3D 開発に必要な最低限の数学知識を知ろう

苦手意識を克服しよう！

はじめに

前回は、新しく覚えることが一度にたくさん出てきてちょっと大変でしたね。

three.js の基本的な機能をざっと紹介したような内容でしたが、とても多機能なライブラリなので盛りだくさんという感じになってしまいました。

さて、今回ですがタイトルどおり数学的な話が中心の講義になります。

しかし真正面から「数学やるぞ！」と言われると、ちょっと気持ちが引けてしまうというひとも実際多いと思います。

> 私も数学は苦手意識があります……

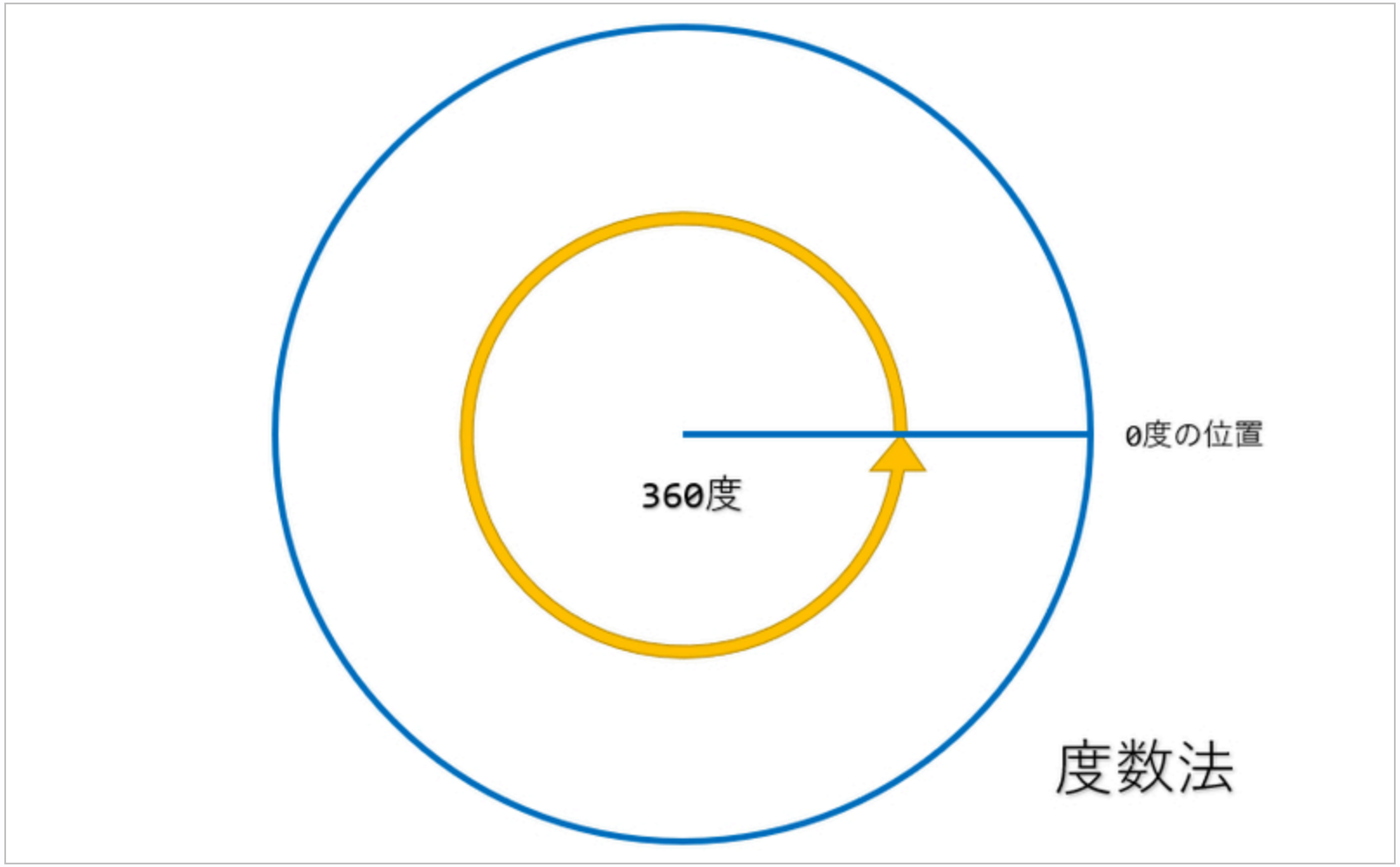
とは言え、3D プログラミングの分野で「数学を無視してレベルアップすることは不可能」です。

これは残念ながら避けようのない事実です…… できる限りわかりやすく解説していきますので、自分のペースで少しずつ、継続して取り組んでいきましょう。

> 図解すると理解が進むことが多いので、可能であればノートや、紙とペン等を用意しておきましょう

ラジアン

日本では多くの場合、角度は度数法を使って表します。360度、といったらこれは度数法です。



一方、プログラミングの世界では角度は度数法ではなく 弧度法 を用います。

弧度法では、角度のことを ラジアン と呼びます。JavaScript でも WebGL のシェーダでも、その他の言語でも、大抵は角度は弧度法（つまりラジアン）を使って処理していくことになります。

ラジアンなんて知ってるし！ という人も多いかもしれませんが、基本なのでしっかりおさらいしておきます。

弧度法では角度は度数法とは違って○○度、という言い方はしません。
ではどうやって角度を表現するのかというと角度の基準が **円（円周）** になります。

円周といえば、円周率。

3.1415926…… と続いていくあれです。数学ではよくパイという記号で表現します。（π）

ではみなさん、円周の公式思い出してみてください。円周を求めるための計算式って、どんなものだったでしょうか。

円周の公式

半径 r の円があるとき、その円周は……

円周 = 2 * PI * r

と表すことができる。

このとき、円の半径（r）が 1.0 だとしたら、次のように表せます。

```
半径 1 の円の円周 = 2 * PI
```

> これがスッとイメージできれば、もうはっきり言ってラジアンは完全理解したようなものです

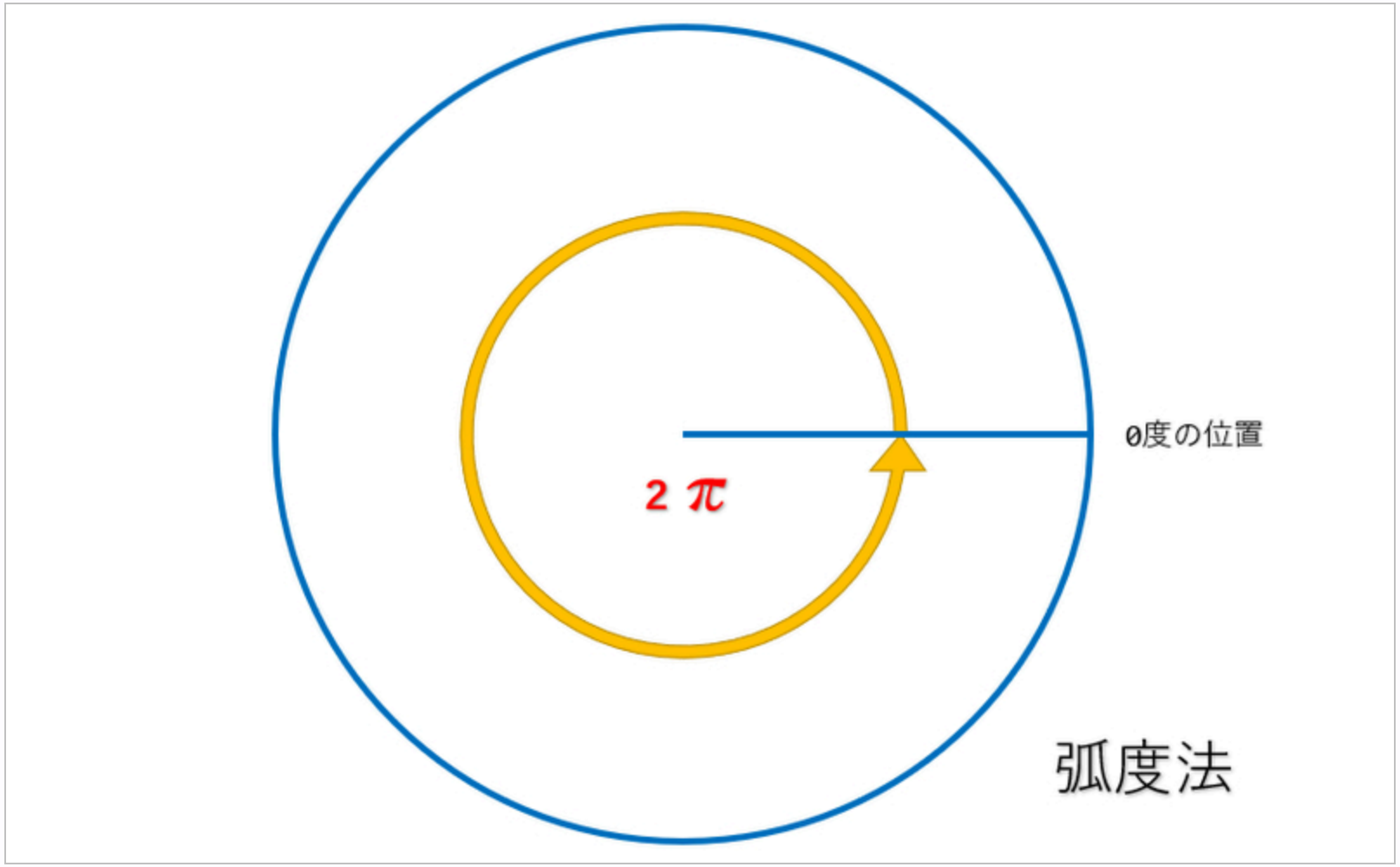
ここで出てきた、**半径１の円の円周は２パイ** という事実、これが超重要です。

