

頂点シェーダ・テクスチャ

3D の表現と 2D の表現

前回のおさらい

前回は GLSL の基本となる項目をざっと全体的に押さえたような内容でした。

その分、ちょっと覚えなくてはならないことが多かったので大変だったかもしれません。

少しずつで構わないので、自分のペースで取り組んでいきましょう！

今回扱う最初のテーマは「頂点シェーダ」になります。

前回の講義でも触れたとおり、シェーダには様々な種類がありますが頂点シェーダはおよそどのようなプラットフォームにも備わっている基本的なシェーダのひとつです。この頂点シェーダをまずしっかりと理解してしまうことが、シェーダを自由に扱えるようになることの第一歩となります。

頂点とはなんなのか

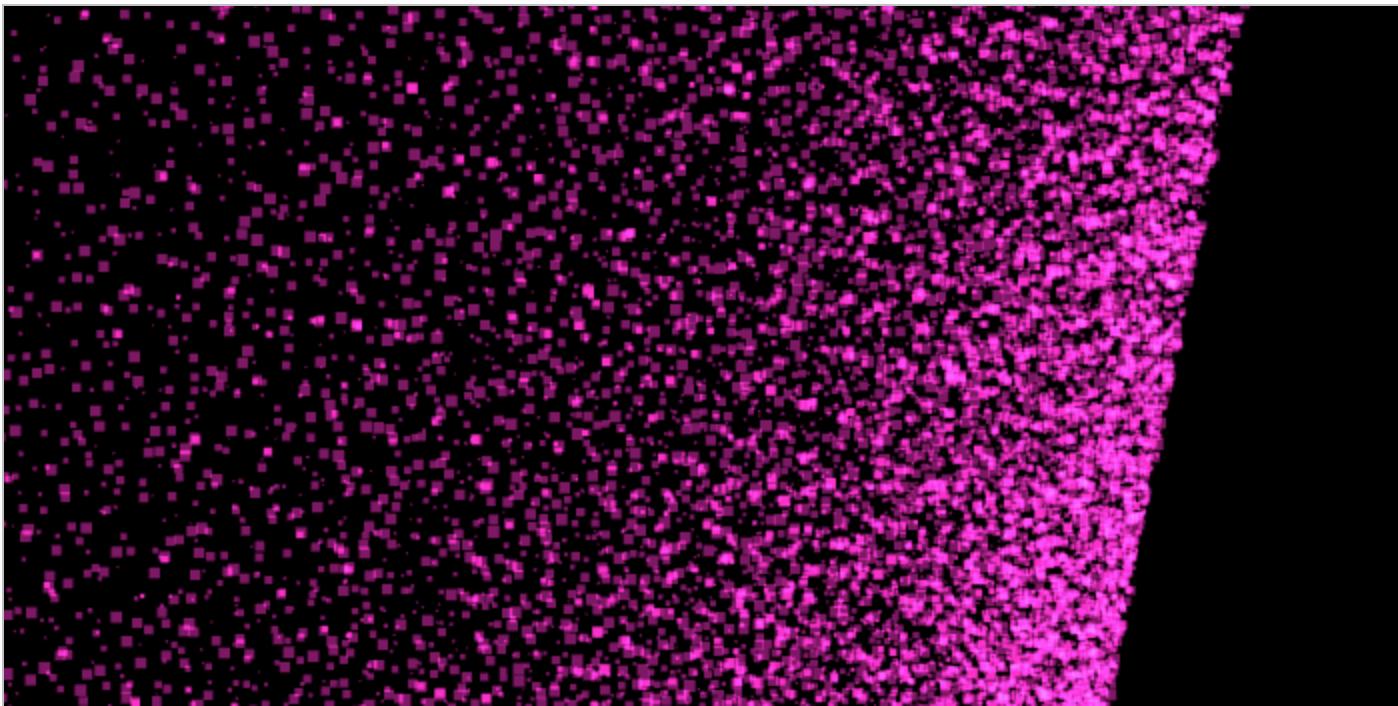
最初はまず 頂点とはなにか ということから考えてみたいと思います。

3DCG では頂点を使ってあらゆる形状を定義しますが、そもそも頂点ってなんなのでしょうか。

頂点は、簡単に言ってしまえば「三次元空間上有る一点」を表す単位のようなものです。

その任意の点は、自分自身が空間のどの位置に存在するのかを表す 頂点座標 を大抵の場合持っていて、これらが様々に組み合わさることで平面はもちろん、立体的な形状をも表現することができます。

点の集合が全てを表現する



たくさんの点の組み合わせで様々なものが表現できる



一見複雑な形状も細部まで見ていけば頂点の集合体

そして、そんな頂点を制御するために存在しているのが今回のテーマでもある「頂点シェーダ」です。

頂点シェーダは頂点が持つ座標の情報をなにかしらのロジックで変換することが主な役割で、多くの場合、ここでは 行列を用いた座標変換 行われます。

特に、3D の立体的な描画を行う場合、行列は欠かせない存在になります

前回のサンプルでは行列は出てきませんでした。つまり「頂点シェーダで行列処理が必須というわけではない」ということをまずは念頭に置きましょう。

しかしそれならばなぜ、一般的な 3D プログラミングでは行列を用いる場合が多いのでしょうか。もったいぶっても仕方ないので答えを書いてしまうと、行列を使うことで 頂点の座標変換が簡単かつ効率的に行えるからです。

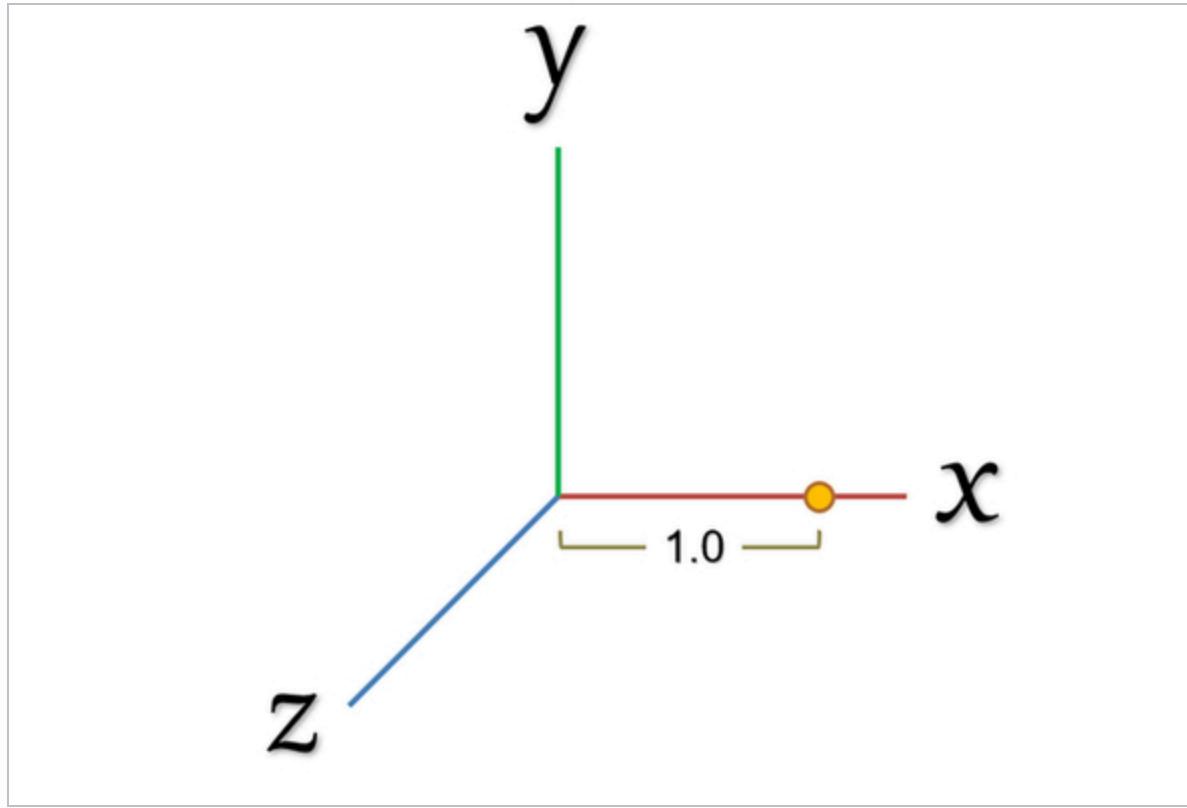
頂点の座標変換について考えてみる

ここでは、最終的に行列のイメージをしっかりとつかむためにも、まずはあえて行列という言葉の意味はいったん隅のほうに置いておき、非常にアナログな感じで 3DCG が描かれるプロセスを追いかけてみることにしましょう。

ここはひとつ、頭を空っぽにして、ちょっとバカバカしく感じるかもしれませんが どうしたら 3DCG を描くことができるか をちょっと真面目に想像してみましょう。

まず三次元空間に頂点がとりあえずひとつ、置かれているとしましょう。これが全てのスタート地点です。

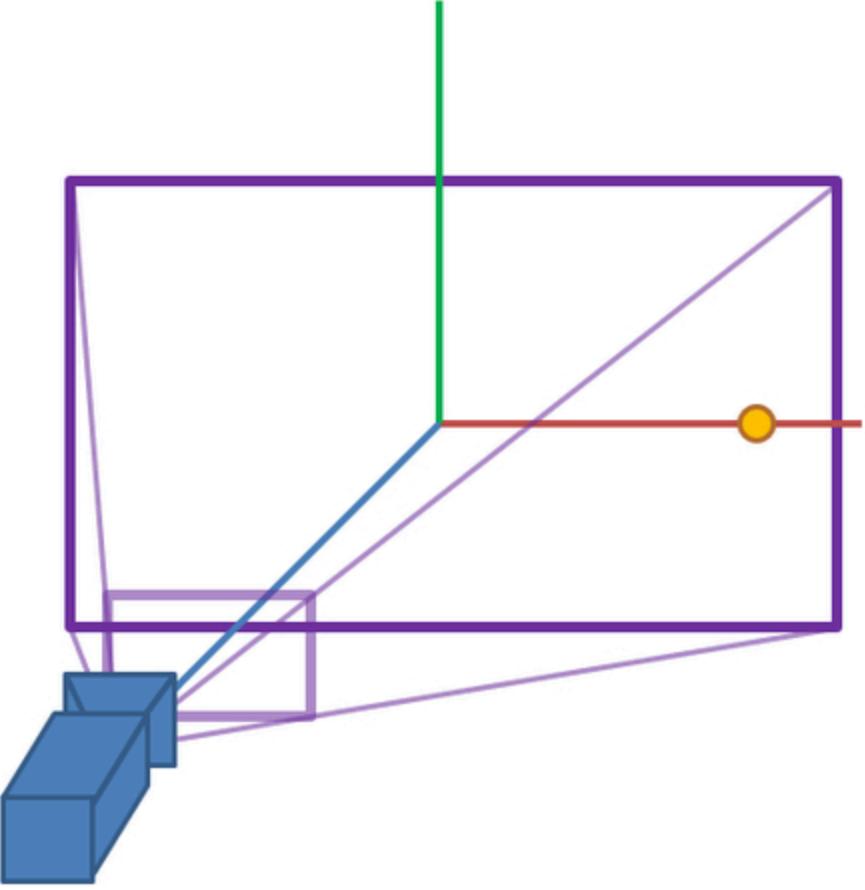
例えば、その頂点の XYZ 座標がここでは **1.0, 0.0, 0.0** だったと仮定します。これを図解すると次のようになります。



