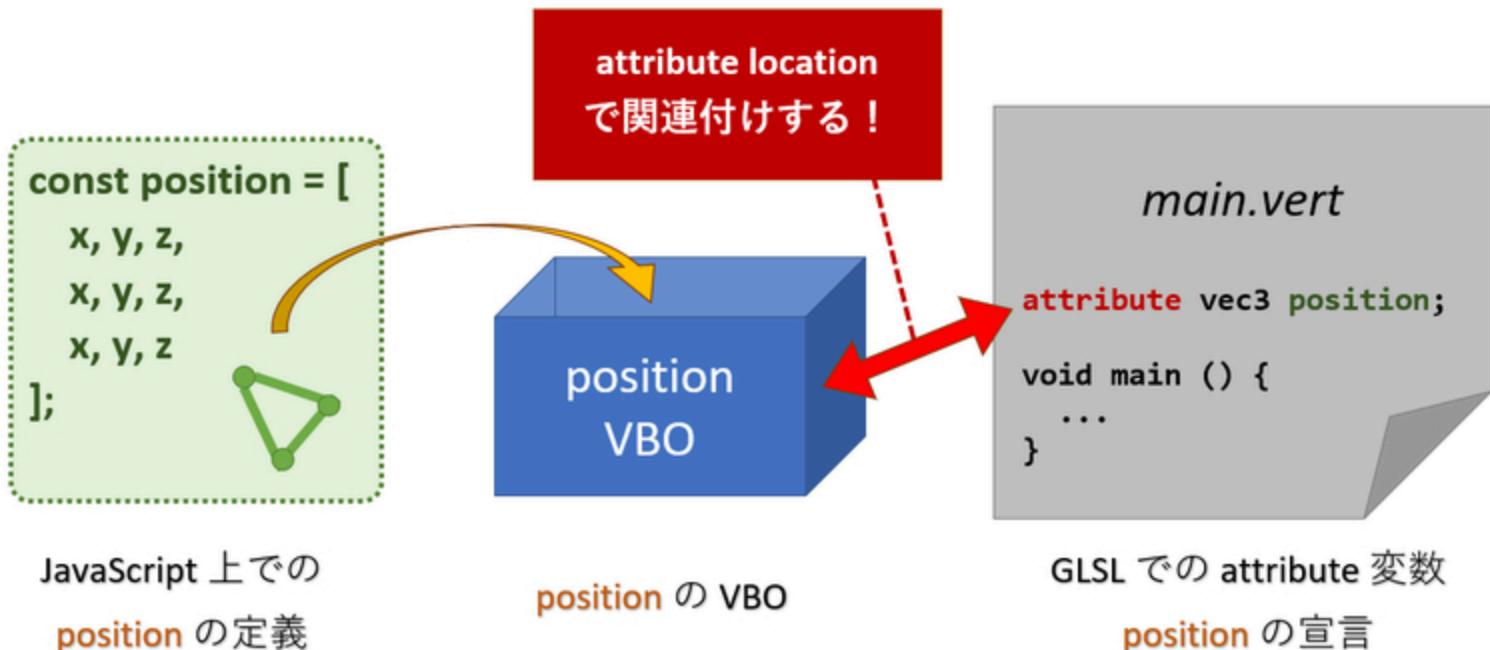
```
const attPosition = gl.getAttributeLocation(this.program, 'position');
```

上記の例では '`position`' という名前の `attribute` 変数のロケーションを、プログラムオブジェクトを経由して取得している

`attribute` 変数のロケーションが取得できたら、あとは「そのロケーションに対してデータ（VBO）を紐付ける」処理を行います。

このあたりはすごく複雑に見えると思いますので、最初はもう「必要な手続きとして形から覚える」ほうがよいのかなと思います。

WebGL API のメソッド名とかはコピー & ペーストすればいい話なので暗記したりする必要はありません



“attribute ロケーションがデータとシェーダ内の変数を紐付ける”

001 (その2)

- シエーダのコンパイルや、プログラムオブジェクトの準備を行い.....
- プログラムオブジェクトを介してロケーションを取得
- ロケーションに対して VBO を関連付けすることでデータを送る
- 最後に描画命令（ドローコール）を発行する
- このとき バインドされている VBO だけ が描画命令の対象になる