度数法では円の一周は 360 度でしたね。弧度法では、一周が 2 パイになるんです！

```
度数法だと……円周 = 360度
弧度法だと……円周 = 2パイ≒（6.2831853……）
```

> 角度の基準が度数とラジアンで異なるだけで、やってることは同じ！

ラジアンでの角度表現。



ラジアンに慣れよう

それでは簡単なテストで試してみましょう。

- 度数法の 180 度は、ラジアンでは？
- 同様に 90 度だとラジアンでは？
- 1 ラジアンって度数法ではおよそ何度？

> 時間がかかってもよいので、度数法と弧度法の相互変換が自力でイメージできれば十分！

JavaScript では `Math.PI` でパイが得られます。 `Math.sin` などの算術系メソッドは 引数をラジアンで 受け取ります。

実際に、ブラウザの開発者ツールのコンソールなどで計算してみましょう。

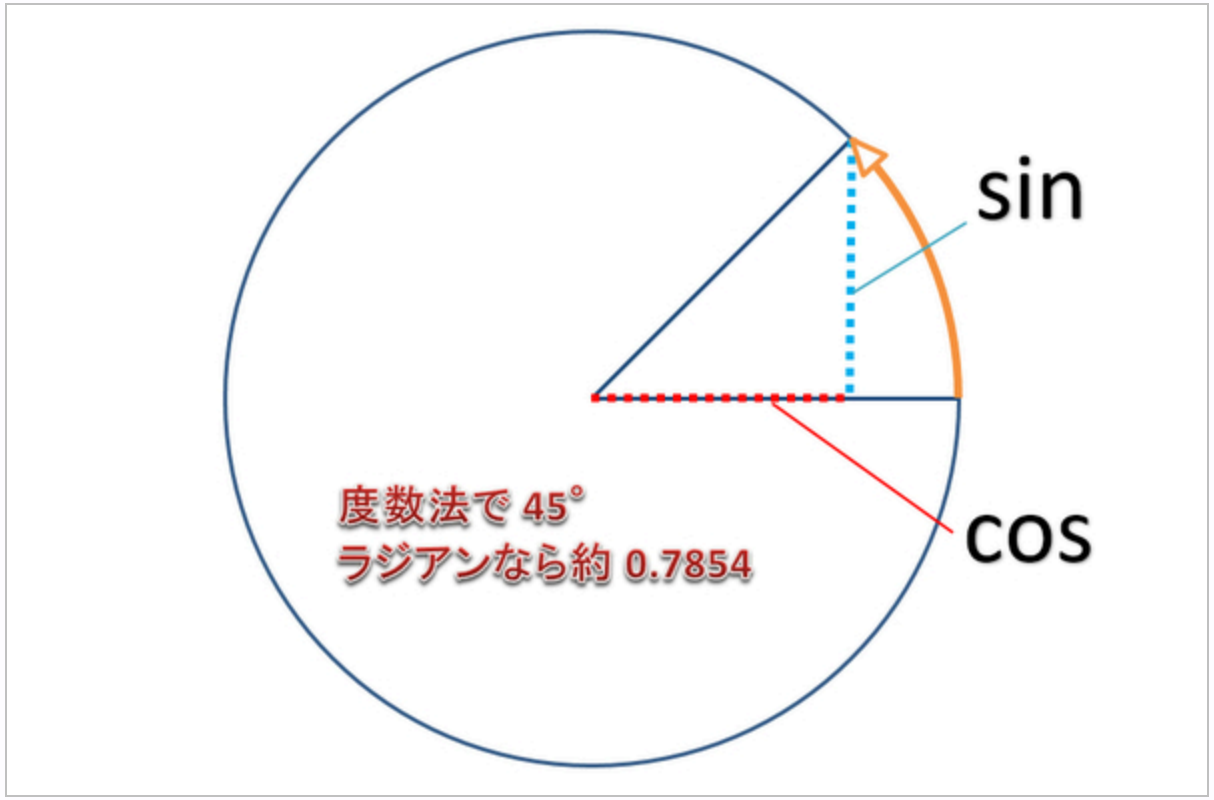
- 60 度のサインはいくつになるでしょう？
- では 270 度のサインは？
- 0 度のコサインはいくつになる？

※一応答えの導出例を書いておきます

```
- 60 度のサインはいくつになるでしょう？
    Math.sin((Math.PI * 2.0) / 360 * 60)

- では 270 度のサインは？
    Math.sin((Math.PI * 2.0) / 360 * 270)

- 0 度のコサインはいくつになる？
    Math.cos(0)
```



サインやコサインとラジアンの関係はめっちゃ簡単に言うと、二次元平面であればコサインが横方向の移動量、サインが縦方向の移動量です。

ラジアンや弧度法は、ちょっと耳慣れない言葉かもしれませんが、単にパイを使って角度を表しているだけだとわかれば、全然怖くは無いと思います。念のため、度数法の度数からラジアンへの変換式を以下に書いておきます。

```
radian = degree * Math.PI / 180;
```

> パイを掛けて 180 で割るだけ！ 簡単！

ラジアンまとめ

- 馴染み深い角度「360 度」のような表現は度数法
- 円周率をベースに角度を表すのが弧度法
- 弧度法では角度はラジアンという言葉で表す
- 360 度 == 2 パイ
- プログラミングでは弧度法を用いるのが基本