このひとつの点（頂点）をなんとかして画面に出すのが最終目的だとする

さて、これを画面に出すためには、まずこの三次元空間を「どこから見ているのか」であったり、「どのくらいの広さを見渡せるのか」であったりを決める必要があります。

一般に、これらのパラメータ類は（人間が頭のなかで直感的にイメージしやすくするために）ひとまとめにして「カメラ」の概念として定義する場合が多いでしょう。カメラを定義することによって、無限に広がる三次元空間を、スクリーンという有限の平面に切り出すことがなんとかイメージしやすくなるわけです。



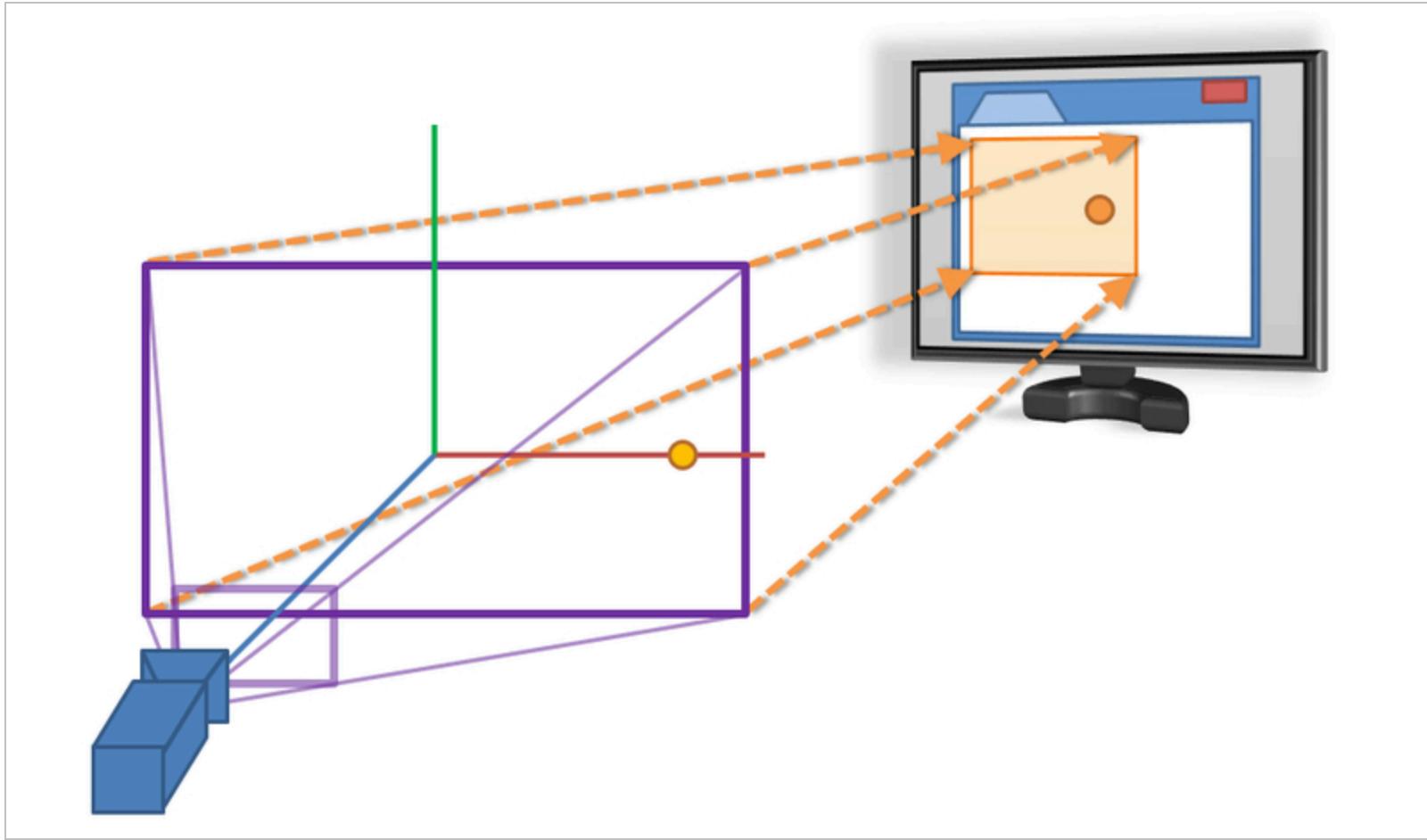
“ カメラを置いて撮影する範囲を定義 ”

この図を見ると、もしもカメラの画角がもっと広かったり、あるいはカメラそのものの位置が前後（あるいは上下左右）に動いたりすると最終的に撮影されるエリアも変化することがわかると思います。

しかも、これだけの情報では三次元空間を撮影するカメラが定義されただけで、スクリーンに描画を行うためには実はまだ情報が不足しているのですが……それがなんだかわかるでしょうか。

答えは、この撮影結果が 映し出されるスクリーン についての情報が足りない、ということです。

スクリーンは縦長や横長かもしれませんし、めちゃくちゃ大きいかもしれませんし逆に小さいかもしれない。もっと言うと、撮影している空間は三次元の立体的な空間ですが、スクリーンは二次元平面ですよね……これも、なんとか変換して次元を減らさないといけません。



“スクリーンにはめこむための変換もしないといけない

実際には、もう少し細かく見ていくといろいろな変換ステップというものがあるのですが、今の流れを見ただけでも、三次元空間にある頂点を二次元のスクリーンに映すのって結構たいへんそうだなあというのがわかるのではないかでしょうか。

これを全部自前で計算してね？って言われたら……点がたったひとつならまだしも、複雑なポリゴンの組み合わせだったりしたら、かなりきついなあと感じるのではないかでしょうか。

WebGL や OpenGL を使っているからといって、なにか得体の知れない魔力の力で変換作業が勝手に行われ、3DCG が出来上がってくるわけではありません。

ここで見てきたような様々な変換は、わたしたちプログラマー自身の手で、愚直にひたすら計算して割り出していく という手順を踏む必要があります。そして初めて CG としてユーザーの目に触れる最終的なグラフィックスが現れるのです。

“自動でやってくれるのではなく、自分でやらないといけない！！”

つまり極端な話、頂点シェーダで以下のようなかんじのことをしないといけないわけです。

```
attribute vec3 position; // 頂点座標
void main(){
    // カメラの位置や、スクリーンの大きさなどを考慮し、頂点が
    // 正規化デバイス座標空間のどこに存在すればよいかを計算する

    vec3 transformed = /* 🔥なにか壮大な計算🔥 */;

    // 変換した座標を頂点シェーダから出力する
    gl_Position = vec4(transformed, 1.0);
}
```

“ だいぶ難しそう…… ”

救世主現る

さて、こんないかにも大変そうな座標変換ですが、とある概念を導入することで非常に簡単にを行うことができるようになります。

そのとある概念こそが、行列 なんです。

行列を使うと、さきほど例に示したような煩雑な座標変換を非常にシンプルに表現できるようになります。

そのひとつの例として、最初のサンプルの頂点シェーダを見てみてください。

ここでは、`mvpMatrix` という名前の `uniform` 変数が出てきていますが、そのデータ型は `mat4` となっていますよね。これこそが、スクリーンに頂点を映し出すための、座標変換の大半を行ってくれる行列です。

先ほどのプロセスの全てではないですが大半はこいつがやります

```
attribute vec3 position;
attribute vec4 color;

uniform float time; // 経過時間
uniform mat4 mvpMatrix; // MVP 行列 @@@

varying vec4 vColor;

void main() {
    (中略)
    // MVP 行列を乗算（列優先）してから出力する
    gl_Position = mvpMatrix * vec4(p, 1.0);
}
```

"@ @" が書いてある場所で定義されているのが mvpMatrix

最初のサンプルの場合、`mvpMatrix` という名前の変数（行列）は頂点シェーダの中で `position` 由来の `p` という変数と乗算されています。

```
gl_Position = mvpMatrix * vec4(p, 1.0);
```

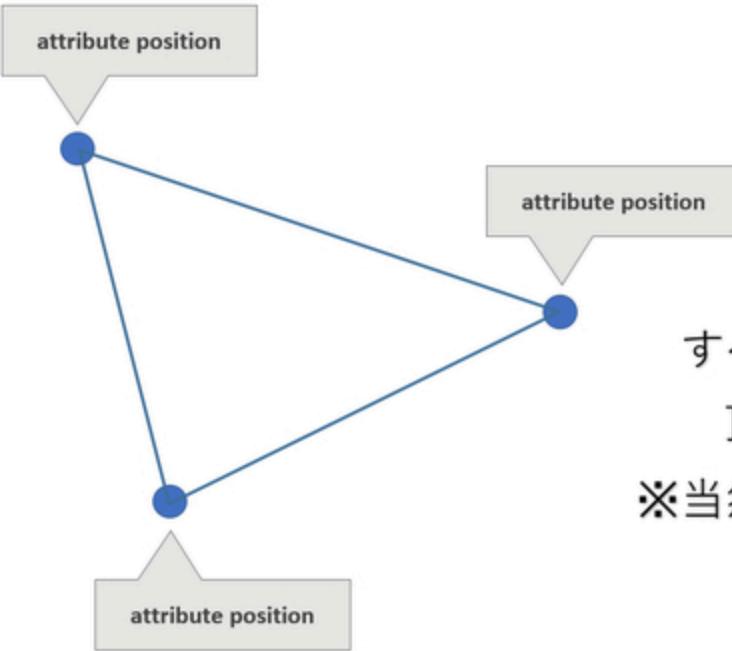
“GLSL では、行列やベクトルは四則演算を演算子を使って記述できます。ただし、掛ける順序に注意！”

さあ、ここから前回の内容を少し思い出しながら考えましょう。

先程の頂点シェーダでは、attribute 修飾子付きの変数 `position` に由来する変数と行列を乗算しています。では、頂点シェーダ側で attribute 修飾子付きで宣言されている変数の役割はなんだったでしょうか。

`attribute` 変数は それぞれの頂点が個別に持つユニークな情報 を参照するためのもの、でした。

つまり、頂点が 1 つずつ頂点シェーダによって処理されるたびに、`position` という名前の変数の中身は毎回変わる（それぞれの頂点が持っている値に置き換わる）可能性があるということです。

