3D 的な表現と照明効果

シェーダを使った表現の基礎

はじめに

前回は、ネイティブな WebGL の API を用いた実装の最初の一歩、という感じで、サンプルの数こそ少なかったですがピュア WebGL のたくさんの手続きが登場した、ちょっと難しい回になりました。

レンダリングパイプラインの話などは、WebGL に限らず、あらゆる CG の基礎になるような内容ですので、余裕のあるときで構いませんのでしっかりと流れを掴んでおくといいでしょう。

さて今回ですが、いよいよ 3D シーンの描画に挑戦する場面が出てきます。

行列を用いた処理など数学的な話題も出てきますが、じっくり解説していきますので必要に応じてノート上で図解するなどしながら考えてみてください。

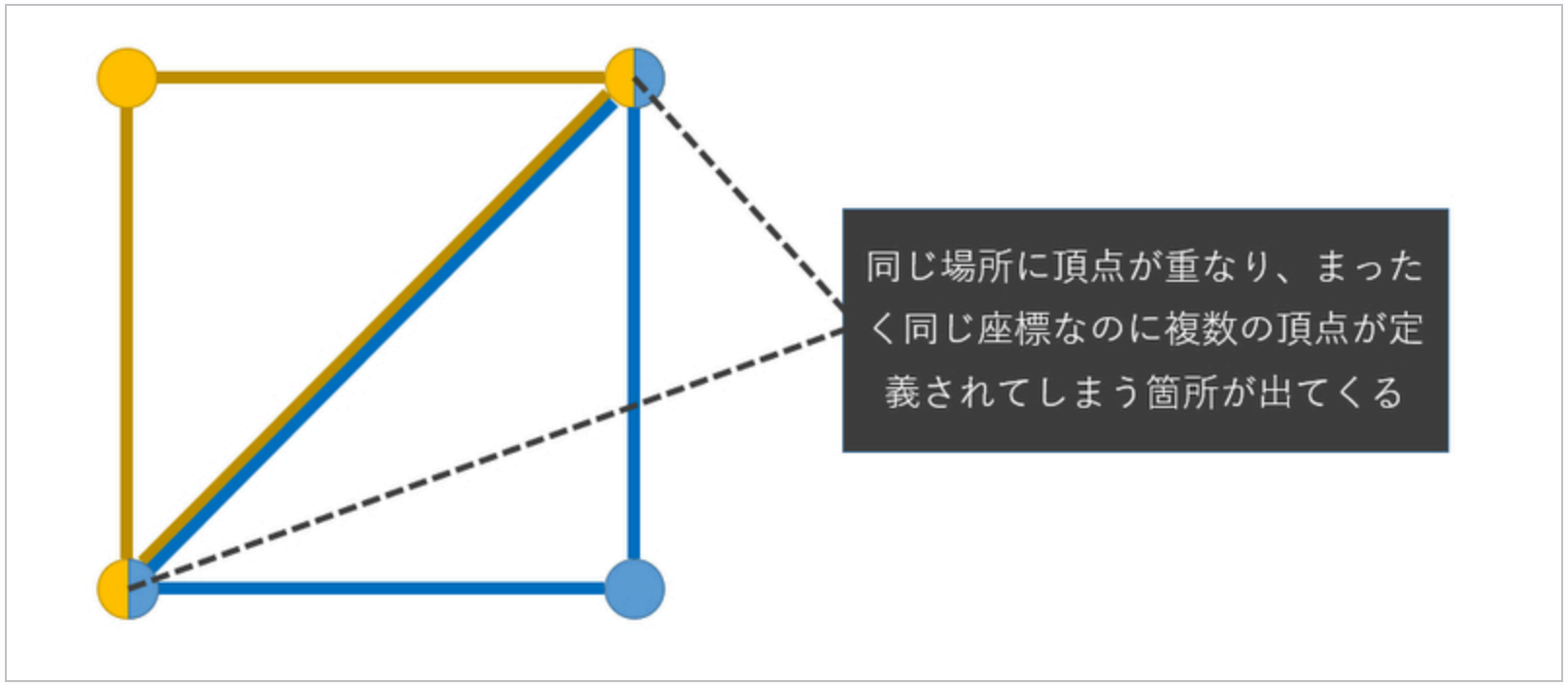
Index Buffer Object（IBO）

今回はまず肩慣らしに、前回の最後に課題として出した五角形の頂点を定義するという内容の、ある種の答え合わせ的なサンプルから見てみます。

課題を実際に自分でやってみるとより伝わりやすいかなと思うのですが、手で頂点を直接定義するのはなかなか大変です。五角形程度ならさほど手間ではありませんが、複雑な形状を手書きで定義するのはそれなりに非効率です。

また `gl.TRIANGLES` というプリミティブタイプ（形状）を用いてジオメトリを構築する場合、三角形を定義するのにその都度 3 つの頂点を準備する必要があります。

そのような場合、同じ座標に複数の頂点を配置するなど一見するとやや冗長に思えるような定義を行わなくてはなりません。



頂点３つで１つのポリゴンを定義する場合、どうしても同じ位置に重複する頂点ができてしまう

頂点が完全に同じ座標にあるにもかかわらず、複数の頂点を重複して定義しなくてはならないのは無駄に感じる人も多いでしょう。

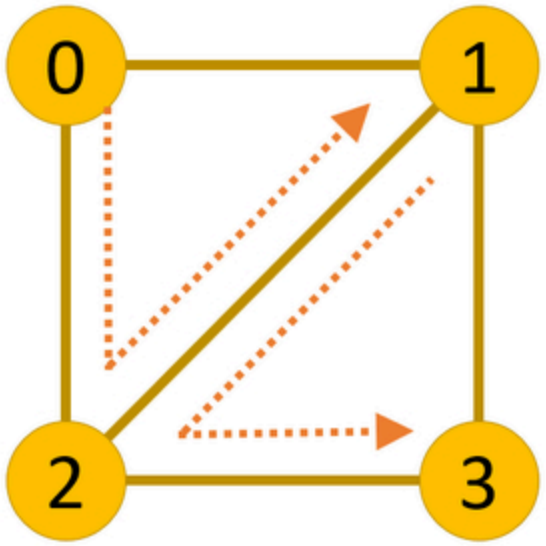
実際、重複する分だけメモリも余分に使ってしまいます。

そこで WebGL には、頂点それぞれのインデックス（連番の添字）を参照しながら描画することで、頂点を何度も使い回すことができる方法がちゃんと用意されています。

それが **IBO（Index Buffer Object）** を利用した描画処理です。

IBO は、頂点が定義された順番（インデックス、 0 から始まる連番）を指定しながら描画を行うために使います。

頂点をどのように結んでいくか、という情報だけを個別のバッファにすることで、GPU 内部ではインデックスを元にポリゴンなどが形成されるようにできます。



頂点を定義すること自体は一度だけでよく、
頂点を定義した順番をインデックスとして、
そのインデックスの番号を指定することで
どのようにプリミティブを構成するかを指定
することができる。

左記の場合：***[0, 2, 1, 1, 2, 3]***

> 定義順がそのまま連番のインデックス番号になる

005

- 005 は五角形をインデックスバッファで描く例
- 基本的には VBO と同じような感じで IBO 生成処理を行う
- VBO と異なる点は引数に指定する定数や、TypedArray の型
- 描画命令はインデックスバッファを使う場合専用のものになることに注意

> 一般に、メモリ効率などの点で IBO を使うほうがメリットが多く、three.js も IBO を使っています

3D の描画を扱うための下準備

さてここからは、いよいよレンダリングされるシーンを 3D にすることに挑戦していきます。

そして、シーンが 3D になるということは、前回の「レンダリングパイプライン」の話のところでも触れたとおり、3D の情報を二次元平面であるディスプレイ（モニター）に出力するための、様々な「座標変換の作業」が必要になります。

一般には 3D の座標の変換には 行列 が用いられます。これは、ベクトルと行列は相性がよく、GPU を使って効率的に計算を行うことができるためです。

今回はまず、この「行列」を用いた一連の処理の流れから見ていくことにしましょう。

> サンプル 006 はかなりシンプルですが……回転する板は確かに立体的には見えるためそれが 3D シーンであることは見てわかると思います

今回より、行列に関する処理を行う必要があるため `math.js` を追加で import しています。

また `geometry.js` というファイルも追加されていて、ここにはいくつかのジオメトリ定義のための機能が実装されています。

math.js の各種算術クラス

- ベクトルと、行列、クォータニオンの基本的な機能を提供する
- メソッドは原則静的（ `static` ）に定義されている
- three.js の算術クラスとは異なりただ単に配列を処理しているだけである点に注意（ `***.x` のようなプロパティは存在しない）

ジオメトリ定義クラス WebGLGeometry

- いくつかの基本的な幾何学構造の頂点定義を行う
- 戻り値のオブジェクトは同じ構造を返すようになっている
- `戻り値.position` のように頂点属性を参照できる
- `戻り値.index` で IBO 用のインデックスも取れる

まずは流れをつかむ

まずは全体的な流れをザッと俯瞰して眺めてみましょう。

最初に注目したいのはジオメトリを定義する `App.setupGeometry` の部分。

前回までは `this.position = [ ... ];` というように、直接配列を宣言して「頂点属性の情報を格納」していました。

今回は、これが `WebGLGeometry` クラスのメソッド呼び出しに置き換えられています。

```
// 板ポリゴンのジオメトリ情報を取得
const width = 1.0;
const height = 0.5;
const color = [1.0, 0.0, 0.0, 1.0];
this.planeGeometry = WebGLGeometry.plane(width, height, color);
```

> WebGLGeometry クラスのメソッドが返す情報をプロパティに保持しておく

`WebGLGeometry` クラスの各種メソッドは、常に戻り値の構成（どのようなメンバを持つオブジェクトを返すか）が同じになっており……

`position` や `index` など、VBO や IBO を作るために必要な情報をオブジェクトの形で返してくれるようになっています。

```
return {
  position: pos, // 座標
  normal: nor,   // 法線
  color: col,    // 色
  texCoord: st,  // テクスチャ座標
  index: idx     // 頂点の結び順
};
```

