WebGL 基礎 .03

行列による座標変換とライティングの基礎

Chapter.01

座標変換とは

頂点の座標変換

WebGL に限らず、一般に 3D プログラミングでは頂点を定義して三次元空間上になにかしらのモデルを描画する。頂点は XYZ のユニークな座標を持っており、当然これは三次元空間にマッピングされる。

しかしここでよくよく考えてみると、その三次元にマッピングされた頂点は、最終的に「二次元のモニター上に描かれる」ことになる。

三次元から、二次元へ、次元を超えた変換がどこかで行われていることがわかる。（なんだか壮大……）

頂点の座標変換

3D プログラミングにおける座標変換は、この三次元空間からスクリーンという二次元空間への変換が主である。そして、この座標変換にはいくつかのステップがあり、これを行うためにしばしば行列が使われる。

ここからは、各ステップに合わせて利用される行列、あるいは座標系などについて見ていく。

ローカル座標系

ローカル座標系、という言葉がある。

これは、一切変換などを行っていない、素の頂点が置かれている座標空間を表す言葉である。

たとえば、モデリングソフトから出力した頂点の座標は、プログラムで読み込んだ直後は、一切の変換などが行われていない状態である。これが、ローカル座標。ローカル座標は、単純に各頂点が原点から相対的にどのような位置にあるのか、という情報だけを持つ。

ローカル座標空間からスクリーン空間へ

いかなる頂点も、最初はローカル座標空間に置かれている。

これを、最終的にはモニター上、つまりスクリーン空間へと変換して持っていってやらなくてはならない。最初に想像する以上に、この道程は非常に長く複数の行程を必要とする。

一部はハードウェアによって自動的に行われる変換もあるが、プログラマが座標変換を正しく行ってやらなくてはならない部分も多い。

先述の通り、これには行列を利用する。

Chapter.02

MVP マトリックス

モデル座標変換（ワールド座標変換）

まず最初に行う変換は、モデル座標変換である。

この変換が終わると、あくまでも原点からの相対距離でしかなかったローカル座標系の各頂点の座標が、三次元の世界の中に置かれる。

DirectX の処理系では「三次元の世界に頂点が置かれる」というところからワールド座標変換などと呼ぶが、OpenGL 系ではモデル座標変換と呼ぶのが一般的である。

ビュー座標変換

ビュー座標変換は、三次元に置かれた頂点を撮影する「カメラを定義」するための座標変換。

モデル座標変換によって三次元空間上に頂点が配置されても、それを撮影するカメラが存在しなかったり、あるいは存在しても全く別の方向を向いていたりすれば、当然モニターには頂点が映しだされることはなくなってしまう。

ビュー座標変換を行うことで、三次元空間を眺めるカメラを正しく定義してやる必要がある。

プロジェクション座標変換

次に登場するのが、プロジェクション座標変換である。

これは、プロジェクションの言葉からも連想できるとおり、スクリーン上に頂点を投影するための座標変換。

たとえば、スクリーンの大きさは横長かもしれないし、正方形かもしれないし、モバイル端末だったりすれば縦長ということもありえる。そういったスクリーンに投影するための諸情報を盛り込むのに、プロジェクション座標変換が必要なのである。

MVP マトリックス

ここで説明した「モデル」・「ビュー」・「プロジェクション」の3つの座標変換は、アプリケーション側（JavaScript）で用意するのが一般的。そして、それぞれの座標変換ごとに、ひとつずつ行列を生成する。

行列の項で触れたとおり、行列は情報をひとつにまとめることができるという特性を持っているので、モデル・ビュー・プロジェクションの全ての情報を、最終的にはたったひとつの行列にすべて凝縮させて持たせることができる。

この3つの行列を掛けあわせた、最終的に出来上がる行列のことを各行列の頭文字を取って「MVP マトリックス」と呼ぶ。

MVP マトリックス

WebGL では、この MVP マトリックスは自力でどうにかして生成しなければならない。別の言い方をすると、WebGL の機能として行列を生成できる機能などは一切備わっていない。

