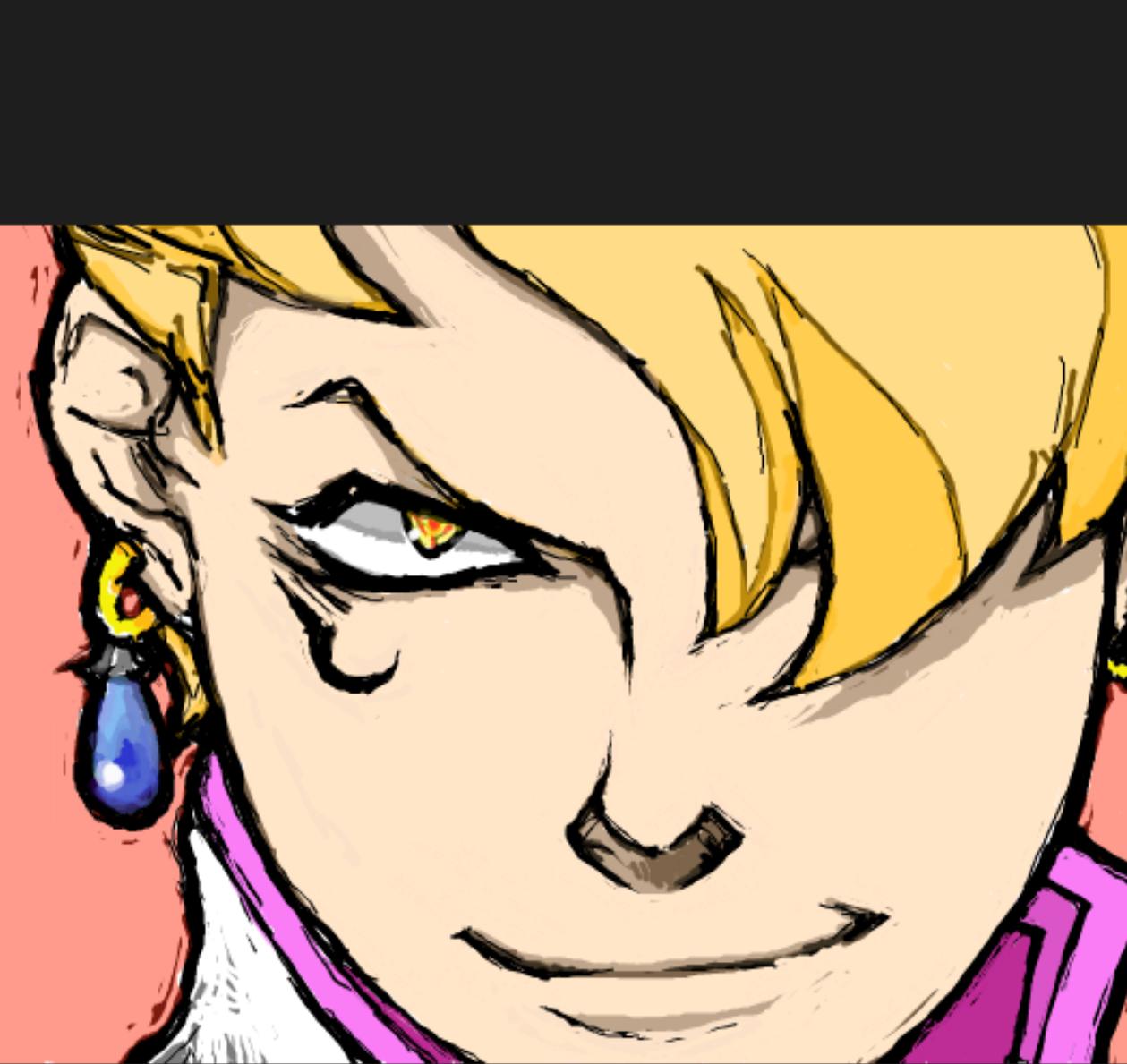


# **GLSL とはなにかを知ろう**

**シェーダと GLSL の概念・基礎知識**



# 自己紹介

杉本 雅広  
(すぎもと まさひろ)

@h\_doxas

# 本スクールの開催概要

本スクールは、WebGL などの OpenGL ファミリで主に利用されている **GLSL** と呼ばれるシェーダ言語を主題に、シェーダの記述に特化した技術を広く学習することを目的としたスクールになります。

シェーダとはそもそもなんなのか、そして WebGL や GLSL との関係とはどんなものなのかななど、シェーダ記述のスキルアップのための基礎から応用までの知識を総合的に扱っていきます。

プラットフォームには、最も手軽に GLSL と向き合える WebGL を使っていきます。文字通り、シェーダや GLSL についてがメインテーマになりますので WebGL や JavaScript に関して少々説明を省いて進めていく形になります。

もし、通常の JavaScript の記述方法や、WebGL 実装部分での疑問点で詰まってしまいそうな場合はどんどん質問していただいて大丈夫です。

さてまずは、シェーダを記述するために最低限必要となる、前提や基礎を固めておきましょう。

後半は実際にみなさんにも手を動かしてコードを記述してもらうフェーズも出てきます。まずは基本的な概念の部分からしっかり押さえていきましょう。

必ずしも暗記する必要はないので楽な気持ちで聞いてください

シェーダとはなにか？

シェーダ、という言葉には非常に広い意味があり、一言で表現するのは難しいです。

ただ、一般に解釈されているシェーダをあえて端的に表現すると GPU  
を利用することで高速に動作 することが特徴の、主にグラフィックスを  
描画するための技術や概念の総称 というふうに言えるかもしれません。

GPU は、**G**raphics **P**rocessing **U**nit の頭文字を取った略語で、主にグラフィックスを描画するための機能に特化したハードウェアです。性能的には、単純計算を大量にこなすことに特化しています。

シェーダは原則として GPU を利用するため、いわゆる普通の CPU 上で動作する JavaScript などと比較すると演算能力が桁違いに高いのが特徴です。

“ CPU は Central Processing Unit の略語 ”



NVIDIA 製 GPU、RTX 4090。  
(発表当時の写真、向かって左  
が RTX 4090)

でかい.....

ウェブの世界においても、WebGL が登場したことによりシェーダを活用した高速なグラフィックスの描画が行えるようになりました。

また、WebGL やシェーダというと一見「3DCG 専用」のように感じるかもしれません、これは必ずしも 3D とは限りません。2D の描画であっても、GLSL の高速な動作が役に立つ場面は多く活用できる場面はたくさんあります。



## PLST | ウォームリザーブパンツ

液体のような表現もシェーダを利用すれば実現可能



@kishimisu

アートとしてのシェーダによる表現も人気が高い

# **3D API**

突然ですが、みなさんがよく目にする 3DCG って、どうやって生成されているか想像できるでしょうか。

なにか特別な魔法によって画面に描き出されているんでしょうか？ なんだかバカみたいな質問に聞こえるかもしれないですが、しかし実際そう問われると意外と答えられなかったりするのではないかでしょうか。

3DCG を描くためにコンピューターが行っていることは極論を言えば 画面上のすべてのピクセルに対して、ピクセルが何色になればよいかを決める作業 です。

オブジェクトの形はこうで、色はこうで、光はこういう方向で当たっていて.....というように、様々なパラメータを考慮しつつひたすら計算するのですが、究極的にやりたいのは「ピクセルの色を決める」を「できるだけ高速かつ大量にこなす」ことなのです。

だからこそ、計算能力に優れた GPU を用いることで得られる恩恵が大きくなります。GPU を用いることで CPU では難しいようなハイエンドなグラフィックスが生成できるのです。