> これらの情報から VBO や IBO を定義すればよい

このように、 `WebGLGeometry` クラスとそのメソッドを用いることで、VBO や IBO 用の情報は手軽に用意できるようになっています。

今後のサンプルではこれらのメソッドを使って VBO や IBO を用意するようにしていきます。

次に `setupLocation` という関数の中身も見てみます。ここでは、前回までと大きく異なる重要な uniform 変数のロケーションを取得しています。

その超重要な uniform 変数が `mvpMatrix` です。

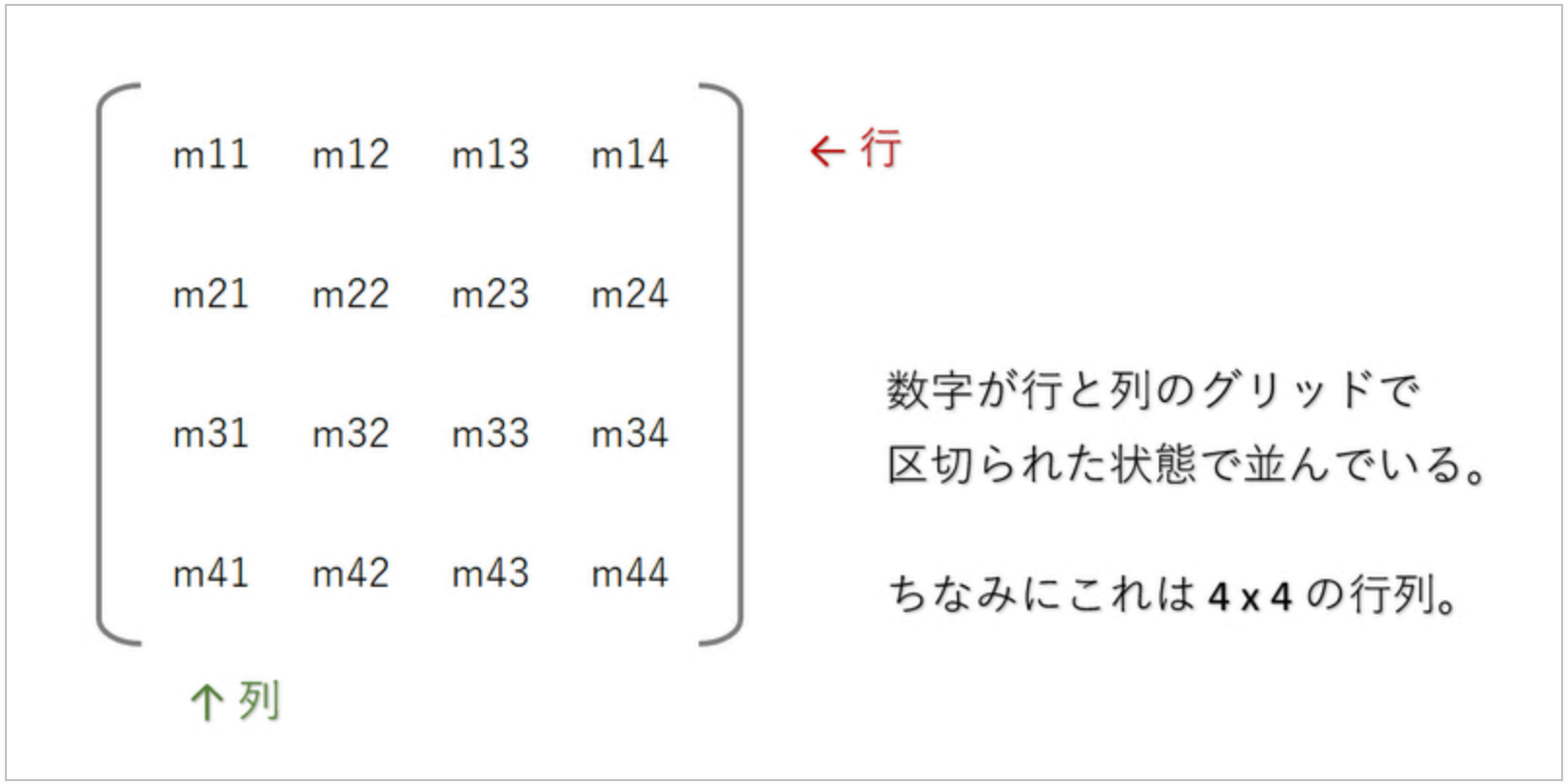
> matrix が「行列」という意味になります

数学における行列は、その難しそうな雰囲気から苦手意識を持っている人が多いかもしれませんが、3D プログラミングを行う上では行列は避けては通れません。

まずは `mvpMatrix` とはなんなのか、そこから深堀りしていきます。

ざっくり理解する行列

行列は線型代数学という数学のジャンルを中心に登場する概念で、文字通り、複数の数値を行と列を使って並べたような構造をしています。



数学的に正しく行列を理解するとなると、当たり前ですが、地道に教科書や参考書で勉強しなくてはなりませんが……

3D プログラミングで行列を扱う場合、まずは「行列の計算の仕方や内部的な演算処理」よりも「その行列がどういう意味を持っているのか」ということから優先的に理解していくのがよいでしょう。

> もちろん正しく数学的に理解できるならそのほうが好ましいと言えます

たとえばプログラミングをしていて `function rotate() { ...` という関数定義があったら、関数の中身を見なくても「この関数は回転に関係したものかな」とおおよそ想像することができますよね。

それと同じように、行列の厳密な計算方法や数学的な意味はいったん後回しにして、行列の種類に応じてなにを実現しようとしているのか、その意図や目的から把握していくようにすると理解しやすいと思います。

> 繰り返しますが、数学的に理解できるならそのほうがよいことは言うまでもありませんので、あしからず

ここまでのまとめ

- 行列は、行と列の組み合わせで数値を並べたもの
- 効率よく計算処理を行うことができるため 3D プログラミングには欠かせない概念であり、頻出する
- 行列がよくわからない場合でも、プログラミングにおける関数のようなイメージで意味や役割から覚えていくのがよい
- 最初は無理せず、まずイメージをつかむことが大切

プログラミングでは、関数に「その関数の効果や意図がわかりやすい名前」をつけて定義してやりますが……

行列にも、その及ぶ効果に応じた名前や、その行列の状態に応じた名前がつけられている場合があります。WebGL でもよく出てくる代表的なものを、いくつか見てみましょう。

単位行列

単位行列は、スカラーの値で言うところの「１」と同じです。

数値の１は、どんな数値と掛け算を行っても結果が変化しません。それと同じように単位行列は他のどのような行列やベクトルと乗算しても、その結果が変化しない特殊な行列です。

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

単位行列。

単位行列では、対角線上に 1 が並んだような構成になっており、これは数値の 1 と同じようにどのような行列やベクトルと乗算しても結果が変化しない形。

対角線上に 1 が並ぶ、単位行列

単位行列は他の行列やベクトルと乗算しても結果が変化しないので、行列を定義するときの「初期状態」としてよく用いられます。

なんの効果も及ぼさないので、最初の状態としては最も都合がよいためです。

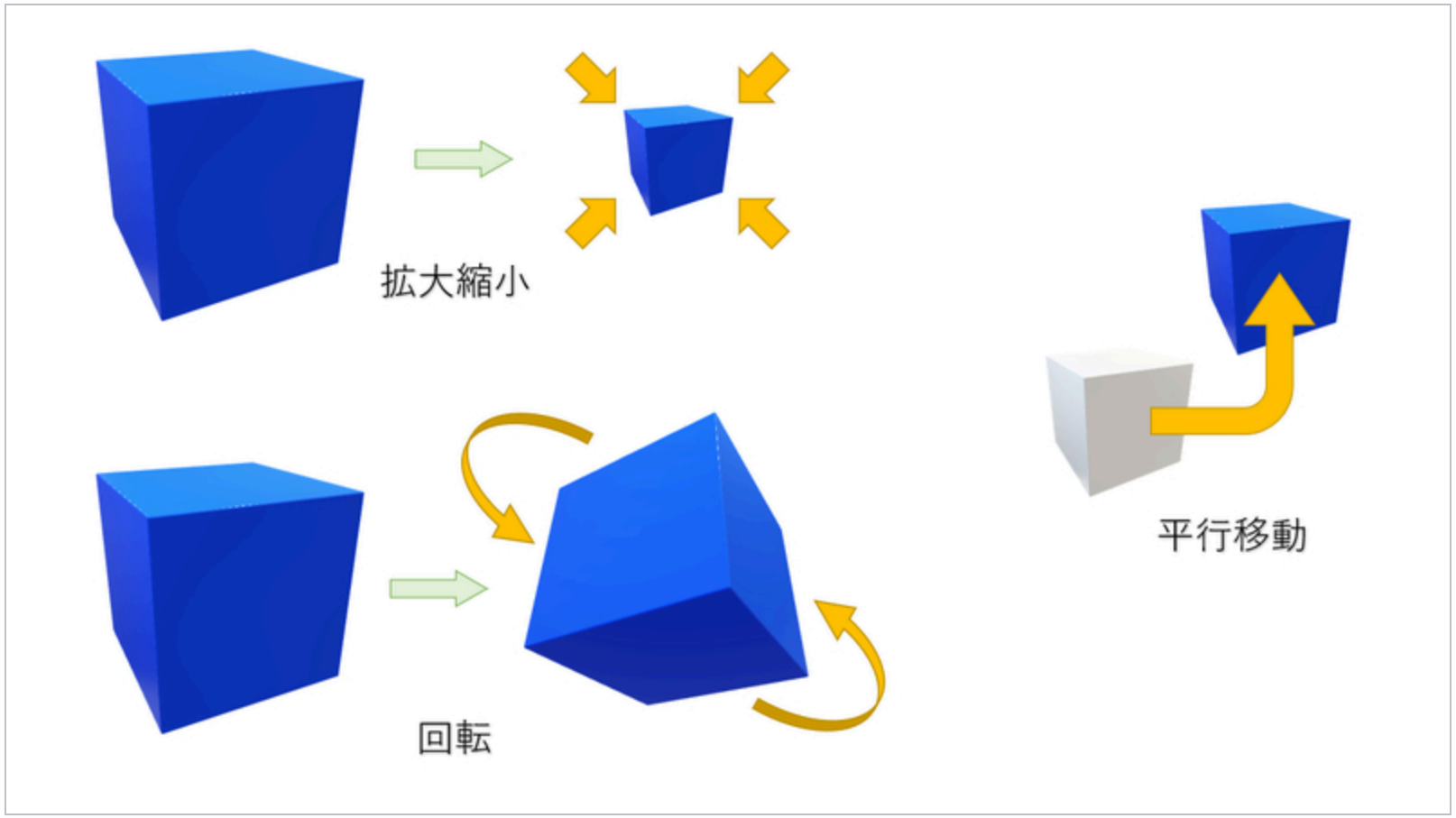
モデル座標変換行列

モデル座標変換行列は、一般によく「ワールド座標変換行列」と呼ばれているものと同じです。OpenGL ではモデル座標変換行列というふうに呼んでいますが同じものを意味しています。

この行列は、単位行列とは異なり、中身にどんな数値が入っているかは一定ではありません。

モデル座標変換行列が持つ「効果」は、ずばり「頂点を三次元空間内で動かす」というものです。

たとえば、左右や前後に頂点を動かす平行移動、あるいは頂点の回転、拡大縮小なども、すべてこの行列によって表現することができます。



いずれもモデル座標変換行列で作用させることができる

実際にプログラムでこれを利用するときは、当たり前ですがどのような移動や回転が適用されているかはそのときどきによって変わります。
（つまり単位行列とは異なり中身は一定ではない）

勘のいい人なら気がついたかもしれませんが、three.js で `Object3D` クラスのインスタンスが持っている `position` や `rotation`、`scale` などは、最終的にこのモデル座標変換行列になります。

ビュー座標変換行列

ビュー座標変換行列には「カメラ（視点）の位置を考慮した変換を与える」という効果があります。

考えてみれば当然のことなのですが、頂点がモデル座標変換行列によってどのように動いたにせよ、それを「どこにある視点で、どのような向きで見ているのか」によって最終的に画面にどんなものが描画されるのかは変化します。

このような「視点によってどういう影響が出るのか」を考慮した変換を掛けてくれるのがビュー座標変換行列です。

ビュー（View）という語感からも、それが視点や、その視点から見える風景に関係しているということが想像しやすいと思います。

プロジェクション座標変換行列

ここで紹介しておきたい行列の最後は、プロジェクション座標変換行列です。

プロジェクション、つまり、プロジェクターで投影したときのような「平面（スクリーン）に頂点を投影するための変換」を行ってくれるのがプロジェクション座標変換行列です。

3DCG は、頂点を三次元的な広がりのある空間に定義するものですが、その描画結果は常に「平面であるスクリーン」に描画されます。

この「スクリーンに投影するための変換（下準備）」を行ってくれるのが、他ならぬプロジェクション座標変換行列です。

> perspective か orthographic かは、この行列によって決まる

ここまでのまとめ

- 行列には、その効果や状態に応じた名前がついているものがある
- 単位行列は、他の行列と乗算してもなんの影響も与えない行列のこと
- モデル座標変換行列は、頂点を直接動かす効果を持つ（平行移動・回転・拡大縮小）
- ビュー座標変換行列は、視点に応じた変換を行ってくれる
- プロジェクション座標変換行列は、スクリーンに投影するための変換を行ってくれる