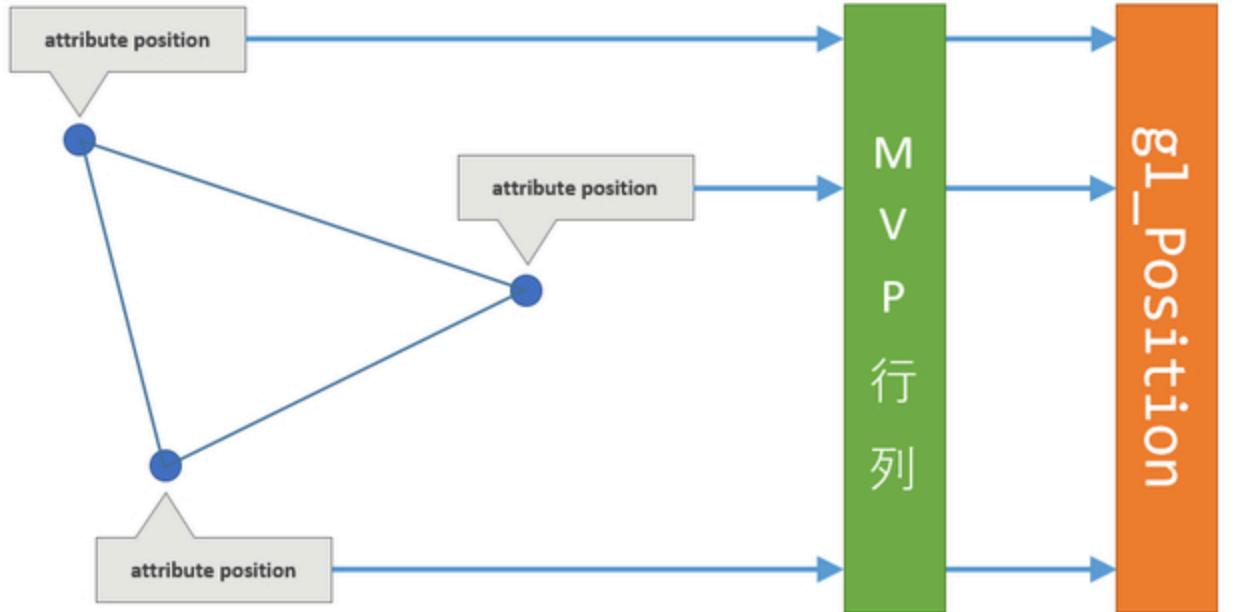


すべての頂点が共通して持っている、
頂点固有の属性値が **attribute** 変数
※当然中身もそれぞれ異なる可能性がある

先ほど記載したサンプルの例のように、行列に関する処理が頂点シェーダに書かれている場合は「頂点ひとつひとつが個別」に、「行列によって座標変換される」ということになります。

つまり、定義されている全ての頂点（描こうとしている全ての頂点）は、漏れなく `mvpMatrix` との演算が行われた上で、`gl_Position` に代入されることになるのです。

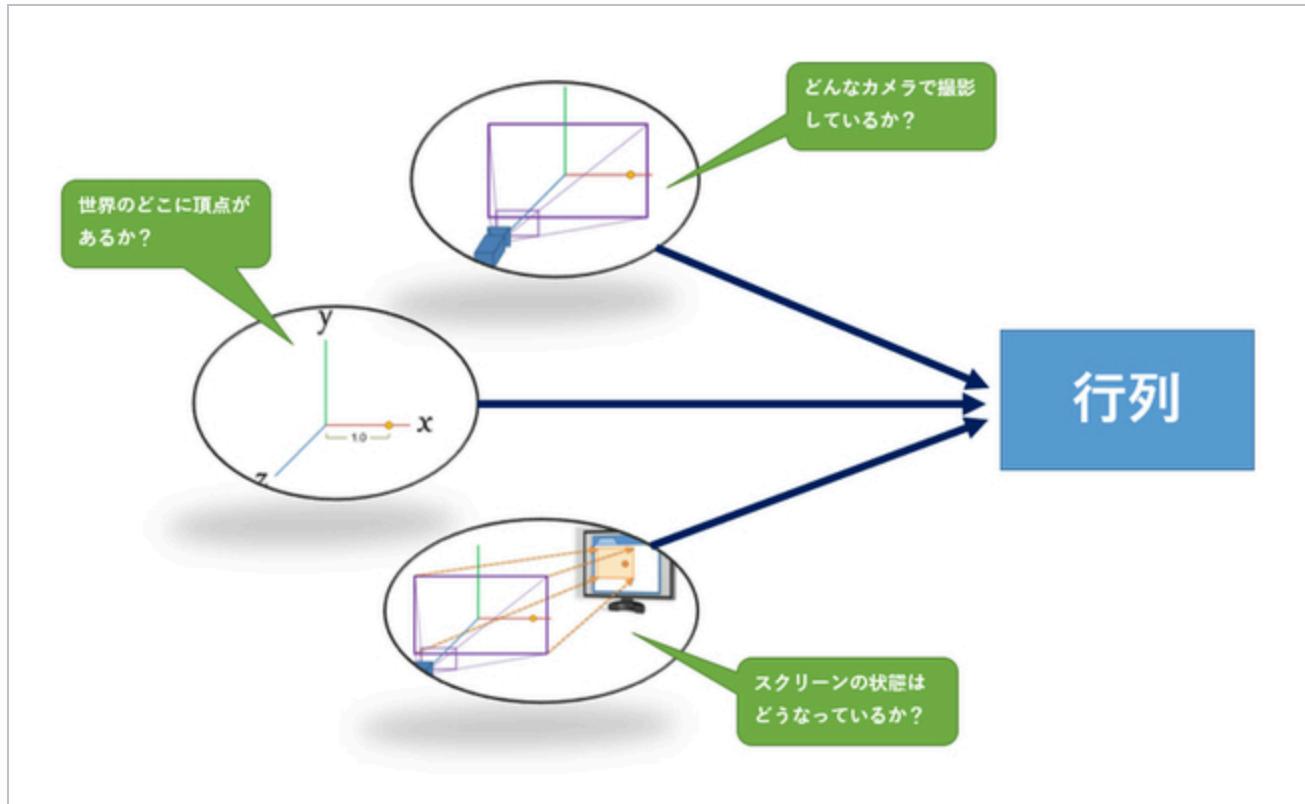
“いま描こうとしている全ての頂点が同じ行列によって変換される！”



頂点シェーダは、各頂点それぞれに対して一律に作用するので、すべての頂点は必ず同じ MVP 行列と乗算されてから gl_Position へと出力されることになる

行列という概念を導入すれば、先ほど図解して説明したような、非常に複雑で膨大な計算（スクリーンに頂点を描くためのいくつかの変換処理）をたったひとつの行列を使うだけで行うことができます。

これこそが、3DCG の世界で行列が重宝される理由だったんですね。にわかには信じがたいまるで手品のような話ですが、実際これは本当のことです。



様々な変換がひとつの行列に集約でき、一度の乗算ですべてを反映できる

そして、ここまで説明してきたことは、実はそのまま 頂点シェーダの主な役割 も同時に表しています。頂点シェーダは、頂点の座標などを attribute 変数として受け取り、それを行列などを用いて変換するのが役目である、と考えればいいわけです。

ときには数万あるいは数十万という膨大な量の頂点が、漏れなく頂点シェーダによって変換されるですから、やっぱりこれは JavaScript などで地道に計算するのは現実的ではなく、高速な GPU を利用したシェーダだからこそ実現できることだと言えるでしょう。

繰り返しになりますが、前回のサンプルを見てもわかるように、ここで例に出したような行列の処理は仕様上は必須というわけではありません。（GLSL 側で行列を使わなくてはいけない、というわけではない）

ただし立体的に見えるようなビジュアルを実現するためには様々な変換が必要であり、それを簡潔に行うための概念として行列を用いる場合が多い、ということは覚えておきましょう。

ここまでまとめ

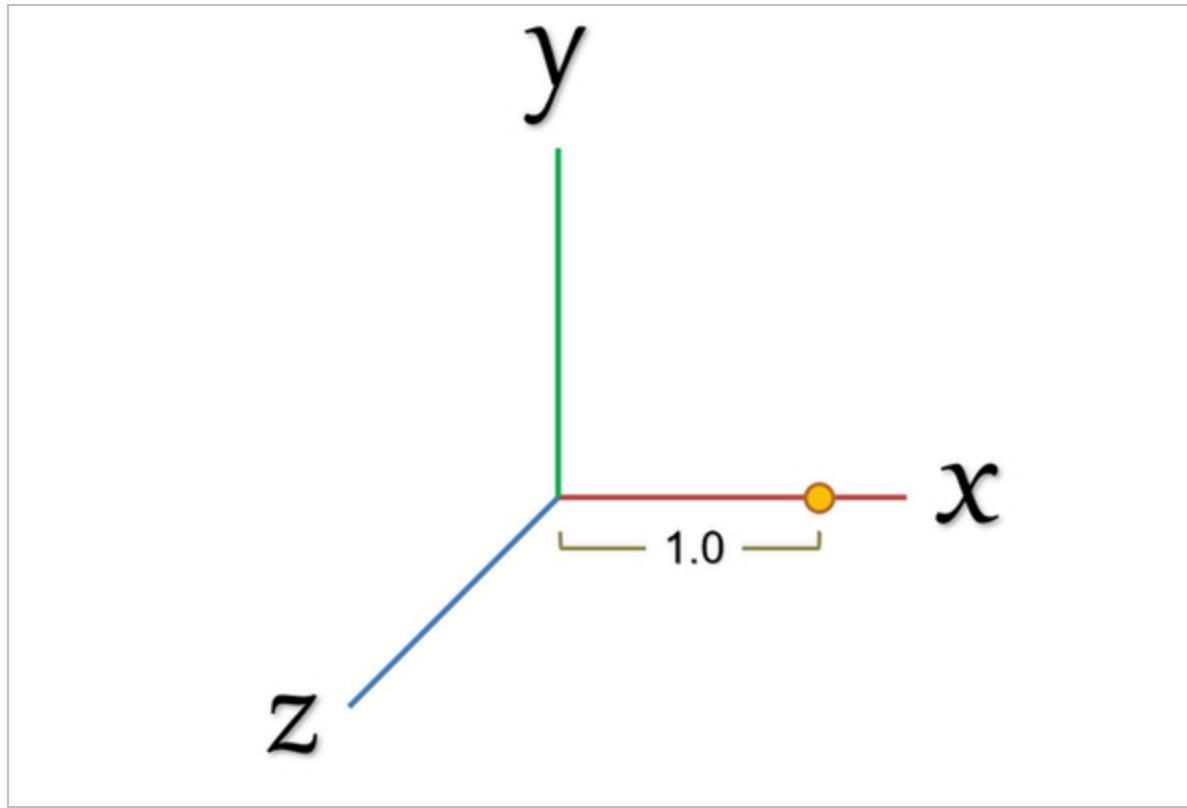
- 頂点シェーダは頂点の座標変換が主な役割
- 座標変換と一口に言ってもカメラやスクリーンの大きさなどが影響する様々なステップがあり複雑で冗長
- それを効率よく行うために都合のよい概念が行列
- 頂点シェーダからは必ず `gl_Position` になんらかの値を出力する必要があり.....
- 一般に MVP 行列と乗算した座標を出力することが多い

“ 頂点シェーダの役割をしっかり認識しておこう ! ”

補足（座標系の話）

さて、ここから最初のサンプルを見ながら行列を用いた処理内容を見ていくのですが、その前に WebGL や GLSL における「三次元空間の表現方法」について少し補足しておきます。

途中で出てきた、三次元空間の XYZ の各軸を表している図がありましたよね。あの図をよく見ると、WebGL や GLSL の三次元空間の「各軸の正方向（プラスの向き）がどちらか」がわかります。



