GLSL で有用な数学のお話

ベクトルや三角関数

はじめに

GLSL スクールでは、当然と言えば当然ですが 3DCG に関係の深い概念がたくさん登場します。

数学も、そのなかのひとつです。

数学、みなさん好きですか？ あるいは得意でしょうか。

私は、実を言うと数学とても苦手意識があって、解説するときはいつもどきどきです。とは言え、GLSL を記述していく上で、これだけは押さえておきたいという基本くらいは説明できると思っていますので、そのあたりをまとめてみようと思います。

はじめに

スクールの講義のなかで、どうしても避けることのできない概念について、ここでは簡単に説明します。完全に今、思いつきでこのスライド作ってます。

次回（第二回）までに完全に習得しろ！ みたいなものではなくて、参考程度に見てもらえたらと思います。もしも数学は割と得意だって方がいらっしゃったら、少し退屈な内容になってしまうかもしれませんので、読み飛ばしていただいても大丈夫です。

逆にあんまり数学よくわからんぞ！ という方は、このスライドをヒントに、あるいはきっかけにして、数学に慣れ親しんでみてください。意外とね、やればなんとかなりますよ！

ベクトル

矢印的なあいつ

ベクトルは二次元でも三次元でも、グラフィックスプログラミングに大いに役立つ概念です。

最初は難しいかもしれないけど、まずはトライしてみましょう。

座標、あるいは位置

ベクトルの話をする前に、もう少し簡単な話を例に考えてみます。

みなさん小学生のときとかに、二次元のグラフでＸ軸とＹ軸を十文字に描き、そこに点を打つ、いわゆるデータをプロットする作業をしたことがあると思います。

Ｘは○で、Ｙは▲です、それではグラフの（○, ▲）の座標に点を書き込んで表現してみましょう―― みたいなやつです。

座標、あるいは位置

このときの、○ だとか ▲ だとかはそれぞれ X 座標とか Y 座標ですね。

二次元の平面は X を横方向、Y を縦方向として考え、文章で表すときには先ほどのように丸括弧を使って (x, y) というような感じに表します。

このときの (x, y) は、要するに座標です。

ふたつの数値をひとつにまとめて「座標」として扱っています。つまり空間のなかのある一点、その位置を表すためにこういう表記を使っているわけです。

ベクトル

で、ここで最初のテーマであるベクトルについて考えてみようと思うのですが、ベクトルの場合も、今見てきた座標の場合と同じように表現する場合が多いです。

たとえばですが、以下のように言ったりします。

「ここにベクトル (3, 2) がありますね、それではこのベクトルを……」

私は、最初これの意味がよくわからなかったです。えっ ベクトルって矢印で表現されるあれのことだよね？ これじゃあ点（座標）じゃん！ みたいな感じでとても混乱しました。

ベクトル

表記の仕方は同じですが、ある一点を座標として表現しているのか、それともベクトルとして表現しているのかは、少なくとも頭のなかでなにかを考えるときは区別しないと混乱します。

ふたつの数値がセットになって (x, y) というような書き方がされている場合は、その内容や文脈から、それが座標を表しているものなのか、ベクトルを表しているものなのか、正しく認識できるようにしましょう。