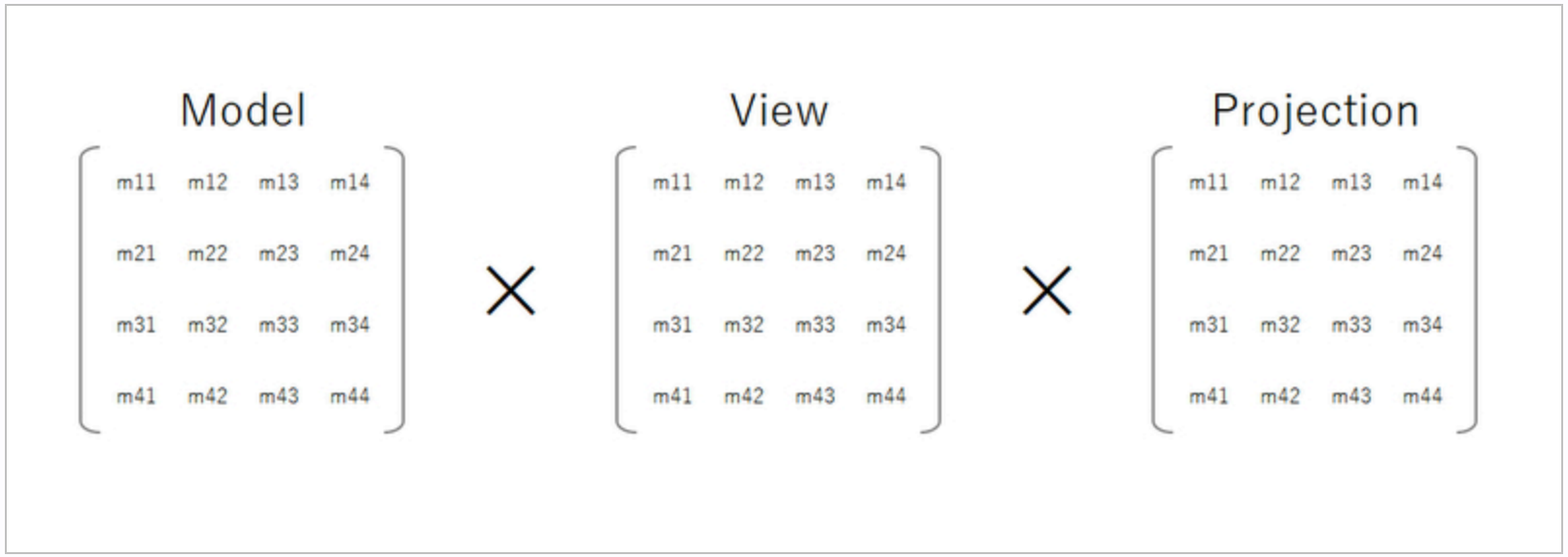
ここで登場した、モデル・ビュー・プロジェクションの 3 つの行列は、英語で表すと Model, View, Projection となり、これらの頭文字をつなげてやると MVP となります。

先程、uniform 変数の名前として `mvpMatrix` というのが出てきましたが、この変数名にある `mvp` は、まさにこのモデル・ビュー・プロジェクションを意味しています。

行列の乗算

行列は「乗算することで複数の効果を組み合わせて合成することができる」という特性を持っています。

この行列の特性を利用して、モデル座標変換行列とビュー座標変換行列、そしてプロジェクション座標変換行列をすべて掛け合わせてやるとMVP 行列を作ることができます。



掛け合わせることですべての行列の効果が１つに集約される（上図はあくまでも雰囲気を伝えたいためのイメージ図です）

この「行列同士を掛け合わせるとそれらの効果が合成される」というのが、最初はめちゃくちゃイメージしにくいのですが……

たとえば、関数を組み合わせているだけだ、というように考えるとまだ雰囲気がつかみやすいかもしれません。

```
function translate (V) { /* なんらかの処理 */ return 移動したV; }
function rotate    (V) { /* なんらかの処理 */ return 回転したV; }
function scale     (V) { /* なんらかの処理 */ return 拡大したV; }

const vertex = translate(rotate(scale([x, y, z])));
```

> これもあくまでも雰囲気を掴むためのイメージですが、行列を乗算していくということは、上の記述例で示したように効果がどんどん重複して適用されていくようなことが起こります

ここまでのまとめ

- モデル・ビュー・プロジェクションの頭文字で MVP
- これらをすべて掛け合わせたのが MVP 行列
- 行列は一種の関数のようなものであり……
- 掛け合わせるとそれらの効果が合成される
- まるで次々と関数を連続で呼び出しているかのように、１つの行列に効果が合成・集約されていく