自前で行列処理を記述できる人は、JavaScript の実装を書けばよい。これが難しい場合は、無理せずライブラリを使うことを勧める。

※ 行列処理が行える JavaScript のライブラリは結構いっぱいある。
拙作のライブラリでよければ以下を使えば最低限の行列処理はできます。
ここからは、以下のライブラリを使ってコードを書いていくのでそのつもりで。
https://github.com/doxas/minMatrix.js

Chapter.03

モデル座標変換

モデル座標変換

モデル座標変換は、ローカル座標系からモデル座標系へ頂点を変換する。このとき、頂点に次のような作用を与えることができる。

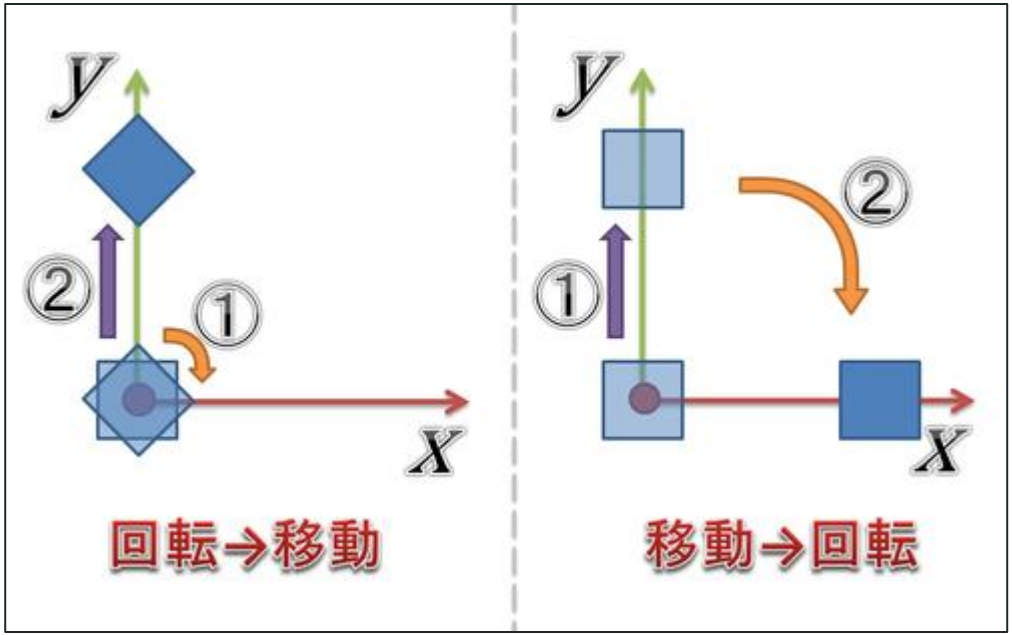
- 平行移動
- 回転
- 拡大縮小

また、これらの作用はひとつではなく、複数組み合わせることが可能。

モデル座標変換の組み合わせ

複数の作用を組み合わせる場合は、その適用する順序に十分注意しなければならない。

特に、拡大縮小や回転といった作用は、常に「原点を中心に作用する」ということをしっかり認識して使うようにしないと、思ったとおりの結果にならないケースが出てくる。



回転と移動の適用順序による違い

モデル座標変換の例

仮に次のような手順でモデル座標変換を行うと、どのような結果になるだろうか。

※ 下記でいうと mMatrix の中身が次々と書き変えられていく

```
var radian = deg * Math.PI / 180;

var axis = [0.0, 1.0, 0.0];

mat.translate(mMatrix, [0.5, 0.5, 0.0], mMatrix); // 平行移動

mat.rotate(mMatrix, radian, axis, mMatrix);       // 回転
```

行列の作用する順序

利用しているライブラリにもよるところが大きいが、OpenGL では行列を「列オーダー」という考え方で処理する。

正しくは行列の計算方法をしっかり勉強するのが一番いいのだが、行オーダーと列オーダーをまったく同じ順序で行った場合、結果が真逆になる。このため、列オーダーで計算する場合には、全部の順序をまったく逆にしてやると、行オーダーで処理した場合と同じような結果になる。