“GPU は性能も価格も様々。当然高価で高性能なものほど演算能力が高い。

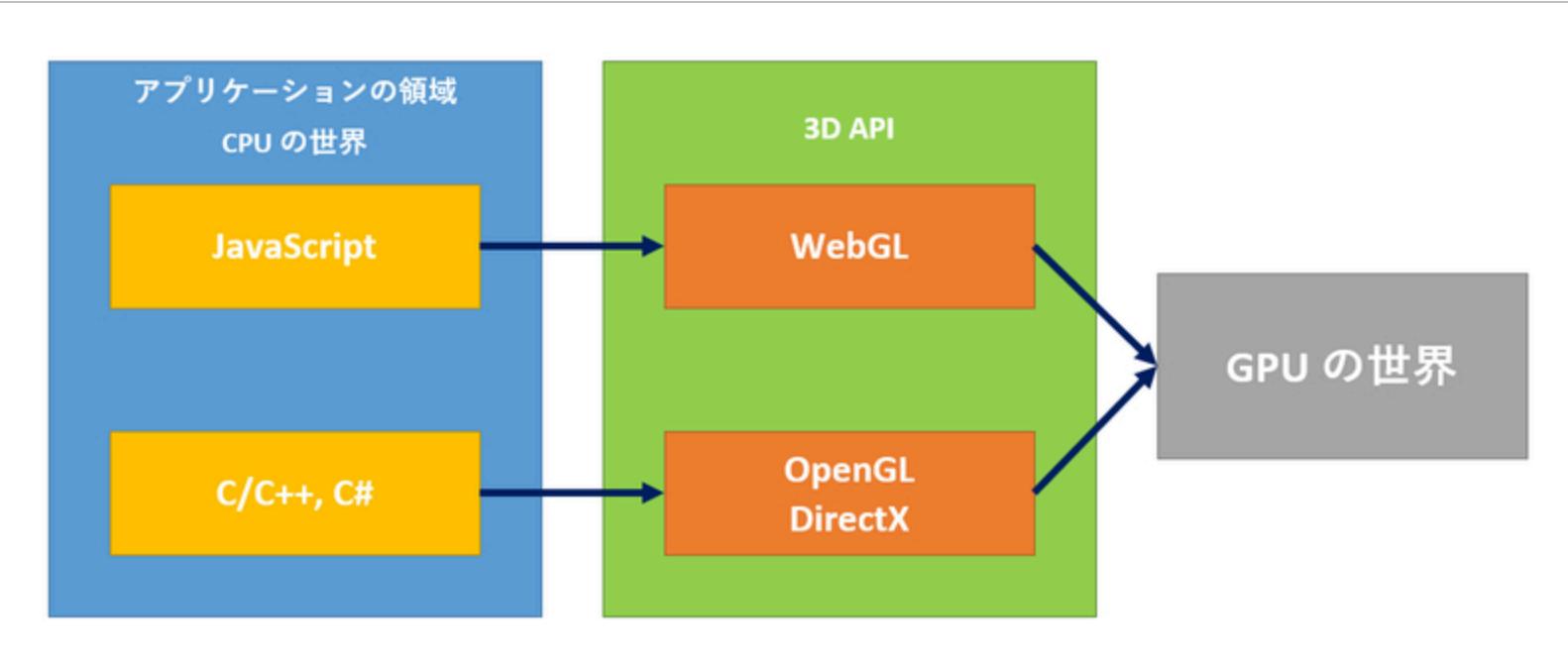
GPU が計算能力が高いことはわかったけれど、実際にそれを活用した開発を行うためには具体的にはどうすればよいのでしょうか。

GPU をプログラム等から利用するには、専用のインターフェース（API）を用います。OS の種類や、その用途に合わせて API にはいくつかの種類があります。

Windows であれば、DirectX と呼ばれる Windows に最適化された高性能な API が存在しますし、オープンソースで提供されている API として OpenGL がよく知られています。

ソフトウェアを設計・開発する我々開発者は DirectX や OpenGL の使い方を覚えることによって、これらの API を通して間接的に GPU の性能を引き出した高いパフォーマンスを誇るアプリケーションを制作することができるようになります。

“ 無数にある GPU のすべてに詳しくならなくても API の使い方を覚えれば済む、ということ ”



GPU は無数に種類があるので、統一したインターフェース（API）で操作できることの利便性が大きい。

ちなみに、近年その名前を耳にすることも多くなつた Vulkan や Metal、DirectX 12、さらには WebGPU などの API がありますが、これらは OpenGL (WebGL) よりもさらにハードウェアに近い低レベルなレイヤーを操作することができる API です。

ハードウェアに近い部分を直接操作することができるため総じて高速ですが、難易度も比例して高くなる傾向があります。

すごい雑な例えをすると、Vulkan なんかはもはや OpenGL そのものを自作するような感じ

# **OpenGL, OpenGL ES, GLSL**

OpenGL はその名の通り、オープンな規格として Khronos によって管理されている 3D API で、モバイル向けの OpenGL ES と共に、現代でも様々なところで利用されています。



“ ES のほうはモバイル向けの軽量な OpenGL 実装です。 ”

ウェブというプラットフォームで利用される WebGL も、やはり世界標準の仕様に則した API です。



“ WebGL はあくまでも JavaScript の API であり、仕様としては OpenGL ES に近い（というかほぼ同じと言ってもよい）”

さて、それではこのスクールの主役である GLSL とはなんなのでしょうか。

**GLSL** ([OpenGL Shading Language](#)) は、OpenGL ファミリーで利用できるシェーダを記述するため言語のことです。OpenGL や WebGL という 3D API がまずあり、その API でシェーダを利用する際に、シェーダを記述するための言語として GLSL を使う、ということです。

わざわざ GLSL という「シェーダを記述するための言語」が用意されているのは、シェーダが GPU 上で動作する独立したプログラム であるからです。

ウェブブラウザというプラットフォーム上で動作するプログラムを記述する JavaScript、というのと同じように、GPU 上で動作するシェーダと呼ばれるプログラムを記述するために GLSL が存在します。

## 3Dを扱えるグラフィックス API（代表的なもの）

DirectX (Microsoft 製品専用)

HLSL

Metal (Apple 製品専用)

Vulkan (Khronos が管理する次世代 API)

OpenGL ファミリー

OpenGL (PC 向け)

GLSL

※バージョンは色々

OpenGL ES (モバイル向け)

WebGL (JavaScript の API)

“ API が変わるとシェーダ記述言語も変わる ”

# ここまで

## までの要点

- GPU は演算能力の高いハードウェアである
- 3D API にはプラットフォームなどに応じていくつかの種類がある
- DirectX や OpenGL、OpenGL ES、Metal、Vulkan など
- これらの API を用いることで GPU のパワーを引き出せる
- OpenGL や WebGL でシェーダを記述するための専用言語が GLSL

つまり GLSL をはじめとするシェーダ記述言語をうまく扱うことができれば GPU を使った高速なプログラムを動かすことができるということ

# シェーダの種類と役割

さて、ここからはもう少し詳しく、シェーダの役割について考えてみましょう。

まずは、現在一般によく利用されているシェーダの種類にどんなものがあるのか、見ていきます。

とは言え、ここではみなさんがこれから学習していく予定の、当スクールで登場するシェーダにある程度限った話をします。

というのは、現在ではシェーダの種類や役割は非常に多岐に渡るのでそれら全てを一度に理解するのは難しいためです。逸る気持ちもあるかもしれません、まずはどんなプラットフォーム上でも利用できる基本的なシェーダから覚えていきましょう。

# 頂点シェーダ

まず最初に登場するのが **頂点シェーダ** です。

このシェーダは、3DCG に欠かすことのできない、頂点の座標変換を主な役割とするシェーダです。

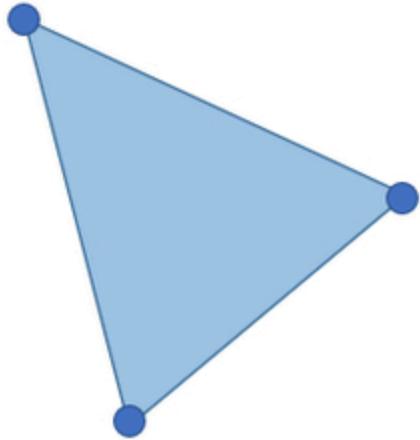
3D プログラミングでは、あらゆるものは頂点によって表現されます。頂点シェーダを利用することで、この頂点の位置や、頂点の持つ様々な情報を自在に制御することが可能になります。

頂点は「3DCG を描画する上で欠かせない」と書きましたが.....

