WebGL 基礎 .02

3D 数学の基本と導入

Chapter.01

ラジアンと三角関数

ラジアンと三角関数

3D プログラミングでは、角度の計算やベクトル演算など、多数の数学に関連する知識を必要とする。

ここでは、まず最初にラジアン、そして三角関数の簡単なおさらいをする。

後半は、ベクトルや行列といった線型代数学の初歩を学ぶ。

度数法と弧度法

一般に角度というと 360度、のように度数法を用いる。

度数法は日常生活では十分に使いやすい角度の表現だが、3D 数学では不都合が多いため「弧度法（ラジアン）」という別の角度表現を用いる。

弧度法では、円の一周（つまり度数法の 360度）を、円周率を用いて 2π （2パイ）と表現し、これを基準にした数値で角度が表現される。

プログラムを書く上では、度数法から弧度法、つまりラジアンへの変換方法くらいは知っておいたほうがなにかとよろしい。

度数法からラジアンへの変換

度数法の度数からラジアンに変換するには、以下のように計算すればよい。

```
var rad = 度数 * Math.PI / 180;
```

※ 細かいことはよくわからなくても、この計算方法だけは覚えておくこと

ラジアンからサインやコサインを求める

JavaScript の Math オブジェクトは、Math.PI 以外にも、サインやコサインといった値を計算できるメソッドを持っている。これらの算術関数には、角度をラジアンで与えなければならない。

```
var rad = 度数 * Math.PI / 180;

var s = Math.sin(rad);

// 角度はラジアンで与えないと、正しい結果が得られない
```

サインコサインに慣れる

サンプル.w04 は、頂点を gl.POINTS を使って点として描画している。

点として描く場合は線やポリゴンとは異なり、点の結ぶ順序などが関係ないので gl.drawArrays で描画する。また、描画される点の大きさは GLSL 側（頂点シェーダ）で指定することができる。

gl_PointSize にサイズを指定すればよいが、最大値はハードウェアによって上下するので注意。

やってみよう

頂点バッファの元となる JavaScript の配列を生成する際に、頂点が円形に配置されるようにしてみよう。

サイン波を使ってみる

サンプル.w05 では、頂点を描画する際に gl.POINTS を使う w.04 と同じ方式で描画している。

頂点のデフォルトの配置は、-1.0 ～ 1.0 までの X 軸上に全部で 100 個の頂点が配置されている。

やってみよう

頂点シェーダを活用して、横一直線に並んでいる頂点を、サイン波のように波打たせるにはどうしたらいいだろうか。

Chapter.02

ベクトルおさらい

ベクトル

ベクトルはよく「大きさと向きを持った量」といったふうに説明される。

これはいったん理解できてしまうと、なんとなく意味がわかるけれども、最初はいまいちよくわからない概念である。

ベクトルについては、数学的には大変ディープな世界が広がっており、そのすべてを理解するのは難しい。ひとまず、3D 数学を扱う上でのベクトルを知るには、まず次元をひとつ落として、二次元空間でのベクトルについて学ぶのがよい。