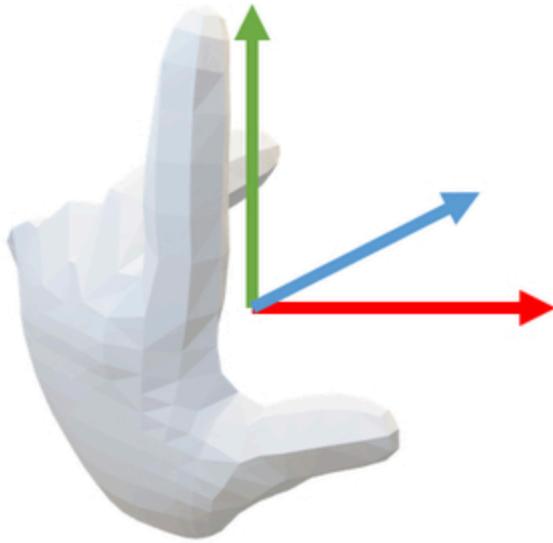
上の図では XYZ の各軸のプラス方向（正の方向）にラインが伸びている

これを見ると、 X と Y は、いわゆる学校とかで教わる二次元の平面図形や、グラフなどの図形と同じプラス方向の向きになっていると思います。要は、右に行くほど X の値は大きくなり、逆に左に行くほど X の値は小さくなる。 Y であれば、上がプラスで下がマイナスですよね。

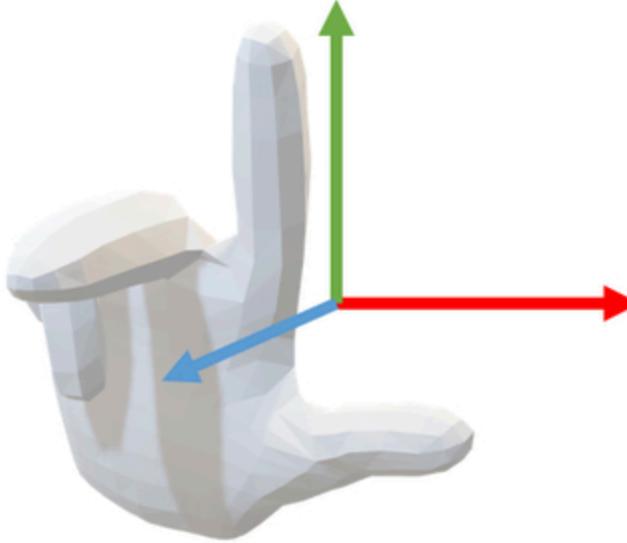
では、 Z はどうでしょう。奥の方向と、手前の方向と、どちらがプラスなんでしょうか。

実は、この Z のプラス方向がどっちを向いているのかは、API やプラットフォームによって表現に差があります。

OpenGL のファミリーは、普通「右手系」と呼ばれ、Z のプラス方向は手前方向です。逆に DirectX は左手系と呼ばれ、Z のプラス方向は奥の方向になります。



左手系



右手系

XY を親指と人差し指で表したときの、中指の指し示す方向が Z の向き！

親指と人差し指で XY の向きを指したとき、左手と右手では中指の指す方向が逆になります。これがそのまま Z のプラス方向に一致するのですね。

OpenGL や WebGL は右手系なので、Z のプラス方向は手前です。もしどっちがプラスの向きだかわからなくなったりしたときは、右手系というのを思い出しながら考えるようにしてみましょう。

実際に細部を見ていく

それでは、実際に行列がどのように世界を変換してくれているのか、ちょっと体験してみましょう。

まず最初のサンプルの JavaScript のほうを見てください。ここでは、大量の頂点を三次元空間上に配置するための、ループ処理が書かれている箇所があります。

```
this.position = [];
this.color = [];
// 頂点を格子状に並べ、座標に応じた色を付ける
const COUNT = 100;
for (let i = 0; i < COUNT; ++i) {
    const x = i / (COUNT - 1);
    const signedX = x * 2.0 - 1.0;
    for (let j = 0; j < COUNT; ++j) {
        const y = j / (COUNT - 1);
        const signedY = y * 2.0 - 1.0;
        this.position.push(signedX, signedY, 0.0);
        this.color.push(x, y, 0.5, 1.0);
    }
}
```

ここでのポイントは、どのような計算がループ処理のなかで行われているのかをまずは落ち着いて見極めることです。

実際に、変数 `i` が 0 の場合や、変数 `i` が一番大きくなるときを計算してみればわかりますが…… 頂点が配置される範囲は XY 平面上 の、-1.0 ~ 1.0 の範囲になっている、という点がここでは重要です。

このサンプルでは、初期配置される頂点の座標は常に XY のいずれも
-1.0 ~ 1.0 の範囲内に収まっている状態になります。

そのことをしっかりと踏まえた上で、続いては頂点シェーダのコードを
次に示すようにちょっとだけ変えてみましょう。

```
// 行列と乗算している処理の部分を.....  
gl_Position = mvpMatrix * vec4(p, 1.0);
```

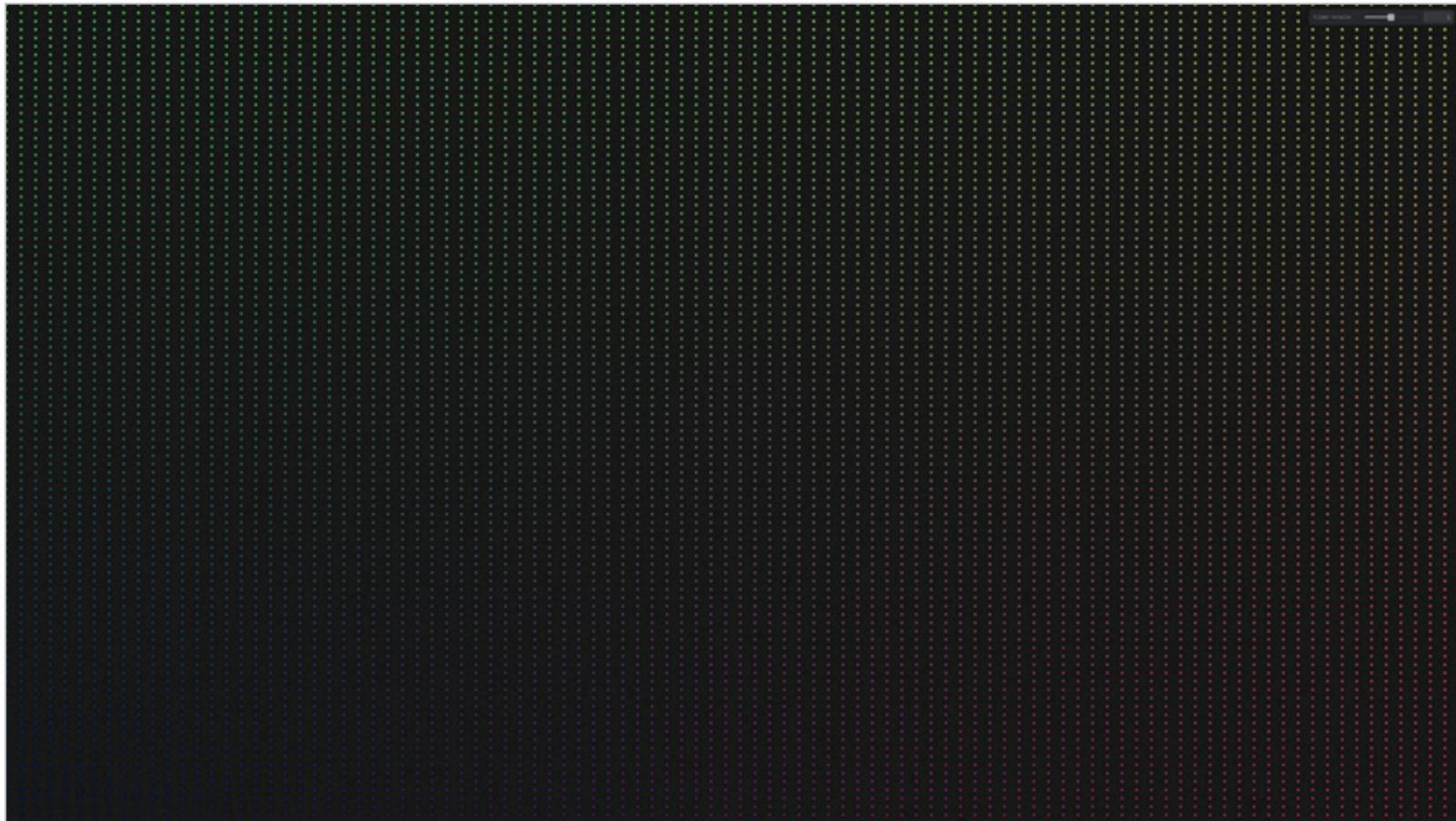
↓↓↓↓↓

```
// 行列と乗算しないでそのまま出力してみると?  
gl_Position = vec4(p, 1.0);
```

“座標変換せずに、そのまま出力したらどうなる？”

これは実際にやってみれば明らかですが……

頂点の座標変換をしないということは、先程で図解しながら説明したような「カメラやスクリーンサイズに応じた変換」などは 一切行われない 状態になってしまいます。ですから最終的な出力結果は、GUI 上のスライダー操作に応じて挙動が変化したりもしませんし、立体的に見えるような回転も一切しなくなります。



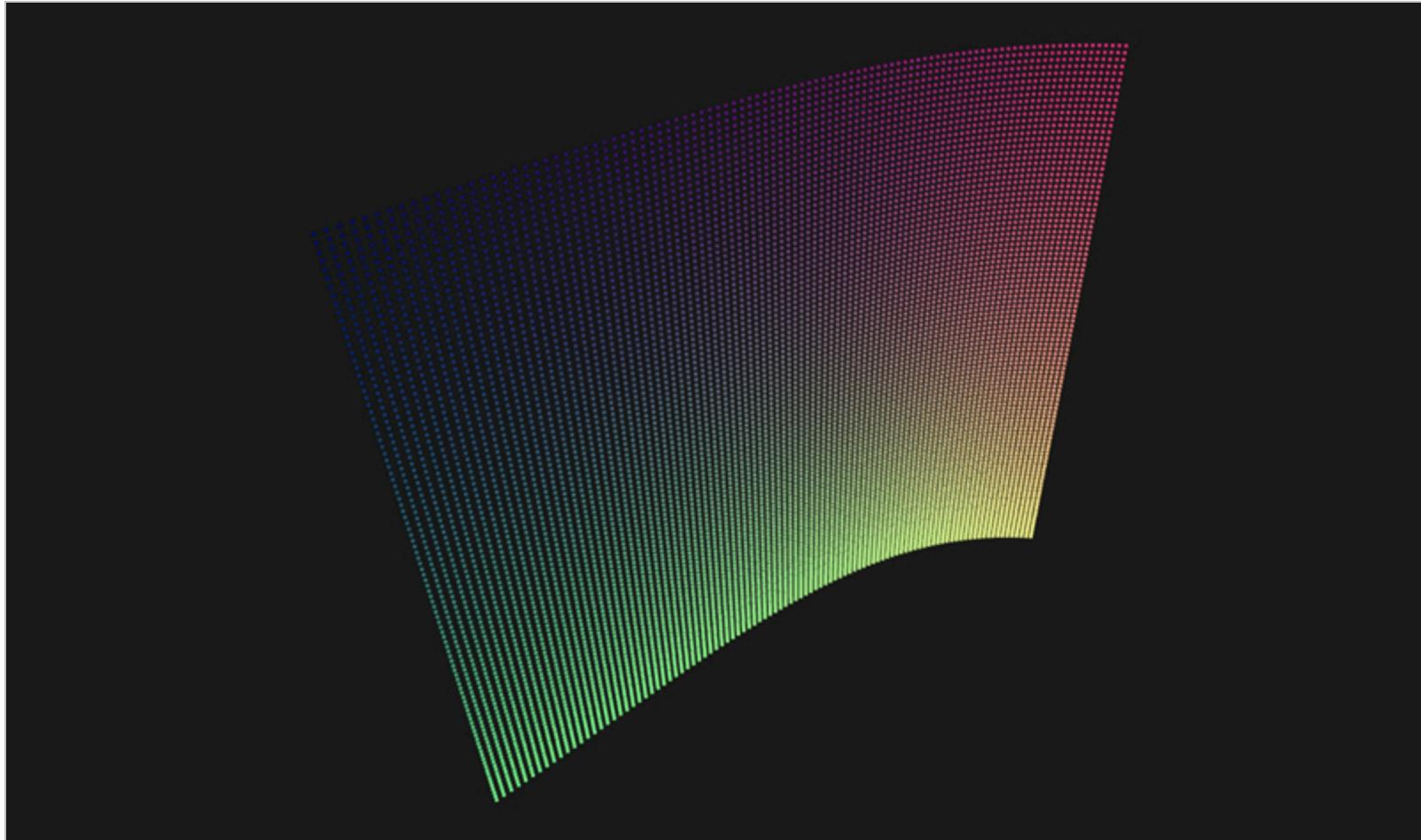
“当然ですが、変換されていないローカル座標そのままの位置に頂点が出ます”