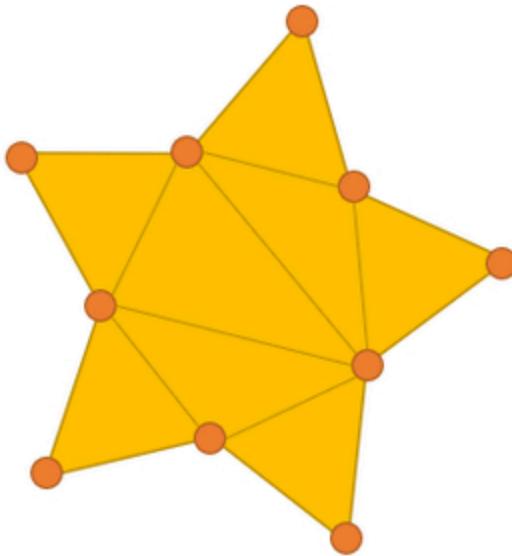
これはちょっと違った言い方をすると、WebGL などの API では「頂点しか描くことができない」というのが正しい表現かもしれません。

つまり、どのような複雑な形状も WebGL や OpenGL では基本的にすべて「頂点を組み合わせる」ことで形状を表現していきます。頂点がひとつで点を、頂点が 2 つで線を、それ以上の個数の頂点を用いて三角形や四角形を表現するのが、3DCG の一般的な概念です。

頂点シェーダは、この頂点ひとつひとつを制御するためのものです。ですから、頂点の個数 = 頂点シェーダが実行される回数 となるのが普通です。



3 頂点



10 頂点

頂点が増えれば増えるほど頂点シェーダの実行回数が増える。

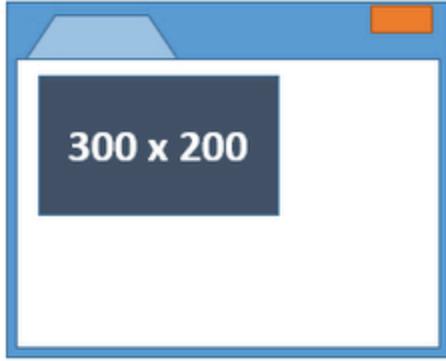
# フラグメントシェーダ

頂点シェーダと並ぶ、シェーダ界の代表的な存在に フラグメントシェーダ があります。API の種類によっては、ピクセルシェーダなどと呼ばれることもあります。

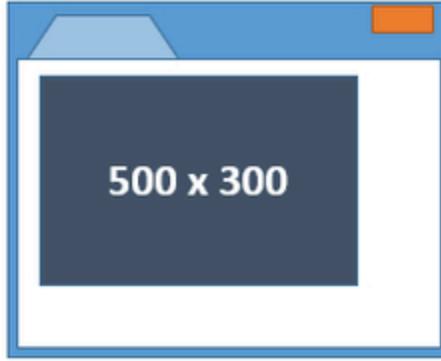
このシェーダは頂点シェーダとは役割が異なり、最終的にスクリーンに描かれる ピクセルひとつひとつ を対象とした処理を行います。

つまり、画面の解像度が大きい場合、当然フラグメントシェーダによる負荷は大きなものになります。

頂点シェーダは頂点の個数に比例して負荷が増えますが、フラグメントシェーダは実行されるピクセル数が増減することで大きな影響を受けます。

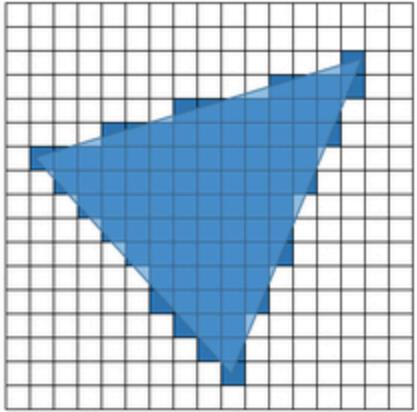


60,000px



150,000px

描画する領域の広さによって塗るべきピクセルの数は大きく変化する。



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

より正確には、頂点によって塗られることになるピクセルひとつひとつ  
がフラグメントシェーダの処理対象となります。

# 頂点シェーダとフラグメントシェーダ

- 頂点シェーダ
  - 頂点の個数分だけ頂点ごとに実行される
- フラグメントシェーダ
  - 描画するピクセル単位で実行される

両者の違いをしっかり認識しておこう

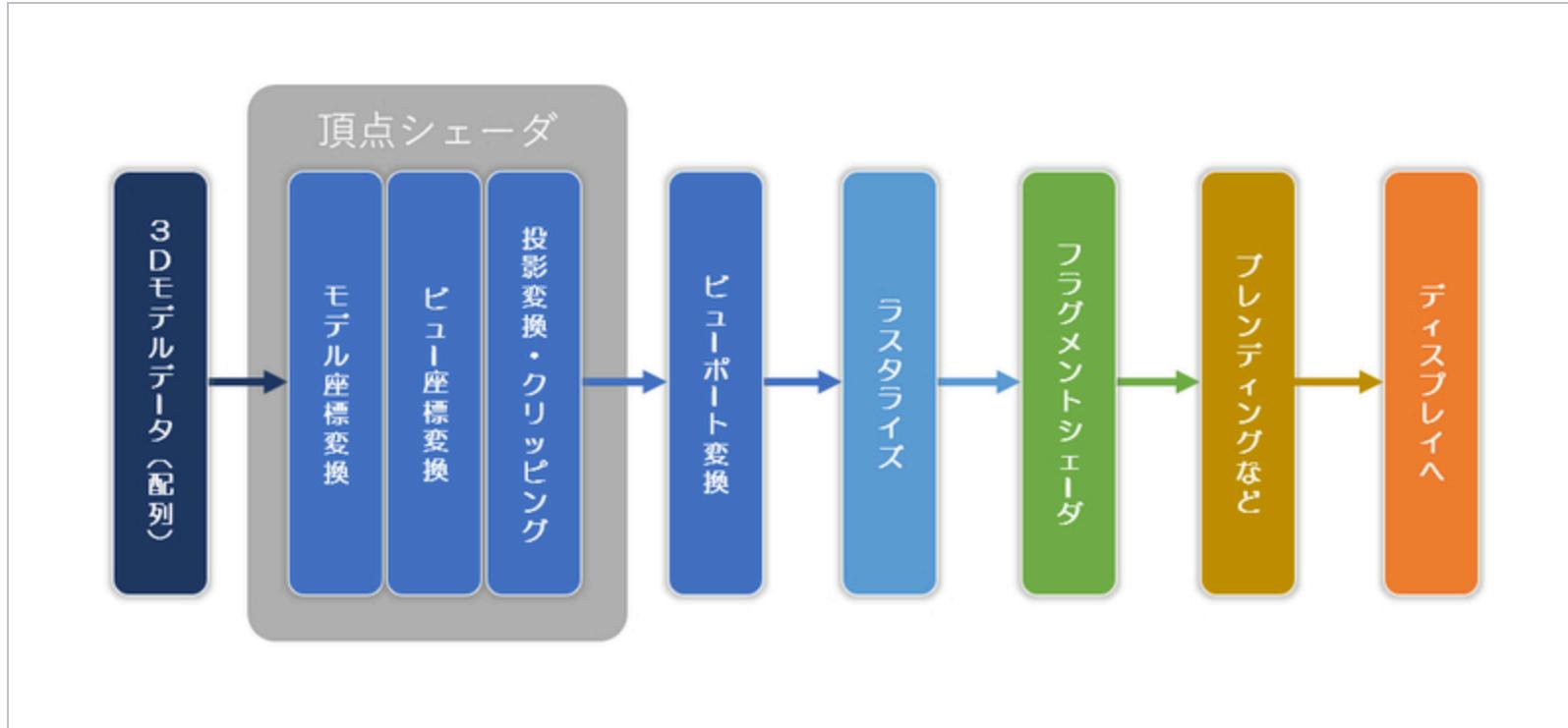
# ここまでの一覧

- シェーダにはたくさんの種類がある
- プラットフォームを限定するものもあり非常に難解な場合が多い
- 頂点シェーダとフラグメントシェーダは特に大切なシェーダ
- 頂点シェーダは頂点の個数に応じて呼び出される
- フラグメントシェーダはピクセルの個数に応じて呼び出される
- WebGL と GLSL の組み合わせの場合は、この 2 種類のシェーダを記述可能

# グラフィックスパイプライン

さて、GLSL を用いれば GPU 上で動作するプログラムを記述することができ、さらにシェーダには（ここではひとまず）頂点シェーダとフラグメントシェーダの2種類があることがわかりましたが、これらはディスプレイに映像が映し出されるまでの過程でどのように利用されるのでしょうか。

ここでは簡単に、GPU 内で行われる処理の一連の流れを概要だけで構いませんので把握しておきましょう。



これは GPU 内部で行われる処理をほんの一部だけ抜粋したものです。

アプリケーション（CPU 側）から情報を受け取った頂点シェーダは、頂点を指定された情報を元に座標変換しパイプラインの次のステージに送ります。そして座標変換された頂点は、ラスタライズなどの処理を経てフラグメントシェーダへと渡されます。フラグメントシェーダで着色などが行われたあと、いくつかの行程をさらに経た後にやっとディスプレイに映像が出力されます。