＊ 座標（○, ▲）→ 座標、あるいは位置

＊ ベクトル（○, ▲）→ 座標ではなくベクトル

向きと量を持つ単位

それでは具体的にベクトルってなんなんでしょう。

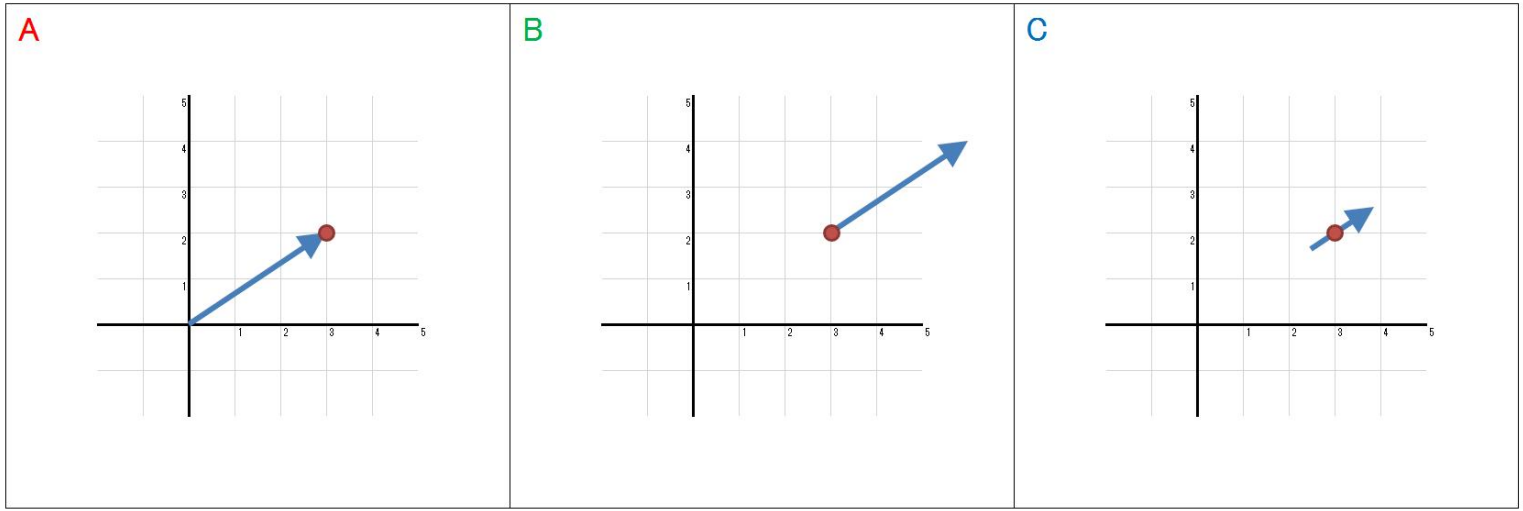
これはまあ数学ちゃんと勉強すれば自ずとわかるのですが、あくまでも 3DCG 的な目線で、グラフィックスプログラミングやシェーダプログラミングにおける中心的な解釈をするなら、ベクトルは「矢印を使って可視化でき、向きと、大きさを表現できる便利な概念」と覚えるのがいいと思います。

唐突にそんなこと言われてもなあ……って思いますよね！ 私は Google 先生にいろいろ訊いてるときにそう思いました。（実話）

向きと量を持つ単位

では実際に、童心にかえったつもりでやってみましょうね。

あるベクトルが (3, 2) と表現できるとします。これをグラフ上にプロットしたときの正しい図は次のうちどれでしょうか。



向きと量を持つ単位

C を選んじゃったひと……

もしいたとしたらちょっと独創的かもしれない……

A と B で迷ったひともいたかもしれませんが、正しくは A です。だいたいのひとは間違えなかったと思いますが、正解は A のように表現すればいいんですね。

さあ、それじゃあここでひとつ冷静になって覚えましょう。

つまり、「ベクトル (x, y)」のように表現できるとき、そのベクトルを矢印でプロットするときは「暗黙で始点は原点になっている！」これが大事です。

向きと量を持つ単位

先ほど、(x, y) のように書いてあるときに、それが座標として表現されているのかベクトルとして表現されているのかをしっかり意識しましょうねという話をしました。

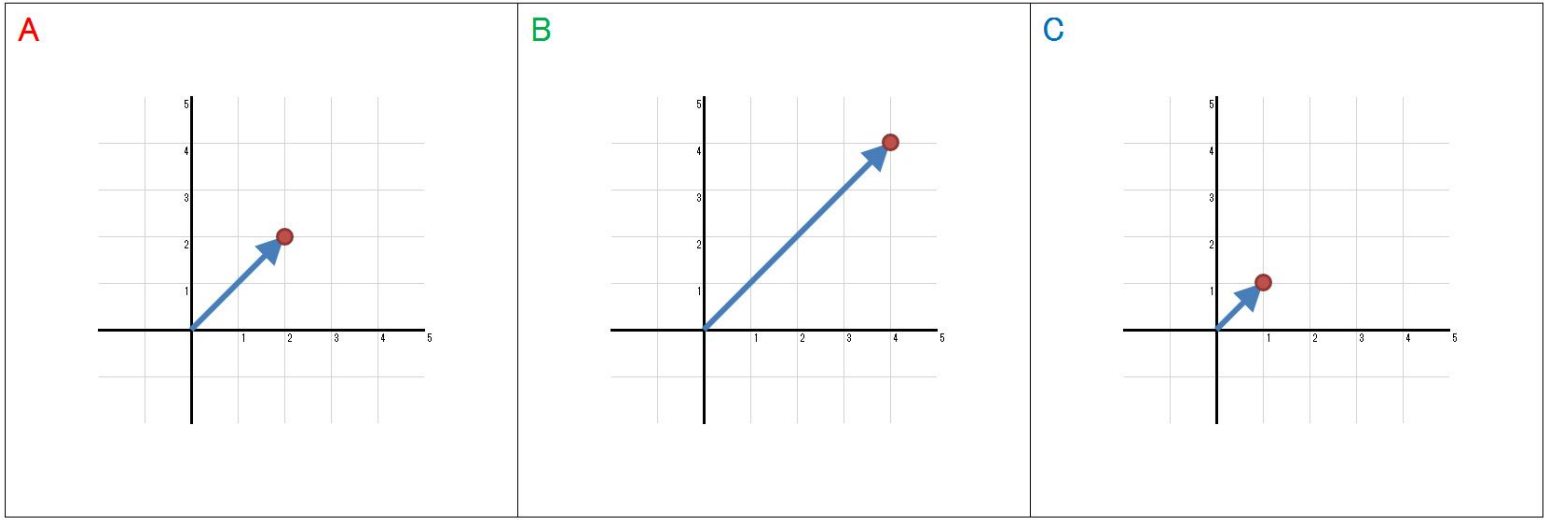
おなじ (x, y) でも、位置としての表現と、ベクトルとしての表現は全然違います。それがベクトルである場合は「原点」から「その座標」へと伸びる矢印としてあたまの中ではイメージできればいいわけです。