この様子を見ると、頂点シェーダのやっていることがだんだんと具体的にイメージできるようになってきたのではないでしょか。

頂点とは、単なる三次元空間上的一点を表す単位のようなものに過ぎません。ですから何も変換を行わなければ、当たり前ですが第一回の講義のサンプルと同じようにそのままの位置で画面に表示されるだけになってしまいます。

3DCG がいかにも立体的に、自然な奥行き感のある 3D シーンを画面に描画することができる原因是、頂点シェーダによって行われる（主に行列などを用いた）「座標変換のおかげ」だったのです。

3DCG の「3D らしさ」を実現するのに、頂点シェーダは大きな役割を果たしているのです。



この立体的な見た目は行列によって実現されている！

行列に関する処理は、JavaScript の基本的な算術関数（`Math` オブジェクトなど）ではサポートされていないので、自分でなにかしらの行列処理を記述しないといけません。

サンプルでは、`math.js` という私が書いた最低限の行列処理を行うことができるライブラリを使っています。

ライセンスフリーなので適当に流用してもらって問題ありません

今回の最初のサンプルでは `WebGLApp.render` メソッドの内部で行列を最初に一気に定義・生成しています。

```
// - 各種行列を生成する -----
// モデル座標変換行列（ここではゆっくりと x 軸回転）
const rotateAxis = v3.create(1.0, 0.0, 0.0); // x 軸回転を掛ける
const rotateAngle = this.uTime * 0.1; // 回転角は時間由来
const m = m4.rotate(m4.identity(), rotateAngle, rotateAxis);
```

(以下続く)

実際には、その都度変更しないような行列の場合は毎フレームわざわざ定義し直さなくとも事前に生成しておいて再利用すればよい場合もあります

ここでは先ほど図解しながら説明したような、カメラや、切り取る空間の情報が一気に行列に対して詰め込まれていきます。それらは最終的に `mvp` という名前の変数に格納され、GLSL へと送られていきます。

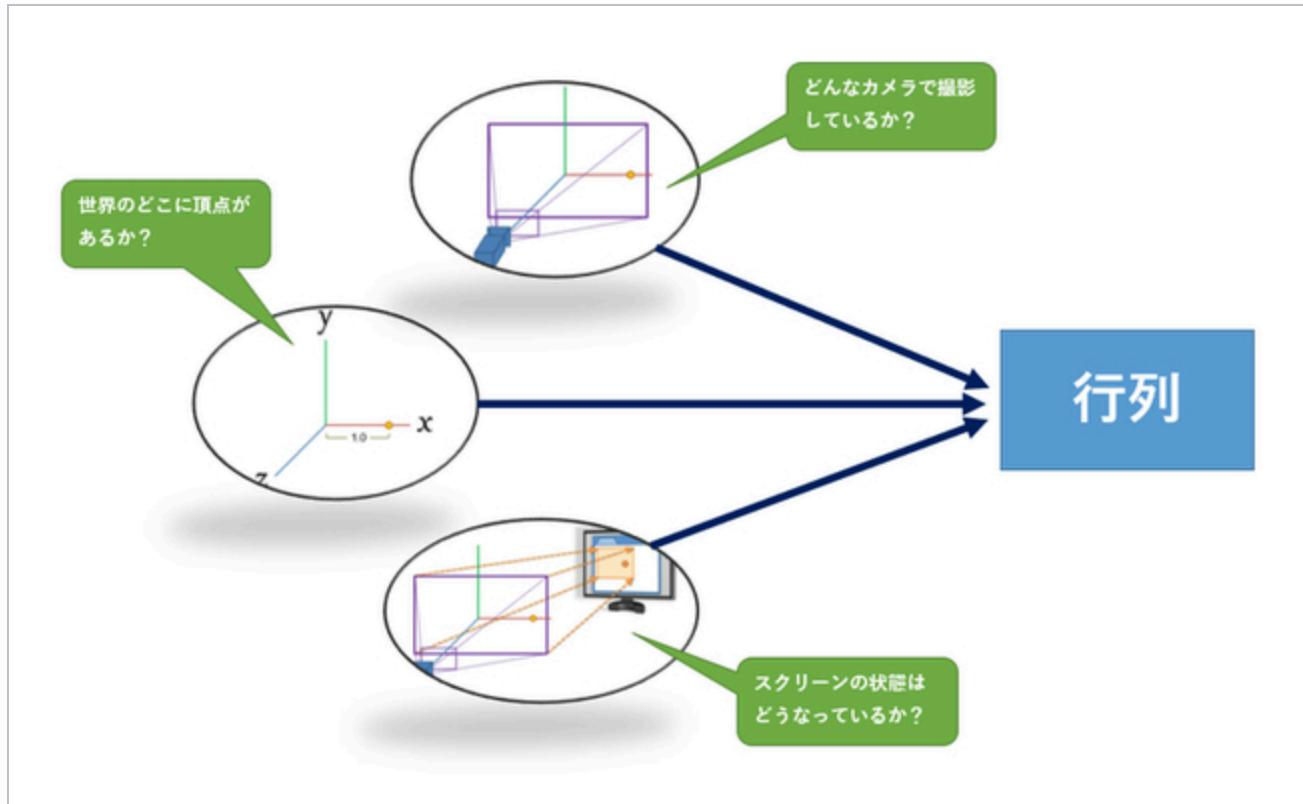
```
// プログラムオブジェクトを指定し、VBO と uniform 変数を設定
this.shaderProgram.use();
this.shaderProgram.setAttribute(this.vbo);
this.shaderProgram.setUniform([
    this.uTime, // 経過時間
    mvp, // MVP 行列
]);
```

もしかしたらちょっと不思議に思う方もいるかもしれないで補足すると、行列には、複数の効果を合成してまとめることができるという特性があります。

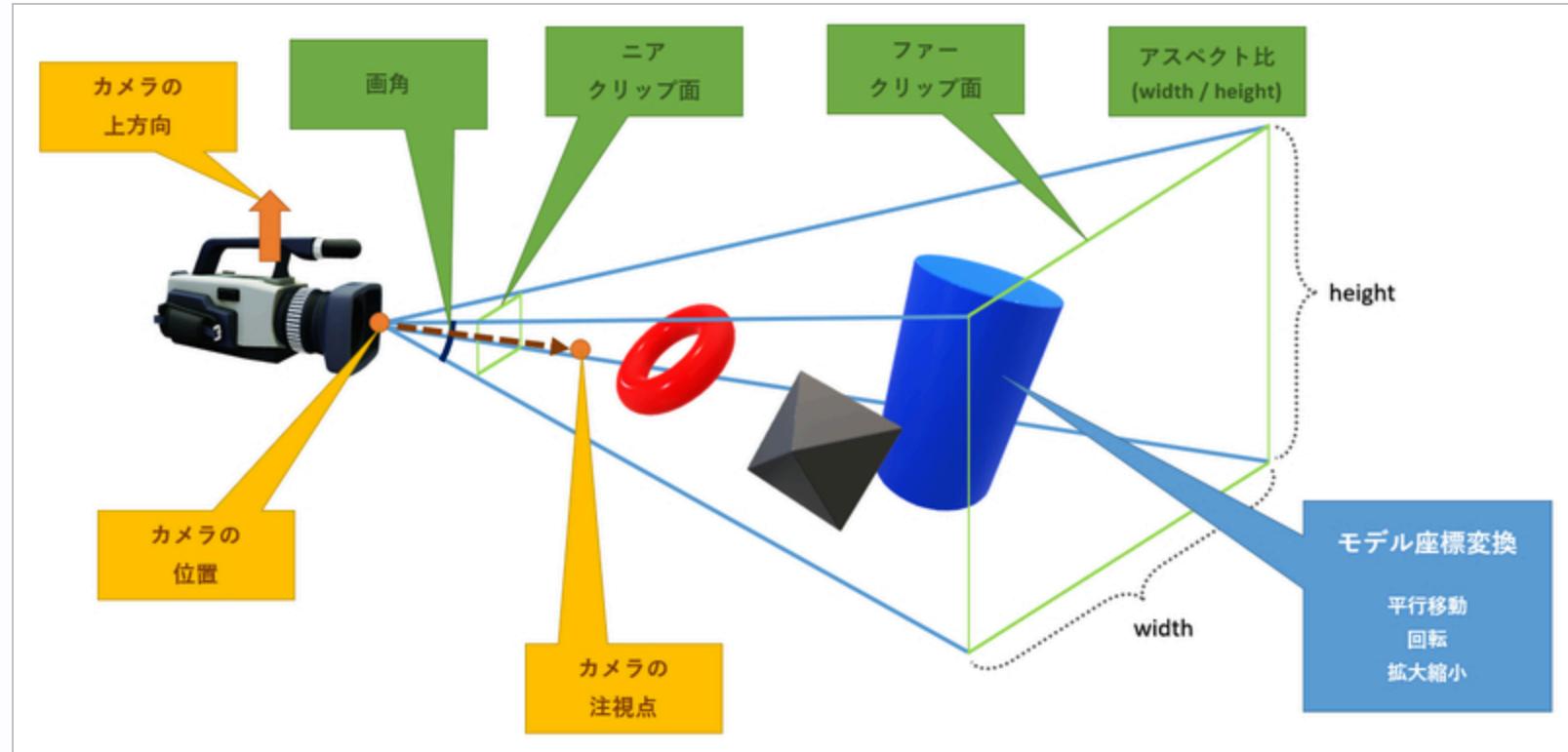
これは 3D が云々とか WebGL がどうしたという話ではなく、単純に 行列の持つ数学的な特性 です。

カメラの情報やスクリーンの情報など、レンダリングに必要となるたくさんの情報を行列に一度格納し、それらを順番に掛け合わせてやることでたったひとつの行列にその効果をまとめることができます。