二次元でのベクトル

ベクトルの難しい話はさておき、ひとまず簡単に理解することを優先する。

ベクトルは先程も書いたように、「大きさと向きを持った量」というふうに説明される。これがどういうことなのか、まずはここから理解していく。

いったんベクトルのことは忘れて、次のことを考えてみる。

「大きさ」を客観的に表現するとき、どのような方法があるだろうか。

大きさを表すには

大きさを表現し、客観的に、第三者にそれを伝えるのに最適なのは数字を使った大きさの表現である。

1より10のほうが大きい、ということは誰しもわかる。このように、大きさは単に数値の大小によって表現することが可能である。

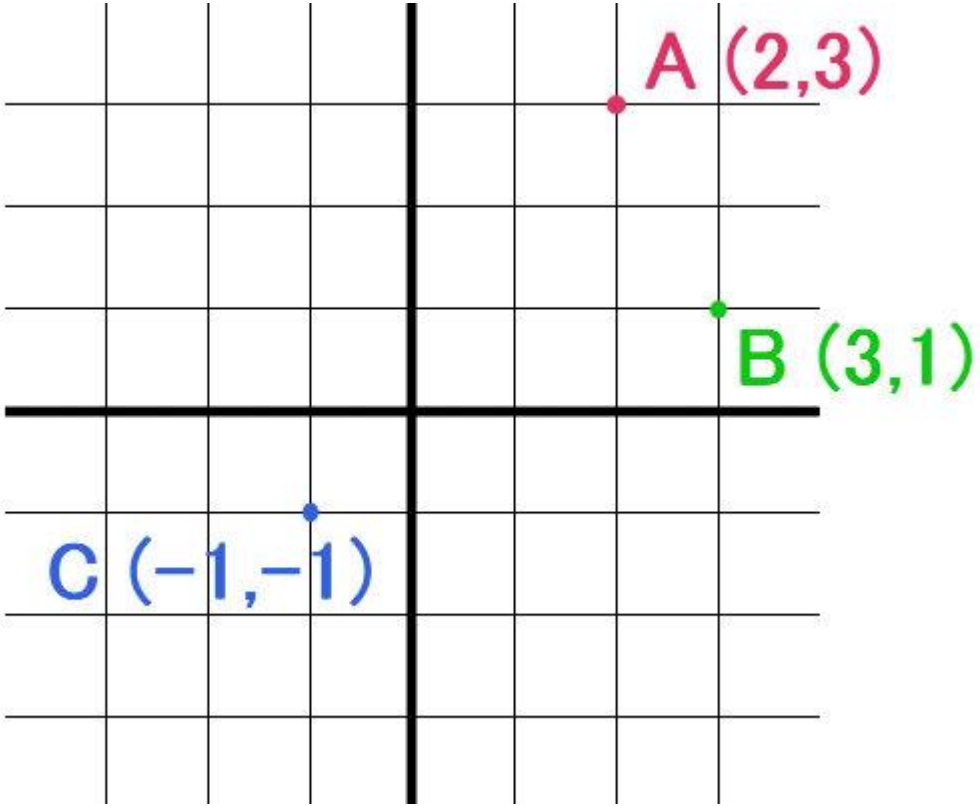
これがまず大前提。

二次元の座標

それでは、単に数値で大小を表現するのではなく、ある平面上の一点を表現するとしたら、どのようにすればよいだろうか。

平面上の座標、つまり位置を表現するには「横方向の量」と「縦方向の量」というふたつの情報が必要になる。

つまり、次のようにすれば、平面上のある一点を表現することができる。