暗黙で原点、つまり (0, 0) の場所から伸びる矢印が出ているイメージができれば、ベクトルはそんなに怖くなくなります。

向きと量を持つ単位

では、ベクトルが「向きと量のふたつの意味を持つ」というところも少し掘り下げましょう。

たとえば、次の ABC の各ベクトルは、同じ向きだと言えるでしょうか？



向きと量を持つ単位

これは、目で見て、直感的に答えはわかったと思います。

正解は「全部同じ向き！」ですね。

A は (2, 2)、B は (4, 4)、C は (1, 1) とそれぞれ表現でき、いずれのケースでも、X 要素と Y 要素がイコールになっていて、なんかルールが似ています。

でも、それぞれの矢印の長さは違いましたよね？

これが、ベクトルが「向き」と「大きさ」を持つことができる単位であると言われる根拠です。

向きと量を持つ単位

つまりまとめると……

ベクトルはＸとＹ、あるいはＺなどを加えたりして二次元や三次元で使うことがができます。さらに、それらの数値の大小によって、向きと、大きさの両者を同時に表現することができます。

向きは、ＸとＹのそれぞれの比率によって決まります。先ほどの例は 1:1 でしたが、これが 0.3:0.7 とかになったら、向きが変わったことがわかります。大きさは、ＸとＹのそれぞれの数値の大小から求めることができます。

どうでしょう。なんか意外と簡単じゃないですか？ あせらず、落ち着いて考えてみてください。

ベクトルの大きさ

どうやって求める？

ベクトルの大きさ、あるいは長さは、ちゃんとやり方さえわかっていれば簡単に求められます。

暗記する必要はないですが、理解していると役に立つこと間違いなしです。

ベクトルの大きさ

ベクトルの大きさ（長さ）は、二次元でも三次元でも、同じやり方で求めることができます。次元が変わっても平気ってなんかすごい！（そうでもないかw）

計算方法も、わりかし簡単です。

＊各要素同士を掛け合わせたあと、全部足して、その平方根を求める

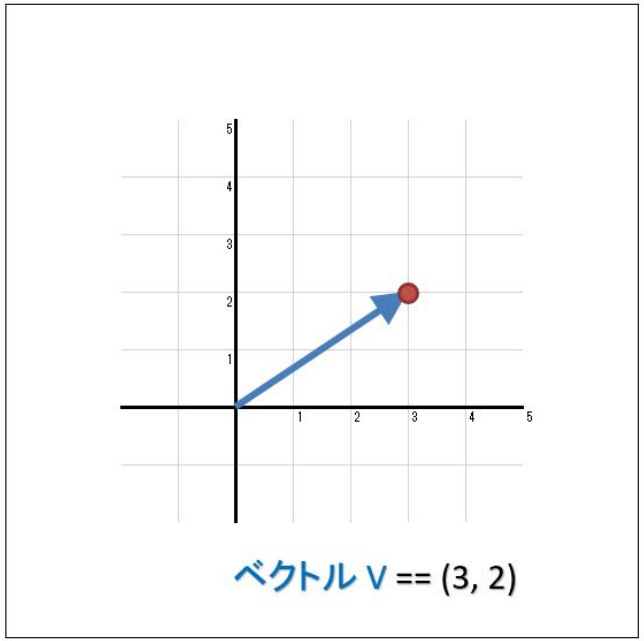
これです。JavaScript のコードにするなら、以下のような感じ。Math.sqrt が平方根を求めるメソッドです。

```
let v = [1, 2, 3];

let length = Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
```

ベクトルの大きさ

たとえば、あるベクトル (3, 2) の大きさ（長さ）を求めるには……

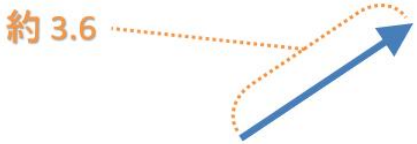



```
let v = [3.0, 2.0];

let len = Math.sqrt(v[0] * v[0] + v[1] * v[1]);

console.log(len);
```