シェーダに送るのは、その合成された後の 全ての情報が詰め込まれた行列ひとつだけ でいいのです。



“全ての効果はひとつの行列に集約できる！”



“これだけの情報がたった一つの行列に詰め込まれている！”

CPU 側で定義した行列は、シェーダに送られた後は当然ながら GPU によって処理されます。

GPU 上で動作するプログラムとなる GLSL 側では、アプリケーションから送られてきた `uniform mat4 mvpMatrix` を 全ての頂点の座標変換に一律で使う ことになります。全ての頂点が、行列の効果によって一様に同じように変換される わけですね。

サンプル 006

- 行列の実態は長さ 16 の配列
- その中にたくさんの情報が（乗算することで）詰め込める
- GLSL では頂点の変換にひとつの行列を用いるだけでよい
- 時と場合により、行列を複数個シェーダに送る場合もある
- 行列の効果により立体的に見えるような描画結果が得られる
- 立体的な空間を扱う場合は「深度テスト」についても注意（※後述）

補足コラム

深度テストとは

深度テストとは、3D 空間を破綻なく表現するために「深度バッファと呼ばれる浮動小数点で埋め尽くされたバッファ」を用いて、深度を確認しながら前後関係を正しく処理する仕組みです。

WebGL の場合、深度テストは既定では無効化されているので意図的に有効化する処理を行う必要があります。

以下のようにすることで、深度テストを有効化したうえで「深度バッファをクリアする際に、そこにどんな既定値を書き込むか」が設定されます。

```
this.gl.clearDepth(1.0); // クリアする際の深度値 @@@  
this.gl.enable(this.gl.DEPTH_TEST); // 深度テストを有効にする @@@
```

さらに、クリア処理を行う際に「色だけでなく、深度も同時にクリアする」という引数の指定を行うことで深度バッファがクリアされます。

普通、深度テストは「最も遠いところが 1.0、近づくほど 0.0 に近づく」という形で表現することが多いです。

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

補足コラム

行列の乗算とは

行列は、単体の数値と同じように掛け算、つまり乗算ができます。

行列を乗算する際の手順（計算方法）は決まっています。以下のサイトで実際に動かしてみることで理解が深まるでしょう。

Matrix Multiplication

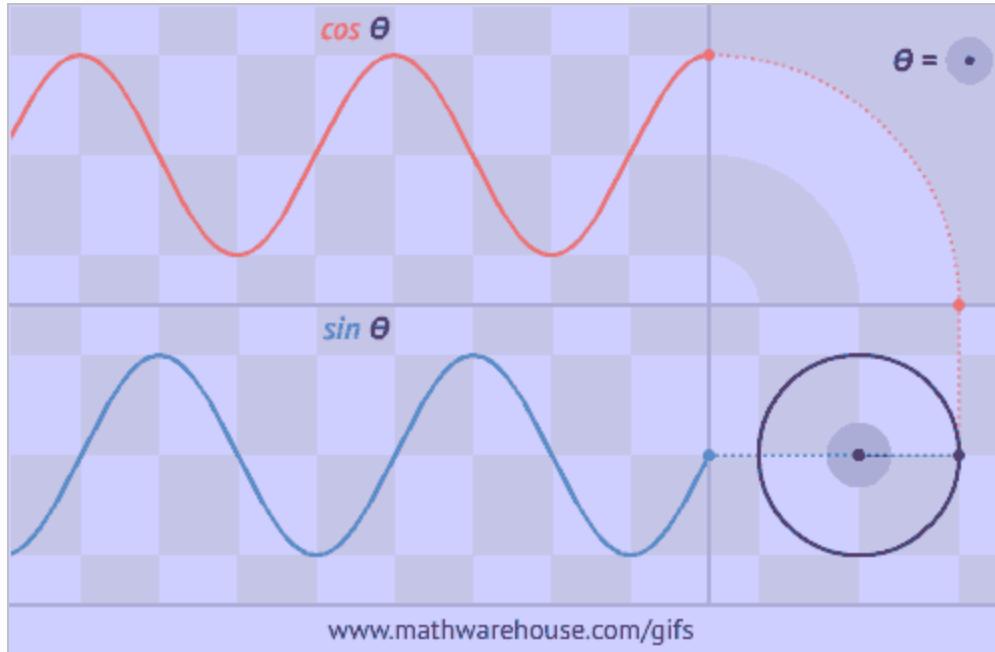
たとえば以下のように行列内に数値を配置してベクトルと乗算すると、座標やベクトルを回転させることができます。

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

$\theta == 90^\circ$ のとき……

$$\cos(\theta) == 0 \quad \sin(\theta) == 1$$

サインとコサインについては以下の図がわかりやすいです。



原典 : [sine-cosine-unit-circle](#)

円運動の X 方向の移動量の遷移がコサイン、Y 方向がサイン

補足コラム

行列はどこで乗算する？

JavaScript で行列を乗算してからシェーダに送って利用するのと.....
行列を個別にシェーダに送り、シェーダ側で乗算するのとどちらが良いのか、という疑問を持たれている方もいらっしゃるかもしれません。

“ 実際、ほぼ毎年この質問が出ます ”

基本的に「JavaScript 側で乗算してシェーダに送る」のが正解です。
なぜなら、頂点シェーダ内で行列処理を行ってしまうと「すべての頂点を処理する際に、その都度行列を乗算する処理を行うことになる」からです。

“すべての頂点でまったく同じ計算を重複して行うことになる”

頂点の数が少ない場合は問題にならないかもしれません、頂点の数が多くなればなるほどシェーダ内で行列処理を行うことのオーバーヘッドが大きくなります。

可能な限り、アプリケーション（JavaScript、CPU 側）で行列を作ったうえで送ってやるようにしましょう。

複数のローカル座標を持つ頂点

頂点の情報を構成する attribute 変数は、実装者がその内容・構成を自由に決めることができます。

これまで、`position` という名前の `vec3` 型の頂点属性（attribute 変数）を用いてきましたが、これは仕様でそうしなければならないと決まっているわけではありません。あくまでも、一般にそのようにすることが多い、というだけなのです。

とは言え、頂点属性が自由に定義できてなにがおもしろいのか、どういうことが可能になるのかは最初はなかなかイメージしにくいかもしれません。

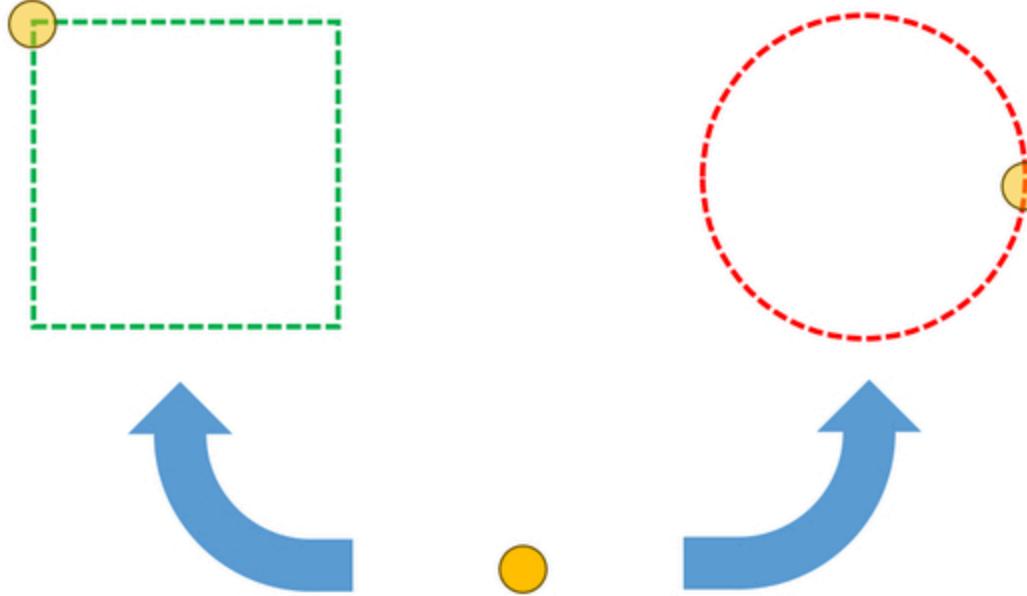
ここでは、頂点にあえて 2つのことなる座標を同時に与えて しまい、それらをインタラクティブに変化させられるようにしてみましょう。

サンプル内、`WebGLApp.setupGeometry` を見ると、平面上に並ぶ頂点の座標と、球の表面に並ぶ頂点の座標、また同様に2つの色を同時に定義しています。

```
this.planePosition = [];  
// 頂点座標  
this.planeColor = [];  
// 頂点カラー  
this.spherePosition = [];  
// 頂点座標  
this.sphereColor = [];  
// 頂点カラー  
(中略)  
this.vbo = [  
    WebGLUtility.createVbo(this.gl, this.planePosition),  
    WebGLUtility.createVbo(this.gl, this.planeColor),  
    WebGLUtility.createVbo(this.gl, this.spherePosition),  
    WebGLUtility.createVbo(this.gl, this.sphereColor),  
];
```

1つの頂点が、同時に二種類の形状と色を持っている状態にしています。

頂点シェーダではこれらすべてを attribute 変数として受け取ることになりますので、理論上、どちらの形状を使うかを自由に選択するようなコードが記述可能ということになります。



頂点はどちらの姿も表現できるよう 2 つの情報を同時に持っている
どの情報をどのように使うかは、頂点シェーダの記述内容次第

シェーダ内でこれらの情報をどう利用するのも自由です

今回のケースでは、2つの情報を持った頂点がどのように振る舞うのかは `uniform float ratio` の値によって決まります。

画面上の GUI のスライダーを操作すると値が変化し、それが `uniform` としてシェーダに送られ、0.0 であれば平面上に、1.0 であれば球状に、その中間であれば両者を補間した結果が使われます。

サンプル 007

- attribute 変数で 2 つの座標・色を受け取る頂点シェーダ
- それぞれの座標は JavaScript 側で一気に定義
- どのような形状となるかは uniform 変数の値によって制御
- GLSL のビルトイン関数 `mix` を使うことで線形補間ができる
- 線形補間は 0.0 ~ 1.0 の範囲の値を係数として用いることが多い

カメラ制御について

ここまでに登場したサンプルでは、いずれも 3D シーンを描画してはいるものの、マウスを使ってインタラクティブにオブジェクトを回転させたりすることはできませんでした。

3D の作例やツールではあまりに当たり前にできるマウスでの 3D シーンへの干渉ですが、自分でこれを実装するとなると実は結構たいへんです。

続いてのサンプルではマウスを使ってカメラを動かすことで 3D シーンに干渉できるようにしてあります。ただし、これを実現するには結構な数学的な知識が必要になりますので、ここではそれを詳しく解説することはしません。（質問するなという意味ではありませんので、もちろん興味があれば聞いてください！）

camera.js というカメラの機能を実装したものを使っていますので、興味があったらソースコードを見てみるのもいいかもしれません。

この自作のカメラクラスは、マウス操作に反応してカメラのパラメータをリアルタイムに更新し、それに応じたビュー座標変換行列を生成してくれます。

```
import { WebGLOrbitCamera } from '../lib/camera.js'; // カメラ制御  
(中略)  
  
this.camera = new WebGLOrbitCamera(this.canvas, cameraOption);  
(中略)  
  
const v = this.camera.update(); // update メソッドがビュー行列を返す
```