ベクトル

さあ、次はベクトルです。

ベクトルは、学校の数学でやったことある人も多いかもしれませんね。
でも、正直言って、私は全く憶えてなかったですね！ 😇😇😇

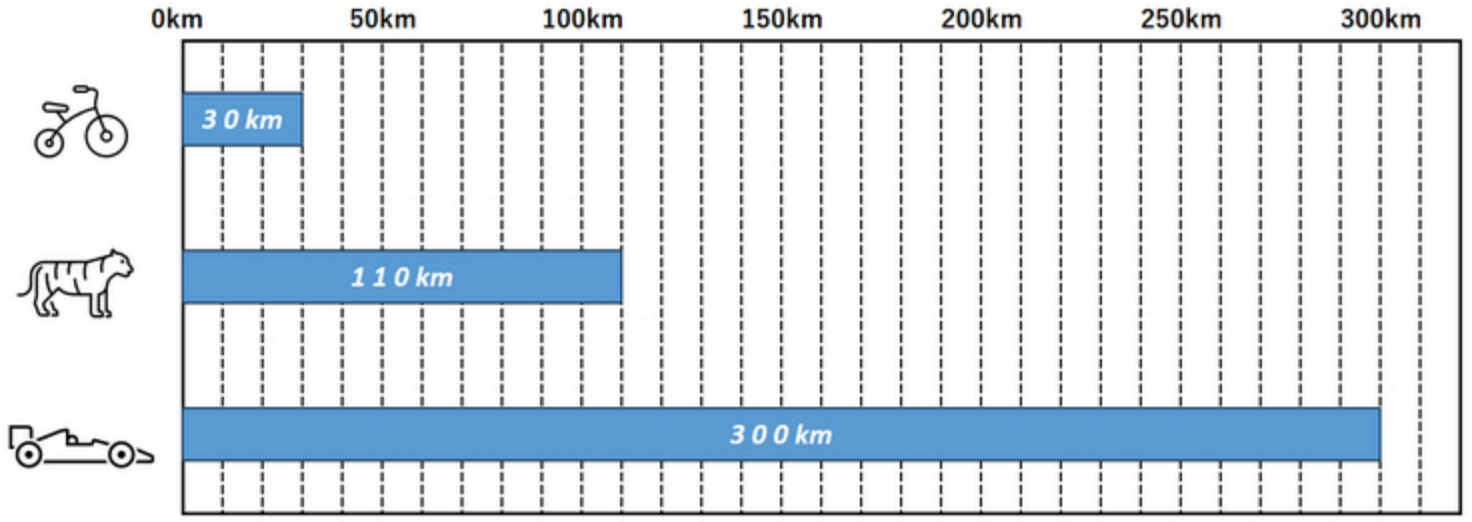
> ここでは 3D 開発に必要となる部分を中心におさらいしましょう

ベクトルとは、よく 向きと大きさを持った量 というふうに言われます。この日本語、いまいちよくわからないですよね。

でも、ベクトルがわかってくると、言い得て妙というか、確かにそういうふうにしか考えられなくなってきます。

それでは、まず最初にいったんベクトルというのは思考の片隅に追いやっておき…… なにかの速度を表す場合を考えてみましょう。

速度なので、わかりやすく時速で考えてみると、速度を表現するためには一次元、つまり単体の数値だけでいろいろなことが表現できることがわかります。



一次元なので、グラフ化する際は水平線ひとつで値の大小を表現できる

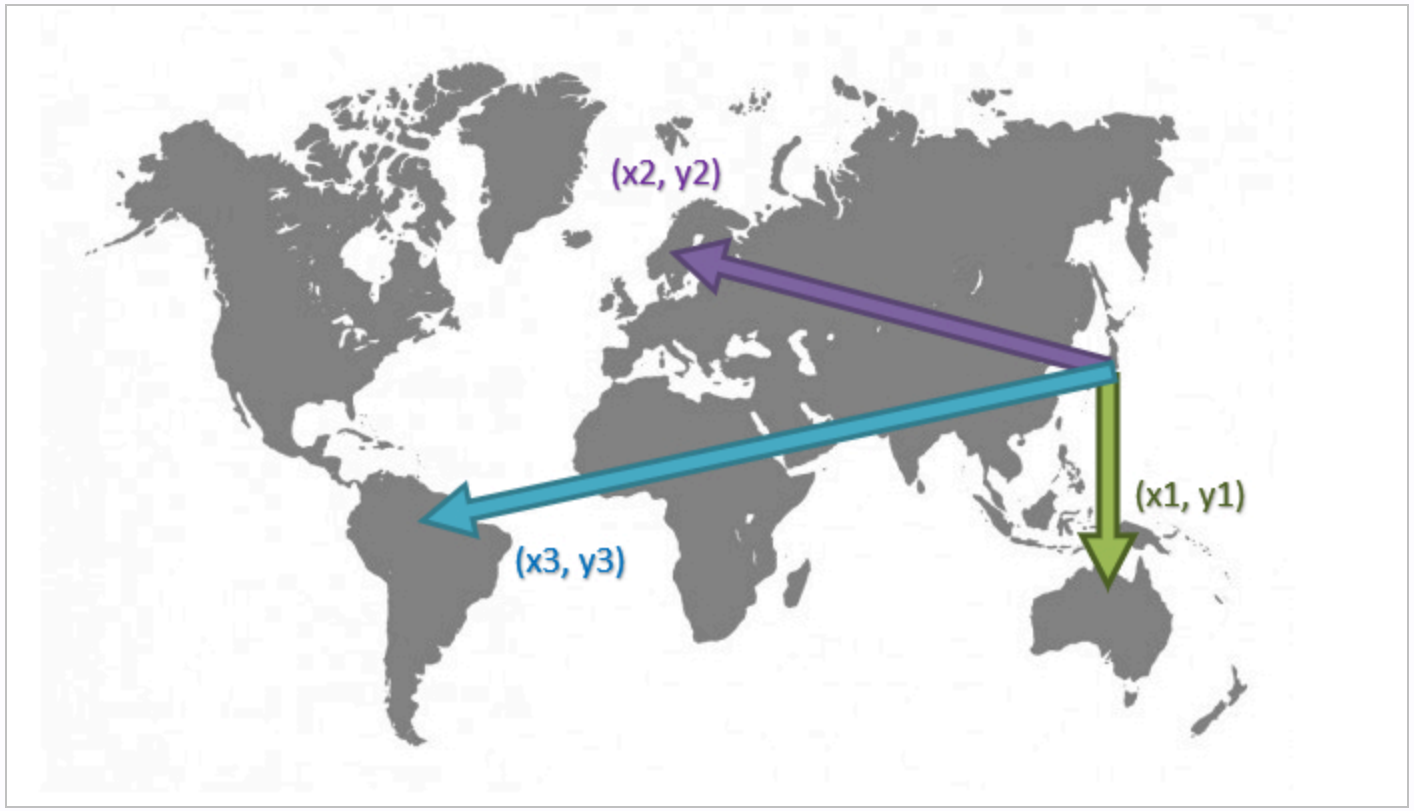
このように、扱う数値がひとつだけの場合は、これは単にひとつの数値です。

ベクトルや行列が登場する線型代数学の分野では、このような単体の値のことを「スカラー」と呼んだりします。

> 単体の値とベクトルとを明確に区別するためにそれぞれに固有名がある

では次に、地図上で目的地を指し示す場合を考えてみましょう。

そうすると、今度は一次元（単体の数値）では表現できないことがわかります。平面的な地図の上で、どの方角に向かうのかということを表すためには 次元を増やして（この場合は二次元で）情報を管理 しなければならなくなります。