→ 答えはルート 13 なので、約 3.6 と求まる



ベクトルの向き

正規化を知る

ベクトルの大きさはわかったけど、向きはどうやって考えればいいの？

その答えは「正規化」というキーワードが握っています。

ベクトルの正規化（単位化）

ベクトルの大きさを知る方法がわかったら、次はベクトルの向きの表現の仕方を覚えましょう。

先ほど例に出したように、ベクトルの向きっていろんな形で表現することができちゃいます。(4, 4) でも、(1, 1) でも、いずれも同じ向きを表していると言えますもんね。これでは同じ方向を表すパターンが無限に定義できてしまいます。

そこで、ベクトルの向きを表現するときは、次のようなルールを用います。

＊ ベクトルの向きは「ベクトルを正規化してから」考える

ベクトルの正規化（単位化）

ベクトルを正規化する、あるいは単位化する、と言われても最初はなんのことやらようわからん！ というひとが多いかもしれません。

でも、これも実は全然難しくはありません。

要するになにをすればいいのかというと、ベクトルを正規化（単位化）するとは「ベクトルの長さを1にする」ことをいいます。

いろんな長さの矢印を無作為に使うんじゃなくて、常に、長さが1になる一定の長さの矢印で向きを表すことにしようね！ と取り決めをしておくというわけですね。

ベクトルの正規化（単位化）

ベクトルを正規化するときは、さっき求め方を覚えたばかりの「ベクトルの大きさ」が活躍します。正規化する場合の方法は次のように。

＊ ベクトルの各要素をベクトルの長さで割る

これがベクトルの正規化の方法です。たったこれだけ！　先ほど同様コードで表現するなら以下のような感じ。簡単だ！

```
let v = [1, 2, 3];

let length = Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);

let w = [v[0] / length, v[1] / length, v[2] / length];
```

ベクトルの正規化（単位化）

このように、ベクトルの向きについて考えるときは、常に正規化されているかどうかを意識することが非常に大切です。

ベクトルは、向きの他に大きさという情報を持つことができてしまうが故に、あらかじめ正規化をしないで「向きに関する計算」に使ってしまうと、それだけで結果がとんでもないおかしな状態になってしまいます。

講義のなかで、ここではベクトルを正規化してます、という文脈が出てきたときは、
「あー、これはベクトルの向きに関する計算なんだな、だから正規化しておくんだな！」というふうに、すんなりと思い浮かべられることがベストです！

ベクトルに関するまとめ

* ベクトルは向きと大きさを表現できる
* ベクトルの大きさは、各要素を掛け合わせてから乗算し平方根を求める
* 各要素を大きさで割ってやると正規化（単位化）ができる
* 正規化されたベクトルは長さ（大きさ）が常に１である
* 正規化されたベクトルで方向に関する処理を行うことにするといろいろとつごうがよい

三角関数

サイン・コサイン
タンジェント

サインとかコサインって日常生活で全然使わない……場合が多いかと思いますが、3D 数学やシェーダコーディングではむしろ必須。

しっかり予習しておくときっと幸せになれます。

利用目的から覚える

三角関数とか言われると、思わず「ウッ……」と怪しげな呻きが漏れる方もいらっしゃるかもしれません。実際、三角関数ってなんなのですかと言われてしまうと、それを簡潔に説明するのはちょっと難しいかもしれません。