行列の定義と合成

さて、MVP 行列がどのようなものか、なんとなく雰囲気がつかめたでしょうか。

ここからはサンプルを見ながら、実際のコードとしてはどんな雰囲気になるのか確認してみましょう。

`App.render()` のなかで、各種行列を生成しているところがあります。

これらは three.js でも似たようなパラメータを定義したりしたので、それぞれの意味をよく考えつつ、落ち着いて確認していきましょう。

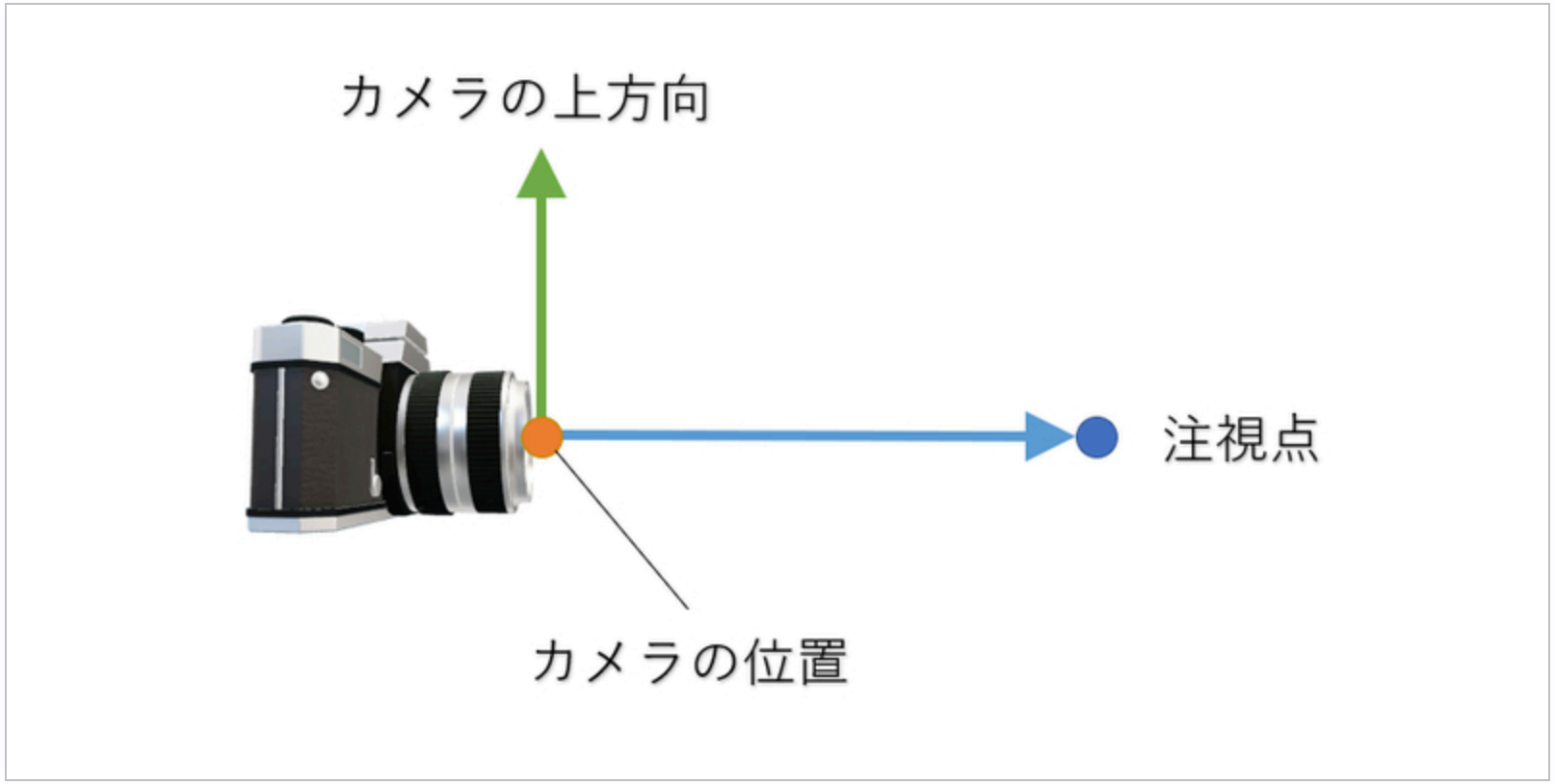
モデル座標変換行列

- ここでは回転を行っている
- 回転量を示すラジアン
- 回転軸ベクトル

> 平行移動、拡大縮小はここでは行っていない

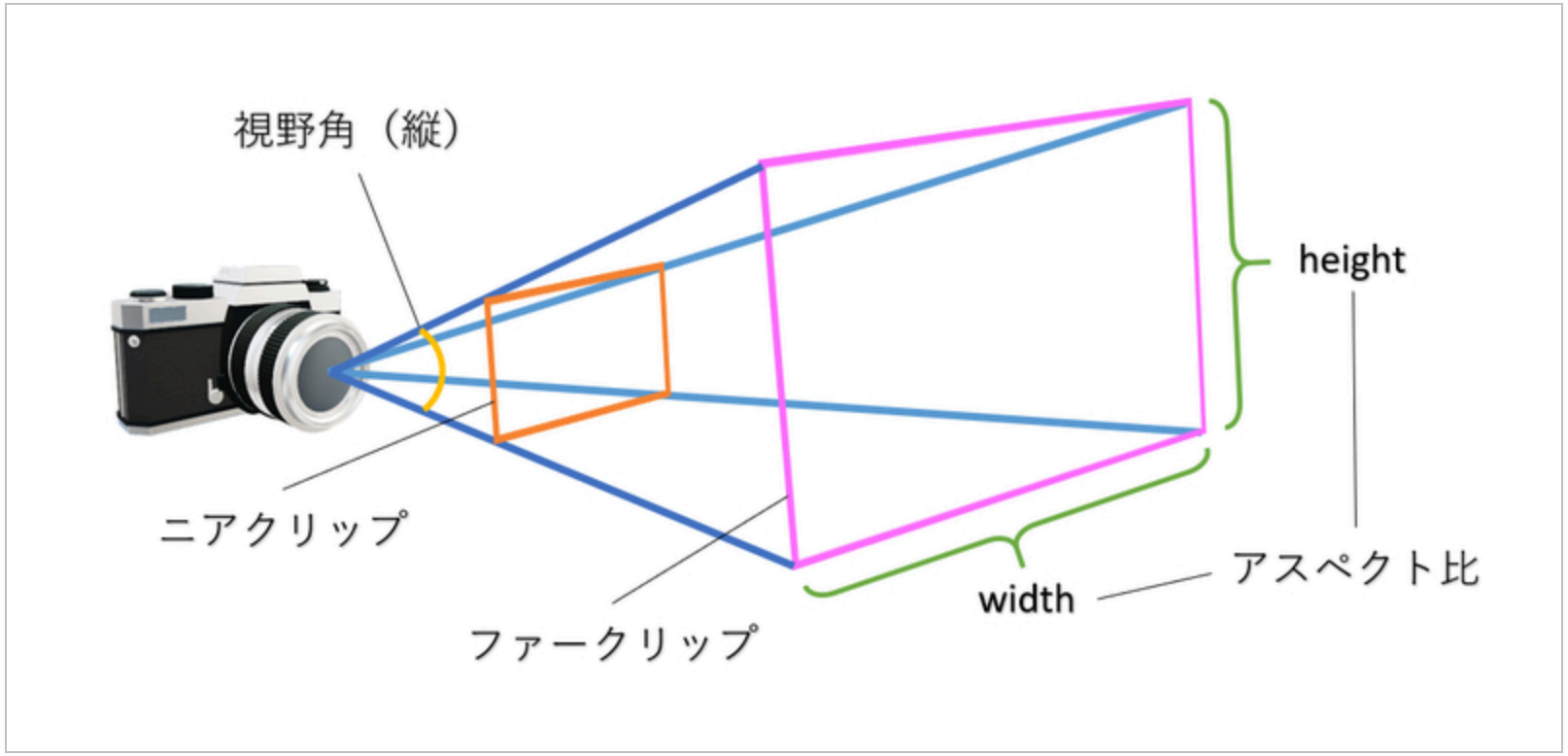
ビュー座標変換行列

- カメラの位置
- カメラの注視点（見つめている場所）
- カメラの上方向（カメラの上面がどちらを向いているか）


カメラの上方向
注視点
カメラの位置

プロジェクション座標変換行列

- 視野角（Y 方向）
- アスペクト比
- ニア・クリップ面
- ファー・クリップ面


視野角（縦）
height
ニアクリップ
width
アスペクト比
ファークリップ

行列同士を掛け合わせる場合は `Mat4.multiply` を使います。

このとき、行列を 掛け合わせる順序 に注意します。

また行列はいわゆるスカラーの場合と異なり交換法則が成り立ちません。

つまり、掛ける順序が変わると 最終的な効果も変わってしまう ので、十分注意しましょう。

行列の掛ける順序は、その行列が行優先なのか列優先なのか……みたいな話があるのですが以下のサイトを使って実際に乗算してみると概要がつかみやすいかもしれません。

Matrix Multiplication

> 行列の乗算方法は難しく考えすぎないようにし、もしわかりにくければいったんそういうものだと納得するのも今の段階ではいいと思います

```
// 行列を乗算して MVP 行列を生成する（掛ける順序に注意）
const vp = Mat4.multiply(p, v);
const mvp = Mat4.multiply(vp, m);
```

> WebGLMath は列優先なので、後ろから前に向かって掛ける

最終的に完成した MVP 行列はシェーダに送ってやり、頂点に対してこの行列を乗算できるようにしてやります。uniform 変数として行列をシェーダに送るときは `uniformMatrix4fv` などを用います。

```
// ロケーションを指定して、uniform 変数の値を更新する
gl.uniformMatrix4fv(this.uniformLocation.mvpMatrix, false, mvp);
gl.uniform1f(this.uniformLocation.time, nowTime);
```

頂点シェーダで頂点と行列を乗算する

JavaScript（CPU）側で MVP 行列を生成する手順はわかりましたが、そもそもどうして行列のような難しい数学の概念をわざわざ持ち出してくる必要があるのでしょうか。

答えを書いてしまうと「大量にある頂点に対して一律に同じ作用を与えるため」に行列の性質が適している、ということが言えると思います。

ジオメトリの定義にもよりますが、頂点の個数は数万単位の非常に大きな値になることがしばしばあります。

そういった状況で大量の頂点に対して一律に同じような変換を掛けるとき、複数の効果を１つの行列に詰め込むことができるという性質はとても都合がよいのです。

サンプルの頂点シェーダのソースコードを見ると、uniform 変数として受け取った行列を使って演算を行い、その結果を `gl_Position` に対して出力しているのがわかると思います。

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

> GLSL でも列優先なので、やっぱり後ろから前に向かってベクトルを掛ける

006

- 頂点を立体的に描画するには、行列を用いる
- モデル座標変換行列で頂点を動かし……
- ビュー座標変換行列でカメラを考慮した変換をし……
- プロジェクション座標変換行列でスクリーンに投影する
- 各行列が持つ効果は、行列の特性により乗算することで合成して１つにまとめることができる
- これを MVP 行列と呼び、シェーダ側で頂点と乗算する

バックフェイス カリング

深度テスト

バックフェイスカリングは、省略して単にカリングと呼ばれることもあります。

three.js ではマテリアルにある `side` プロパティに対してバックフェイスカリングを設定することができましたが、ネイティブな WebGL では WebGL のコンテキストに対してこれを設定します。

> つまり影響範囲が WebGL コンテキスト全体に及ぶ点に注意

バックフェイスカリングの設定例は以下のような感じ。WebGL コンテキストに対して `enable` メソッドを使って設定します。

ちなみに無効化する場合は `disable` メソッドを使います。

```
gl.enable(gl.CULL_FACE);  // 有効化
gl.disable(gl.CULL_FACE); // 無効化
```

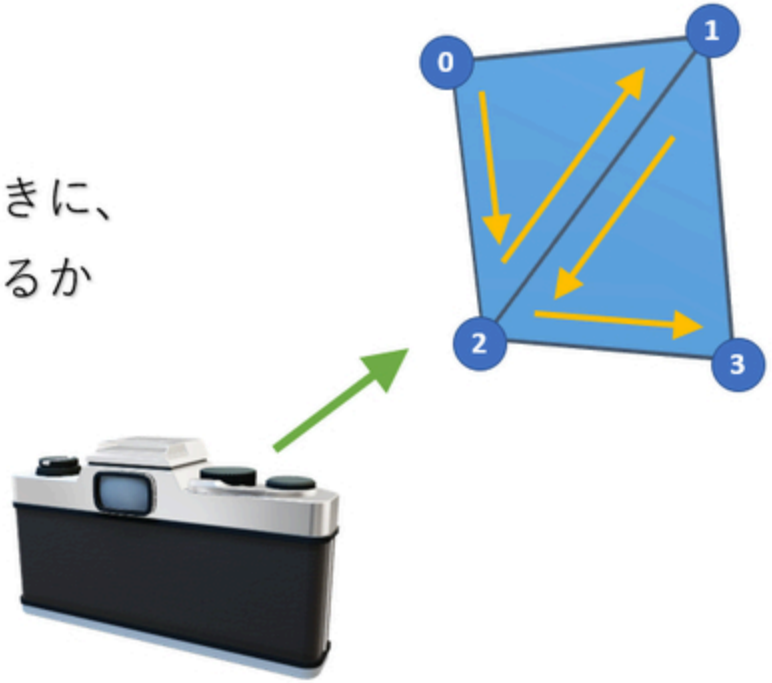
バックフェイスカリング（既定値は無効）を有効化すると、裏面ポリゴンが画面に描画されなくなる ということが起こります。

いままであまり深く言及してきませんでしたが、実は頂点を結んで作られる三角形ポリゴンには裏表の概念があります。

ポリゴンの表と裏は「どのような順序で頂点を結んだのか」によって決まります。

これは、カメラから見たときにどういう結び順で見えるか という意味です。

カメラから頂点を見たときに、
結び順がどのように見えるか



あくまでも、カメラから見てどう見えるかがカリングの判断基準

バックフェイスカリングの場合と同じように、WebGL のコンテキストに対して `enable` 等のメソッドで設定を有効化・無効化するものとして、深度テストがあります。

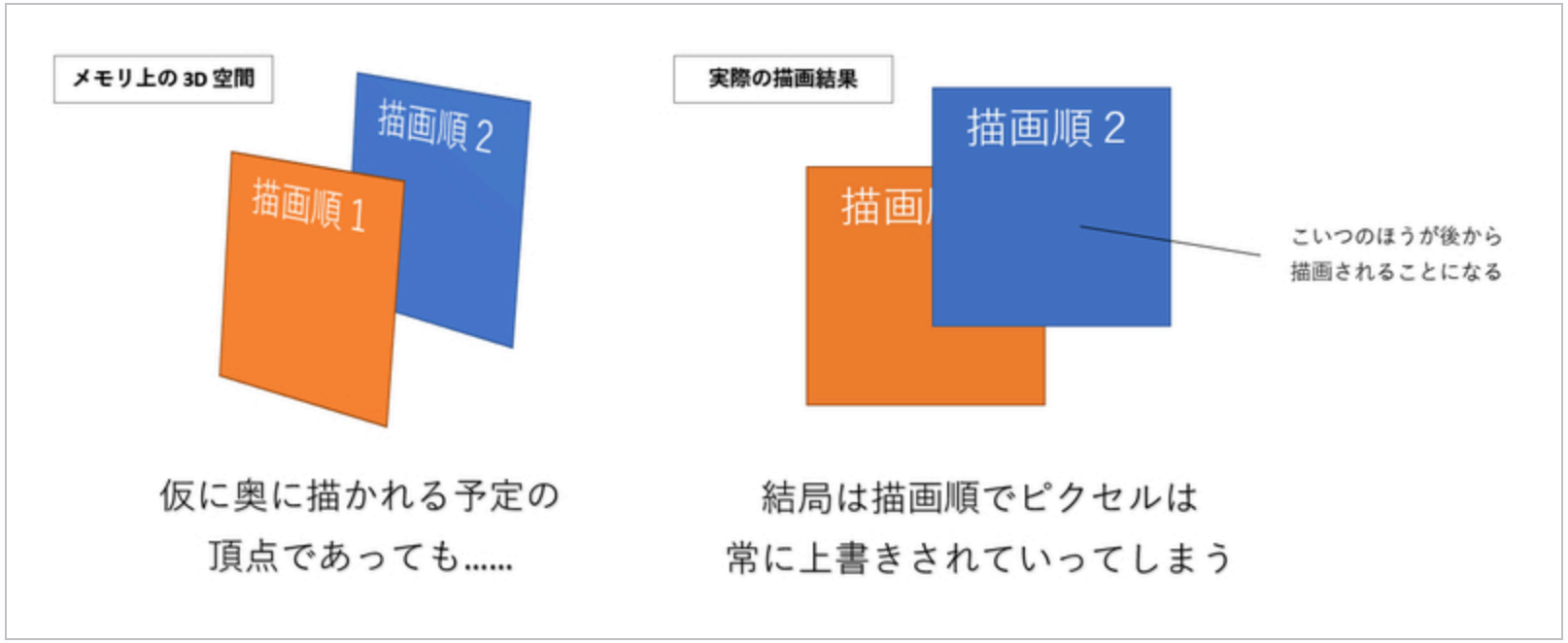
WebGL に限らずどのような 3D API にも大抵これと似た仕組みが備わっており、現代の CG における非常に重要な技術のひとつです。

深度テストは、フェイスカリングと同じように WebGL の既定では無効化されています。

意図的に有効化された場合に限り深度テストは有効になり、深度バッファと呼ばれる「深度値を書き込むためのバッファ」を活用した前後関係のチェックが働くようになります。

それでは仮に、深度テストを有効化していない場合は、前後関係ってどのように処理されるのでしょうか。

実は、WebGL のような 3D 描画を行うことができる API であっても、基本的には HTML の DOM のレンダリングなどと同じように 最初に定義されているものから順番に 描画が行われます。ここでいう「定義されている順番」というのは、VBO に格納されている「頂点情報の定義順」です。



深度テストが無効だと、奥行きとは関係なく描画順によってピクセルは上書きされてしまう

007

- WebGL のコンテキストに対して設定するいくつかのフラグは `gl.enable` や `gl.disable` を使って有効化・無効化する
- バックフェイスカリングを有効化すると裏面が描画されなくなる
- 面の裏表は、カメラから見た頂点の結び順によって決まる
- 深度テストを有効化すると奥行きが考慮されるようになる
- 深度テストはピクセルレベルで行われ、テストに通れなかった（遮蔽物がある）場合はピクセルが破棄される

> 深度テストが行われるのは、レンダリングパイプライン的にはフラグメントシェーダ実行よりも後です

カメラ実装について（オマケ）

さて、行列を用いて頂点を 3D で処理することができるようになり、さらにシーンが 3D になったことで深度テストやカリングといった新たなトピックも登場し、いつものことですが初めて聞くような言葉がたくさん出てきて認知負荷が高まっていますね……