平面上のある一点を表すためには？

複数の数値をまとめてひとつの表現とする

このように、複数の数値をまとめてひとつの塊として扱うと、平面上にある特定の座標を表現することができる。

先ほどの例は二次元だったが、三次元の場合も考え方は同じ。

三次元空間上の特定の一点を表現するために、ベクトルは非常に都合がよい概念である、ということがわかるのではないだろうか。

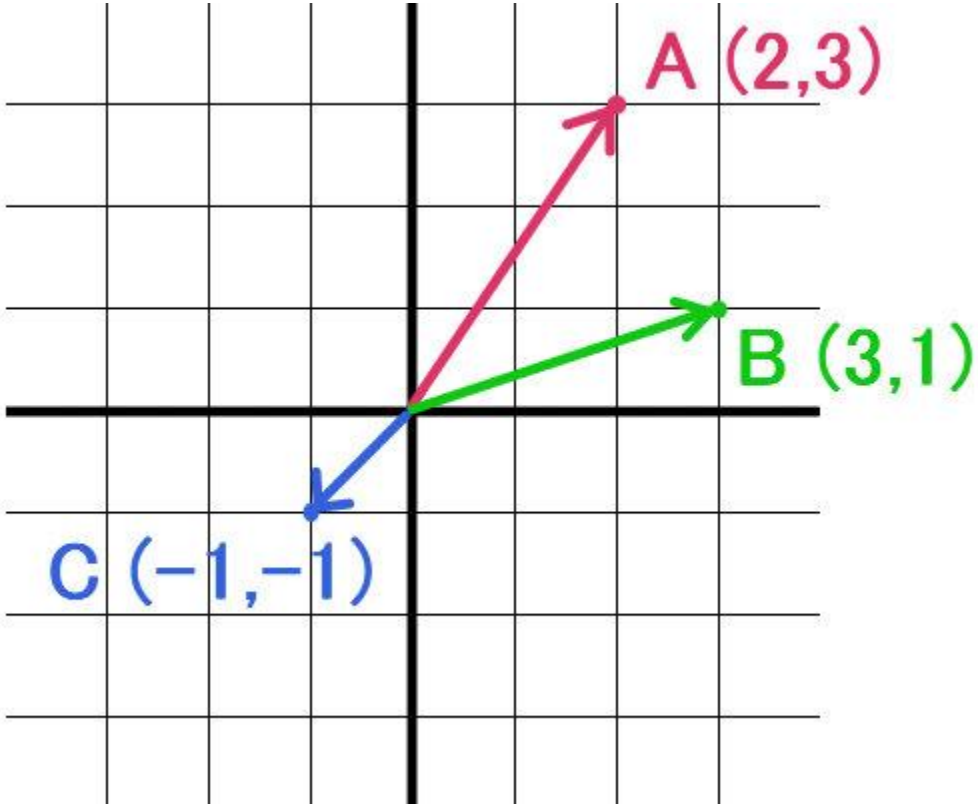
原点から伸びるベクトル

ベクトルはプログラミングのコード上では、先ほどのように複数の数値の塊として表現される。

しかし、視覚的にベクトルを表現する場合には、矢印を利用してベクトルを表現することが多い。

これは、原点と呼ばれる、「座標が (0, 0) となる地点」から、数値のプロットされた座標へ向かって矢印を伸ばすことで表現する。

たとえば、先ほどの図を矢印を用いたベクトル表現にすると、次の図のようになる。



ベクトルを矢印で表現

Chapter.03

ベクトルの向きと大きさ

ベクトルの向きと大きさ