このパイプラインの行程全体をグラフィックスパイプラインと呼び、非常に多くの行程があります。

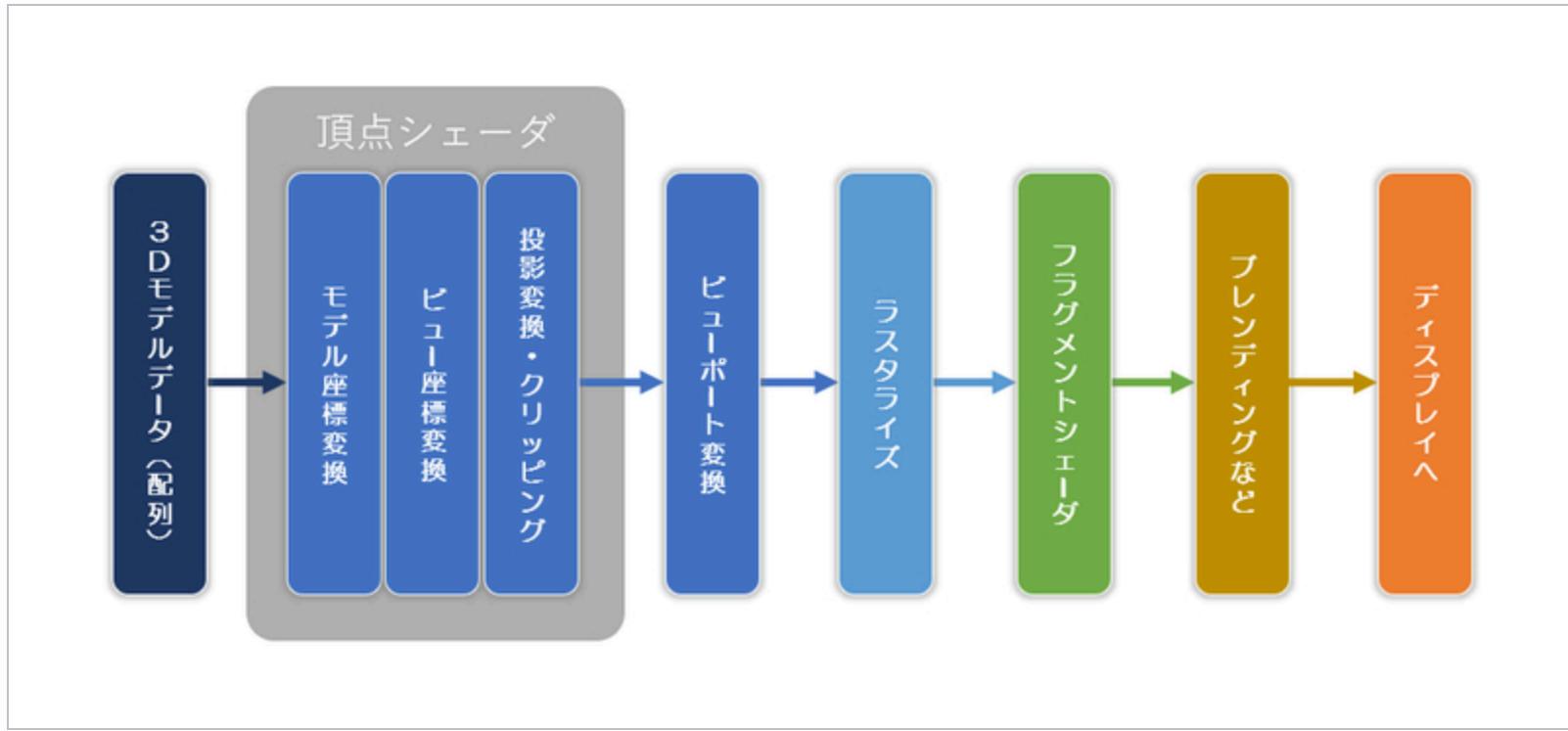
“レンダリングパイプライン、と呼ぶ場合もあります。”

頂点シェーダ、フラグメントシェーダはもちろんのこと、それら以外のシェーダについても基本的にはグラフィックスパイプラインの何かしらの処理を置き換えたり、拡張したりするものであるのが普通です。

とは言え、レンダリングのフロー（処理順序）そのものはある程度決まっています。

ここで最も重要なポイントとなるのは、頂点シェーダとフラグメントシェーダのうち、先に呼び出されるのは常に頂点シェーダであるということです。

これはどうしてかと言うと、頂点シェーダによって頂点が動いたり変形したりするなかで、当然画面に映るものと映らないものが出でてきます。頂点シェーダがまず最初に形を作り、その後の行程で色を塗るべきだと判断されたピクセルに対してのみ フラグメントシェーダが実行されるようになっています。



先程の図を見ても、パイプラインの仕組みから考えても必ず先に頂点シェーダが実行されることがわかります。

もし仮に、頂点シェーダで処理した結果、ひとつも頂点が画面上に描かれなかつたとしたら……どうなるのでしょうか。

そのときは、塗るべきピクセルがひとつもないということになりますので、フラグメントシェーダは一度も実行されません。

逆に言えば、画面上の全てのピクセルを頂点やポリゴンが覆っている場合、全てのピクセル上で何かしらのフラグメントシェーダが実行されます。

フル HD の  $1920 \times 1080$  だと、約 214 万ピクセルです。気軽に掛け算をひとつフラグメントシェーダに追記すると、214 万回の乗算処理が GPU によって処理されることになります。

“ そう考えると GPU って半端ない処理能力ですよね..... ”

# ここまで

## までの要点

- CG がレンダリングされる過程には順序がある
- その処理の流れを一般にグラフィックスパイプラインと呼ぶ
- つまり、シェーダが実行される順序もやっぱり決まっている
- 頂点シェーダが必ず先に実行され.....
- 塗るべきピクセルが確定したあと、その色を決めるためにフラグメントシェーダがピクセル単位で実行される

“ 実際にはポリゴンが重なってる場合もあるのでやり方次第でいくらでも重くなる（ディスプレイの物理ピクセル数以上の負荷になることもある） ”

**サンプルを見ながら考えていこう**

それでは、GLSL を実際にどのように記述していけばいいのかを把握するためには、少しずつコードのほうも見ていきましょう。

まず最初は、サンプル 001 からです。こちらのサンプルは、主に WebGL 側の処理の流れを把握するためのもので、普段 JavaScript を書いてらっしゃらない方にはちょっと難しい部分があるかもしれません。

とは言え、C++ などで同様の実装を OpenGL で組む場合でも、基本的な流れはこれとほとんど同じになります。

Unity や TouchDesigner などの場合は、こういった基盤処理は全てアプリケーション側で担保してくれるので意識することはほとんどないでしょう。しかし、こういったアプリケーションの下地の部分を知っておくことはけして無駄にはならないと思いますので、コメントを参考にしながら特にその 一連の処理の流れ を把握するようにしましょう。

# 001 を読み解くまでの要点

- `WebGLUtility` は WebGL の API を呼び出すユーティリティクラス
- `ShaderProgram` はシェーダに関する情報をまとめて保持するクラス
- これらは `three.js` などと同じ一種のライブラリのようなもので.....
- サンプルのメインとなる実装は `WebGLApp` クラスに記述されている
- シェーダのソースコードは外部ファイル化されていて.....
- それらは非同期に読み込まれることに注意する

# WebGL コンテキスト

WebGL では、`canvas` 要素から取得した `WebGLRenderingContext` と呼ばれるコンテキスト・オブジェクトを経由して様々な処理を行います。

ちなみに、サンプルでは `WebGLApp.init` のなかで `WebGLRenderingContext` を取得しています。

```
// canvas から WebGL コンテキスト取得を試みる
this.gl = this.canvas.getContext('webgl', option);
```

スクールのサンプルでは `gl` という変数や、`WebGLApp.gl` (`this.gl`) がこの WebGL のコンテキストを格納するための変数として統一されています。