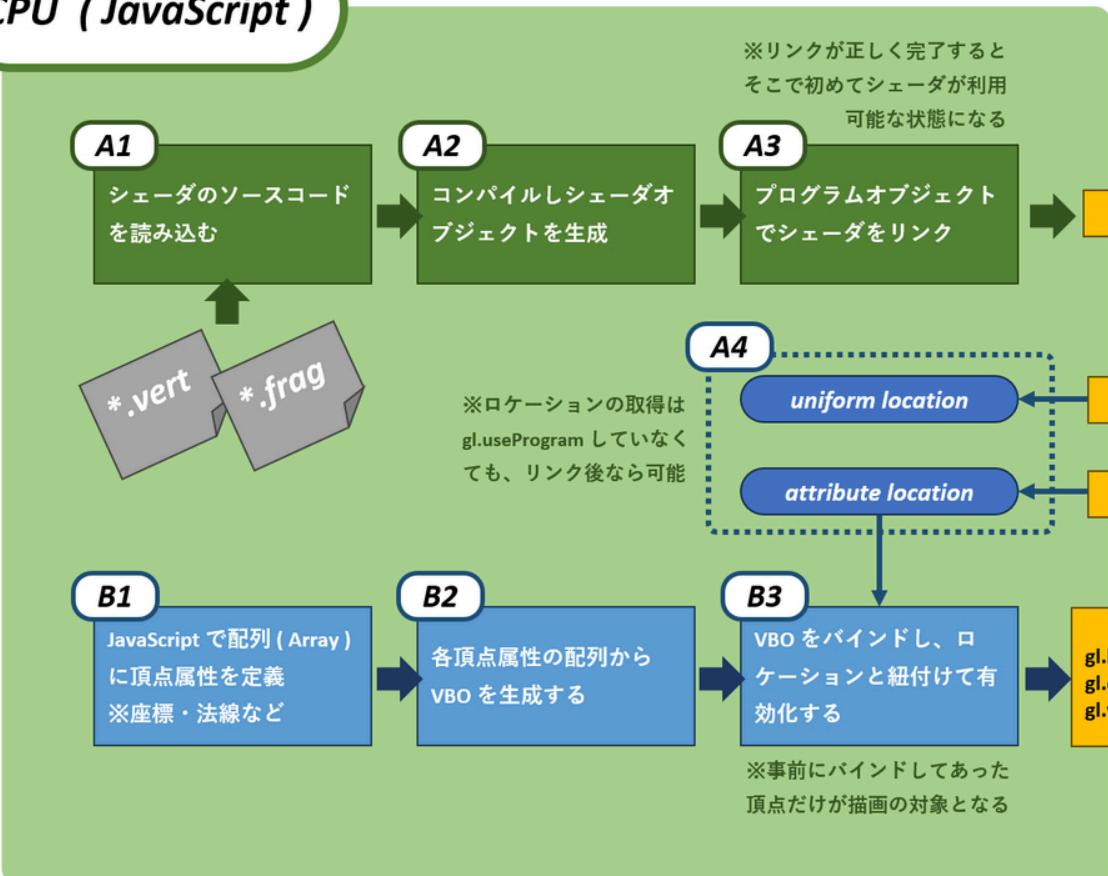
“Canvas2D のように「任意の図形を、描画する」という考え方ではなく「そのときバインドされている VBO だけが、描かれる」という考え方”

001 (その3)