ただこれに関してはちょっとずつ慣れていくしかありませんので、最初はとにかく「鉄板な設定はこう！」というように決め打ちしてしまうのもひとつの手です。

> 何度も繰り返し勉強することで記憶はどんどん定着します

また、3D 空間を扱うようになると、自分が習作として作った実装が正しく動作しているのかなど、検証する作業もちょっと大変になります。

もう少し具体的に言うと、カメラをマウスで操作できるようにとか、そういった UX の部分を作っておいてやらないと、検証するのさえ一苦労という状況になりがちです。そこで、ここでは技術的な説明はしませんが、私のほうで実装したマウスコントロールを導入しています。

カメラをマウスで操作できる、というのは 3D ツールでは当たり前すぎる機能なので軽視されがちですが、自分でやってみると本当に……いろいろと難しい点が多いです。

もし、3D プログラミングの中級者以上を目指すのであれば、自作のカメラ実装を行ってみると一気にレベルアップできると思います。行列やベクトル、クォータニオンを駆使してがんばる感じなので、仮にそれができなかったとしても肩を落とす必要はまったくありません。あくまでも、チャレンジできそうならやってみる価値があるよ！ というような感じのものですね。

008

- three.js の OrbitControls みたいな感じのカメラ実装
- 左ボタンドラッグ操作で、注視点を中心に回転する
- 右ボタンドラッグ操作で、現在の視点を平行移動する
- ホイール操作等で、ズームイン・アウト
- `camera.js` に実装が書いてあります

> あくまでも参考程度に

ライトと陰影効果

さて、続いては立体感を演出するのに欠かせない「ライトによる照明効果」を実装してみましょう。

three.js ではライトオブジェクトをシーンに追加するだけで簡単に照明効果を実現できましたが、ネイティブな WebGL API の実装では、どのようにこれらの処理を行っていけばよいのでしょうか。

なんとなく想像がつく方もいるかもしれんが、ピュアな WebGL の場合のライトは、シェーダを利用するなどして自前で計算を行い、なんとかする しかありません。

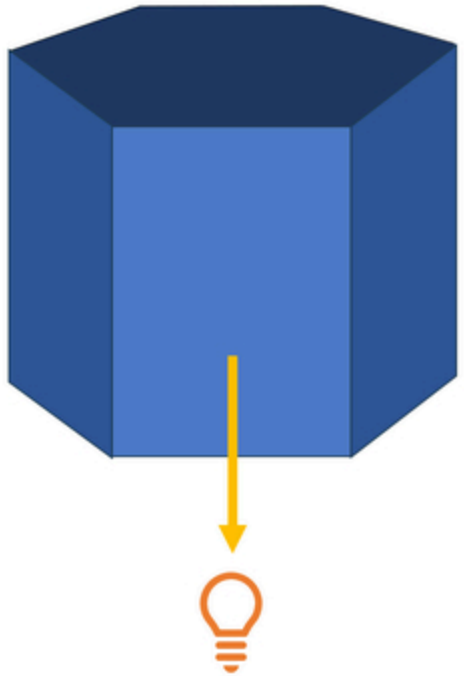
WebGL では固定機能パイプラインはありませんので、勝手に都合よく陰影がついたりはしないのです。

> 陰影を付けたければ、そういうシェーダを書く必要がある

とは言え、いきなりそんなことを言われても、どうやってこれを実現するのかは想像が難しいと思います。（私は難しかったです）

そこでたとえば、次のような状態を思い描いてみてください。まずは、六角柱の正面から平行に光が真っ直ぐに降り注いでいる状態をイメージします。

> 次ページの図を見ながら考えてみましょう



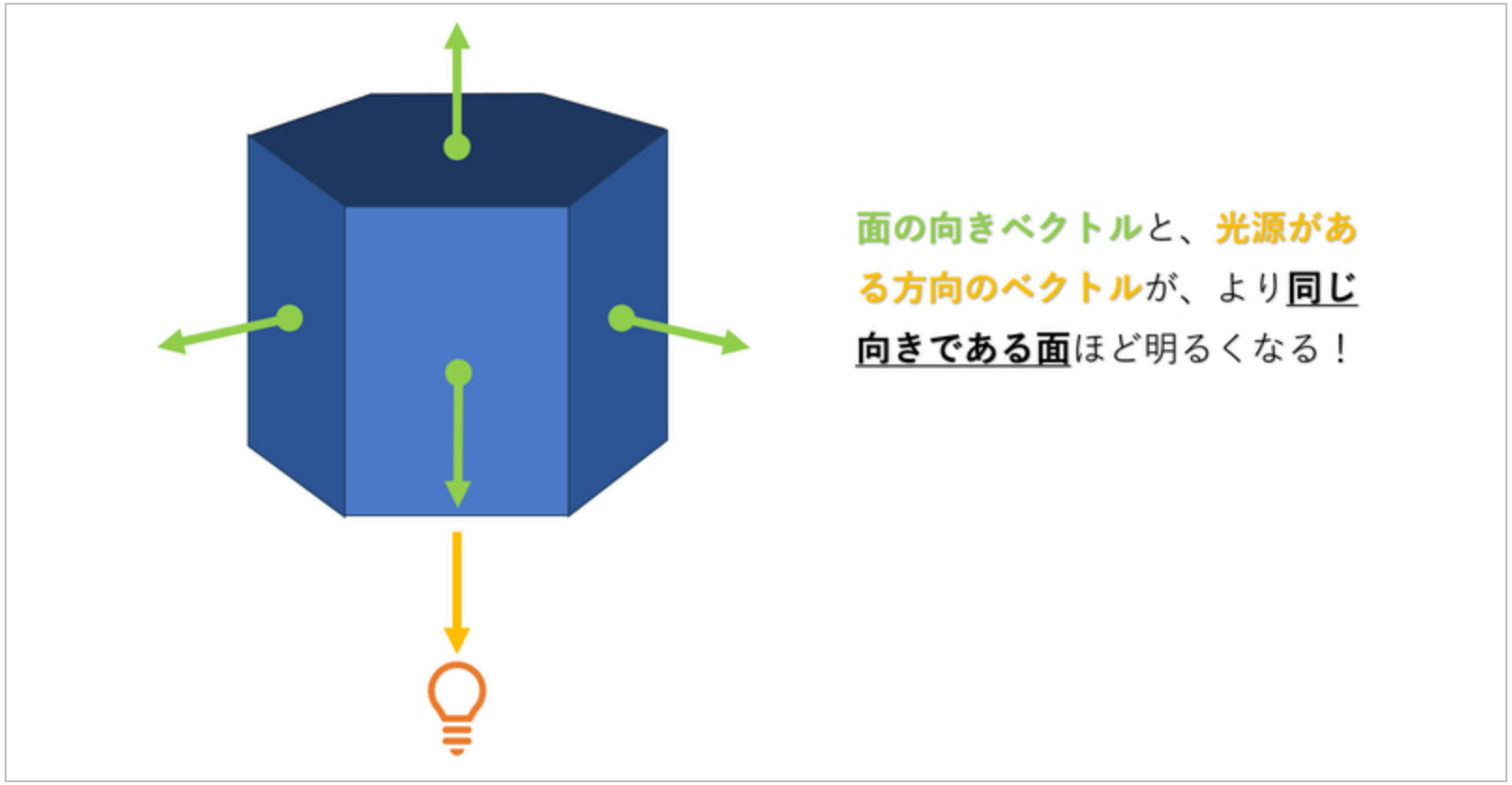
光源が黄色矢印の方向にあるとして、六角柱の各面の明るさに差が生じるのはなぜだろう？

それぞれの面ごとに明るさに差が出るのはどうしてでしょう？

どうして六角柱の各面は、陰影の付き方が違うのでしょうか。

暗くなる、明るくなる、の指標っていったいどこにあるんでしょう。面の大きさ？ 面の形？ それとも……？

> 答えは次のページに



「面の向き」と「光の向き」こそが明るさを決める要因になっている！

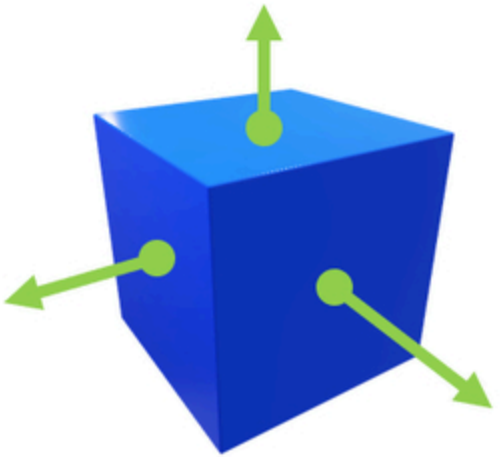
つまり、照明効果を実現するためには「面がどちらに向いているのか」という情報を使って、「光と面の向きとの、相対的な関係」を考えてやるわけです。

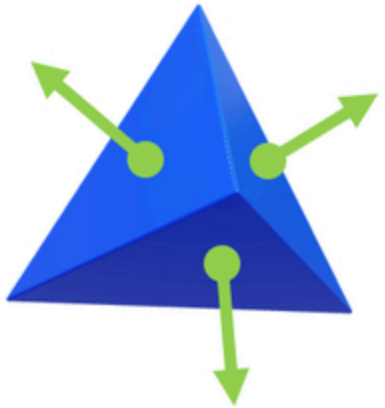
そして、この「面の向き」を表すために用いられるのが **法線** です。

法線は、向きだけを表すために用いられるベクトル です。← 重要！

法線や、あるいは頂点法線のようなキーワードは 3DCG の分野では頻出する単語なので、すでにその名前を聞いたことがあるというひとも多いかもしれません。

> 英語では Normal と書きます

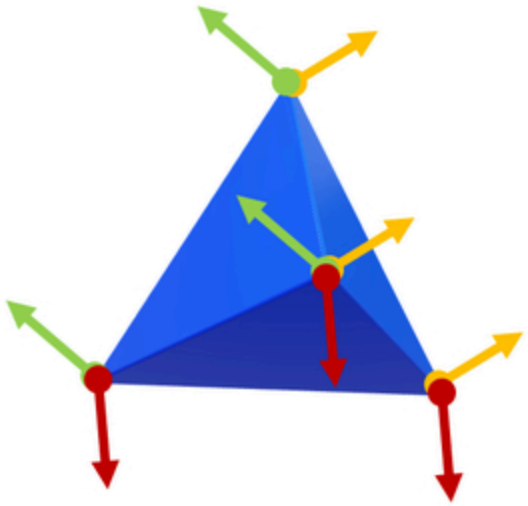




面に対して垂直な向き、これが面法線

ただし、少し思い出してみてほしいのですが、3DCG の世界では描かれるのは常に「頂点のみ」でしたよね。ですから、通常は 頂点に法線という頂点属性を持たせてやる ことでシェーダへ法線を持ち込みます。

つまり、これまで頂点に座標、色、などを持たせてきましたが、ライトを実現するためにはこれらに続き「法線という頂点属性」を頂点に持たせてやることになります。



色の場合と同じように、法線もやっぱり頂点同士で補間されてしまうため、面の法線を完全に一致させるためには、面を構成する頂点すべてが同じ法線を持っている状態にしてやる必要がある。

※つまり同じ座標に複数の頂点を置くことになる

補間された法線でも問題ないケースもあります（たとえば球）

話が複雑になってきたので、ここまでの内容を簡単にまとめます。

- ライトを実装するには、なんらかのロジックで陰影を計算しなければならない（魔法のような力で勝手に陰影がついたりはしない！）
- 「面の向き」と「光の向き」の関係を用いてやればよさそう
- 「面の向き」を表すベクトルを「法線」と呼ぶ
- なんとかしてシェーダ内に法線を持ち込みたいので、頂点属性として法線を定義し、VBO 経由でシェーダに渡す

先ほども書いたように、頂点法線とは要するに「向きを表すベクトル」です。

今は三次元空間上にある頂点を扱っているので、向きに関するベクトルである法線も、やはり XYZ の 3 つの要素を持つベクトルとして定義します。これを踏まえて、サンプル 009 の「頂点シェーダのコード」をまずは見てみます。