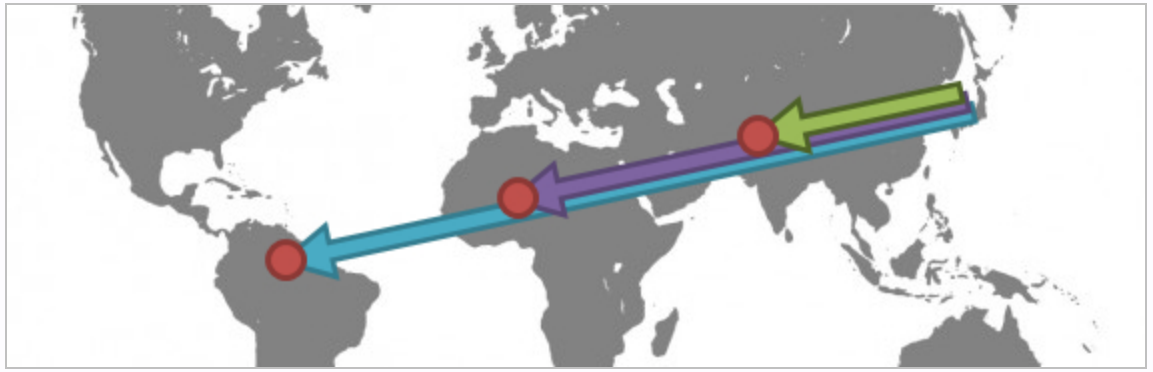
方向を表現するには複数の次元が必要になる

これが噂の「ベクトル」です。

スカラーの場合とは異なり、X と Y のような複数の数値を組み合わせて定義されるものをベクトルと呼ぶわけです。

さらに言うと、ベクトルには最初にも書いたように「向き」と「大きさ」があります。

まったく同じ向きのベクトルでも、その大きさ（長さとも言う）が違う場合があるんです。これがベクトルが 向きと大きさを持つ量 と言われる理由です。



> 全部が同じ向きだけど、大きさは様々

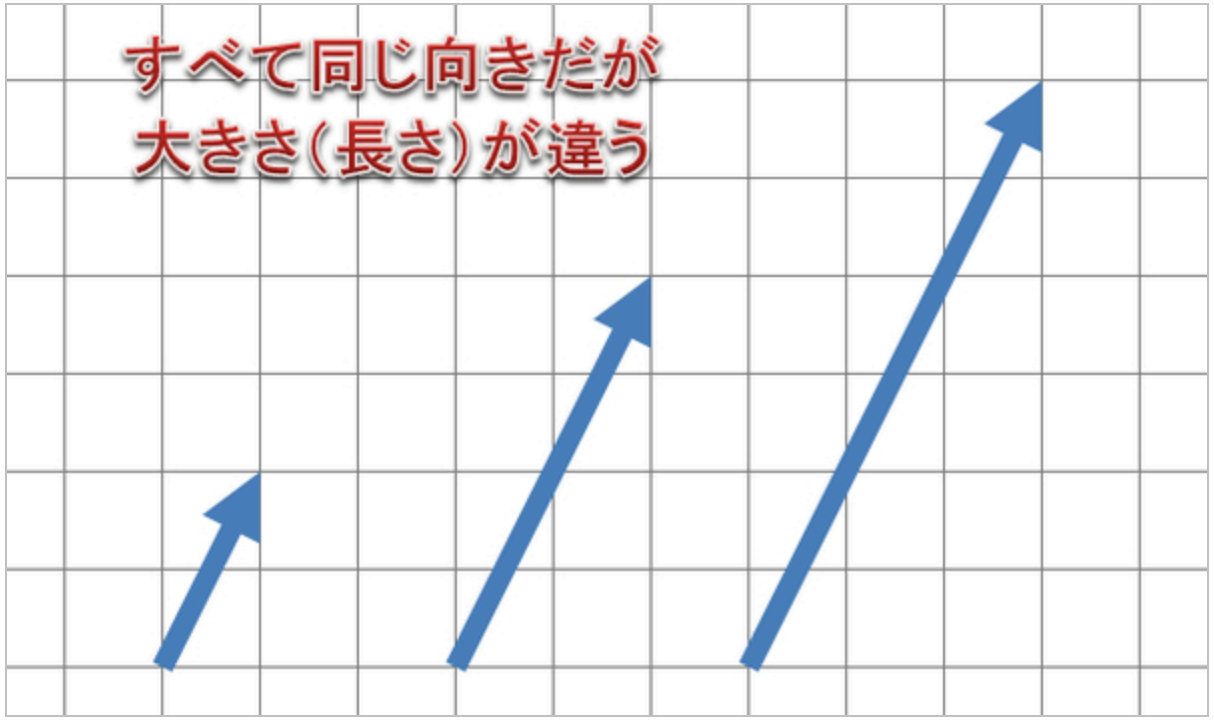
ベクトルまとめ

- ベクトルは複数の要素が組み合わさって定義される
- XY なら二次元ベクトル、XYZ で三次元ベクトル
- ベクトルには向きがあり……
- 向きが同じでも大きさが違うことがある
- このことからベクトルは向きと大きさを持つ量、と呼ばれる

単位ベクトル

ベクトルは、向きと、大きさを表現できる便利な概念であることがわかりました。

先ほども書いたとおり、同じ向きのベクトルでも、その大きさは様々に定義できます。



> 向きと大きさという概念は常にベクトルにセットで含まれている

しかし 3D 数学では時折、ベクトルの 向きだけに注目したい場合 というのが出てきます。

そんなときに出てくるのが **単位ベクトル** と呼ばれる特殊なベクトルです。

特殊なベクトルと言っても、実際には全然難しくはありません。

単位ベクトルとは、ただ単に 長さが 1 のベクトル のことを言います。

サインやコサイン、あるいはラジアンを考えるときもそうでしたが、半径が 1 の円を基準にいろいろ考えましたよね。

あれと同じように、ベクトルも基準となる長さを 1 とすることで、向きや角度と一緒に扱いやすくなります。

> 別の言い方をすると、長さを 1 にするという前提条件を設けることで、長さ以外の「向きという属性だけに注目することができる」とも言えます

そして驚くべきことに、どんな長さのベクトルであってもそれらは 必ず単位ベクトルに変換 できます。

数値はその数値自身で割ってやれば 1 になる という一般的な数値の性質と同じように、ベクトルの長さを求めその長さでベクトル自身を割ってやる、という手順を踏むことでベクトルを単位化することができます。

ベクトルの長さを求め単位化する

まずは、ベクトルの長さを算出する方法です。ベクトルの長さ（大きさ）は、各要素同士を掛け合わせてからその平方根を取る、という方法で取得することができます。

```
ベクトル V = [x, y];
ベクトル V の大きさ = Math.sqrt(x * x + y * y);
```

> いわゆるピタゴラスの定理

たったこれだけの計算で、ベクトルの長さ（大きさ）を測ることができます。

そして、先程も説明したように、その大きさでベクトル自身を割ってやれば、これで長さが 1 のベクトル、すなわち単位ベクトルを得ることができます。

まとめると、ベクトルを単位化するときは以下のようにします。

これは二次元でも三次元でも、あるいはそれ以上に次元が増えても同じで、各要素同士をかけ合わせて平方根を取り（これが大きさ）それで各要素を割ってやれば OK です。

```
ベクトル V = [x, y];
ベクトル V の大きさ = Math.sqrt(x * x + y * y);
V の単位ベクトル = [x / 大きさ, y / 大きさ]
```

このように、ベクトルを基準となる長さ（1）に揃えることを **単位化**、あるいは **正規化** と言います。

もし今後の講義のなかで「ここでベクトルを単位化（正規化）してます」という風に出てきたときは、長さを 1 にそろえているのだなと思ってもらえればいいわけです。

単位ベクトルが必要なときとは？

また、単位化する必要がある場面とはどういうときか、ということについてもちゃんとここで理解してしまいましょう。

単位化すると、ベクトルの長さが 1 になります。この状態は、数学的には 向きだけに注目する のにとても都合の良い状態です。

> なぜ都合が良いのかは続けていくうちに少しずつわかってくると思います

つまり、ベクトルの大きさはいったん無視して どっちを向いているのかだけに注目したいとき ベクトルを単位化します。

今はイメージしにくいかと思いますが、ベクトルの単位化は 3D 数学ではいたるところで登場しますので、ベクトルの向きに注目する場合単位化する ということをしっかり覚えておきましょう。

実際に使いながら覚えていこう

さて、ラジアンやそれを用いて求めるサイン・コサイン、そしてベクトルとその単位化（正規化）など、いろいろ出てきましたが……

これらは普段の生活のなかで常用するものではないと思いますので、意識して使っていかないとなかなか身につきません。ここからは、実際にこれらを使ってどのようなことが行えるのかを見ていきましょう。

まずこれらの数学的な知識を活用しやすいように、土台となるサンプルを作ったものがサンプル 017 です。

実行してみれば明らかですが、シーンには地球と月を模した球体が置かれています。月は X 軸上に重なるようにすこし動かした位置に描画されるようにしています。

サンプル 017 では特に新しい技術などは出てきておらず、これまでに取り組んできた内容を組み合わせただけです。

テクスチャのための画像ファイルを複数読み込む必要があるので、無名関数を組み合わせてロード処理を行っています。今回の講義では、これをベースにいろいろと改造しながら数学の理解も深めていきましょう。

017

- 地球と月とを配置する、ベースとなるサンプル
- テクスチャ用に画像を複数読み込む
- テクスチャが異なるためマテリアルも複数用意
- あくまでもベースなので、スペースキーを押しても地球も月も定速で回転するだけ