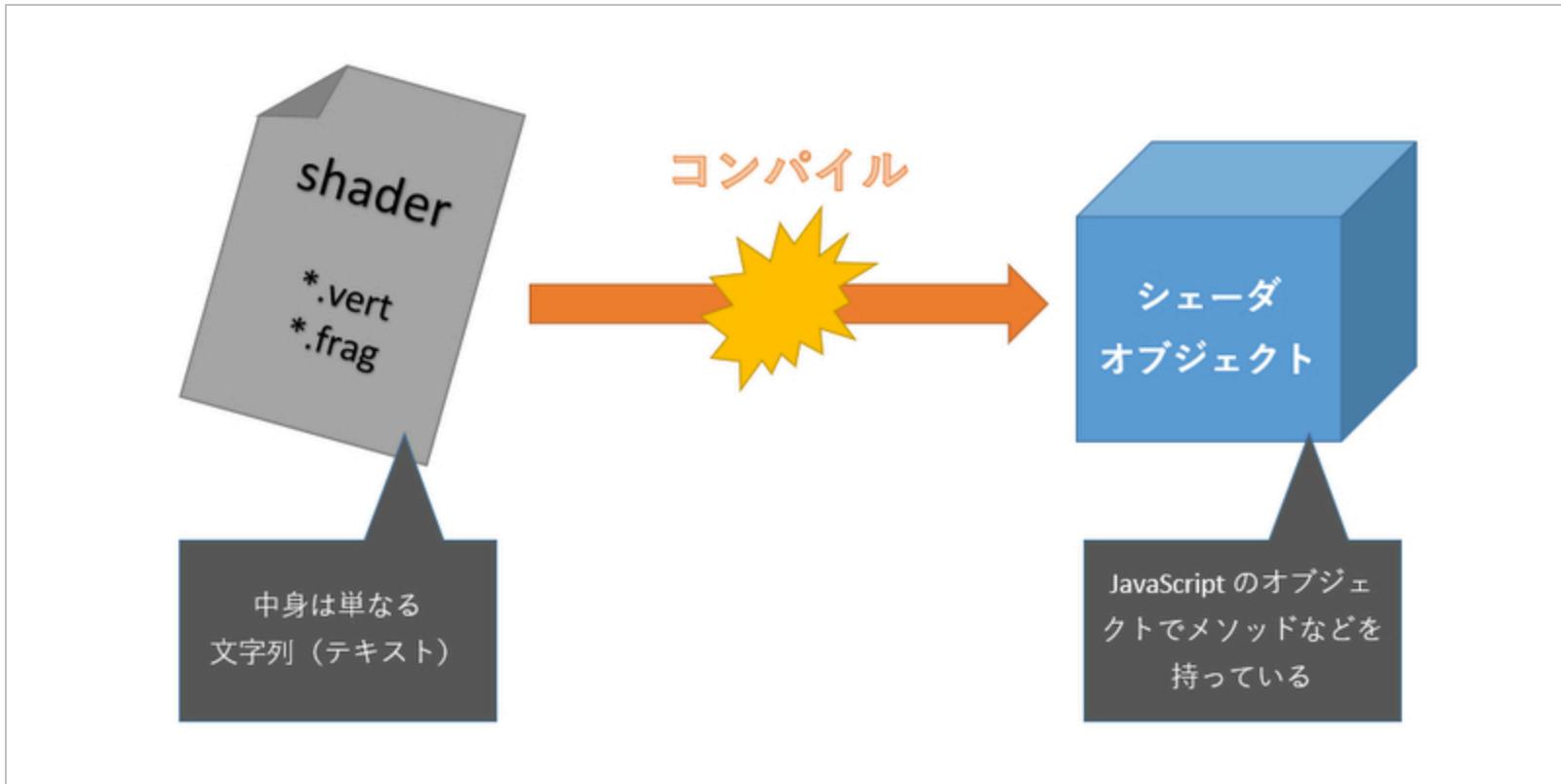
ですから、コードのなかに `gl.xxxxxx` というように `gl.` (または `this.gl.`) から始まる記述があったときは、それが WebGL の API をコールしているもの のだな、と考えるようにしてください。

“ サンプルとして独自に実装したクラスのメソッドなのか、本来の WebGL API のメソッドなのか、そこを勘違いしてしまうと理解の妨げになります ”

# シェーダの読み込みとコンパイル

WebGL や OpenGL では、シェーダを利用するためには「GLSL で書かれたシェーダのソースコード」が必要になります。

シェーダとは一種の小さなプログラムなので、GLSL のソースコード（これは単なる文字列）を JavaScript 側でコンパイルするという手順が必要です。コンパイルされたシェーダは、単なる文字列の羅列から、そこで初めて実体のあるシェーダオブジェクトになります。



“ファイル由来でなくても、単に変数に文字列を代入したものなどでもよい”

サンプルでは、`ShaderProgram` という独自のクラスのインスタンスを生成する際に、その内部でコンパイルを行っています。

```
// シェーダのソースコードをテキストとして読み込む
const vs = await WebGLUtility.loadFile('./main.vert');
const fs = await WebGLUtility.loadFile('./main.frag');

// シェーダの情報をひとまとめにしたオブジェクトを生成する
// ※ShaderProgram は webgl.js に実装されている独自のクラスであることに注意
this.shaderProgram = new ShaderProgram(this.gl, {
  (中略)
});
```

# プログラムオブジェクト

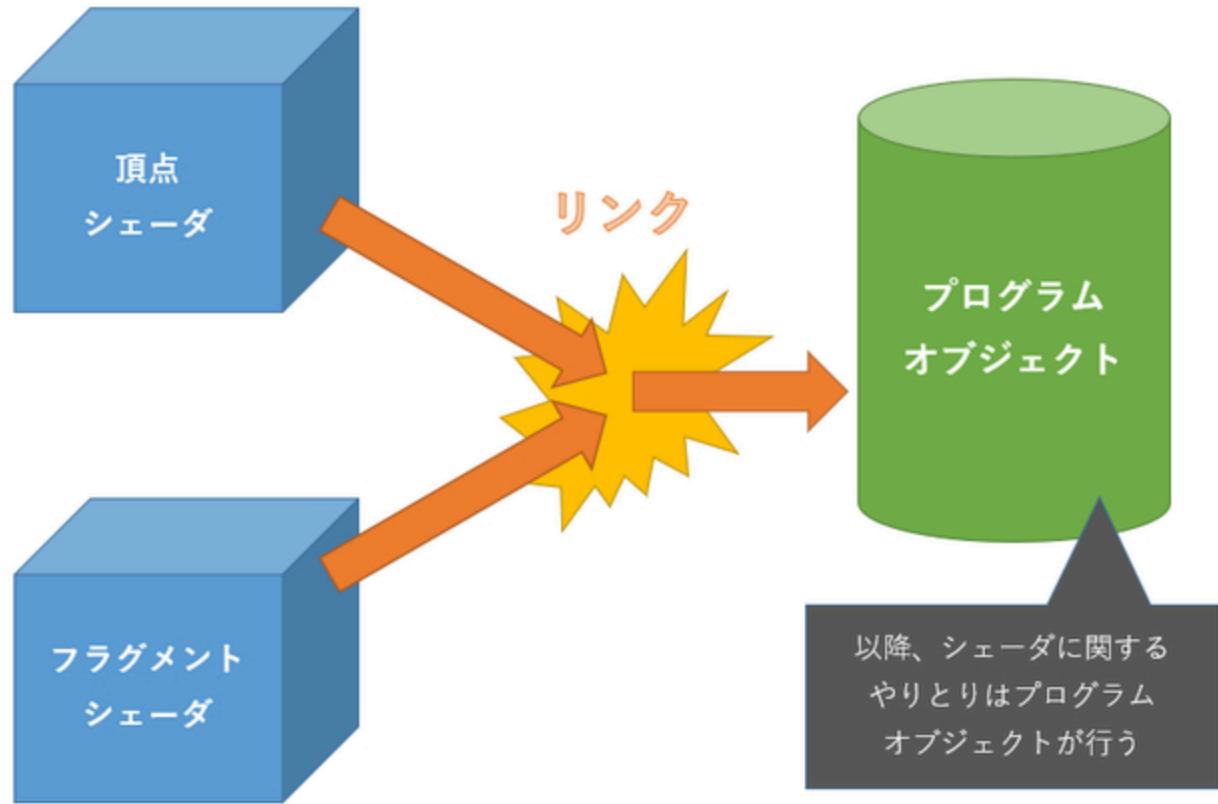
さて、シェーダオブジェクトがコンパイルされて生成されても、実はこれですぐさまシェーダが使えるようになるわけではありません。

シェーダには、先にも触れたとおり頂点シェーダとフラグメントシェーダという異なる種類のものが複数あります。これらの種類が異なるシェーダは双方でデータの受け渡しなどを行う場合があるため これらを管制する役割を持つ別の登場人物が出てきます。

シェーダ同士、あるいはシェーダ（GPU）とCPUとを結ぶ役割を持つ、管制塔のようなオブジェクト……

これを プログラムオブジェクト と呼び、コンパイル済みのシェーダオブジェクトをこのプログラムオブジェクトを利用して関連付ける（リンクする）ことにより、やっとシェーダを使った描画を行うための準備が完了します。