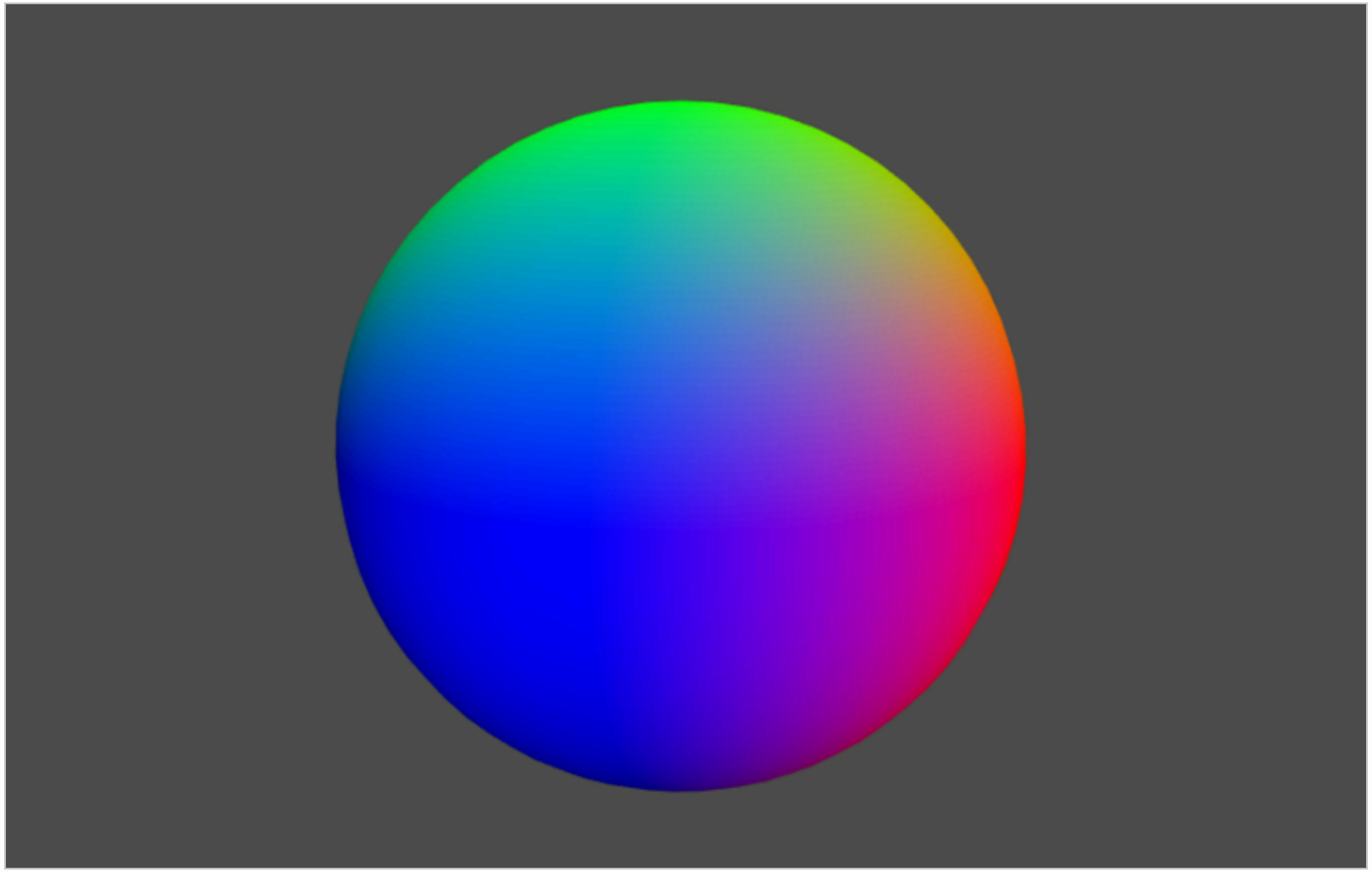
```
attribute vec3 position;
attribute vec3 normal; // 頂点法線 @@@
attribute vec4 color;
```

> vec3 型で attribute 変数に法線が追加されている

ちなみに、本当に法線が正しくシェーダに渡っているのかを「目で見て確認」することも慣れるとある程度はできます。

要は色として出力してしまい、その結果を見ます。

> 法線は本来は色ではなく単なる向きベクトルですが、それをあえて RGB に出力することでベクトルの内容を脳内補完する感じですね



XYZ を RGB に置き換えてじっくり観察してみると、たしかにそういう色の付き方をしているのがわかるのではないでしょうか。

法線とライトベクトル

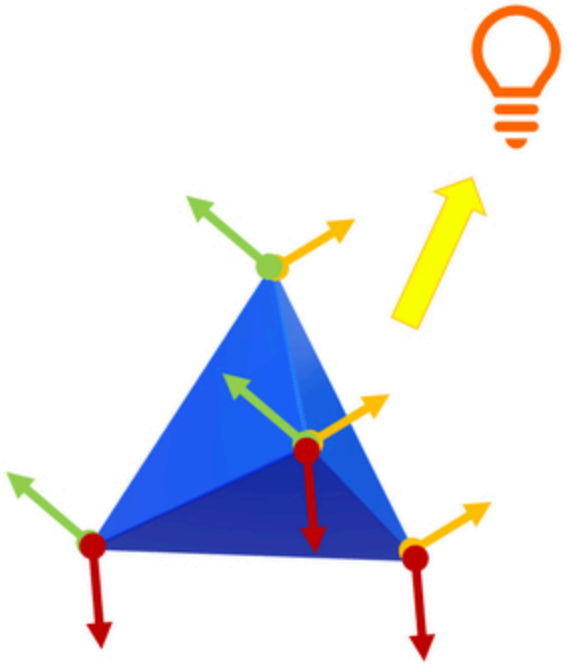
法線を色として出力するなどして、無事にシェーダに法線を渡すことができているのを確認できたら……

続いては、これを実際にライトの計算に利用していきます。

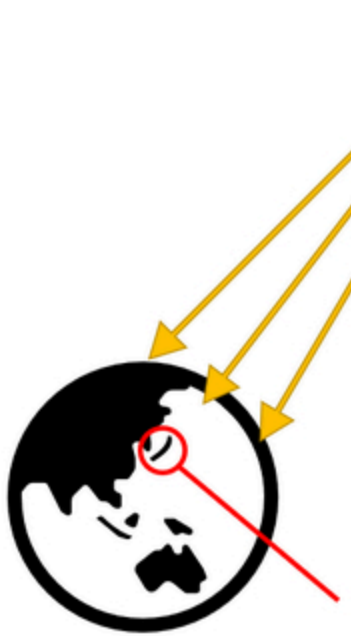
先ほど六角柱の例で図解したときの様子を思い出してみると、法線だけではなく「光がどちらから降り注いでいるのか」という情報が同時に存在しなければ陰影計算を行うことはできないことがわかると思います。

この光の向きは一般に ライトベクトル と呼ばれます。

> 光の向きであるライトベクトルと法線とを比較して色付けする



最初はちょっとイメージするのが難しいのですがライトベクトルは「光源が始点となるベクトル」**ではなく**、「光源へと向かうベクトル」で考えるのが一般的です。
この**「光源へと向かうベクトル」と「法線」**を利用してライティングの陰影を計算します。



光源が十分に離れた距離から降り注いでいる場合、
光線の傾斜は限りなく小さくなってしまう。
つまり、地表では太陽の光はほぼ平行に降り注い
でくる。これを模したのが平行光源。

地表レベルでは太陽光は、もはやほぼ平行になってしまう

ライティングには様々な手法がありますが、最もシンプルで簡単な「平行光源によるライティング」では、ライトベクトルが常に一様であると考えて陰影付けの処理を行います

また、ライトベクトルは常に単位ベクトルであるべきです。

なぜなら、しつこいようですが「ライトベクトルは法線と同じように向きだけに注目すればいいもの」だからですね。ですからシェーダ内で、ベクトルを単位化してから利用します。

それらのことを踏まえて、サンプル 009 のシェーダのソースコードを見てみましょう。

ここで出てきている `normalize` という GLSL の関数が ベクトルを単位化する 関数です。

```
// 法線とライトベクトルで内積を取る @@@
float d = dot(normalize(normal), normalize(light));
```

ここで出てくる `dot` は、ベクトルの内積 を計算できる GLSL の関数です。

内積ってなんだったっけ？ という方もいるかもしれませんが…… 実際はすごく簡単ですので過度に恐れおののく必要はありません。

内積の計算方法をおさらいすると、以下のような感じ。

```
V = [x1, y1, z1];
W = [x2, y2, z2];
V・W === (x1 * x2) + (y1 * y2) + (z1 * z2);
```

> 内積では XYZ をそれぞれ掛けて全部足す

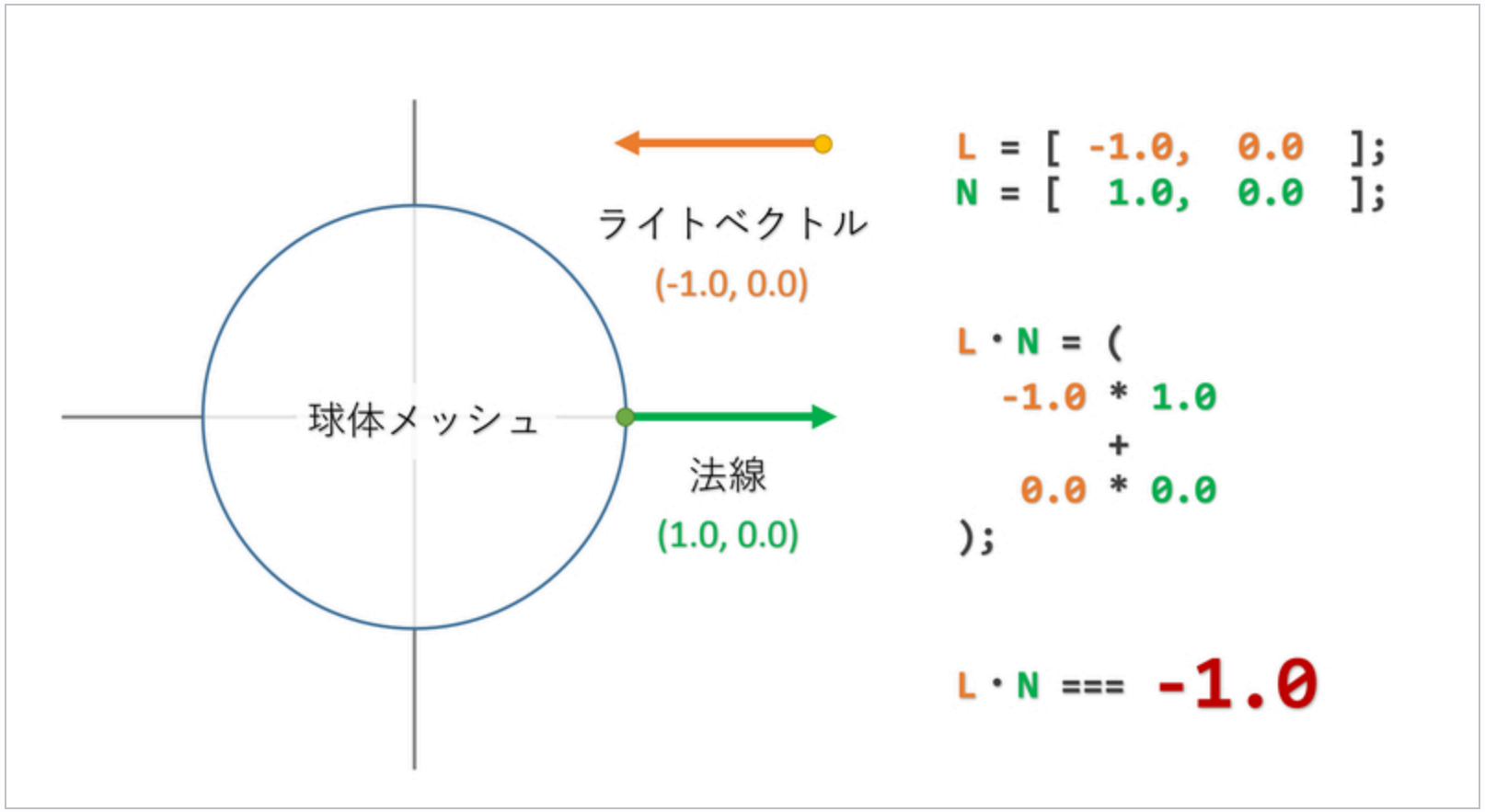
さて、この内積の計算方法だけを見ると、なんでこれでライトが実装できるのかちょっと不思議に感じますね。