サイン・コサインを使って月を動かす

サンプル 017 がどのような状態なのか確認できたら、まずはラジアンと、それを用いたサイン・コサインによる実装から始めていきます。

サンプル 018 では、時間の経過をラジアンとみなして、それを元にサインとコサインを求めています。

今回のサンプルのように、時間の経過を利用するタイプの実装では、やり方は様々考えられますがレンダリング開始直前の時刻を変数に確保しておくのがわかりやすいでしょう。

最初に変数に開始時刻（のタイムスタンプ）を取得しておけば、あとからその差分をもとに経過時間を求めることができます。

JavaScript の `Date.now()` を用いると現在時刻のタイムスタンプを得ることができます。

このタイムスタンプはミリ秒単位の整数値なので `Date.now()` を呼び出すたびにそれらの差分を計算するようにしてやれば、どのくらい時間が経過したのかを調べることができます。

> タイムスタンプの単位はミリ秒であることに注意（めちゃでかい整数）

記述例

```
// 最初に時刻を保持しておく
const startTime = Date.now();

（中略）

// ミリ秒単位の大きな整数なので 1000 で割って秒単位にする
const nowTime = (Date.now() - startTime) / 1000;
```

なお、three.js にはこういった処理を行いやすくするための `THREE.Clock` がありますので、これを使って同様のことが行なえます。

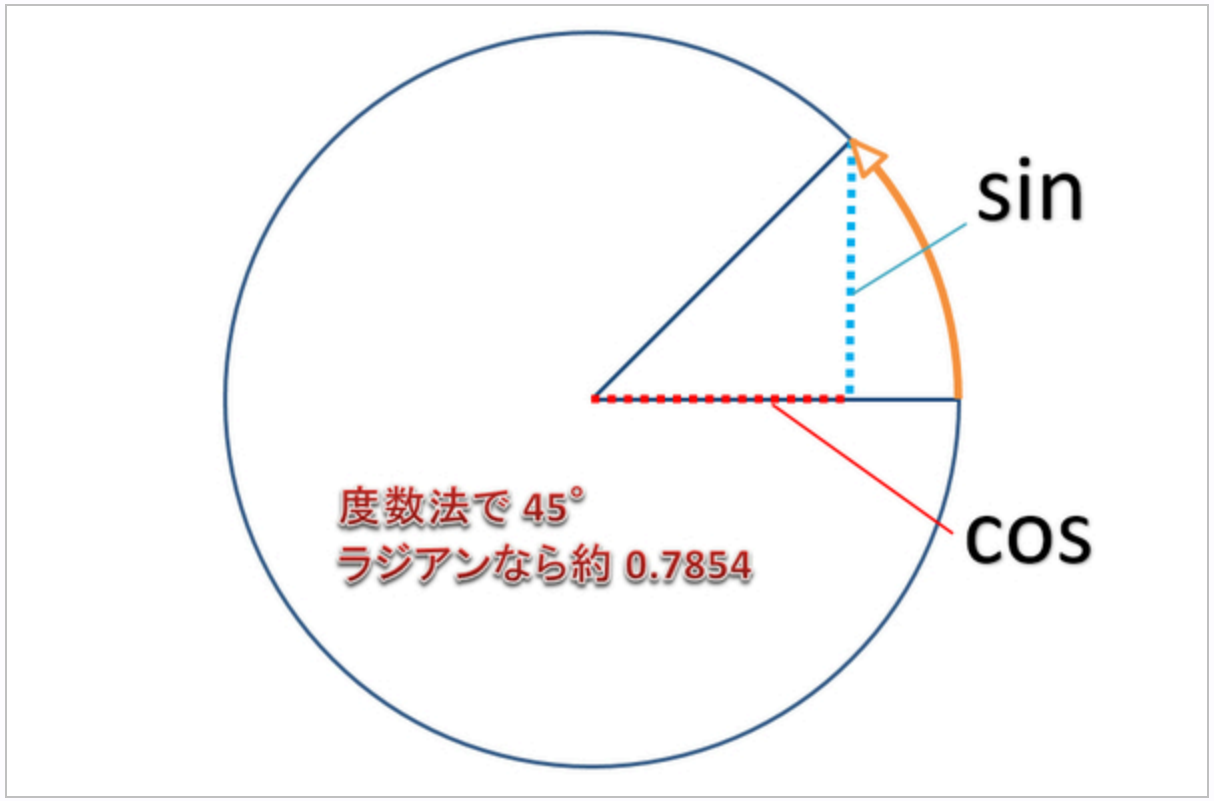
```
// インスタンスの生成
const clock = new THREE.Clock();

// 経過時間を得る（単位は秒）
const nowTime = clock.getElapsedTime();
```

参考： THREE.Clock

無事に経過時間が得られたら、あとはこれを `Math.sin` などに与えて、サインやコサインを求めます。

次のページの図を見ながら、サインとコサインの結果がいったいどういうことを意味しているものなのか把握しましょう。



二次元なら、サインが Y、コサインが X に相当する

018

- 時間の経過を知りたい場合は「開始時間」を保持しておき……
- `Date.now` との差分を求めることで経過時間がわかる
- three.js では `THREE.Clock` を使えば同様のことができる
- サインやコサインの結果の範囲は -1.0 ～ 1.0 なので……
- 必要に応じてスカラー倍してから使う

> サインやコサインの結果がなぜ -1.0 ～ 1.0 になるのか説明できるでしょうか？ また、時間の経過をラジアンとみなしているとしたら、一周するのに何秒掛かるでしょうか？

スクリーン空間と 3D 空間

サインやコサインで得られた値を、月の座標位置に設定してやることで「円運動」を行わせることができました。

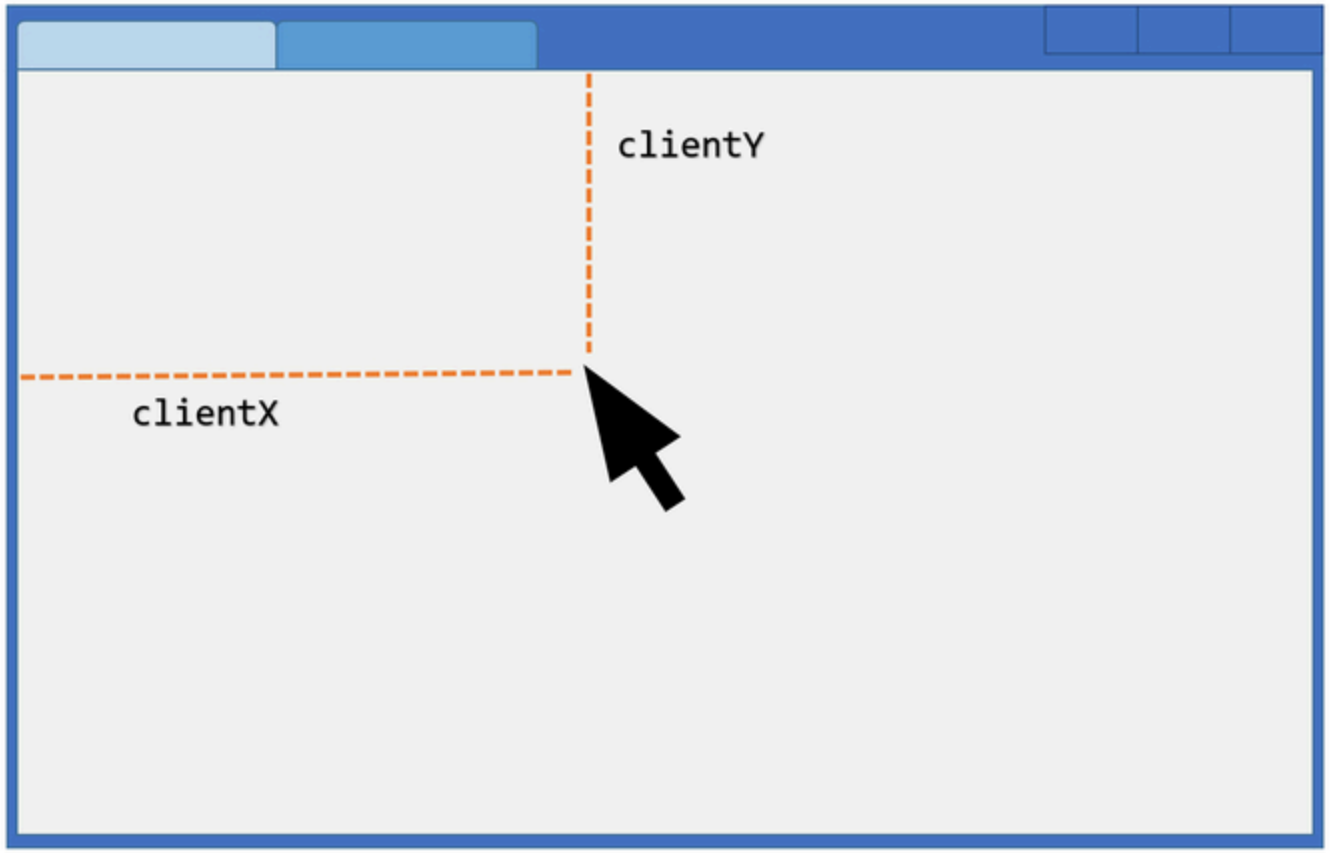
続いては、マウスカーソルの位置をリアルタイムに検出しながら、それを 3D シーンに反映させるような処理を行ってみましょう。