サンプル 008

- 頂点にランダムな属性値を与える
- ランダムな値を使って不規則性を演出
- 頂点シェーダ内の処理は一見するとめちゃくちゃ複雑だが、単純計算なので落ち着いて考えること
- カメラ制御は数学的な知識を結構必要とする
- ひとまずスクールオリジナルのカメラクラスを使ってマウスで制御できるようにする

動作確認やデバッグ時、マウスでカメラに干渉できるだけでぐっと開発がしやすくなります

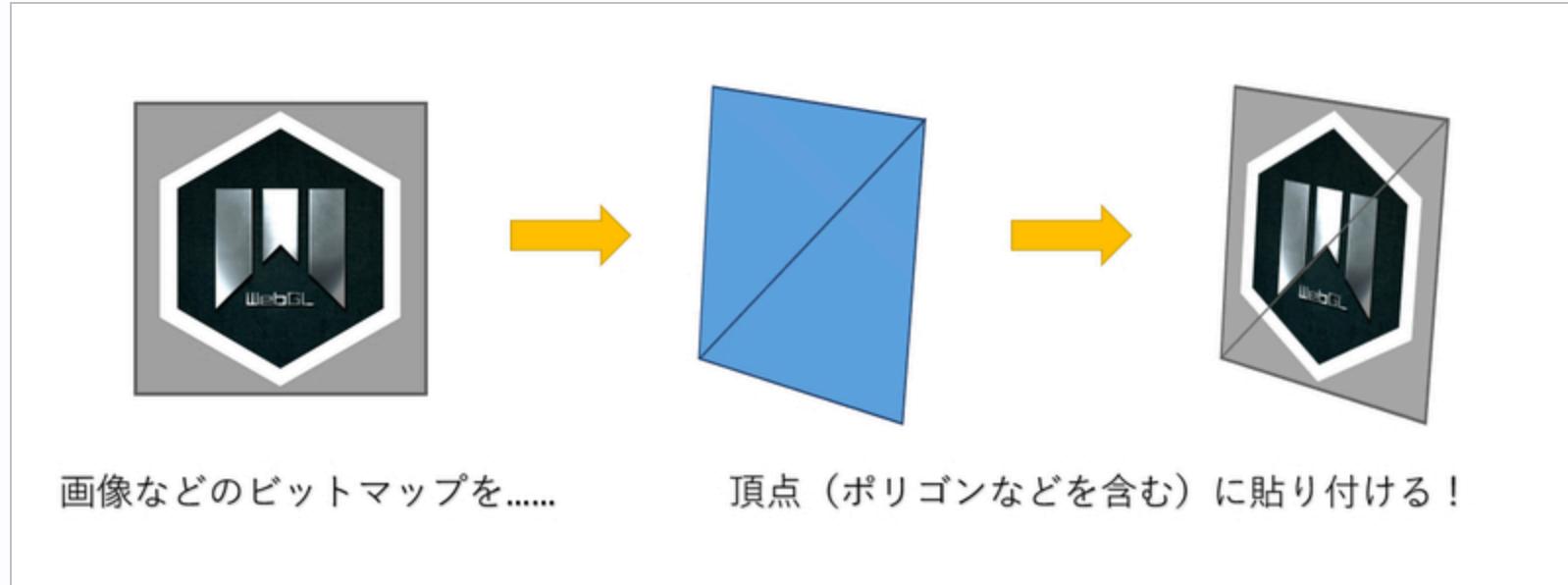
テクスチャ座標

さて、続いてのテーマは テクスチャ です。以降のいくつかのサンプルでは、画像ファイルを読み込んでそれを利用した表現が行われています。

WebGL や OpenGL、あるいは DirectX など含め CG 一般の用語として、こういったビットマップを扱う概念を総称して「テクスチャ」と呼んでいます。

基本的には「テクスチャ」と言った場合は（特殊な例外は一部あります
が）画像などのなにかしらのビットマップを用いた処理だと考
えていいでしょう。

WebGL の場合はウェブページに読み込む画像形式としてよく使われる
JPEG や PNG などをテクスチャ用のリソースとして利用できます。さら
には、`canvas` 要素に描画した結果や動画ファイルなどをテクスチャに使
うことも簡単にできます。



“ WebGL の場合は画像の読み込み処理は JavaScript 側で行う ”

テクスチャをしかるべき手順を踏んでセットアップしてやると、GLSL側で「色の情報をテクスチャから取り出す」ことができるようになります。以下はその記述例。

```
uniform sampler2D textureUnit; // テクスチャユニット  
varying vec2 vTexCoord; // テクスチャ座標  
  
(中略)  
  
// テクスチャから色を読み出す  
vec4 samplerColor = texture2D(textureUnit, vTexCoord);
```

sampler2D から「テクスチャ座標」を使って「色」を読み出す

ここで唐突に出てきた用語があるので、まずはその意味を覚えましょう。

まず、GLSL の内部ではテクスチャはサンプラー（もしくはサンプラー）と呼ばれます。なんで「テクスチャ」じゃなくて「サンプラー」なのか.....という疑問を持つ方もいるかもしれません。

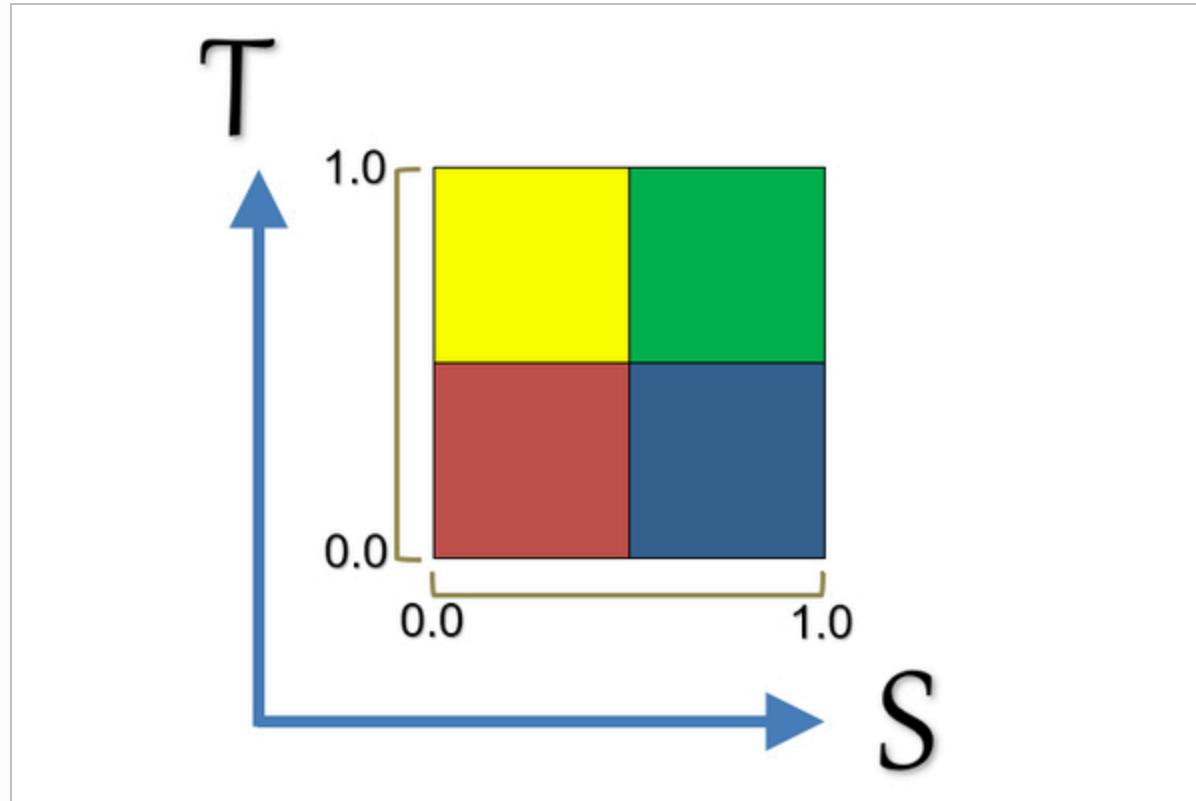
テクスチャは画素の集合体ですが、そこから色をサンプリング（抽出や採集の意）する際、どのようなアルゴリズムで色を取り出してくれればよいのかを指定でき、これをサンプリング設定と呼びます。

GLSL 内部では、このサンプリング設定は変更することができず、
WebGL の API を使って JavaScript 側で設定します。テクスチャを扱う
画面では必ずこのサンプリングの設定も含めて扱うため、GLSL 内部で
は `sampler2D` というデータ型で表現されるようになっています。

また具体的なテクスチャの利用方法としては、GLSL で `sampler2D` というデータ型の変数から色を抜き出すために使われるのが `texture2D` 関数です。

この関数は第二引数に テクスチャ座標と呼ばれる `vec2` の座標データ を受け取ります。このテクスチャ座標はモデリングソフトなどでは、よく UV という名前で呼ばれるものと同じです。

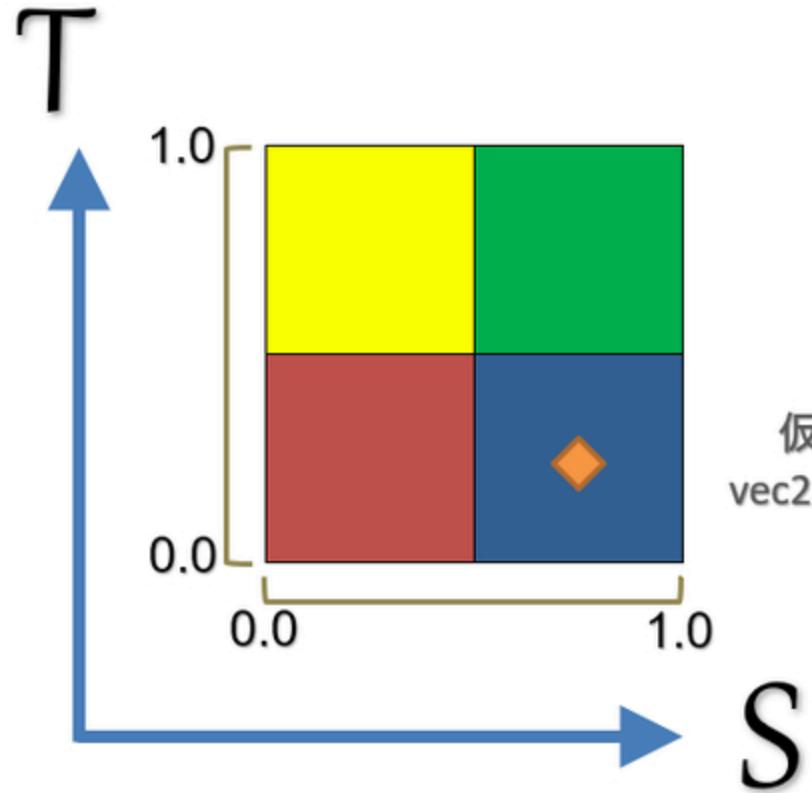
左下を原点として横方向が S、縦方向が T で表される。



黄・緑・赤・青の4色には深い意味はなく、単にそういう模様の画像を読み込んだ図、とみなして見てください

テクスチャ座標は、常に `0.0 ~ 1.0` の範囲で指定することに注意しましょう。

これまで頂点の座標（位置）は常に三次元でしたが、テクスチャは 平面的なビットマップ ですのでテクスチャ座標も同様に 二次元座標 であることも大事なポイント。`S` や `T` といった記号の意味がちょっとわかりにくければ、`X` や `Y` などのイメージしやすい記号で考えてしまっても問題はありません。



仮に、テクスチャ座標が
 $\text{vec2}(0.75, 0.25)$ だとした場合

この図を例に取ると $\text{vec2}(0.75, 0.25)$ という位置を参照することで青い色が vec4 として取得できる、といったイメージです。

テクスチャ座標は、一般に「頂点自身がテクスチャ座標を持っている」という形にしてシェーダに送ります。

つまり、attribute 変数として `texCoord` といったような、テクスチャ座標を表すような名前の頂点属性を定義してやり、それをもとにテクスチャを参照できるようにしてやります。

“coord”は、英語で「座標」を意味する Coordinate の略です”

テクスチャを参照した結果得られるのは「色の情報」なので.....