でも逆に考えてみると、簡潔に説明するのが難しいということは、それだけ「様々な場面で役に立つ汎用的な知識である」とも言えると思います。

三角関数は、その数学的な意味を覚えようとするよりも、まず使えることによってなにが嬉しいのか、その利用目的や利用することのメリットから覚えるのがいいでしょう。

三角関数だけど円をベースに考える

三角関数というくらいなので、三角関数には三角形が出てきます。

でも個人的には、三角関数は円を思考の中心に置いて考えたほうがわかりやすいなあって思います。

いまひとつ三角関数とか言われてもよくわからないぞって人は、ここはひとつ騙されたと思って、一度あたまのなかをクリアにした状態で読み進めてみてください。

ラジアン

角度を表すあいつ

三角関数をしっかりと理解するためにも、まずはこのラジアンから理解しておきたいところです。

全然難しくないです。ほんとに。

ラジアン（弧度法）

三角関数に触れる前に、まずは前提知識としてラジアンから押さえましょう。

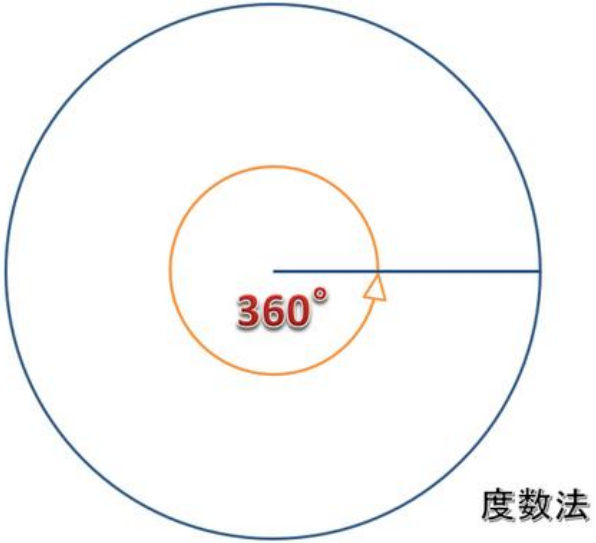
ラジアンとは、角度を表現する方法のひとつです。一般的に、角度って 360 度とか、度数を使って表現することが多いですよね。90 度で直角になるやつ。

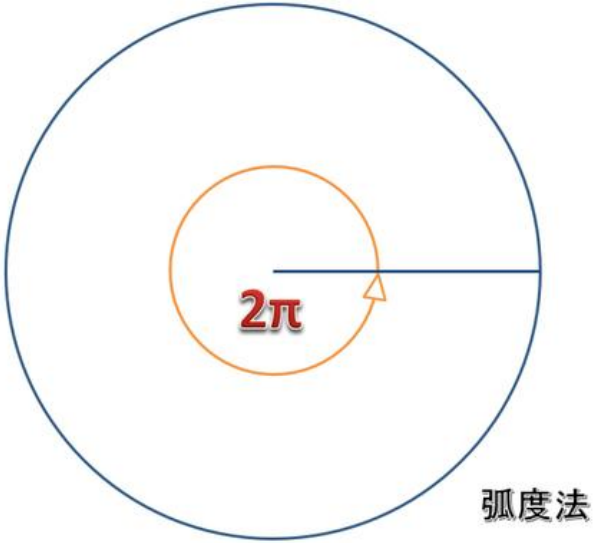
この、度数で角度を表現するのが「度数法」です。

一方で、ラジアンは弧度法と呼ばれています。弧度法なんて言われると難しそうって思うかもしれないですが、全然難しくないです。度数法では円の一周を 360 度と表しますが、弧度法では円の一周を「2パイ」と表します。

ラジアン(弧度法)

度数法と弧度法のそれぞれの違い。





ラジアン（弧度法）

要するに、円の一周を度数で表すのかパイを使って表しているのかが違うだけです。単位が違うだけですし全然難しくないですね。

度数法の 360 度が、弧度法では 2 パイになっている、ただそれだけです。

つまり、数学の話が出てきた時に「角度はラジアンで与えてください」みたいなふうに言われたときは、度数法の度数を、ラジアンだったらいくつになるのかなって考えてみればいいだけなんですね。

ラジアン（弧度法）

たとえば、度数法でいう 90 度をラジアンで表すには……

円の一周が度数法では 360 度なので、90 度は、その四分の一ですよね。

だから 2 パイの四分の一がラジアンで言う 90 度に相当します。だから以下のようにすればいいですね。

```
let rad90 = 2 * Math.PI / 4;
```

簡単ですね……簡単すぎますね……これが、ラジアンです。それ以上でもそれ以下でもなく、ラジアンってのはただ単にこれだけのことです。

サイン
コサイン

数学的意味より使い方

先ほども書いた通り、サインやコサインはその利用目的やメリットから覚えるのがよいです。

これもやっぱり、わかってしまえば全然難しくないです。

サイン・コサイン

さて、ラジアンがわかったところで、次にこんな実験をしてみましょう。

JavaScript の算術関数クラスには Math.sin や Math.cos があります。

これらの関数は角度を「ラジアンで」受け取ります。

そのことを踏まえて、0 度のところから、90 度ずつ加算していき、180 度、270 度で、それぞれどのようなサイン・コサインが得られるか調べてみます。

まず最初は、サインをいくつか求めてみます。

サイン・コサイン

```
let s_0   = Math.sin(0);                    //   0度のラジアンはやっぱり 0
let s_90  = Math.sin(2 * Math.PI / 4);      //  90度のラジアンでサインを求める
let s_180 = Math.sin(2 * Math.PI / 4 * 2); // 180度のラジアンでサインを求める
let s_270 = Math.sin(2 * Math.PI / 4 * 3); // 270度のラジアンでサインを求める
```