JavaScript では `mousemove` や `pointermove` イベントを設定することで、カーソルが画面上で動いたことを検出することができます。

このときのカーソル位置は `addEventListener` に登録したリスナーに引数として渡されてくるイベントオブジェクトを通じて調査することができます。

```
window.addEventListener('mousemove', (event) => {
  // マウスカーソルのスクリーン空間上での位置
  const x = event.clientX;
  const y = event.clientY;
   (中略)
```

`MouseEvent.clientX` が、スクリーンの左上角を原点とするカーソルの X 座標、同様に `clientY` が Y 座標。



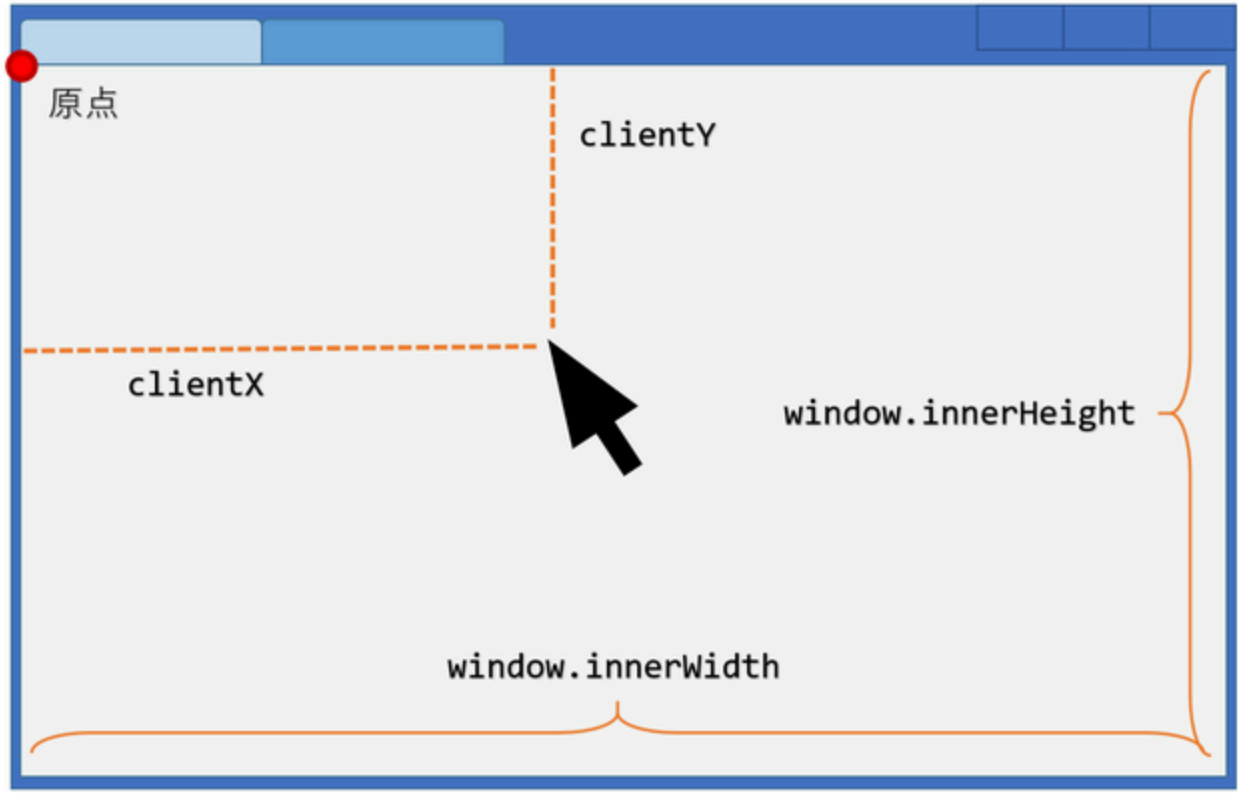
画面の左上を原点にした座標系なので左側からの距離が clientX 、上側からの距離が clientY となる

イベントオブジェクトから得られる `clientX` や `clientY` といった値は「スクリーン空間上のピクセル単位」の座標です。つまり、スクリーン座標系という座標系における値です。

これを 3D 空間で扱いやすくするために、画面の幅や高さを使って変換してやります。

このような「自分が扱いたい座標系に応じて、座標を変換してから使う」という処理は、3D プログラミングでは結構よく出てきます。

最初はちょっと違和感というか、なぜそれをしなければならないのかわかりにくいかもしれませんが、今回の場合は「原点の扱いやスケール感を揃えたいので変換したい」という意図があり、このような座標変換を行っています。