ベクトルは、複数の数値が一塊になったものである、というふうにひとまず理解できた。

ただ一般的に、ベクトルはやはり「向き」と「大きさ」を持つものとして考えられるようにしておくことが、プログラミングを行う上でも理解としてより正しい。

向き、そして大きさ、これはどういうことなのか、ここからさらに理解を深めていく。

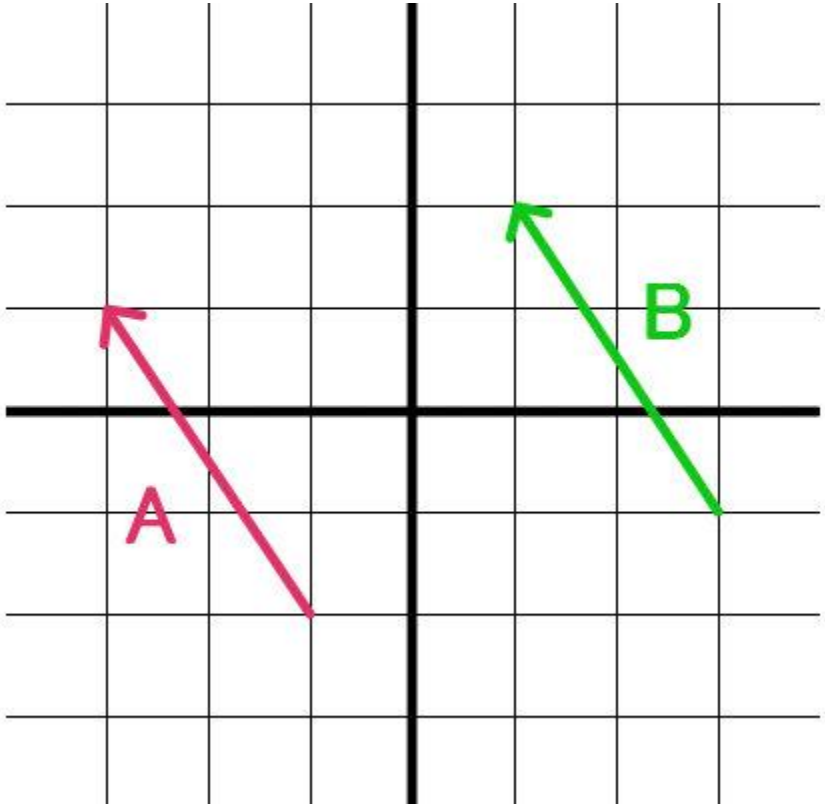
ベクトルの向き

ベクトルは向きと大きさを同時に表現できる。これがどういうことなのかわかれば、ベクトルに関する理解が深まる。

ここでは再度、ベクトルを二次元で考えてみる。

次に出てくるふたつのベクトルは「同じ向きである」

○か？ ×か？



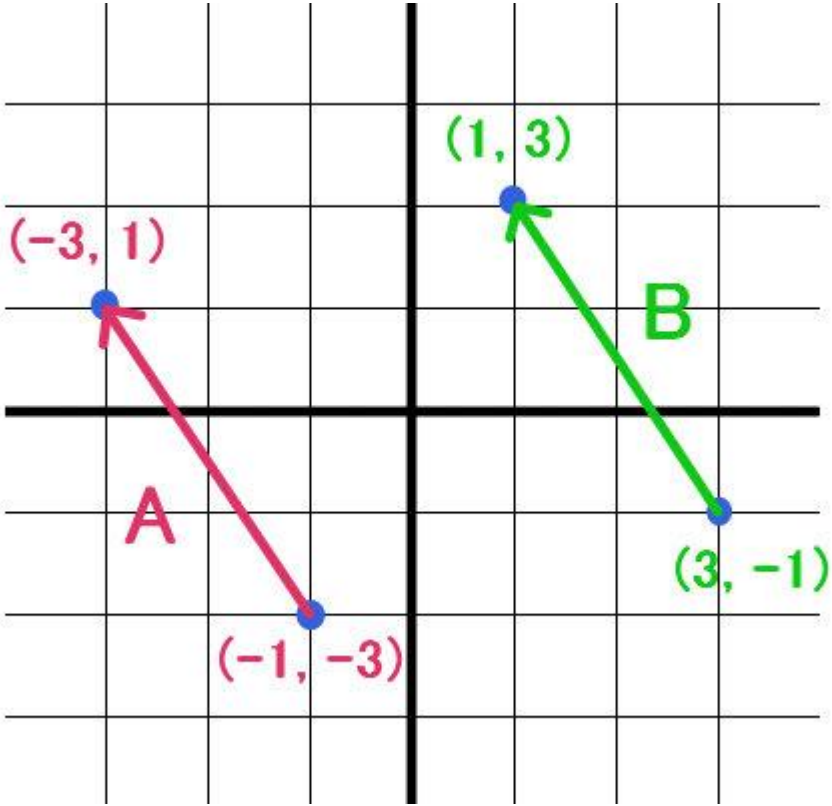
ベクトル A とベクトル B は同じ向きか？

ベクトルの向き

先ほどのベクトル A とベクトル B は、言うまでもなく向きは同じである。

人間は目で見て、これが間違いなく同じ向きだと視覚的に判断できるが、これを数学的に同じ向きだと判断するには、どのようにすればいいだろうか。