少々わかりにくいが、このような理由から、WebGL では素直に列オーダーを使う場合、行列処理は「見た目上の処理手順だけは真逆にする」ので注意。

行列の作用する順序

つまり、先ほどの処理を振り返ってみると、コードの記述の順序としては、上から見ていくと translate ➔ rotate という順序で実行している。だがこれは実際には、rotate ➔ translate という順序で作用する。

つまり、原点で回転したのち、モデルが平行移動するというわけである。

```
var radian = deg * Math.PI / 180;

var axis = [0.0, 1.0, 0.0];

mat.translate(mMatrix, [0.5, 0.5, 0.0], mMatrix); // 平行移動

mat.rotate(mMatrix, radian, axis, mMatrix);       // 回転
```

モデル座標変換に慣れる

サンプル.w06 では、モデル座標変換を使って、平行移動と回転を行っている。

やってみよう

translate、rotate、scale の三種類のモデル座標変換用のメソッドの使い方を理解しよう。

また、それらを適用する順序によって、どのような差が生まれるのか、しっかり理解しておこう。

Chapter.04

ビュー座標変換

カメラという概念を持ち込む

ビュー座標変換は、モデル座標変換によって三次元空間上に配置された「頂点」という存在を撮影することができるカメラを定義する。

ビュー座標変換も、やはり行列によって定義するが、カメラを定義するためのパラメータとして次に示すものを決めてやる。

- **カメラの置かれている位置（vec3）**
- **カメラの見つめている場所（vec3）**
- **カメラの上方向を表したベクトル（vec3）**

いずれも XYZ を使って表現するため、JavaScript 上では配列として定義し、行列生成を行う。

ビュー座標変換行列の生成

以下に示すように、カメラに関する各情報を lookAt に与えることで、ビュー座標変換行列を生成することができる。

```
var camPosition    = [0.0, 0.0, 5.0]; // カメラは z 方向に 5.0 の位置にあり……

var centerPoint    = [0.0, 0.0, 0.0]; // カメラは原点を見つめており……

var camUpDirection = [0.0, 1.0, 0.0]; // カメラの天面は真上方向を向いている

vMatrix = m.lookAt(camPosition, centerPoint, camUpDirection, vMatrix);
```