“プログラムオブジェクト、という名前は非常にナンセンスでわかりにくいのですが、仕様上そういう名前なので頭の片隅に憶えておきましょう”



以降、シェーダというよりは、それを内包したプログラムオブジェクトが GPU とやりとりする役割を担う。

サンプルでは、先程も登場した `ShaderProgram` クラスのインスタンス生成時に、内部的にプログラムオブジェクトの生成まで一気にまとめて行ってしまう作りになっています。

ですから `ShaderProgram` クラスのインスタンス内部には、シェーダオブジェクトの他、それらを管制するプログラムオブジェクトも同時に含まれます。

つまり `ShaderProgram` クラスを使う限りは、あまり「シェーダオブジェクト」や「プログラムオブジェクト」を気にしなくても大丈夫です！

# ここまで的小まとめ

- サンプルでは `WebGLRenderingContext` を `gl` という変数名やプロパティで保持する設計になっている
- つまり `gl.xxxx` や `this.gl.xxxx` という記述を見かけたら WebGL の API を直接呼び出しているとみなせばよい
- シェーダのソースコードをコンパイルしたり、プログラムオブジェクトを生成したりといった処理は `ShaderProgram` クラスが内部でまとめて行っている

シェーダの書き方などの本質的な部分になるべく注目・集中してほしいので  
このような設計にしています

# WebGL の描画までの流れ

シェーダの初期化が完了したら、GLSL 側でどんな名前の変数を使っているのか、その変数のデータ型はなんのか.....

そういうことを確認しながら、双方で正しいやりとりが行えるよう準備をしていきます。

GLSL は GPU 側で動作するプログラムです。一方、JavaScript は CPU 側で動作しています。

このように異なるハードウェア間でデータをやりとりするためには、ちょっと面倒ですがしっかりと手順を踏んでデータの通り道を構築しなければなりません。

サンプルでは、先程からたびたび登場している `ShaderProgram` クラスが、なんとこの開通作業も内部でうまいことやってくれるように実装しています。

ですが、ここはシェーダの挙動を理解する上で非常に重要な部分ですので、少し詳しく説明します。

GLSL 側の変数名がたとえば `attribute vec3 position;` と定義されているとき、この変数は「"position" という名前の "attribute" 変数である」と表現します。

"attribute" の部分は修飾子と呼ばれていて、その変数がどういう役割を持っているのかを表しています。その詳細な意味は後述するとして、ここでは、この "position" という名前の変数にデータを送る手続きを例に考えてみましょう。

GLSL の世界に定義された `attribute vec3 position;` にデータを送るために、まずこの変数の ロケーション を取得します。

ロケーション (location) は文字通り変数がどこにいるのかを指示する目印のように機能し 送りたいデータとロケーションをセットで処理することで、JavaScript で定義したデータを GLSL がいる GPU 側に送信することができます。

繰り返しになりますが実際にデータを送信する処理は `ShaderProgram` クラスがよしなにやってくれるように作っています

ロケーションの取得には、前述のプログラムオブジェクトが活躍します。

プログラムオブジェクトに正しくシェーダがアタッチされている状態で、プログラムオブジェクトに対して「この名前の変数のロケーションはどうなってるの？」というふうに尋ねてやれば、該当する変数のロケーション情報を得ることができます。

“プログラムオブジェクトは CPU 側と GPU 側の通訳みたいな感じですね”

記述例としては、以下のような感じ。

```
const ロケーション = gl.getAttributeLocation(プログラムオブジェクト, 'position');
```

取得したロケーションの情報は、後々、実際にそこにデータを送信する際に必要になります。

繰り返しになりますが、お配りしているサンプルの実装では `ShaderProgram` クラスがロケーションの取得などの面倒な手続きはすべて行ってくれます。インスタンスを生成する際に、引数から必要な情報を指定してやります。

```
this.shaderProgram = new ShaderProgram(this.g1, {  
    (中略)  
    attribute: [  
        'position', // ← attribute 変数の名称  
    ],  
    stride: [  
        3, // ← vec3 型の変数なので、要素数3を指定  
    ],  
});
```

# ここまで的小まとめ

- シェーダが動作するのは GPU の世界
- JavaScript が動作するのは CPU の世界
- CPU から GPU の世界に干渉するには.....
- プログラムオブジェクトを使って、データ送信時の目印となる「ロケーション」をまず取得する
- データを送信する処理を行う際に、ロケーションが必要になる
- サンプルでは、このあたりはまるっと `ShaderProgram` クラスがやってくれる

暗記の必要はないので、雰囲気をつかみましょう！

# **GLSL の変数**

座学が長くて覚えないといけないことも多いのでちょっと大変ですが、実際にシェーダで利用することになる GLSL の変数と修飾子 についても、確認しておきましょう。

GLSL でも、JavaScript などと同様に基本的には開発者が自由に変数を定義できます。ただし、GLSL に独特な変数宣言のルール、というのがあるので、ここからはそれを見ていきます。

GLSL では変数を宣言する際に、代表的なところでは以下のようないものを指定します。

- 変数のデータ型
- 変数の名前
- 変数の種類を表す修飾子（無い場合もある）

JavaScript とは違い型定義はかなり厳密です

# 変数のデータ型

- int: 整数
- float: 浮動小数点
- bool: 真偽値
- vec系: ベクトル (vec2 ~ vec4 があり中身は float)
- bvec系: 真偽値ベクトル (bvec2 ~ bvec4 があり中身は bool)
- ivec系: 整数ベクトル (ivec2 ~ ivec4 があり中身は int)
- mat系: 行列 (mat2 ~ mat4 があり中身は float)
- sampler系: サンプラー (始めはテクスチャと読み替えても良い)

このあたりは利用する際にその都度説明しますので、暗記しなくても大丈夫です（整数と浮動小数点数との違いくらいは意識しておくとなにかとトラブルが減らせます）

# 変数の種類を表す修飾子

GLSL では変数の型とは別に、その変数の持つ意味合いを表現するための「修飾子」が用意されています。

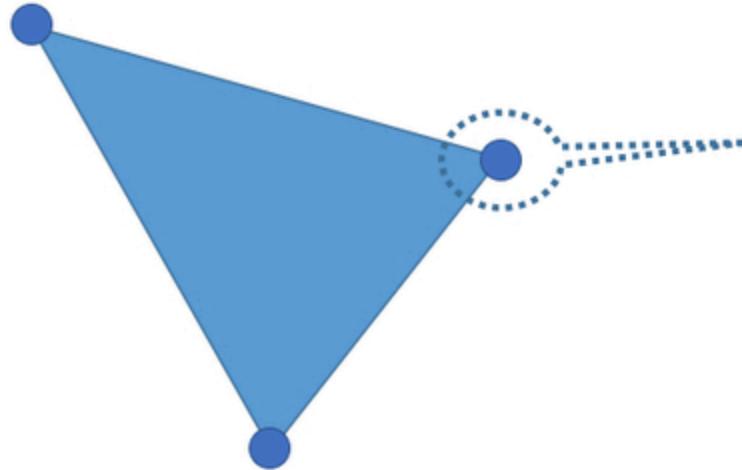
実は、この部分がシェーダを理解する上で 非常に重要 です。最初は大変だと思いますが、本当にこれがわからないとなにも始まらないというくらい大切なことで、ここでしっかりと把握しておきましょう。

# attribute

先程もチラッと登場した attribute 修飾子は、アプリケーション（JavaScript）から送られてきた 頂点の情報 が格納される変数に用いられます。

頂点シェーダは、頂点ひとつひとつに対してそれぞれ実行されます。頂点が 個別に持っている座標や色などの情報 は、頂点シェーダ内では attribute 修飾子付きの変数として定義します。

```
// 記述例
attribute vec3 position; // この頂点は vec3 型の position という名前の属性を持つ
```



頂点の位置は？（xyz など）

頂点の色は？（RGBA など）

といった頂点ひとつひとつに固有の情報については、attribute 変数で定義する。このような頂点に固有の情報のことを一般に「頂点属性」と呼ぶ。

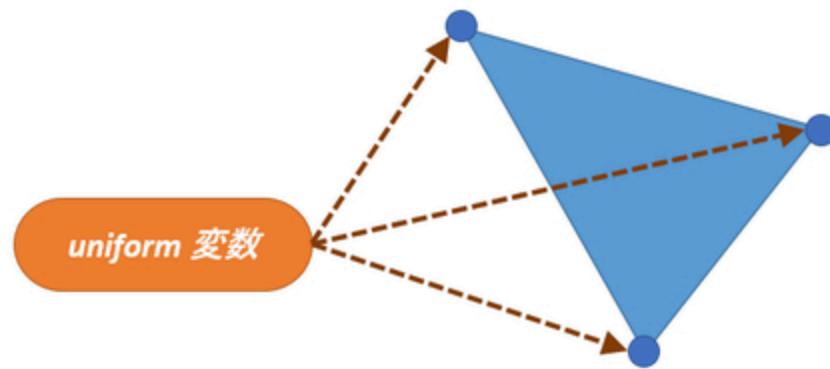
attribute 変数では、それぞれの頂点がどのような固有の情報を持つのかが表現できます。