次の図をよく観察しながら、どのような計算を行えば、数学的にふたつのベクトルの向きが同じだと言えるのか、考えてみる。



どうすれば数学的に同じ向きだと証明できるか？

ベクトルの向き

先ほどの図をよく見ながら考えてみると、次のように計算すれば、ベクトルがどのような向きなのかが求められる。

ベクトルの始点と、終点というふたつの座標に注目する。

始点から、終点へと伸びるベクトルは、次のように計算すればよい。

始点から終点へと伸びるベクトル ＝（終点x － 始点x，終点y － 始点y）

先ほどのベクトル A も、ベクトル B も、このルールによって計算すると ……

いずれも （−2，3） となる。つまり、全く同じ向きをしているということがわかる。

ベクトルは2地点の座標があれば求められる

このように、ベクトルは常にふたつの地点の座標から求めることができ、これは三次元の場合でも同じである。

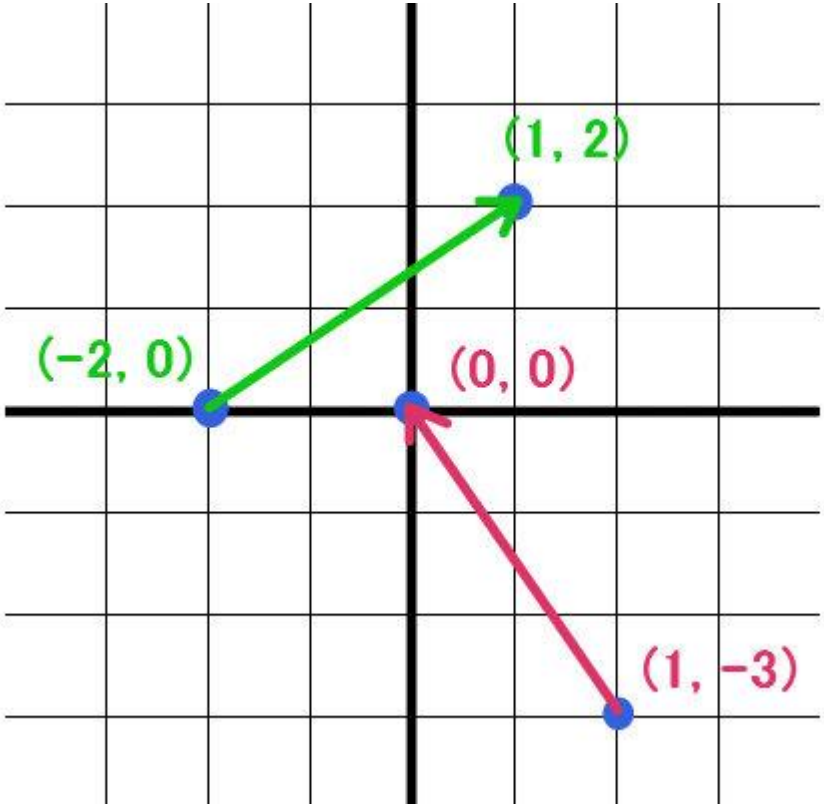
この計算の仕方がわかっていれば、ふたつの地点を結ぶようなベクトルを簡単に算出することができる。