さて、この結果ってそれぞれどうなるんでしょうか。

引っ張ってもしょうがないので、答えを書いちゃうと、次のようになります。

```
s_0   ===  0.0
s_90  ===  1.0
s_180 ===  0.0
s_270 === -1.0
```

なーんか意味ありげな感じですが、よくわからんですね……では続いて同じようにコサインも求めてみましょう。

サイン・コサイン

```
let c_0   = Math.cos(0);                    //   0度のラジアンはやっぱり 0
let c_90  = Math.cos(2 * Math.PI / 4);      //  90度のラジアンでコサインを求める
let c_180 = Math.cos(2 * Math.PI / 4 * 2); // 180度のラジアンでコサインを求める
let c_270 = Math.cos(2 * Math.PI / 4 * 3); // 270度のラジアンでコサインを求める
```

さて、結果は次のようになります。

```
c_0   ===  1.0
c_90  ===  0.0
c_180 === -1.0
c_270 ===  0.0
```

なーんかこっちも意味ありげな感じですが、やっぱりまだよくわからんですね。でもこれを、二次元のグラフにマッピングしてみると、よくわかります。

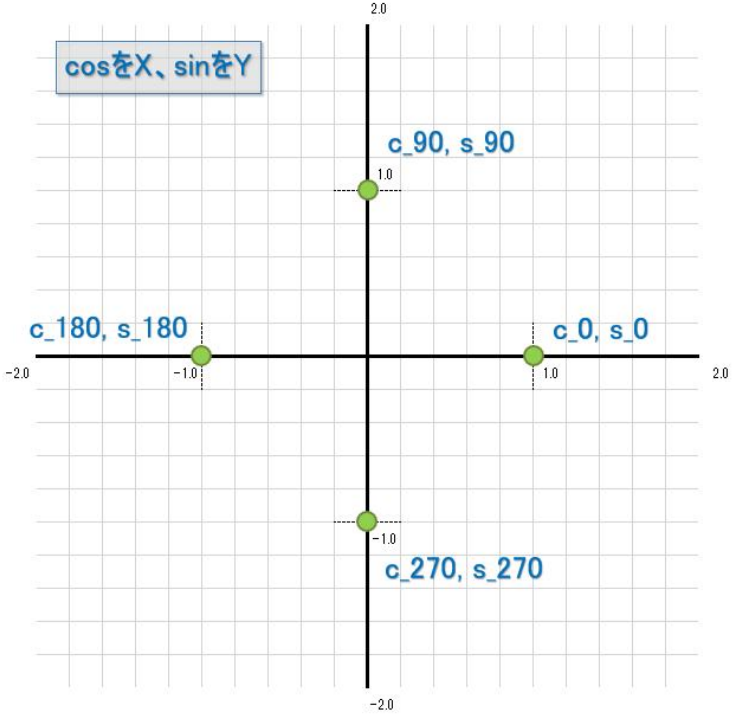
サイン・コサイン