# uniform

uniform 修飾子は、アプリケーション（CPU 上の JavaScript）から渡されるグローバル変数です。

attribute 修飾子付きの変数が個々の頂点が持つ固有の情報を扱うものだったのに対し、uniform 変数は すべての頂点に対して一律となるグローバル変数 のように振る舞います。

```
// 記述例  
uniform mat4 modelMatrix; // modelMatrix という名前の行列用グローバル変数
```



uniform 変数は、一種のグローバル変数であり、シェーダ内ではいつどこから参照しても、基本的に同じ内容となる。  
頂点シェーダ内でも、やはりすべての頂点が一律に uniform 変数の内容を参照・利用することができる。

uniform 変数で全体に影響するパラメータなどを制御できる。

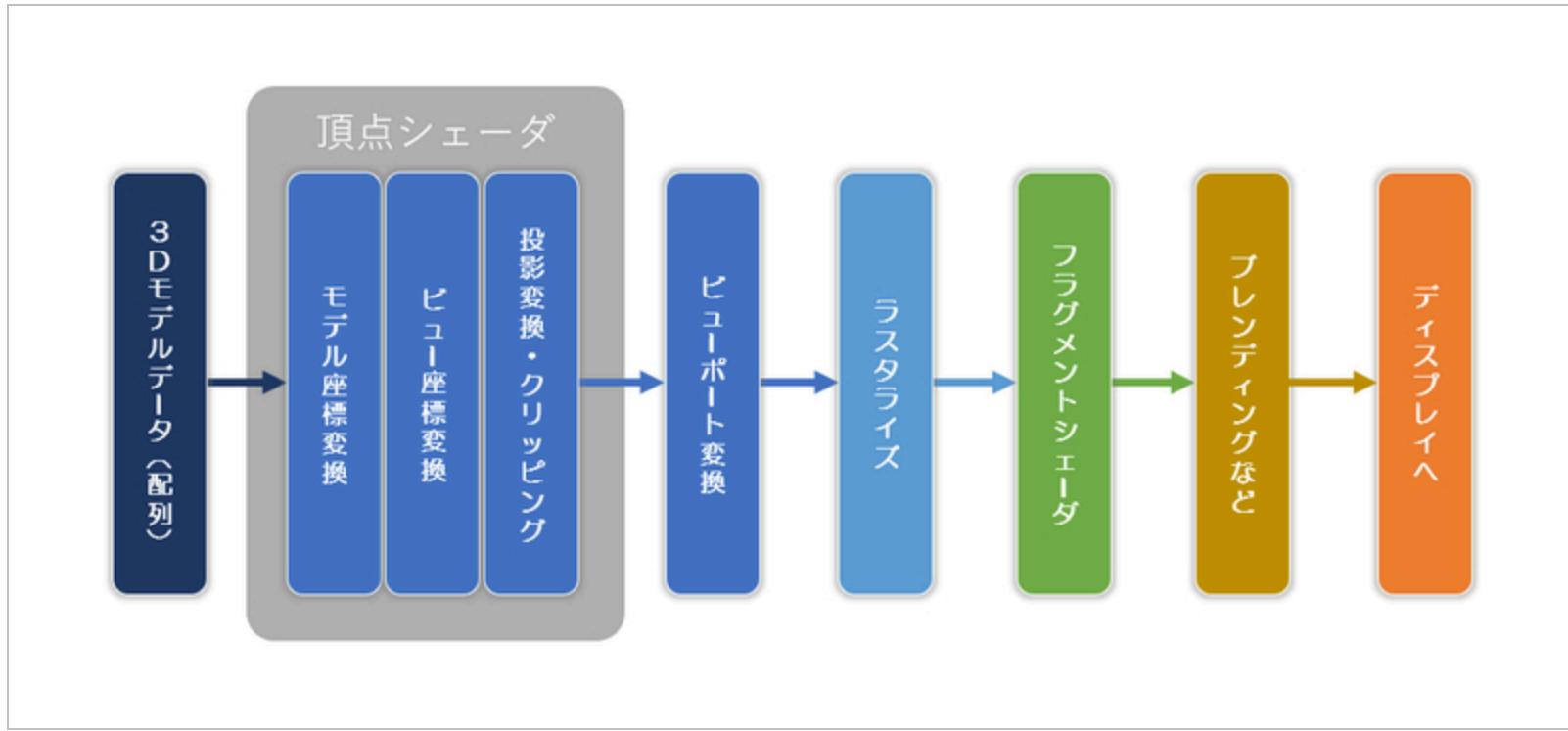
# varying

`varying` 修飾子付きの変数について考える際は、グラフィックスパイプラインの構造を思い出して考えてみましょう。

グラフィックスパイプラインでは、必ず頂点シェーダが先にありました。その後、フラグメントシェーダへと値が受け渡され、ピクセル単位での処理が行われます。このとき「値の受け渡し」に使われるのが `varying` 変数です。

// 記述例

```
varying vec4 vColor; // vColor という名前の色を受け渡すための変数
```



頂点シェーダからフラグメントシェーダへ、パイプライン上でデータを受け渡すのに `varying` 変数が使われる。

# GLSL のストレージ修飾子まとめ

- attribute は、頂点それが固有に持つ情報を格納
- uniform はアプリケーションから送られてくるグローバルな情報
- varying は頂点シェーダからフラグメントシェーダへの橋渡しをする

それぞれの役割は非常に重要なのでしっかり覚えておきましょう

# いよいよ仕上げ！

## JavaScript 側の一連の流れを見ていく

※各処理の詳細はサンプル内のコメントも参考に。ここではまず、ざっくりと流れを把握してください

# 頂点の定義と VBO

JavaScript 側の `WebGLApp.setupGeometry` メソッド内に、頂点を定義している箇所があります。

頂点を定義する処理自体はシンプルに配列を利用して定義すればよく、  
GLSL 側で `vec3` で宣言された `attribute` 変数に渡すデータ なら、  
JavaScript 側でも `xyz` でデータがワンセットになるように要素数を調整する 必要があります。

GLSL 側で `vec2` なら、2 個ワンセット、`float` なら単体の値に

```
this.position = [  
    0.0, 0.0, 0.0, // 1つ目の頂点の XYZ  
    -0.5, 0.5, 0.0, // 2つ目の頂点の XYZ  
    0.5, 0.5, 0.0, // 3つ目の頂点の XYZ  
    -0.5, -0.5, 0.0, // 4つ目の頂点の XYZ  
    0.5, -0.5, 0.0, // 5つ目の頂点の XYZ  
];
```

GLSL 側で `vec3` で定義されている頂点属性（attribute 変数）なら、  
JavaScript 側の定義も 3 つワンセット（配列の長さが 3 の倍数）になる  
ように。

なお、頂点の定義自体は単に JavaScript の配列を用いれば問題ありませんが、最終的に GPU にこれを送る際には専用のバッファにデータを詰め込む必要があります。

この、頂点の情報を詰め込んだバッファのことを VBO (Vetex Buffer Object) と呼びます。

以下のコードが VBO にデータを格納している部分です。生成した VBO は描画を行う直前に `ShaderProgram` に渡すことになるので、  
`this.vbo` に一度代入しておきます。（詳しくは後述）

```
// 定義した頂点の情報（頂点属性）は VBO に変換しておく  
// ※ WebGLApp.setAttribute で処理するために配列に入れています  
this.vbo = [  
    WebGLUtility.createVbo(this.gl, this.position),  
];
```

# canvas 要素の大きさ

DOM としての canvas 要素は、このサンプルでは「ウィンドウサイズと同じ」になるようにしています。

```
resize() {
  // ウィンドウサイズぴったりに canvas のサイズを修正する
  this.canvas.width = window.innerWidth;
  this.canvas.height = window.innerHeight;
}
```