ベクトルの大きさ

次にベクトルの大きさについて考える。

先ほどと同じように、ふたつのベクトルがある。これらは同じ大きさだと言えるだろうか。

先ほどまでは向きだったが、今度は大きさを比較する。



ふたつのベクトルの大きさは同じか？

大きさを求める方法を探す

先ほどの向きの場合と同様に、なんとなく見た感じは同じ大きさのように見える……かもしれない。

これを数学的に同じ大きさであると証明する方法には、どのようなやり方が考えられるだろうか。

これは先に答えを書いてしまうが、先ほど向きを求めたときのように、まずはベクトルの始点が原点にあった場合の、ベクトルの純粋な向きを求めることからスタートする。

原点を始点とする状態に変換

先ほどの要領で計算してみる。

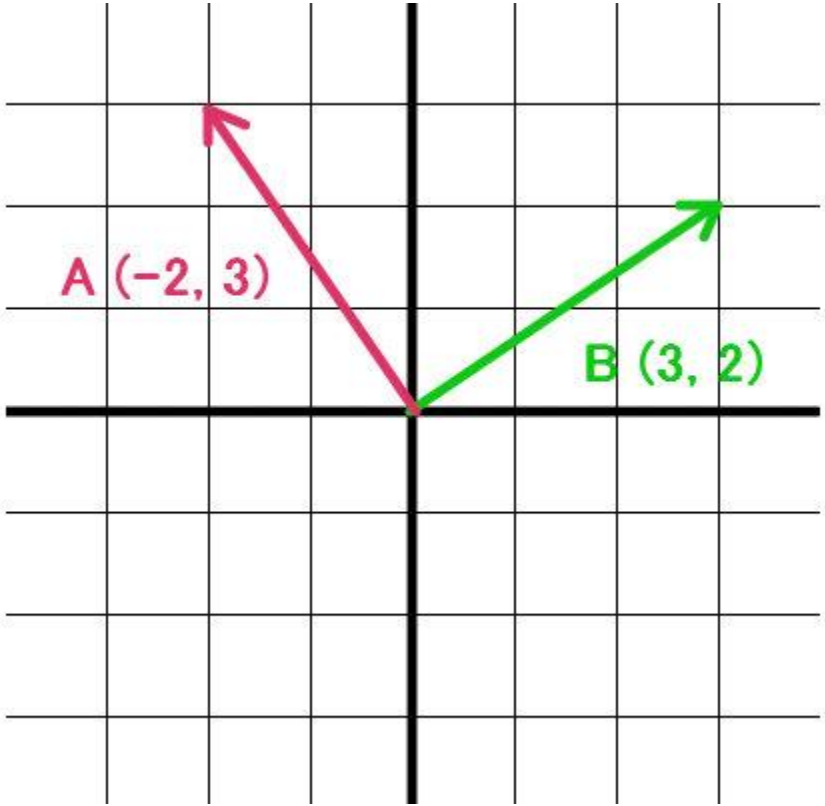
このように計算することで、ベクトルが原点を始点とする、純粋な向きを表すベクトルになる。

ベクトル A （赤い矢印）は……

```
(0 - 2, 0 - (-3)) = (-2, 3)
```

ベクトル B （緑の矢印）は……

```
(1 - (-2), 2 - 0) = (3, 2)
```



始点を原点に持ってきた状態になった

べき乗と平方根の計算を使って大きさを求める

ベクトルの純粋な向きがわかったら、それをもとに次のように計算することで、ベクトルの大きさを求めることができる。

これにより、先ほどのふたつのベクトルは同じ大きさであることが証明できた。

ベクトルの大きさを求めるには、 x と y をそれぞれ二乗して加算し、その平方根を取る。

ベクトル A → `Math.sqrt((-2 * -2) + (3 * 3))` **= √13 = 約 3.6**

ベクトル B → `Math.sqrt((3 * 3) + (2 * 2))` **= √13 = 約 3.6**

ベクトルの基礎

このように、ベクトルは2点の座標から簡単に求めることができ、大きさを調べたい場合には、要素をそれぞれ掛けあわせて合算し、平方根を取ればよい。

この計算方法は二次元だけでなく、三次元でも同様なので、とっさに思い出せるようにしっかり覚えておくこと。

- 終点 - 始点 とすることでベクトルが求められる
- 各要素を二乗して加算し平方根を取れば大きさが求められる

ベクトル補助関数

ベクトルに関する知識をおさらいする意味も兼ねて、いま覚えたことを JavaScript の関数として記述してみる。

やってみよう

2つの座標を受け取り、ベクトルを返却する関数、そしてベクトルを与えるとその大きさを返す関数、これを実装してみよう。

※ひとまず二次元で！

Chapter.04

押さえておきたいベクトル知識

ベクトル予備知識

ここからはベクトルに関する押さえておきたい予備知識、キーワードなどを解説。

- 単位ベクトル
- ベクトルの正規化（単位化）
- ベクトルの内積
- ベクトルの外積

単位ベクトル

単位ベクトルとは、大きさがちょうど 1.0 になるベクトルのことである。

ベクトルを用いる場合、その「向きだけ」に注目したい場面というのが存在する。そういったケースでは、ベクトルの大きさがそろっていないと、うまく計算ができない場合がある。