カメラの定義に慣れる

カメラの定義では、位置や注視点はイメージしやすいが、カメラの上方向というのが今ひとつ、つかみにくい概念かもしれない。

やってみよう

ビュー座標変換を行っている箇所を適宜修正し、どのようなレンダリング結果への影響があるのか調べてみよう。

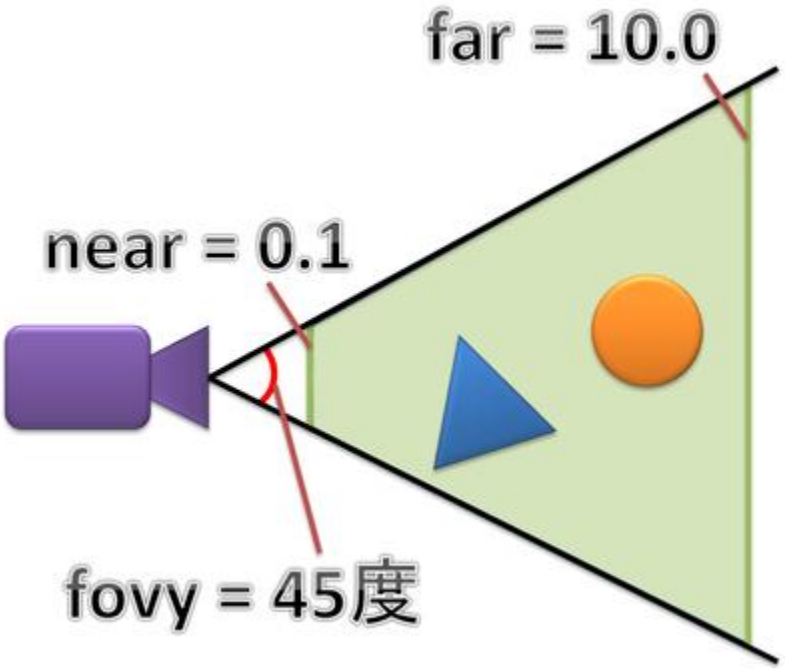
Chapter.05

プロジェクション座標変換

プロジェクション座標変換（射影変換）

MVP のうちの最後のひとつ、プロジェクション座標変換。

ここでは、スクリーンへ正しく投影するためのいくつかのパラメータを与えた行列を生成するが、考え方としては「どのように空間を切り取るか」というふうに考えるのがわかりやすい。



どのように空間を切り取りシーンに収めるか

プロジェクション座標変換行列の生成

プロジェクション座標変換行列を生成するためには、次に示すようなパラメータを指定する。

```
var fovy = 60;   // 切り取る空間の視錐台の角度

var near = 0.1;  // ニアクリップ面の距離

var far  = 10.0; // ファークリップ面までの距離

var aspect = canvas.width / canvas.height; // スクリーンのアスペクト比

m.perspective(fovy, aspect, near, far, vMatrix);
```

ファークリップ面の設定は割りとデリケート

プロジェクション座標変換の指定を誤ると、本来なら確実にカメラに収まるはずの頂点が突然見えなくなったりすることがある。これは結構ハマりやすいので注意。

- ファークリップ面までの設定距離が短すぎて遠くのものが消える
- かといってファークリップを大きくしすぎると他の不具合が増える

このあたりは慣れもあるし、こうすれば絶対間違いない、という手法はないので、本来映るべきはずのものが消えたりする場合は、プロジェクション座標変換が影響している可能性もあるんだな、程度に認識しておくとよい。

Chapter.06

行列を乗算する

行列の乗算

列オーダーなので行列を真逆の順序で処理する、というのは、行列同士を乗算する場合にも同じことが言える。

乗算する順序を間違えると、途中経過が正しくても、最後の乗算して合成する部分で結果が変わってきてしまう場合がある。

たとえばモデル・ビュー・プロジェクションの3つの行列を掛け合わせるには、次のように書く。

行列の乗算

まず最初に pMatrix（プロジェクション）に vMatrix（ビュー）を掛け、最後にそれらの結果に対して mMatrix（モデル）を乗算する。

```
m.multiply(pMatrix, vMatrix, vpMatrix);

m.multiply(vpMatrix, mMatrix, mvpMatrix);
```