それでは、ベクトルのときと同じような感じで、紙かなんかに十文字を書いてグラフ用紙を作り、そこに先ほどのサインとコサインの結果をプロットしてみます。

このとき、サインをＹ軸に、コサインをＸ軸に当てはめてみます。

すると、驚くべきことが起こるのです……

サイン・コサイン

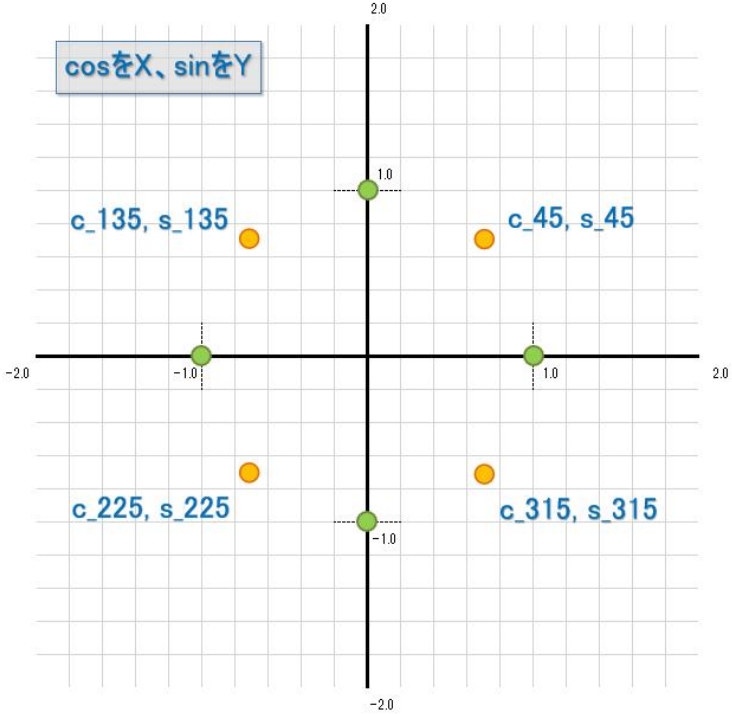


コサインをＸに、サインをＹに見立てて二次元のグラフ上にプロットしてみると、左記のようになります。

0 度が右側にあり、そこから 90 度毎に回転した場所に点が打たれているように見えますよね。（角度とリンクしている！）

ではもっとわかりやすくするために、45 度ずらした位置にもサインとコサインの値をプロットしてみましょう。

サイン・コサイン



さあどうでしょうか。

もうこれで、完璧にわかったかな？

そう、サインやコサインの結果というのは、実はその与えられたラジアンを元に「円を描くように結果が変化していく」のですね。

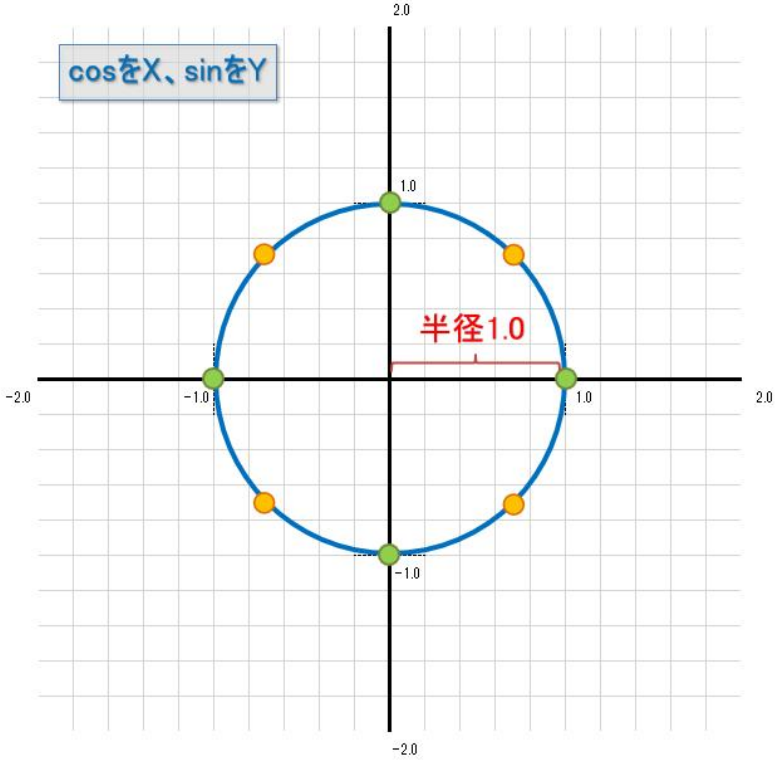
サイン・コサイン

サインやコサインのこの「指定されたラジアンの円周上の位置を返す」という性質がわかっていれば、角度に関する様々な計算に応用できます。

たとえばシューティングゲームで、ボスが大量に弾を発射するような処理を書きたいと思っているときに、今回のサインとコサインの性質がわかっていれば、放射状に敵弾をばらまくみたいなことが簡単にできるようになります。