そこで、そういったケースではベクトルの大きさを一定に揃えるために、ベクトルを正規化（単位化）する。正規化（単位化）とは、ベクトルの大きさを 1.0 ちょうどになるように補正すること。

ベクトルの正規化

正規化は、大きさを求める処理と似ている。

というか、求めた大きさで割るだけである。

ベクトル `(3, 5)` を正規化するには……まずベクトルの大きさを求める。

`Math.sqrt(3 * 3 + 5 * 5) = 約 5.83`

大きさが求められたら、大きさで各要素を割る。

単位化ベクトル `= (3 / 5.83, 5 / 5.83) = (0.5145, 0.8575)`

※ これで正規化完了。これを二次元の平面にプロットすると、原点からその座標までの距離がほぼ `1.0` になっているはず！

ベクトル正規化関数を作る

先ほどの計算方法をよく思い出しながら、ベクトルの正規化関数を作る。

やってみよう

せっかくなので、三次元ベクトルを正規化する関数を作ってみよう。

平方根は、Math.sqrt を用いれば求めることができる。

ベクトルの内積と外積

ベクトルには足し算や引き算の他に、内積と外積という特殊な計算方法がある。

内積や外積を数学的に正しく理解し使いこなすのはなかなか大変だが、3D 数学で「どういったときにそれが必要なのか」に注目すれば、ひとまず細かい数学の定理は置いておいて、実践的に内積や外積を使うことができる。

ここでは、数学的な理解を深めることよりも、実際にどういった場面でそれが利用されるのか、それがどうして便利なのかについてを重視して話を進める。

ベクトルの内積

ベクトルの内積は、二次元でも、三次元でも、用途がだいたい同じである。

内積は、二次元なら X と Y、三次元なら XYZ の各要素を、全てそれぞれに乗算してから足し合わせる。つまり次のようにする。

ベクトル A とベクトル B の内積を求める例

内積の結果 ＝ Ax * Bx + Ay * By + Az * Bz

つまり、それぞれ乗算した結果を、全部足し合わせるだけ！

なにに使うの内積

内積は、ふたつのベクトルが「どれくらい同じ向きになっているのか」を調べるときや、ふたつのベクトルが「成す角度」を調べたい時に使う。

正規化されたベクトル同士の内積は、その計算結果が必ず -1.0 〜 1.0 の範囲に収まる。これがとても重要。

どれくらい同じ向きになっているのか、ということは、内積の結果を見ればすぐにわかる。-1.0 の場合、それぞれのベクトルは完全に真逆を向いている。一方もしも結果が 1.0 だったとしたら、ベクトルは完全に同じ方角を向いている。

これは、実際に計算してみればわかる。

内積計算関数を作る

本当に、正規化されたベクトル同士の内積は、向きが同じかどうかを判定するのに使えるのだろうか。

実際に関数を設計し、試してみる。

やってみよう

ふたつのベクトルが与えられたとき、両者の内積を計算する関数を作ってみよう。

また、正規化する関数を活用しつつ、ふたつのベクトルの向きが一致している量を正しく計算できるか、試してみよう。

ベクトルの外積

ベクトルの外積は、内積とは異なり二次元と三次元で大きく変わる。

特に大きいのが、その計算結果。

二次元での外積は計算結果が実数になるが、三次元の場合は結果がベクトルになる。
そして、内積と比較すると計算方法が複雑になるため、計算方法を暗記するとなるとちょっと大変。