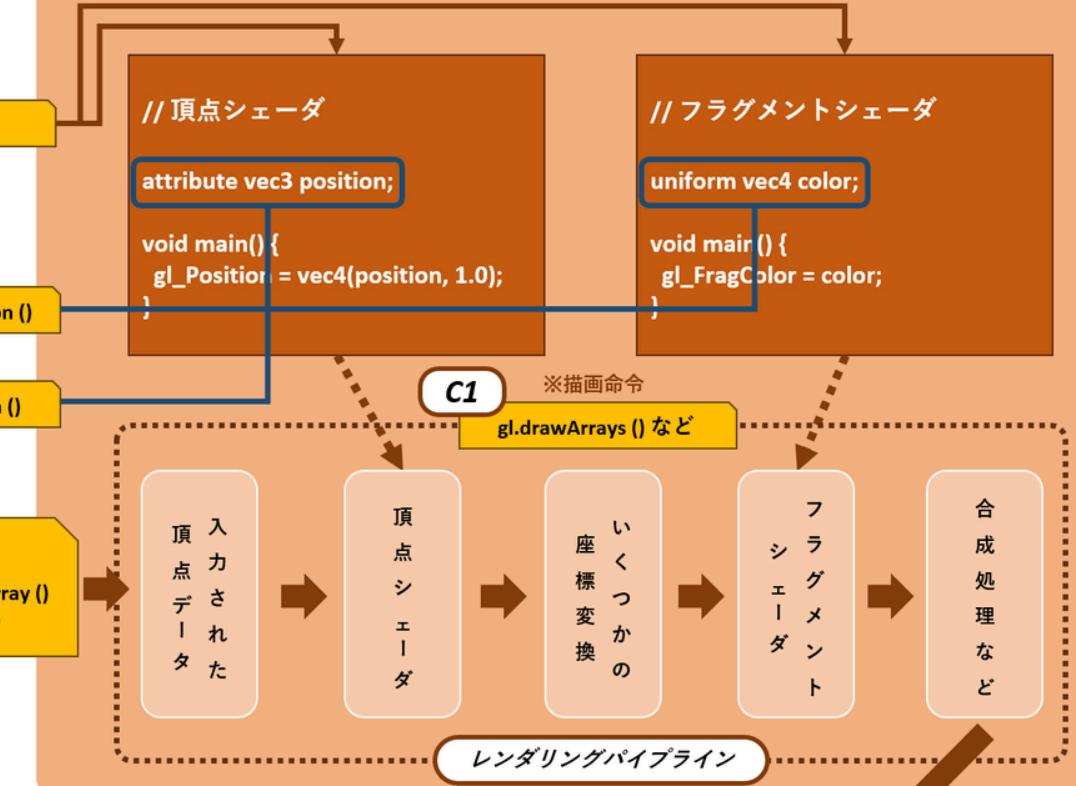
- GLSL 側には、必ず `main` 関数を定義する
- これがシェーダのエントリポイントになる
- 頂点シェーダでは `gl_Position` に頂点の座標を出力
- フラグメントシェーダでは `gl_FragColor` に色を出力

シェーダについては、順を追って詳しく解説していきます

CPU (JavaScript)



GPU (GLSL)



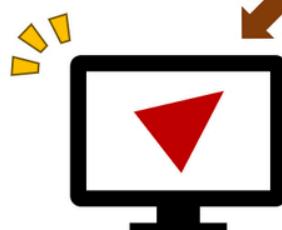
A の流れでシェーダ（プログラムオブジェクト）をセットアップし、あらかじめ GLSL 内の変数の「ロケーション」を取得しておく。

B の流れで描画する ジオメトリの頂点属性を定義し VBO を生成、「attribute location」に則した内容で バインド・有効化する。

最終的に 描画命令を発行する (C) と GPU 側でレンダリングパイプラインによってグラフィックスが生成され、出力される。

CPU と GPU はそれぞれ物理的に異なるハードウェア（メモリなどは自然には共有されない）であることに注意する。

※ ここでは uniform location については説明を省略しているが、attribute location と同様 JavaScript 側から汎用的な（頂点属性以外の）データを GPU に転送する際に利用する