実際には、頂点属性 (attribute) として入力されてきたテクスチャ座標を `varying` 変数としてフラグメントシェーダに送り、フラグメントシェーダ側でテクスチャの色を読み出すというフローになります。

言葉で説明されてもややこしいと思うので、実際にコードを見ながら考えてみるほうがわかりやすいと思います

まずはテクスチャ座標の定義。

```
// テクスチャ座標を定義 @@@  
this.texCoord = [  
    0.0, 0.0,  
    1.0, 0.0,  
    0.0, 1.0,  
    1.0, 1.0,  
];
```

JavaScript 側で 0.0 ~ 1.0 の範囲になるようにテクスチャ座標（二次元ベクトル）を定義しておき、これを VBO にしてシェーダに送り、attribute 変数として GLSL 内で参照できるようにしておきます。

続いて、VBO の情報を受け取る頂点シェーダ。

```
attribute vec2 texCoord;  
varying vec2 vTexCoord;
```

(中略)

```
vTexCoord = texCoord; // フラグメントシェーダに送る
```

頂点シェーダからは、フラグメントシェーダへと `varying` 変数として渡してやります。わざわざフラグメントシェーダにテクスチャ座標を渡すのは、実際に色を読み出して使うのは多くの場合フラグメントシェーダであるためです。

最後に、フラグメントシェーダ内で `texture2D` 関数を使って色を読み出します。

```
uniform sampler2D textureUnit; // テクスチャユニット  
varying vec2 vTexCoord; // テクスチャ座標  
  
(中略)  
  
// テクスチャから色を読み出す  
vec4 samplerColor = texture2D(textureUnit, vTexCoord);
```

ちなみに、頂点シェーダ内でもテクスチャを参照して色を読み出すことはで
きますが、それはなにか特殊なことをしたい場合に限られます

サンプル 009

- テクスチャを利用するには VBO 等と同様バインドが必要
- GLSL 側ではテクスチャは `sampler2D` という型になる
- テクスチャをサンプリングするのはフラグメントシェーダ側で行う
- テクスチャ座標は `0.0 ~ 1.0` の範囲
- S が横方向、T が縦方向を表す二次元座標で、原点は左下
- ここではポリゴン化するために `gl.TRIANGLE_STRIP` を使っている

サンプルの補足

この補足は、より深く理解したい人向けの補足です。

ちょっと難しい内容を含むので、もし厳しいなあと感じたら、無理に暗記したりする必要はありません。

まずテクスチャは、基本的に「覚えることが多く、やや扱うのが難しい概念」です。今回は理解を助ける意味で、あえて説明していないテクスチャに関する設定項目がたくさんあります。

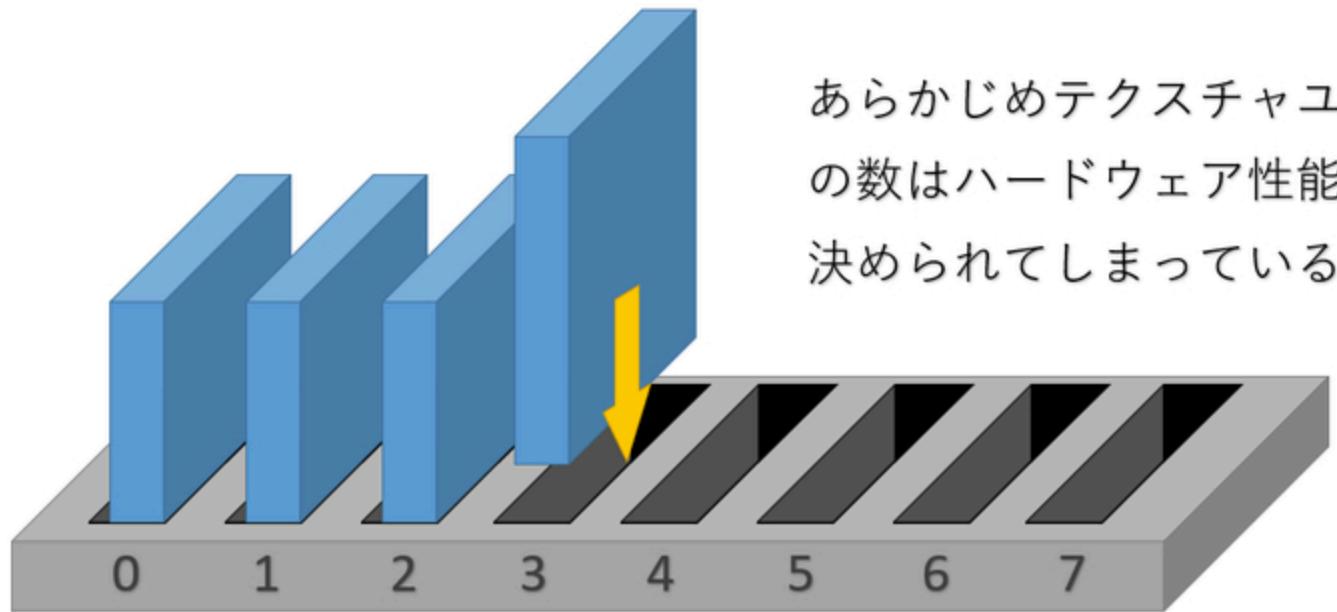
このあたりは最初の段階ではひとまず理解できていなくても、テクスチャを使うこと自体にそれほど大きな問題とはなりませんので、徐々に理解が深まってきたら少しづつ勉強していくといいと思います。

参考：wgld.org | WebGL: テクスチャパラメータ

そして、テクスチャを複数使う場合にちょっとわかりにくいのが、テクスチャユニットの概念。

テクスチャユニットは、いわば「カートリッジを差し込むスロット」のようなものです。差込口の数はハードウェアごとにあらかじめ決まっていて、例えばモバイル端末だったりすると、仕様上の最低必要数である8個だったりします。

“ 一般に PC などはもっと多いことがほとんど ”



あらかじめテクスチャユニット
の数はハードウェア性能により
決められてしまっている

テクスチャがバインドされると、スロットの差込口がどんどん埋まっていくようなイメージ。

テクスチャは「頂点を描画するときにその時点でバインドされているもの」が使われます。ですから複数のテクスチャを用いる場合でも、テクスチャユニットの0番にバインドし直してから頂点を描画、というように順番に処理していくけば、実際にはテクスチャユニットの最大数以上のテクスチャを用いて絵作りをすること自体は可能です。

ちょっと違った言い方をすると、テクスチャユニットの最大数とは、あくまでも「同時にバインドできるテクスチャの最大数」であることに注意しましょう。

テクスチャを使っていろいろやってみる

さて、続いてもテクスチャを使ったサンプルです。

次のサンプルはテクスチャを複数同時に利用するケースの実装例です。ここではスライダーを操作してパラメータを自由に変えることができますが、このパラメータが uniform 変数としてシェーダに逐一送られるようになっています。



このような処理は「テクスチャを複数同時にシェーダ内で参照」できないと実現できないため、複数のテクスチャユニットを消費して初めて実現できます。

サンプル 010

- あらかじめユニット 0 とユニット 1 にテクスチャをバインドしておく
- uniform 変数で 2 つのユニット番号を送っておく
- シェーダ内では同時に 2 つのテクスチャから色をサンプリング
- mix 関数を使って色を線形補間する
- GLSL の内部では座標も色も共にベクトルで表現できるため同じように計算することができる

続いてのサンプルも、やはりテクスチャを使った表現の例です。

ここではさらにもうひとつ、同時に利用するテクスチャを増やしています。

つまり合計 3 つのテクスチャを同時に使うシェーダ

合計 3 つのテクスチャを同時にシェーダ内で参照するのですが、ここでは先程のサンプルと同様の「色を合成する処理」を行う際に「第三のテクスチャの明度によって動作に差が生まれる」ようにしています。

フラグメントシェーダで色を合成する処理の部分が、一見するとちょっとわかりにくいかもしれません。しかし実際にはすごく単純な計算を行っているだけなので、図解するなどしながら落ち着いて考えてみましょう。

サンプル 011

- テクスチャを合計 3 枚、同時に利用する
- JavaScript からは色の合成係数を送る (0.0 ~ 1.0)
- `clamp` を使って最大値と最小値の範囲外にならないように調整している
- `ratio` が、0.0 の場合、0.5 場合、1.0 の場合など、いくつかの数値で実際に計算してみるとわかりやすいかも

サンプルの余談

同時にテクスチャ座標 자체を水平にずらしてみるなど、適度に変化をつけてやることでよりかっこよくなります。

参考 : [Distortion Hover Effect | Codrops](#)

まとめ

さて、今回は頂点シェーダを主なテーマとしつつ、数学に関するキーワードや概念、さらにはテクスチャといった概念など、多くのトピックが登場した回となりました。

頂点シェーダは主な役割が「座標変換」である場合がほとんどなので、頂点シェーダ単体でビジュアルを作るというよりは、複雑で緻密なグラフィックスを作るための事前処理を頂点シェーダでいかに記述するか、ということを考え方のポイントになるでしょう。

行列を用いて頂点の座標を変換する処理は、最初のうちは特に、なにがどうなってるのかいまいち感覚としてつかみにくいです。

しかし、なんの座標変換もしない状態で出力したものと、行列による座標変換を行った結果とを見比べると、それがいかに重要な処理を行っているのかが想像できると思います。

頂点シェーダが自在に操れるようになれば、生き生きとした、ダイナミックな頂点の「動き」を実現できるようになります。

今回併せて解説したテクスチャなどをうまく活用しつつ、動きのある表現に少しずつで構わないので挑戦していきましょう。

今回の課題は、テクスチャを使った表現に挑戦してみましょう。

テクスチャとポリゴンを組み合わせて、フラグメントシェーダで表現を工夫するのもよいでしょうし、テクスチャと点、あるいは線と組み合わせてもなにか面白いことができるかもしれません。

たとえばプリミティブ（形状）がポリゴンでなくても、それこそたとえば点として頂点を描いていてもテクスチャからはまったく同じように色を読み出すことができますし、それを利用して表現の幅を広げることができます。

先入観にとらわれず自由に発想を広げてみましょう。