# 色の指定は 0.0 ~ 1.0 で指定

WebGL や GLSL では RGBA は全て 0.0 ~ 1.0 の範囲で表します。

CSS や HTML では 16 進数を使って `#FFFFFF` のように表しますが、  
 GLSL では `vec4(1.0, 1.0, 1.0, 1.0)` が完全に不透明な白に相当します。背景をクリアする色の指定を行っている `gl.clearColor` などでも、同様の値の範囲で色を指定します。

```
// 背景を何色でクリアするかを 0.0 ~ 1.0 の RGBA で指定する
this.gl.clearColor(0.1, 0.1, 0.1, 1.0);
```

# **viewport** は、描画される領域のこと

WebGL や OpenGL では、描画する領域のことをビューポートと呼びます。

ウィンドウサイズや、WebGL 等のレンダリング対象になっている canvas の大きさが変化するときには、普通はビューポートにもそれを反映するための再設定が必要になります。ビューポートを小さく設定して描画してみるとどんなことが起こるか、実際に試してみるとビューポートがどういう意味を持ったものなのかわかりやすいかもしれません。

サンプル 001 でのビューポートの定義は `WebGLApp.render` メソッド内で行っています。

```
// WebGL 上のビューポートも canvas の大きさに揃える  
gl.viewport(0, 0, this.canvas.width, this.canvas.height);
```

# ShaderProgram 経由で頂点データを送る

サンプルでは `ShaderProgram` クラスに VBO を渡すことで、頂点の情報  
をシェーダ側に送ることができるように実装してあります。

より具体的には `ShaderProgram.setAttribute` に VBO を渡せば OK で  
す。 (ここで先程 `this.vbo` に代入しておいた VBO を渡している)

```
// どのプログラムオブジェクトを使うのかを明示する
this.shaderProgram.use();
// attribute 変数に VBO を設定する
this.shaderProgram.setAttribute(this.vbo);
```

# 描画命令を発行する

WebGL や OpenGL では、描画を実際に行うのは GPU の仕事です。ですから「描画を行うために必要なデータ」は、事前に GPU に全て渡しておいた上で……

最終的に描画命令（`gl.drawArrays` など）が発行されたとき、その時点で GPU に転送済みだったデータ が描画に使われます。

データを送ってから描画命令（ここでは `gl.drawArrays` が相当）を発行すると、その時点で「GPU に送られている情報」を利用してレンダリングが行われます。

```
// 設定済みの情報を使って、頂点を画面にレンダリングする  
gl.drawArrays(gl.POINTS, 0, this.position.length / 3);
```

# ここまで的小まとめ

- 頂点を定義して VBO を作っておく
- canvas 要素や viewport の大きさを指定する
- 色は `#fffffff` のような形式ではなく 0.0 ~ 1.0 の範囲で指定する必要がある
- `ShaderProgram` を有効化し VBO を渡すと GPU に送ってくれる
- 描画命令（`gl.drawArrays` など）を呼び出すと、その時点で GPU 上に転送済みのデータを使って描画が行われる

**CPU 側の処理の次は GPU 側 !**

**GLSL で何をやっているのかを見ていく**

# `gl_` の接頭語が付いているものはビルトイン

ビルトイン変数とは定義や宣言を行わなくても、はじめから利用できる変数のことです。

`gl_Position` であれば、これは頂点座標を格納するためのものですし、`gl_FragColor` であれば、これは最終的に画面に出力される色の情報になります。

ビルトイン変数はいろいろありますが、ひとまずここでは最低限の知識として3つ紹介します

## **gl\_Position (頂点シェーダ限定)**

グラフィックスパイプラインを説明した場面で、頂点シェーダから次のステージへと処理が移り変わっていく様子を図解して見せました。

つまり「頂点シェーダからはなにかしらの情報を出力する必要がある」ということになります。

ここで言う、出力すべきデータの受け取り口が `gl_Position` です。

`gl_Position` に出力する（代入する）情報は `vec4` のデータ型で、ざつくり言えば、頂点の 座標の情報 です。

# gl\_PointSize（頂点シェーダ限定）

`gl_PointSize` は頂点シェーダから確実に出力しなければいけない変数ではありませんが、頂点を「点として描画する」際にその大きさを指定するために使います。

原則として、点として描画する際は必ず指定します。

“ 指定しない場合の挙動は仕様上未定義で、ハードウェア依存であるため、点として描画する場合は指定することが好ましい ”

## **gl\_FragColor** (フラグメントシェーダ限定)

フラグメントシェーダからは、色に関する情報を出力します。

`gl_FragColor` は文字通り、RGBA を意味する `vec4` のデータでここに  
出力した色がそのままピクセルの色として扱われます。

**サンプルを見ながら理解を深める**

これから他のサンプルも一気に解説していきますが、最初はわからなくても落ち込む必要はありません。

ちょっとずつ、自分のペースで理解していけば問題ありませんし、そのために Discord などを用意して質問も受け付けられるようにしているので、とにかく気負いしすぎないことが大切です。

“楽しむことが一番大事！”

またサンプル内には多くのコメントが記載してあります。

スライドにはある程度の概要や、図解したほうがわかりやすい概念の説明などが多いです。実際のコードの記述方法や細かな意味は、むしろサンプルのコメントを読んでもらうほうがわかりやすいと思います。

## サンプル 002

- 頂点属性 (attribute 変数) に頂点カラーを追加
- 頂点属性の追加は手順が多いので落ち着いて 1 つずつ確実に
- 頂点カラーはまず頂点シェーダが先に受け取る
- 逆に attribute を直接フラグメントシェーダに渡す方法はない
- どうにかして、頂点の色をフラグメントシェーダへ渡す必要が生じる
- そこで varying 変数を使って、色の情報を渡す

“ attribute 変数や varying 変数の違いをしっかり意識しよう ”

# サンプル 003

- 一種のグローバル変数である `uniform` 変数を追加
- `attribute` 変数とは手続きが異なるので注意
- サンプルの実装では `ShaderProgram` 経由で直接データを送れる
- `uniform` 変数の `type` の指定は残念ながら暗記するしかないが、一定のルールはあるので慣れれば比較的簡単

```
GLSL で float => uniform1f  
GLSL で vec2  => uniform2fv  
GLSL で vec3  => uniform3fv  
GLSL で mat4  => uniformMatrix4fv
```

## サンプル 004

- カーソルに連動して動くサンプルを作ってみる
- アプリケーション（CPU）側でマウスカーソルの動きを捕捉し.....
- カーソルの動きを検出したらそれを正規化した上でシェーダに送る
- ただしマウスの動きの影響は頂点ごとではなく、すべての頂点が一律に影響を受けるようにしたいので attribute ではなく uniform を利用する（この考え方には早く慣れておくと幸せになれる）

“attribute にすべきなのか uniform にすべきなのか、判断できるようにしておくことが大切”

## サンプル 005

- GLSL のソースコードにコメントが書いてあります
- 書いていくうちに自然と慣れるので、無理してすべてを暗記はしなくても大丈夫
- ただしエラーに悩まされることのないよう、ある程度は目を通しておくのがよい
- 公式のチートシートなども必要に応じて活用しよう
- チートシート：[webgl-reference-card](#)
- より詳しい仕様：[The OpenGL ES Shading Language](#)

上記は WebGL 1.0 + GLSL ES 1.0 の組み合わせの場合です

次回へ向けて

さて、初回からなかなか覚えることが多くハードな内容でしたね……本当に、お疲れさまでした。

シェーダという概念は、まずその存在としての意味の理解、そしてその構文などの知識の理解と、一度に覚えなければならないことが非常に多いです。どのようなプログラミングの世界でも同じだと思いますが、最初は、これらのことを地道に少しずつ理解していくしかありません。

大切なことは どんどん動かしてみること だと思います。

自分で修正した結果が、どのような形で描画結果へ反映していくのかをよく観察し、その結果を見ながらさらに考察を重ねていく。これが大切です。

なかなか自発的に課題を見つけてこなしていくのは大変だと思うので、講義の最後に次回に向けての宿題を出すようにします。

課題の提出は必須にはしません。ただ、課題に挑戦することが上達や理解向上への近道であることは間違いありませんので、できればトライしてみてください。

“ 完成した課題は Discord の report チャンネルでどんどんシェアしよう ! ”

今回は、まず頂点の個数を単純にどんどん増やしてみましょう。

今はまだ奥行きのある状態を扱っていないので、ひとまず XY 平面上に大量に配置してみましょう。（数千～数万程度）規則正しく並べてみてもいいですし、ランダムに配置してみても面白いかもしれません。マウスとのインタラクションなども実装できたらなお良いでしょう。

小さな疑問でも遠慮せずに Discord 等でどんどん聞いてください～！