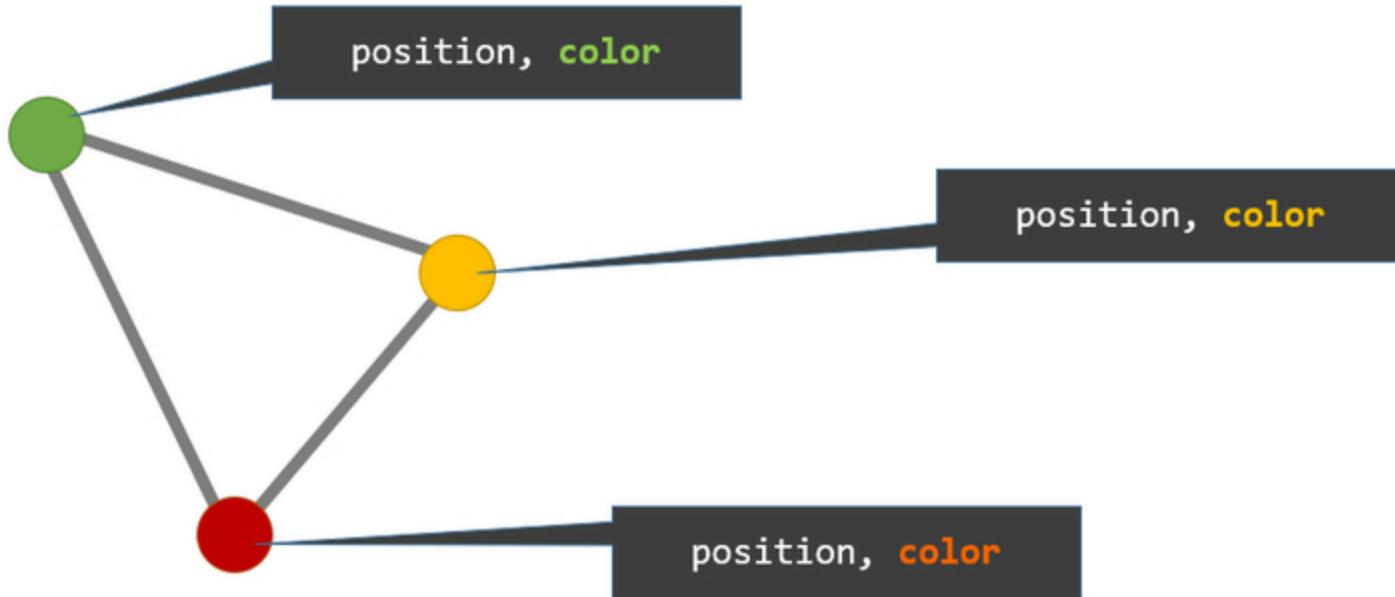


頂点属性を追加してみる

続いての 002 のサンプルでは、001 のときには 1 つだけだった「頂点属性」を追加してみます。

別の言い方をすると「attribute position だけではなく、attribute color を使えるようにする」ということです。

頂点自身が、座標の他に色の情報を持っている状態にする



頂点それぞれが固有に「色」の情報を持つ

位置だけじゃなく色も持っている状態

002 では、まず JavaScript 側で「頂点属性として色の情報を定義」します。これは `script.js` に該当する処理が記載されています。

```
this.color = [  
    1.0, 0.0, 0.0, 1.0, // ひとつ目の頂点の r, g, b, a カラー  
    0.0, 1.0, 0.0, 1.0, // ふたつ目の頂点の r, g, b, a カラー  
    0.0, 0.0, 1.0, 1.0, // みつつ目の頂点の r, g, b, a カラー  
];
```

ここで追加された処理を抜粋すると……

- 配列を使って、色の頂点属性を定義
- 色の頂点属性用の VBO を生成
- 色の頂点属性用にロケーションを取得
- 描画を行う前に有効化の手続きをしておく

これが頂点属性を追加するたびに必要となる基本的な手順

GLSL 側では、attribute 修飾子付きで変数を追加します。（
`main.vert` を参照）

JavaScript 側でロケーションを取得するために指定している変数名と、まったく同じスペルになっているかどうか、気をつけて記述しましょう。

```
attribute vec4 color; // 大文字小文字の違いは区別されます
```

さて、ここまで手順で attribute としての（頂点属性としての）色は追加できましたが……

WebGLにおいて、頂点シェーダはあくまでも頂点の座標変換をするのが仕事であり、最終的に出力される色を各ピクセルに対して処理するのは、フラグメントシェーダの仕事です。

以下のように 001 の頂点シェーダを見ると、どこにも色に関する設定箇所がありません。これは頂点シェーダは出力する色を制御する仕組みを持っていないからです。

```
attribute vec3 position;  
  
void main(){  
    // gl_Position は頂点がどのように描かれるかを決める座標の出力 (vec4)  
    gl_Position = vec4(position, 1.0);  
}
```

頂点シェーダからは「画面上に直接色を出力する方法がない」ので、頂点が持っている色の情報をなんとかしてフラグメントシェーダに渡してやらなくてはなりません。

ここで登場するのが varying 修飾子 です。

`varying` 修飾子を使うと、頂点シェーダからフラグメントシェーダへと任意のデータを渡すことができます。

両者は「同じ変数名、かつ、同じデータ型」であることに注意しつつ、002 のシェーダのソースコードを見てみましょう。

002

- JS 側では、VBO やロケーション周りの処理を新規に追加
- シェーダ側では、単純に attribute 変数を追加することに加え.....
- 頂点シェーダからフラグメントシェーダへと値（ここでは頂点カラー）を渡したいという状況が発生した
- そのようなケースでは varying 修飾子付きの変数を利用して、フラグメントシェーダに値を渡すことができる
- 両シェーダ間で変数名とデータ型を一致させておく

第三の修飾子 uniform

ここまで、GLSL における `attribute` 修飾子は「頂点属性」を表すものであり、`varying` 修飾子は「シェーダ間でデータを渡すためのもの」であることを見てきました。