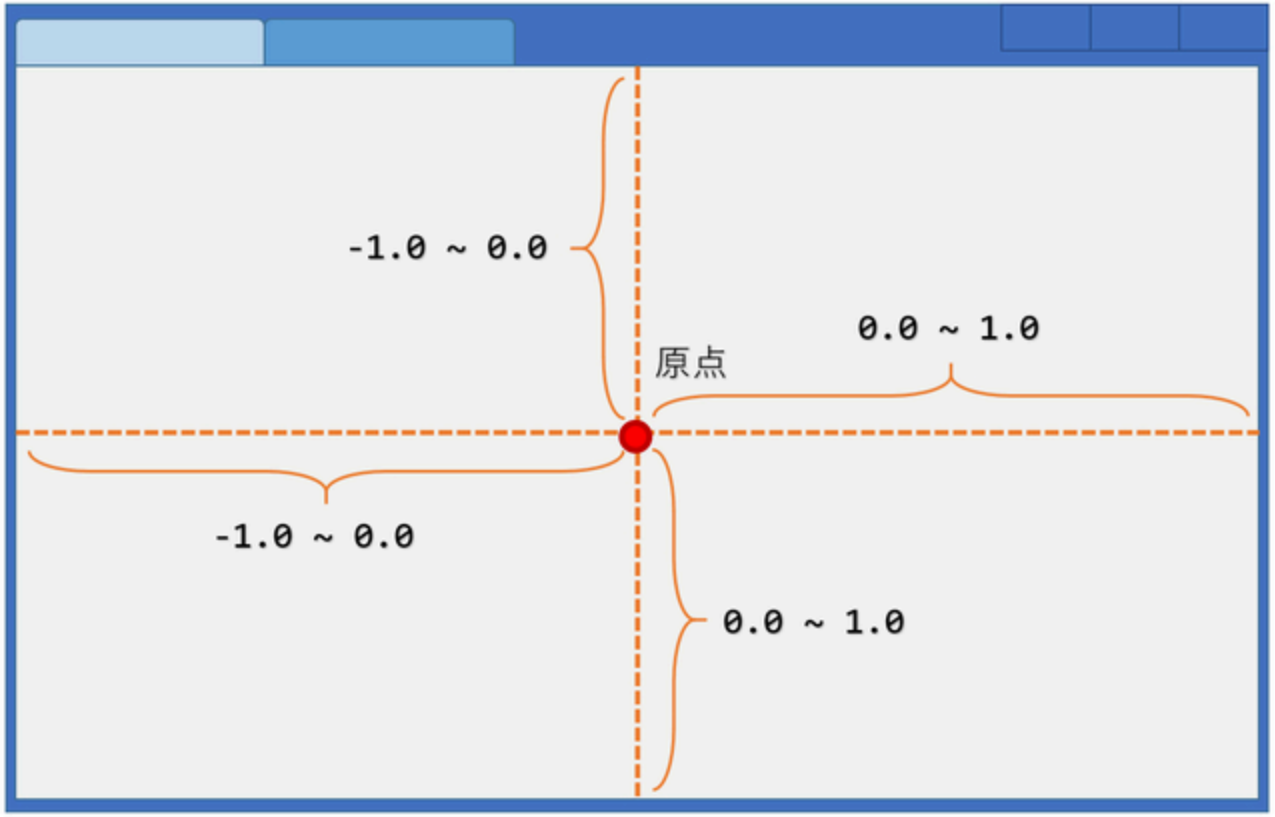


clientX / innerWidth で、マウスカーソルの X 座標を元に、0.0 ～ 1.0 の範囲の結果を得られる

clientY / innerHeight で、マウスカーソルの Y 座標を元に、0.0 ～ 1.0 の範囲の結果を得られる

その結果を、２倍してから、１を減算する

最終的に、原点が画面の中心にあり正負にそれぞれ 1.0 の幅を持つ座標系へ変換できる



最終的に、原点が画面の中心にあり正負にそれぞれ1.0の幅を持つ座標系へ変換できた場合でも、3Dの座標系とはYの扱いが反転している点には注意！

※場合によりYだけ反転して使えばよい

019

- マウスカーソルの位置を検出して処理を行う
- スクリーン空間は左上角が原点で、ピクセル単位
- 原点の扱いやスケール感が、3D シーンとはまるで異なっている
- 一度マウスカーソルの座標を -1.0 ～ 1.0 に変換する
- 変換したことで、任意に定数倍して使いやすい形に

> 座標変換は 3D に限らずグラフィックスプログラミングの基本中の基本であり、3D では信じられないほどいろんなところでこのような座標変換が出てきます（難しく感じるかもしれないですが、簡単な計算であることが多いので落ち着いて考えましょう）

ベクトルの単位化を利用した位置指定

さてカーソルの位置に応じて月の座標を決めることができるようになりましたが…… この状態では「月が場合によって地球に激突！」みたいなことが起こってしまいます。

ここでもし「月は常に、地球から一定の距離を保つ」という処理を行う必要があるとしたら、一体どのような計算を行えばいいでしょうか。

> 実際の月も、地球から常に一定の距離を保っていますよね

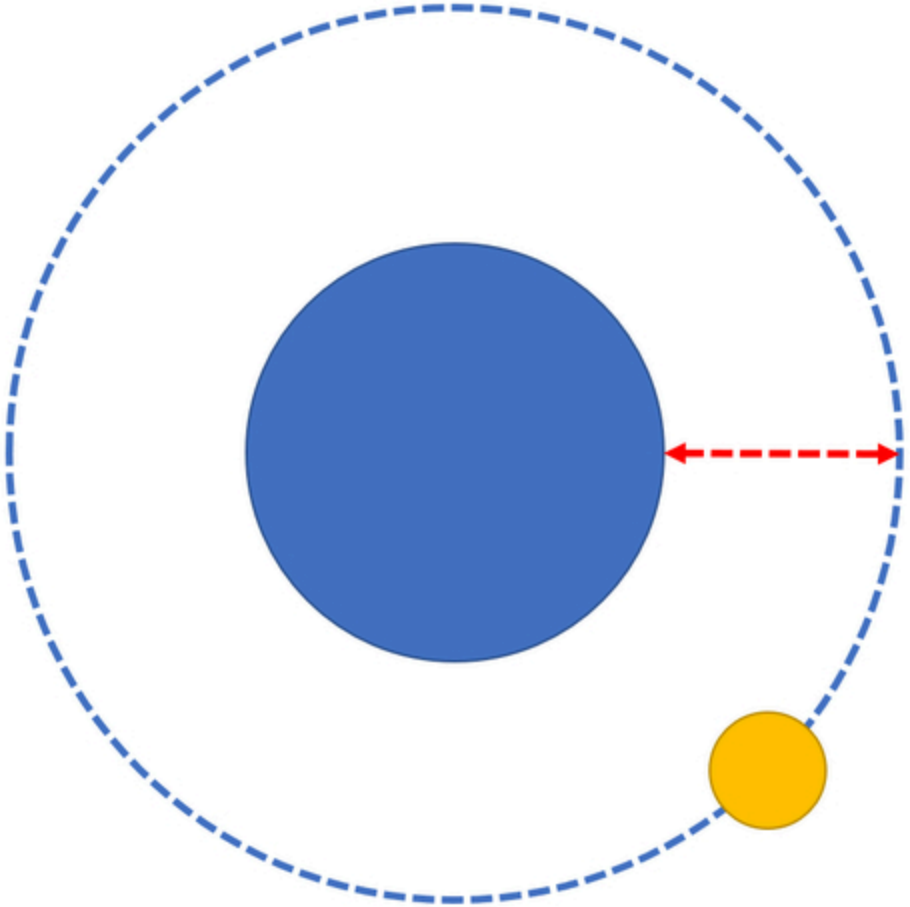
こういうときに役に立つのが「ベクトル」の知識です。

月と地球の相対的な距離の関係性が崩れないようにするために、マウスカーソルの位置を「ベクトルとみなして、単位化してから利用する」ことでこれを実現します。

ベクトルの単位化とは「向きだけに注目したい場合に用いる」と説明したかと思いますが、今がまさにそのときです。

月と地球の距離を一定に保つ、つまり「距離」は常に一定になる前提です。であれば、注目すべきは「地球から見て、月がどっちの方角にいるか」だけです。

> ちょっと言葉で書くと紛らわしいけど落ち着いて考えよう！



赤矢印で示されている「距離」については、今回の場合常に一定であることが前提になっているので、気にしなくてよい。

問題は、地球から見たときに月がどの方角にあるべきなのかなので……
つまり「向き」だけを考えればよい！

おさらいになりますが、ベクトルの単位化は以下のようにすればできます。

```
ベクトル V = [x, y];
ベクトル V の大きさ = Math.sqrt(x * x + y * y);
V の単位ベクトル = [x / 大きさ, y / 大きさ]
```

ベクトルは単位化することで「長さが 1 のベクトル」になるので、あとはこれに定数を掛けてやるだけで、常に一定の距離を保つような処理が実現できます。

最初はどうしても難しい感じがするかもしれませんが「この場合はどっちの方向にいるかだけがわかればいいので…… 単位ベクトルが必要だな！？」というように、徐々に自然と考えられるようになってくると思います。

020

- XY それぞれを乗算してから足し合わせて……
- 平方根を取ることでベクトルの長さがわかるので……
- XY それぞれを長さで除算（割る）ことで単位化が行える
- 単位化することで向きだけを考えればよくなり……
- 距離感は、あとからどのような定数を掛けたかによって自由に決められる

> three.js ではベクトルの単位化などもメソッドで簡単に実現できます

ベクトルの始点と終点

これまでのサンプルでは、マウスカーソルの位置ベクトルを「画面の中心を原点とした座標系」として求めて利用してきました。しかしベクトルは、なにも原点から伸びる形しか存在しないというわけではありません。