なにに使うの外積

ここでは三次元での外積についてのみ述べるが、先述の通り、外積の結果は三次元ではベクトルになる。

この結果として得られるベクトルは、与えられたふたつのベクトルに垂直なベクトルである。これが重要。

外積が必要になる時の多くは、ふたつのベクトルに垂直なベクトルを求めたいとき。
（ちょっと難しい話をするとバンプマッピングなどで使う）

Chapter.05

ベクトルと行列

行列

行列は、英語では Matrix と呼ぶ。

数値を行と、列からなる塊として扱う概念で、ベクトルが一次元だったのに対し、行列は二次元で数値をまとめたものである。

ベクトルと行列

ベクトルは一次元

V = (x, y, z);

行列は二次元

$$m = \begin{bmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{bmatrix}$$

行列を使うと何が嬉しいのか

行列を使うと、一言でいうと「ベクトルや座標の変換」ができる。

そして行列の優れた特性のひとつが「行列にいくつもの情報をまとめて格納できる」ことである。

たとえば、座標を変換したいと考えた時に、座標を拡大して、そこから回転させて、なおかつそのあと移動させる、ということをしたい場合があるとする。普通に考えれば、三回の変換計算が必要になりそうに見える。

しかし行列を用いる場合、そのすべての情報を行列ひとつにまとめて格納しておくことができ、ベクトル（座標）と行列を掛けると全てが適用された結果が一発で得られる。

行列を使うと何が嬉しいのか

行列が持つ、この「情報をまとめて格納しておける」などの特性が、三次元の座標計算にとって非常に都合が良いので、3D 数学や 3D プログラミングでは行列を頻繁に利用する。

行列の計算方法は、ベクトルの外積と同じようになかなか複雑なので、暗記していつでも手入力可能なように暗記したり、あるいはそれが可能なレベルの高い知識を持っていることはとても難しい。

普通は、行列処理を行ってくれるライブラリなどを活用してプログラミングを行う場合がほとんどである。それで困るというケースは、ある程度高度なことをしようと考えた時や、限界まで最適化をしたりする場合だけであることがほとんどなので、あまり過度に心配しなくてもよい。

3D プログラミングにおける行列

行列は、数値が二次元に配置されている数値の塊であることはすでに説明した。

行列には、行と、列、という概念があり、行数や列数は同じ数でなければならないというわけではない。通常 3D プログラミングでは 4x4の行列を用いることが多い（ちゃんと理由はある）が、行列の概念としては3x4や2x2などの行列も、もちろん行列として間違っているわけではない。

3D プログラミングにおける行列

3D 数学で行列が4x4なのは、三次元ベクトルを扱う上でそれが都合がいいからである。

行列を使うと、例えば頂点の座標を、移動させたり、拡大するように動かしたり、回転したように動かしたり、といったことができる。この代表的な3つの動作のうち、回転と拡大縮小は、3x3の行列で表現することができる。しかし、平行移動を表現するときに、4番目の要素があったほうが都合が良いのである。（他にも4番目の要素を使うシーンはある）

そこで 3D プログラミングでは一般に XYZ にもう一次元加えた4x4の行列を使うのである。

単位行列

ベクトルの話をした時に、単位ベクトル、というものが出てきた。

行列にも、単位行列というものがある。単位行列は、行列と行列で乗算を行った時に、計算結果が一切変化しない行列のこと。

より砕けた言い方をすると、数字の 1 は、どんな数に掛けても結果が変わらない。それと同じように、どのような行列に掛けても、結果が変わらないのが単位行列である。

行列をプログラムのなかで初期化するときには、普通この単位行列を用意する。

行列と交換法則

一般的な数値による乗算では、5x10も、あるいは10x5も、結果は同じであり、左辺と右辺の数字を交換しても結果は変わらない。一般にこれを交換法則と呼ぶ。

しかし行列を用いた計算では、交換法則が成り立たないため、かける順序を変えてしまうと、最終的に得られる結果も変化してしまう。

行列の細かな乗算方法を覚えていなくても、行列を用いた乗算では交換法則が成り立たず掛ける順序がとても重要なのだ、ということは覚えておいたほうがよい。

付属サンプル

- w04（頂点の点描画）
- w05（100個の頂点が並んでる版）