※ これはあくまでも、今回使っている自作ライブラリの場合。

MVP マトリックスの完成

こうして、モデル・ビュー・プロジェクションの3つの行列を乗算し、MVP マトリックスが無事に完成。

ここで完成した MVP マトリックスは、シェーダに送ってやり、その行列の内容を参照しながら、頂点シェーダによって各頂点に対して座標変換が行われる。

これは、実際に頂点シェーダでどのようなコードが書かれているかを見るとわかる。

頂点シェーダ内での行列処理

MVP マトリックスを使って、ローカル座標系にある position が変換されている。

```
attribute vec3 position;

uniform mat4 mvpMatrix;

（中略）

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

ここでもやはり、ベクトルに行列を掛けるのではなく、行列に対してベクトルを掛けている。掛ける順序が直感と逆になるのは、シェーダ内でも同様。

Chapter.07

シンプルなライティング

シンプルなライティング

MVP マトリックスは、あくまでも頂点の座標変換に対して使われる。

しかし、行列の使いみちは必ずしも頂点の座標変換だけではなく、ベクトルや座標の変換に広く利用することができる。

ここからは、ライティング（照明効果）の実現を通して、行列を使ったベクトルの変換にもう少し踏み込んでみる。

平行光源

3DCG では、ライティングは非常にポピュラーな技術。

たくさんの研究がされてきたので、古典的な方法から最新の技術まで幅広く様々な手法が存在する。まずは、最も単純な平行光源によるライティングを通して、ベクトルや行列の処理にさらに深く関わっていく。

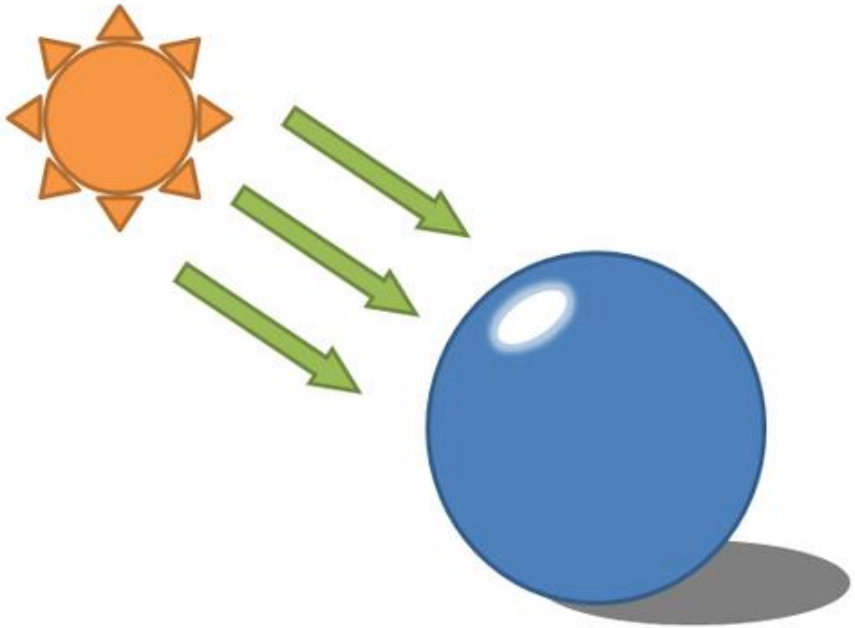
平行光源は、無限に遠いところから降り注ぐような、太陽などの光を再現することができるシンプルなライティング手法。

平行光源

平行光源では、ライト（光源）は単に「向きを持ったベクトル」として定義する。

無限遠から一定方向に降り注ぐ光なので、向きの情報だけあればいいのである。

JavaScript からは、このライトの向きを uniform 変数としてシェーダへ送る。シェーダ側では、このライトの向きの情報を受け取り、頂点がどのように光に照らしだされるのかをシミュレートする。



無限遠から降り注ぐ光（平行光源）

頂点属性の追加

ライトベクトルは単に配列で定義した情報を uniform としてシェーダに送ればいいだけである。

しかしライティングを行うためには、ライトの光の向きだけでは情報が足りない。正しくライティングを行うためには、頂点の側にも「ある情報」を持たせなければならない。

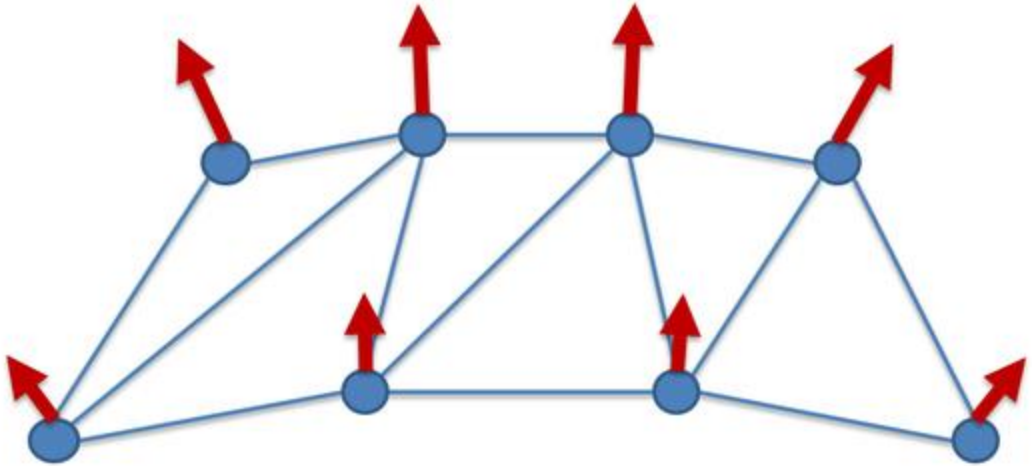
それが「頂点法線」である。

頂点法線

頂点法線とは「頂点がどの方角を向いているのか」という情報を表す情報で、3DCG ではいたるところで利用するとても大事な頂点に関する情報。

ライティングも、正しく処理を行うために頂点法線が必要となる処理のひとつである。

光が頂点にぶつかった際に、頂点がどのような方角を向いているかによって、光をどう反射するのかが決まるため頂点法線が必要になる。



頂点法線の概念図

頂点に属性を追加する

頂点には、これまで頂点の座標位置と、頂点の色の情報を持たせ、attribute 修飾子付きの変数としてシェーダ内で参照していた。