ここでは、法線もライトベクトルもシェーダ内で **単位化している** ということが本質を理解するための重要なポイントになります。

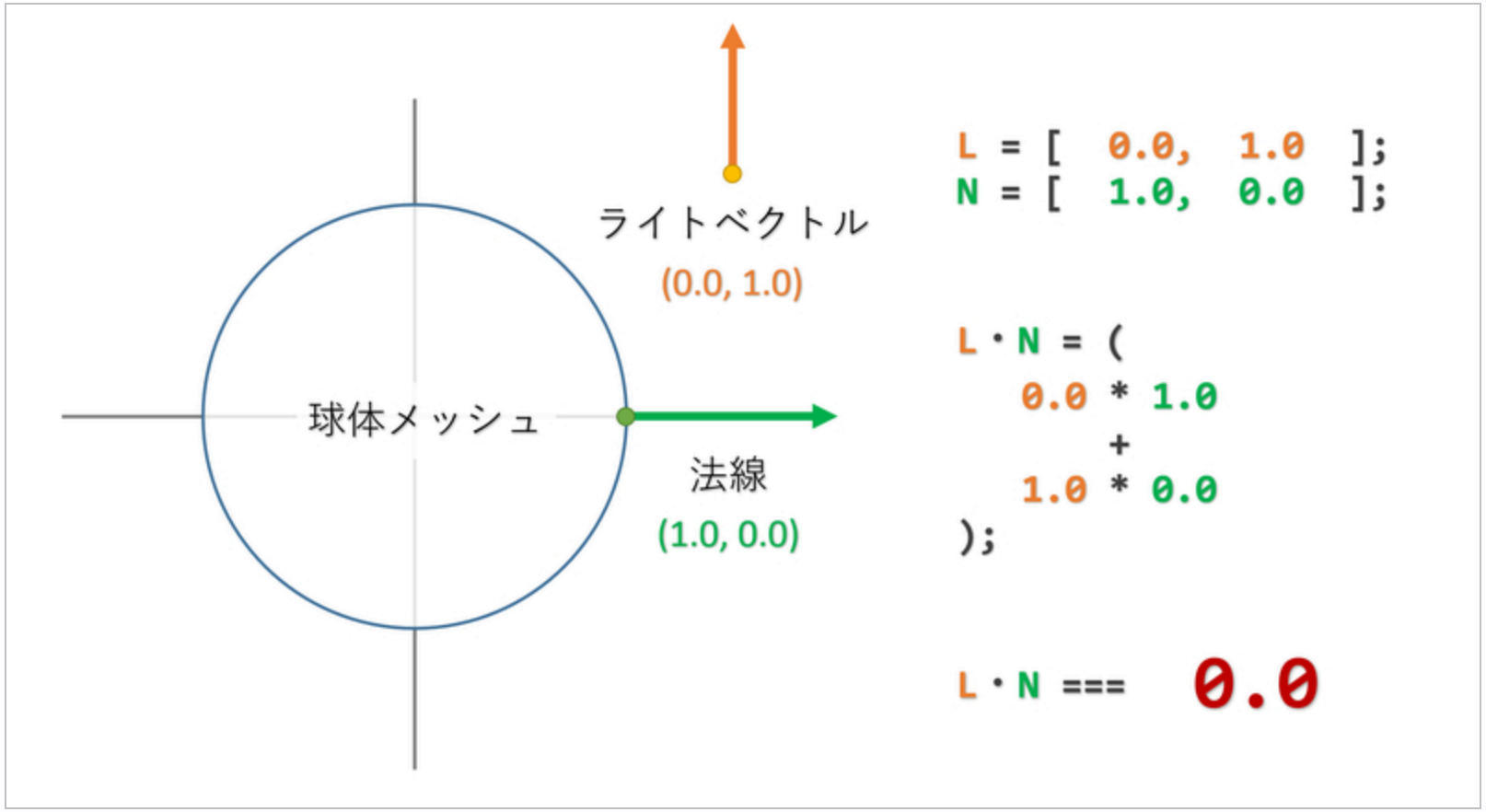
こういうことを考えるときは、なるべく単純なケースを考えてみましょう。

たとえば、法線はまっすぐプラスの X 方向に、ライトベクトルがまっすぐマイナスの X 方向を指し示したベクトルになっていると考えてみます。

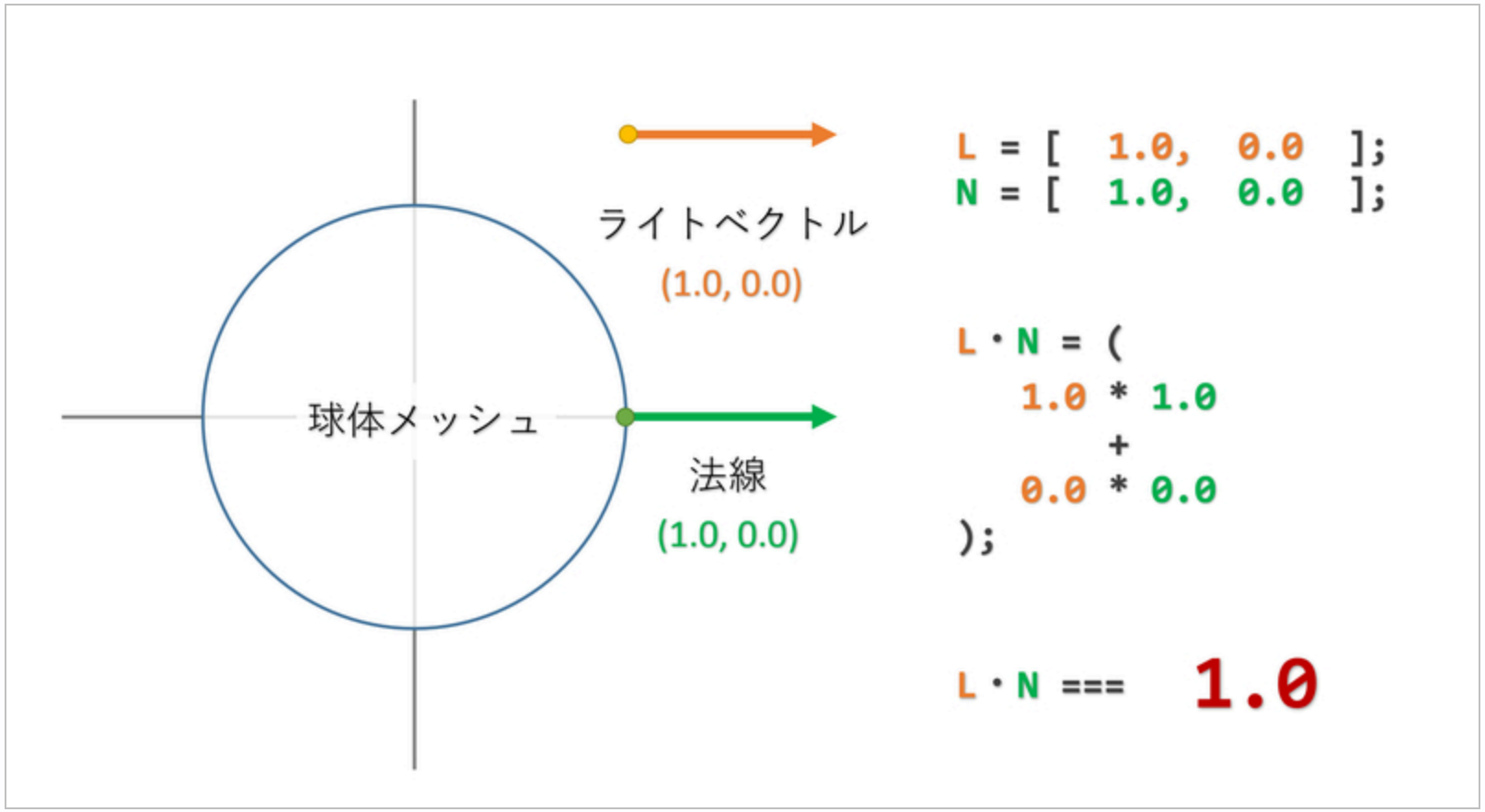
> 話をわかりやすくするために、二次元で図解しつつ考えてみます



ライトベクトルと法線が完全に向かい合っている場合。内積の計算方法に当てはめてみると結果は `-1.0` になる。



それでは次に、こんなケースではどうでしょうか？ ベクトルが互いに垂直になるような向きの関係になっている場合。



最後は、ベクトルが完全に同じ向きになっている場合。この場合、内積の結果は `1.0` となりました。

さて、どうでしょうか。

どうして内積でライトが実装できるか、ちょっとずつ見えてきませんか？

ライトの実装では、内積が持つ **ベクトルがどの程度同じ方向か** を調べられる性質を利用して、ライトの光の当たり具合をシミュレーションしています。

全く同じ方向を向いている正規化されたベクトル同士の内積は 1.0 になります。逆に、正反対の方向を向いているベクトル同士なら、結果は -1.0 になるんですね。

> 内積のこの性質がたまたまライトのような効果を生んでいるだけ！

009

- ライトベクトルはすべての頂点に対して等しく同じ効果を及ぼす
- つまりライトベクトルは attribute ではなく uniform で定義するか、または定数で定義する
- 法線やライトベクトルは向きだけに注目したいので必ず単位化する
- 光のあたり具合いはライトベクトルと法線で内積を計算して求める
- 現実の光を物理的に再現しているのではなく内積の性質を利用してそれっぽい陰影をつけているだけ