また、先ほどのプロットした図をさらによーく観察してみると……

サイン・コサイン

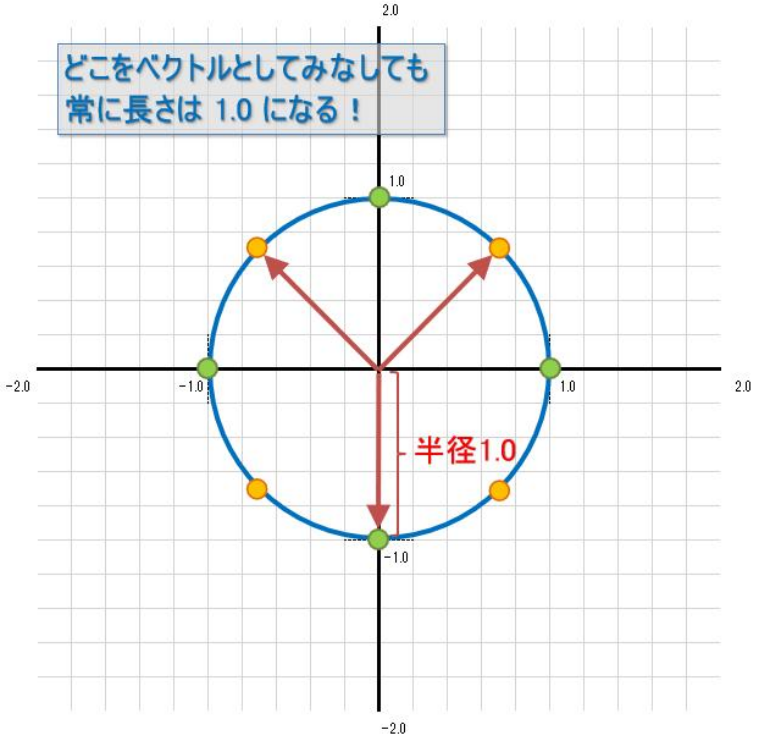


サインとコサインによって描かれる円周は、半径を１とした円になっていることがわかりますね。

半径が、1.0……

ということは、サインとコサインで求められた位置に向かって原点から矢印を伸ばしてベクトルを定義すると、そのベクトルの長さは常に 1.0 であるということになります。

サイン・コサイン



つまり、ベクトルの項目のところで出てきた「ベクトルの方向を考えるときは常に正規化されたベクトルを使う」という前提は、サインとコサインの結果から作ったベクトルなら、最初からその条件が満たされているわけです。

サインとコサインで求めた値は、後から正規化などの手順を踏むこと無く、そのまま正規化済みのベクトルとして計算などに用いることができるんですね！

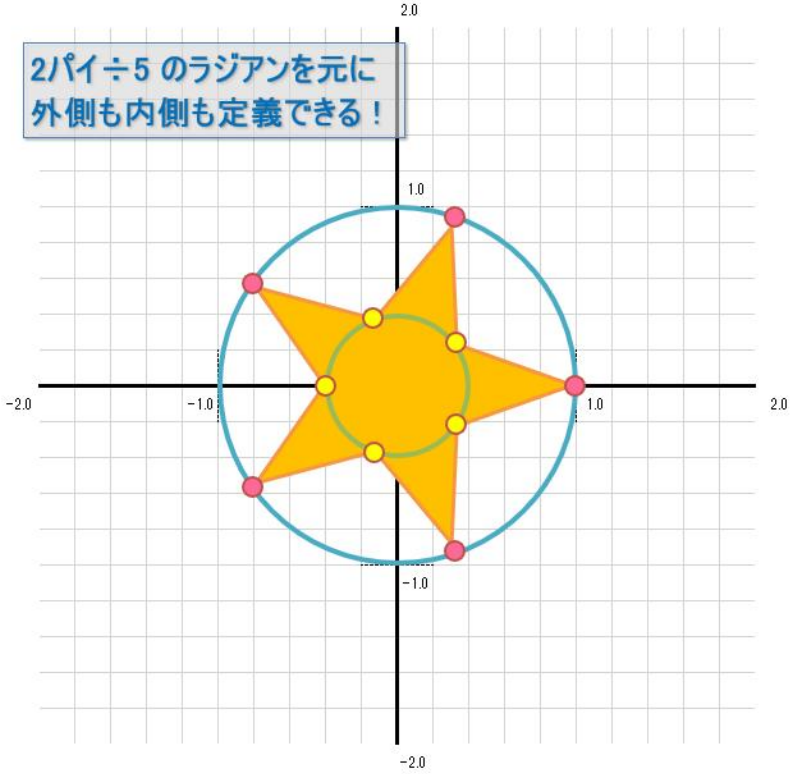
応用問題！

さて、それではサインとコサインの性質がわかったところで、第一回の講義のスライドにあった、五芒星（★）の形をポリゴンで描画することができるか、っていうのを考えてみましょう。

星型の場合、星の尖っている部分の先端がキレイに 5 等分されて円周上に配置されていると考えることができます。

また、凹んでいる部分の角も、やっぱりキレイに 5 等分されて円の円周上に配置されていますよね。

応用問題！



外側の始点位置を０度の場所にして、内側の凹んだ位置については 180 度の場所を起点にします。
そこから２パイを５等分した分だけのラジアンを使って順番にサインとコサインの値を求めてやれば、それで頂点の位置は求まります。

内側の場合は、半径を小さくするのも、地味にポイント！（サインコサインの結果に 0.5 とかを掛ければそれだけ半径の小さな円周軌道になる）

最後に

本講義のなかで、いたるところでベクトルや、サインやコサインといった概念が登場してきます。そんなときに、焦らず落ち着いて、スムーズに理解できるようになっていると、余計なことに思考が邪魔されない分、余裕を持って講義を受けることができると思います。

具体的にサインやコサインを使ったらどう嬉しいのかっていうのは、これからまた講義のなかでいろいろな例が出てくると思いますので、まずは簡単なところから、自分なりに予習してみたりするのもいいと思います。

五芒星を描いてみたり、あるいはシェーダエディタでサイン波を描いてみたりするのでもいいでしょう。わからないことは、いつでも質問してください！