これと同様に、シェーダ内で法線を参照できるようにする必要がある。

まずは JavaScript 側で法線の情報を定義し、VBO を生成するところを拡張する。

VBO を追加する手順を確認する

VBO を新しく追加し、法線をシェーダ内で参照できるようにするにはどうしたらいいか、考えてみる。

一通り自力でがんばってみて、答え合わせのつもりでサンプル.w07 を見てみるとよい。

やってみよう

まずはサンプル.w06 を使って頂点法線を自力で追加するにはどうしたらいいか考えてみよう。

答えは 07 で確認できます。

逆行列

ライティングを正しく行うためには、ライトベクトル、頂点法線、これに加えてもうひとつ必要なファクターがある。

それが「逆行列」である。

逆行列は、数学的にはやや理解するのが難しいが、ここでは単に「全く逆の作用を与えることにより影響を打ち消す行列」というふうにイメージするのがよい。

逆行列

しかしライティングを行うためにどうして逆行列が必要になるのだろうか。

実は、逆行列がなくてもライティングがうまくいく場合というのもある。それは、頂点がローカル座標からまったく動いていない場合。そういったケースでは逆行列を使おうが使うまいが、ライティングの結果は変わらない。

しかし頂点が動いている場合、たとえば回転している場合を考えてみると、どうだろうか。

逆行列

原点を中心にした回転によって頂点が動いた場合、当然頂点の「座標は移動する」が、法線や色は、それに追従して変換されないまま移動していく。

すると、座標は動いているのに、法線は内容がまったく変化していない、という状態になるため、ライトの当たり方が非常に不自然なものになる。

具体的には、動かないはずのライトが、モデルと一緒に動いているかのようにレンダリングがされてしまう。

※ たぶんその状態を実際に見たほうがわかりやすい

ライトベクトルの変換

これに対処するために必要となるのが、逆行列である。

逆行列は、モデル座標変換行列から生成する。そうすることで、モデル座標変換とはまったく逆の作用を持つ行列が生成できるので、これをライトベクトルに対して変換処理する行列として利用する。

これで、頂点が動いてしまった場合でも、相対的に法線を調整することができるようになる。

※ 図を書くなどして、おちついて考えるが吉

ライトベクトルの変換

最後にシェーダの記述について。

シェーダ側では、ライトベクトルを正規化してから、これを逆行列で変換する。

変換後のライトベクトルと、法線とで内積を取る。内積が何を計算できるものだったのか覚えていれば、ここはイメージがしやすいはず。

内積は、ふたつのベクトルがどの程度同じ方向を向いているか、を数値化することができ、真逆のベクトルなら -1.0、全く同じ向きなら 1.0 という結果になる。つまり、ライトベクトルと法線が、どの程度同じ方向を向いているのかを数値化するわけである。

ライトベクトルの変換

内積の結果は、ベクトルの向き次第でマイナスの数値になることもあるため、シェーダ内では clamp というビルトイン関数を使って数値がマイナスにならないようにクランプ（数値の丸め処理）している。

あとは、これを照度の指標として、最終的に出力する色に乗算してやればよい。

ライティングは最初すごく難しいと感じる部分なので、落ち着いて考えること。

付属サンプル

- w06（行列の各種処理）
- w07（ライティングを行っているもの）