GLSL には、これら 2 つの他にもう 1 つ修飾子の種類があります。これを uniform 修飾子 と言います。

GLSL 側で uniform 修飾子を付与された変数は「CPU から直接入力を受け取る」ことができます。

たとえば、プログラム実行開始からどれくらいの時間が経過したのか？などは、GLSL 側では調べる方法がありません。ですからシェーダ内で時間を使いたい場合、CPU 側の実装（ここでは JavaScript）からシェーダに対して値を送ってやらなくてはなりません。

`uniform` 変数は、GLSL 側では「JavaScript（CPU 側）から値が送られてくる変数」というふうに考えるだけでいいので、それほど扱いは難しくありません。

しかし、JavaScript 側から値を送る際には、ちょっとだけ手間がかかります。より具体的には「送るデータの型に応じて、メソッドを使い分けなくてはならない」というなかなか手厳しい仕様になっています。

003

- JavaScript から任意の値を GLSL に渡したい！
- そんなときは uniform 修飾子付きの変数を利用する
- GLSL 側では、単純に修飾子を付けて変数を宣言するだけ
- JavaScript 側から送る際はまず最初にロケーションを取得し.....
- 送りたいデータの型によってデータ送信用メソッドを使い分ける

慣れてしまうと結構簡単なのですが最初はちょっと覚えるのが大変かも

(付録) uniform のタイプ指定

- 数字は要素の数を表している
- GLSL 側で `vec4` なら 4 を使うし `vec2` なら 2 を使う
- `f` は float の f なので、つまり浮動小数点であることを表す
- もし f の代わりに `i` が使われていたら int、つまり整数のこと
- `v` は vector、つまり平たく言うと配列のことを表している
- これらを総合すると.....GLSL 側で `vec4` なら `4fv` を使う、ということ！

(付録) uniform のタイプ指定

- 代表的な uniform のタイプ指定文字列

GLSL で int → 1i

GLSL で float → 1f

GLSL で float の配列 → 1fv

GLSL で vec2 ~ vec4 → 2fv ~ 4fv

GLSL で mat2 ~ mat4 → matrix2fv ~ matrix4fv

ジオメトリを定義してみよう

これまでに何度も言ってきたことではありますが、最初から全てを完璧に覚えるのは誰にとっても難しいことです。

まずは、どんどん手を動かし、自分なりにトライ＆エラーを繰り返し、体験を積み重ねることで徐々に考え方がまとまってくると思います。

サンプル 004 は、初期状態では 003 と大して描画結果は変わりません。

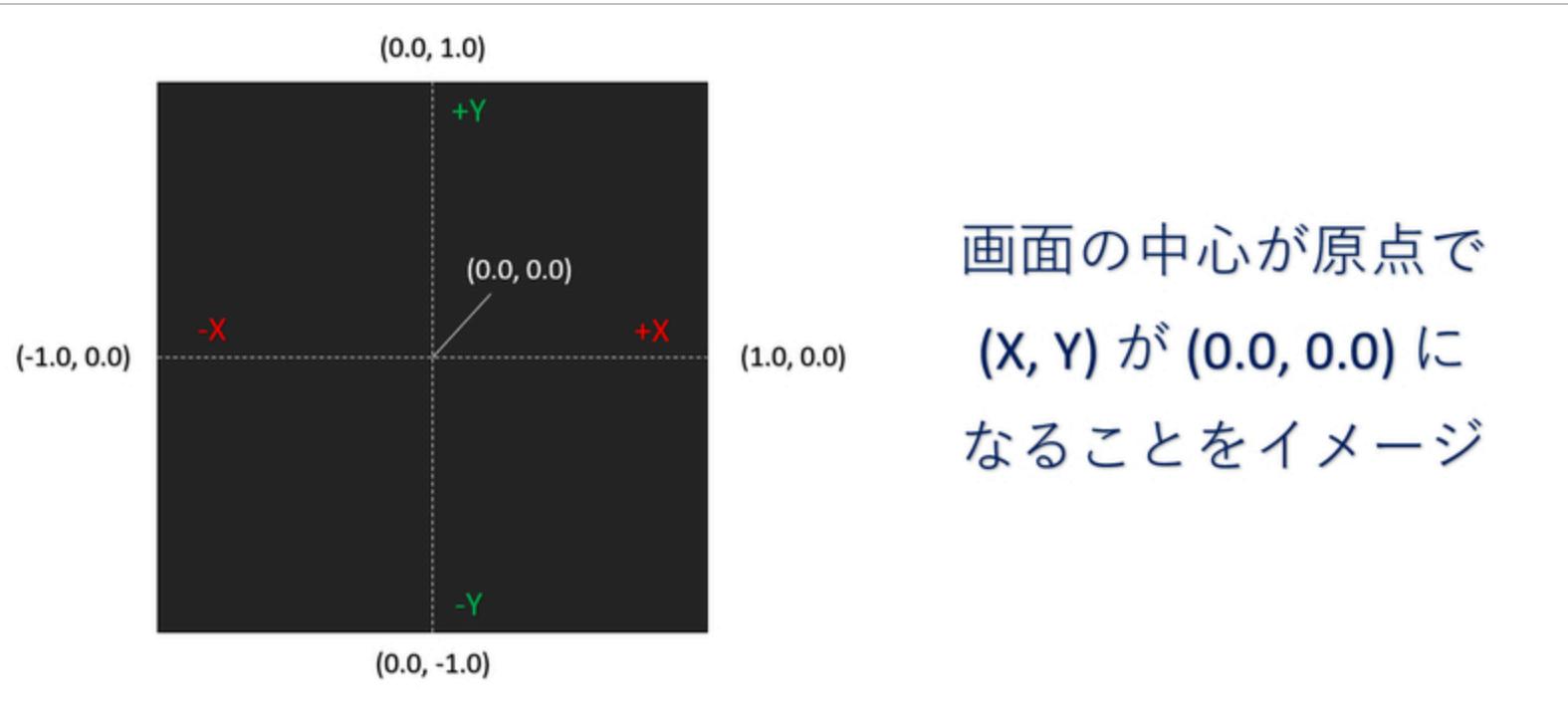
このサンプルは、みなさん自身で修正を加えてもらうことで完成する、課題サンプルになっています。