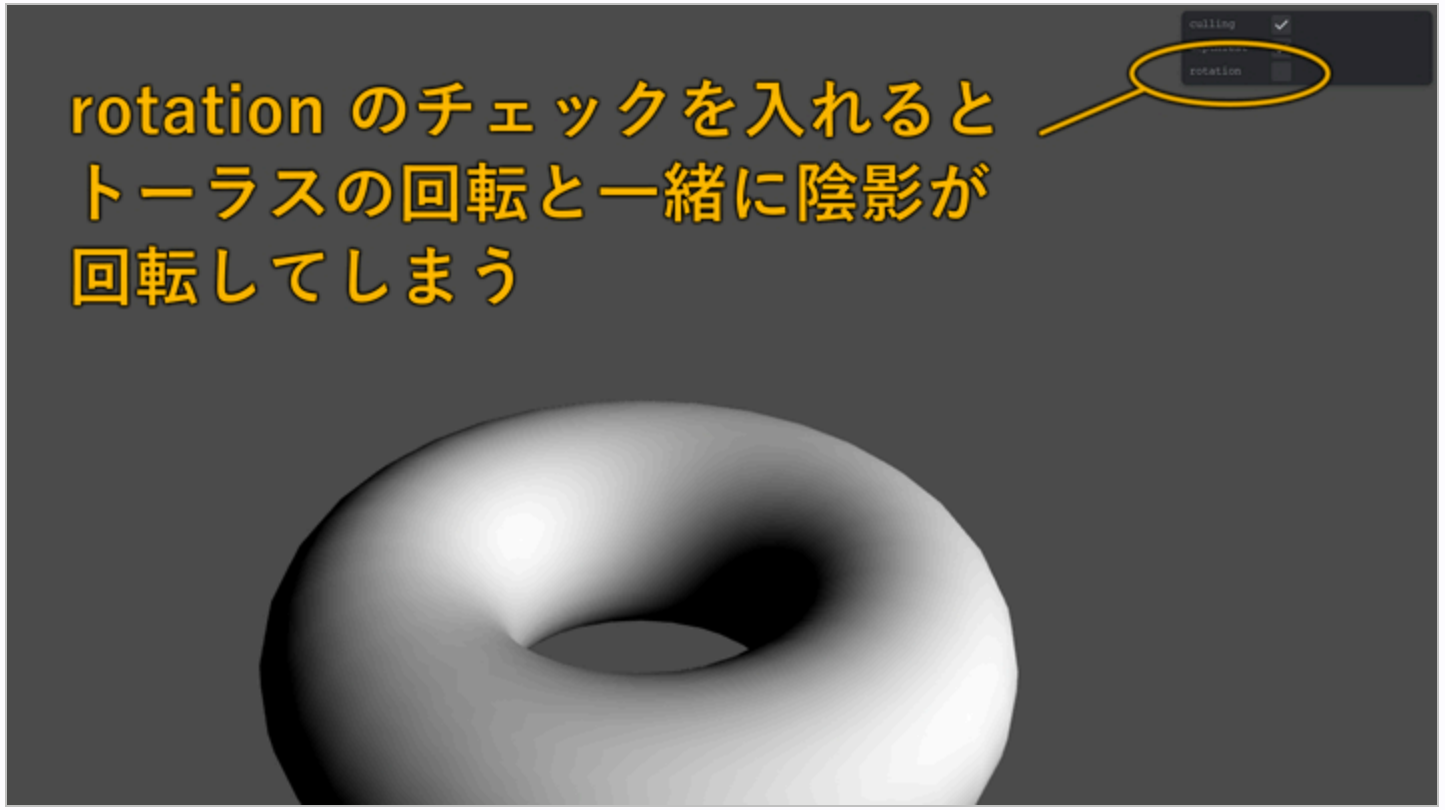
法線とモデル座標変換

さて、ベクトルの内積を用いることで、まるでライトで照らし出したかのような陰影が表現できました。

three.js でやる場合と違い、ピュア WebGL の場合はライトもシェーダを使って自前で実装してやらなくてはならず、かなり大変ですね……

しかし、サンプル 009 のライトの実装って実は不完全な状態です。

どういうことかと言うと、モデルを回転させたりした場合に、おかしなことが起こってしまうのです。



ライトは動かしてないのに、動かしたのと同じことが起こってしまう

なぜこのようなことが起こるのか、私は WebGL を学び始めた当初、まったく意味がわからなかったです。

だから経験的に、みなさんもなんでこんなことが起こっているのか最初はイメージしにくいんじゃないかなと思います。でも実はとても単純なことが起こってるだけです。

描画される 3D モデルが回転している、という状態は、実際には頂点がモデル座標変換の作用により回転している状態です。

これは頂点シェーダ内でモデル座標変換の効果を含んでいる MVP マトリックスを頂点座標の変換に使っているからですね。

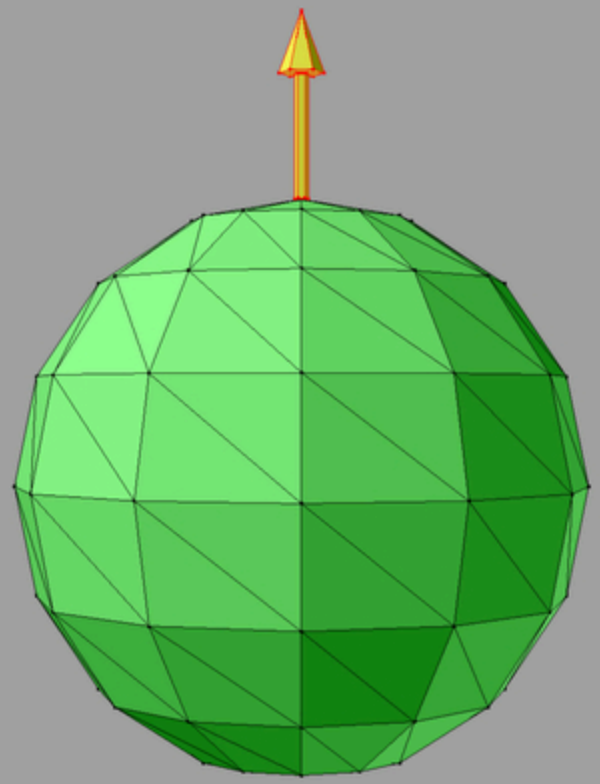
```
// 頂点座標（position）と行列を演算している（ここに回転の作用が含まれる場合がある）
gl_Position = mvpMatrix * vec4(position, 1.0);
```

頂点の「座標（つまり `position`）」については、MVP 行列と演算を行っているため変換されていますが、よくよく考えてみると、009 のサンプルでは法線に対してはいかなる処理も行なっていません。

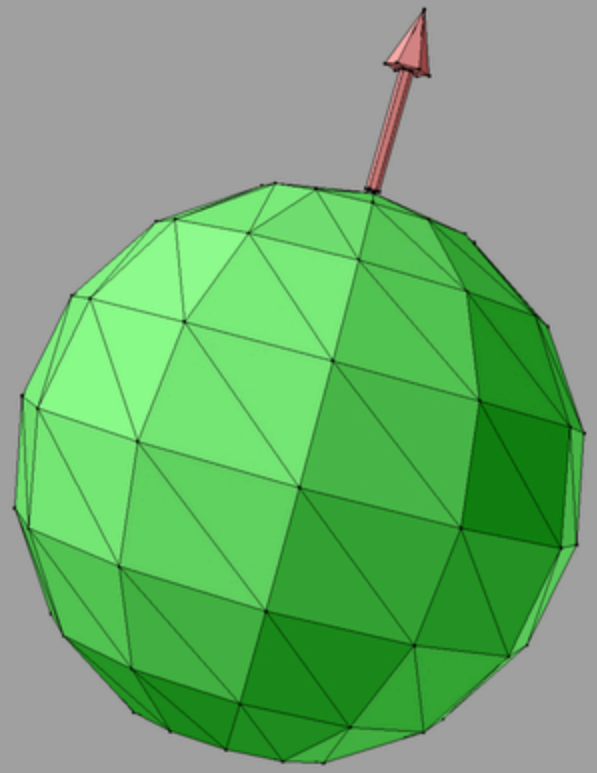
つまり、頂点の「座標位置」は動いているのに 法線はそのまま元の値を維持している 状態になっています。

> たとえば球体の北極点の位置にある法線で例えて考えてみると……

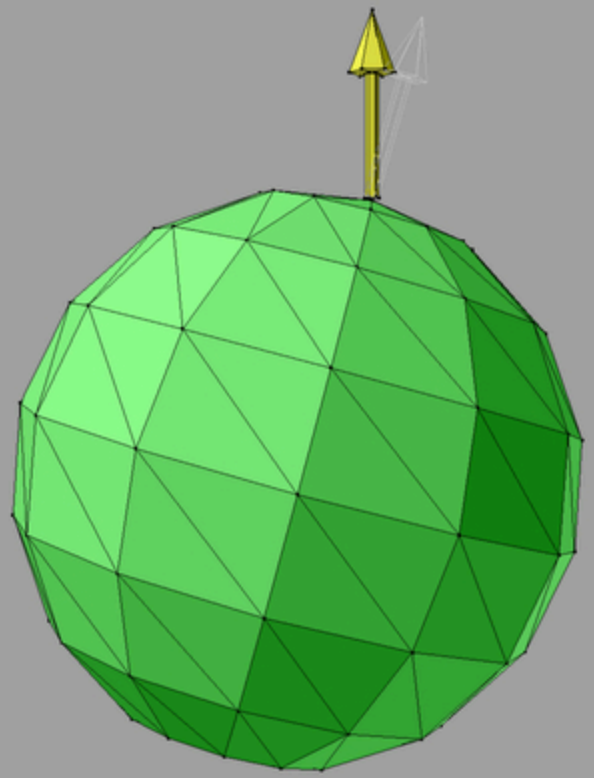
北極の位置の法線は
真上を向いているはず



球が回転したら、理想的には
法線もそれに合わせて
回転してほしいが……



実際には法線は一切変換を
掛けていないので、この図の
ようになっている



ライトが回転しているように見えましたが、実際は 法線が変換されていないのでライトの影響が固定された状態 になっているのです。

これを解消するには、頂点がモデル座標変換で移動したり回転したりしても問題が起こらないように、法線のほうも同時に（破綻しないように）変換してやらないといけません。

モデル座標変換の影響に合わせて、法線を正しい向きへと変換する……これは「法線を変換するための行列を定義」して行います。そのために必要な行列は モデル座標変換行列から生成 します。

これは 3D 特有の変換なので学術的な名前は（たぶん）ありませんが、よく「Normal Matrix」というふうに呼ばれます。

法線変換用の行列を生成する具体的な方法ですが、モデル座標変換行列の逆転置行列 を使います。

モデル行列の、転置行列の、逆行列です。

> 文章で書かれてもなんのこっちゃという感じですが……

ちなみに、逆行列というのはいわゆる一般に「逆数」と呼ばれる数のように、A に B を掛けて C になるとき、C に掛けると結果が A に戻る、というような効果を得られる行列のことです。

行列 A を乗算して結果が B になるとき、B に A の逆行列を掛けるともとに戻る、という感じ。`math.js` 内に `Mat4.inverse` として逆行列を生成するメソッドが実装してあります。

なお、行列の転置とは、行と列を入れ替えた状態にすることを言います。`math.js` に `Mat4.transpose` が実装されており、これを用いて転置できます。

```
// モデル座標変換行列の、逆転置行列を生成する @@@
const normalMatrix = Mat4.transpose(Mat4.inverse(m));
```

> モデル座標変換行列を、逆行列にしてから、転置する！

シェーダ側で uniform 変数としてこの法線変換行列を受け取り、これを頂点法線の変換に使います。

```
// 頂点シェーダ内で法線を変換
vNormal = (normalMatrix * vec4(normal, 0.0)).xyz;
```

なぜこれで正しく法線の向きが補正されるのかは、行列についてしっかり勉強していくとわかるのですが……

よくわからない、という場合はここでは深く考えず、いつものように行列は関数のようなものだとひとまず考えておいてもまったく問題はありません。

010

- モデル座標変換の影響は頂点の座標だけでなく法線にも適用しなければならない
- ただし普通にモデル座標変換行列を法線に対して適用しても正しい結果を得ることはできない
- 法線変換用の行列はモデル座標変換行列から生成する
- モデル座標変換行列の逆転置行列を使えば正しく法線を変換できる
- シェーダ内で法線を変換してから、変換後の法線で陰影計算を行う

さいごに

さて、今回は初めて本格的に GLSL を使ってシェーダを書く形になりましたので、かなり新鮮な体験ができたのではないでしょうか。

その分、ちょっと普段使わない思考が多く、難しく感じた方も多かったかもしれません。

ベクトルを行列で変換したり、ライトの陰影計算を行うためにベクトルの内積や正規化（単位化）が出てきたり……

スクールのこれまでの講義で少しずつ扱ってきた題材が、ここへきて一気に相互に関連しあいながらグラフィックスを描く処理に使われています。少しずつでも構いませんので、丁寧に復習していきましょう。

今回の課題は、ちょっと変化球的な内容になりますが……

頂点シェーダで行っている陰影計算を、フラグメントシェーダ上で行うにはどうしたらいいかを考えて実装してみましょう。varying 変数をうまく利用して、必要な情報をフラグメントシェーダへと正しく渡せるかどうかがカギになります。

答え合わせは、次回の講義の最初でしっかり行いたいと思いますので、
もしうまくいかなくてもくれぐれも落ち込まないようにしてください！

簡単すぎてモノ足りないぜ！　という方がもしいたら、点光源の実装に
挑戦してみても面白いかもしれません。