004 のサンプルの内容を改造して、形はいびつな感じでも構わないので、三角形ではなく五角形が描画されるように頂点の数を増やすことに挑戦してみましょう。

ここで失敗することは、当然ありえることですし恥ずかしいことじゃありませんから、まずは気軽にトライしてみましょう。

ポイントになるのは、以下のことをしっかり意識すること。

- WebGL ではポリゴンは三角形で表現する (`gl.TRIANGLES`)
- つまり四角形なら、三角形 2 枚を組み合わせて表現する
- 描画領域の中心が X も Y も 0.0、つまり原点
- X のプラス方向は右、マイナス方向は左
- Y のプラス方向は上、マイナス方向は下
- X も Y も、画面に映っている領域は -1.0 ~ 1.0 の範囲



画面の中心が原点で
(X, Y) が $(0.0, 0.0)$ に
なることをイメージ

004

- WebGL のポリゴンは三角形までしか定義できない
- 各頂点属性のストライドと配列の要素の数に注意する
- ストライドが 3 なら 頂点数 * 3 の配列に.....
- ストライドが 4 なら 頂点数 * 4 の配列になる
- 頂点属性のストライドがいくつであれ、頂点の個数分のデータが必要なので注意すること (position が 10 頂点分なら、color も 10 頂点分必要になる)

さいごに

今回はかなり内容が難しかったかと思います。

ほんのポリゴン一枚出すのにも、本当に大変なのが 3D API の世界.....この先ちょっと不安だなあと感じてる方も、実際少なくないかもしれません。

今回のスライドにはたくさんいろいろな概念や用語が出てきましたね。そのなかでも、特に重要なものをまとめると以下のようになります。

- シェーダは GPU 側にいる
- シェーダが使うデータは GPU にあらかじめ送っておく必要がある
- GPU に送られるデータにはいくつか種類があり.....
- VBO (attribute) として送るものと uniform として送るものがある
- それらの方法の違いにより GLSL 内部では修飾子が変わる

最低でも、今箇条書きにしたところはイメージできるようにしておきましょう。

逆に言えば、WebGL の細かな処理手順に多少の不安があっても、現時点ではひとまず大丈夫。不安になるよりも、まずは自分が触れる範囲から少しづつ触れていきましょう。

今回の課題は、まずは 004 のサンプルにもあった、五角形を作るという課題に挑戦することから始めてみましょう。

もし五角形がうまく作れたら、次は五芒星（星型）を描いてみるというように、少しずつ、複雑な形に挑戦してみましょう。

“ サイン・コサインを駆使すれば扇形や円形も実現できるはず…… ! ”

比較的見た目に派手なことができるようになるのは、もう少し先です。

今積み上げたものが、確実にそこへつながっていきますので、ちょっと地味ですがしっかり地固めをしていきましょう。