地点 A と地点 B というように、ベクトルの 始点 となる位置と 終点 となる位置がわかっていれば、相対的な距離を求めることでベクトルは任意に定義できます。

サンプル 021 では、まさにこれに相当することを行っています。

カーソルの位置に連動して動く月は従来どおりですが、人工衛星が「月を追尾するように動く」ようになっています。

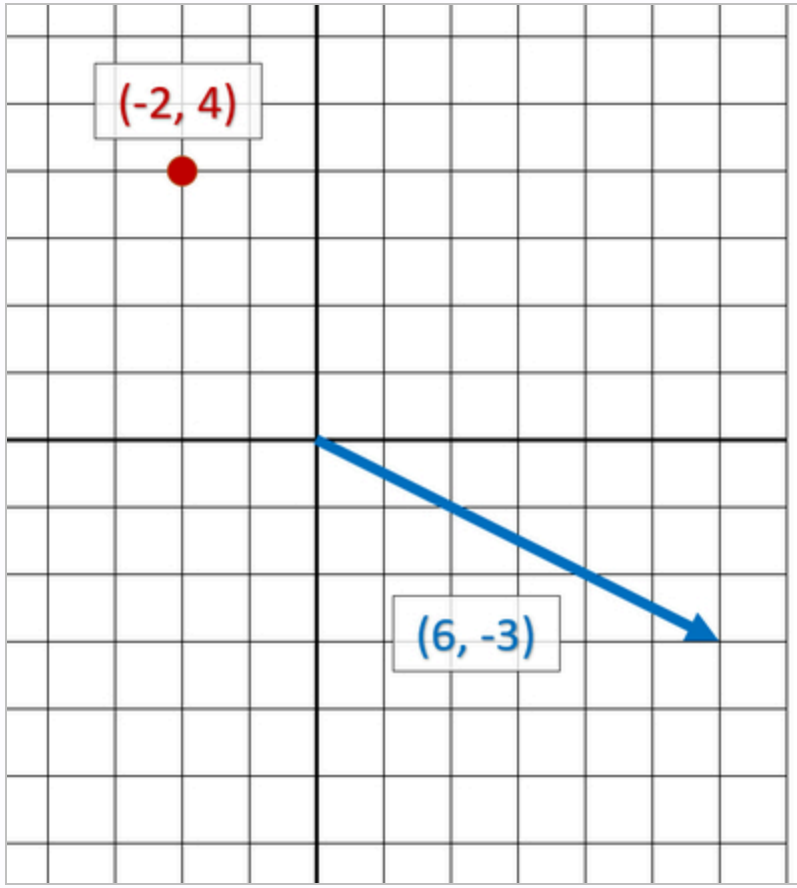
実際に動かしてみると、なんだかこんな挙動をさせる実装を作るのはすごく難しそう！ って思うかもしれませんね。

でも、考え方は実は結構シンプルです。

> 落ち着いて考えればきっと理解できます！

ベクトルは、原点から任意の座標に向かって伸びる場合や、ベクトル（x, y）のように表記される場合、これは暗黙で 始点は（0, 0）、つまり原点である と仮定して話をしています。

（0, 0）の位置を始点にして、任意の（x, y）に向かって伸びるベクトルですよ～ というふうに暗黙に定義しているわけです。



①

座標 (x, y) という場合、単純にそれは位置を表している。

②

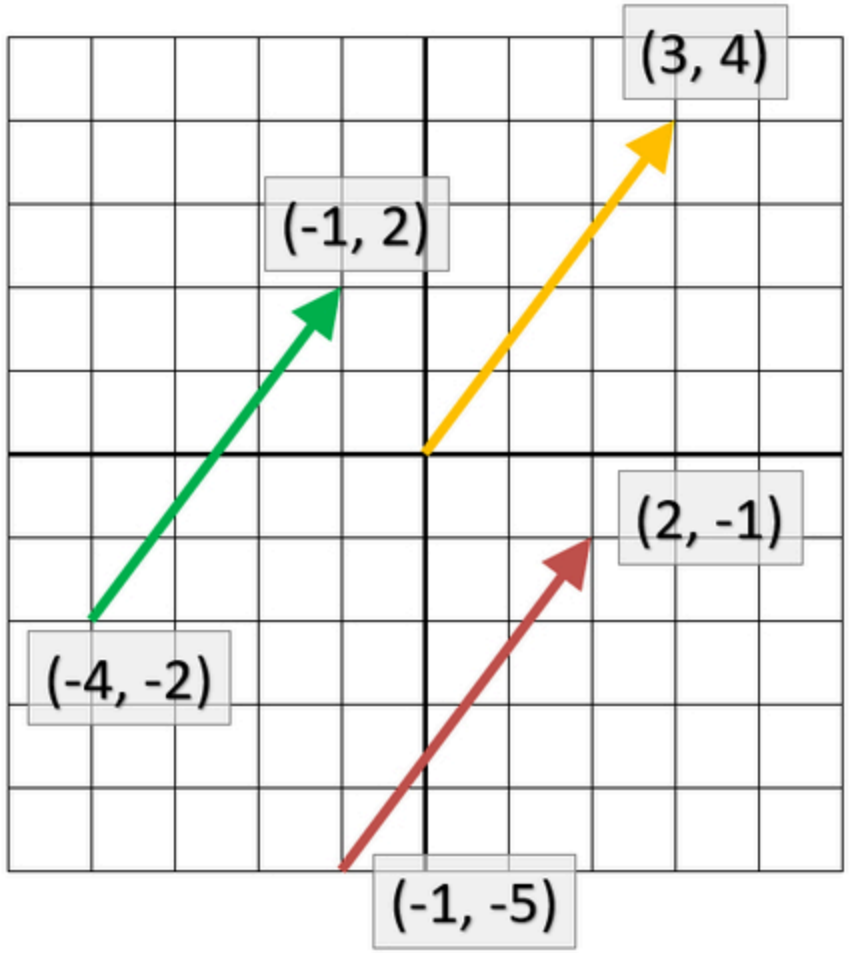
ベクトル (x, y) という場合、暗黙で原点を始点としたベクトルを表している。

“表記方法は同じでも、意味するところはまったく違う”

一方で、地点 A と地点 B のように、始点も終点も任意の座標になっている場合は……

これも図に描き起こすなどして冷静に考えてみると自明なのですが、お互いの相対的な距離を求めればベクトルは簡単に定義できます。式で書くなら、以下のようにすればいいですね。

```
任意のベクトル = [終点.x - 始点.x, 終点.y - 始点.y];
```



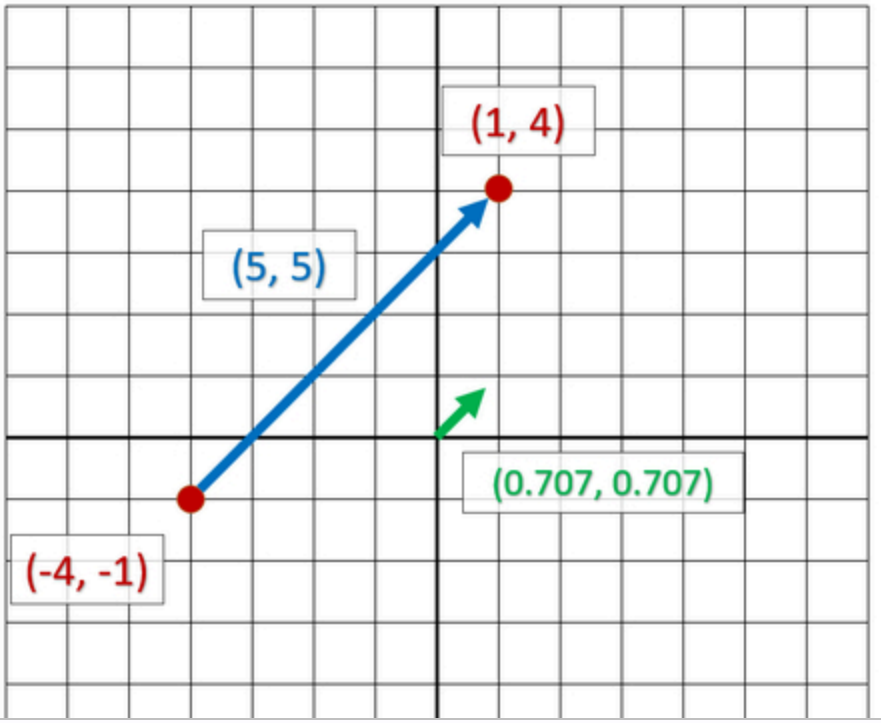
終点 - 始点 で、
どれも同じベクトルになっている！

終点の記載しかないときは、暗黙に原点が始点になっているだけ！

このように、任意の２点が求まる時、ベクトルは簡単に定義できるのですね。

そしてベクトルが定義できるということは、そのベクトルの方向にオブジェクトを進ませることもできるはずです。終点 - 始点、という計算で求めたベクトルを単位化し、月を追尾する人工衛星がどっちに進めばいいのかを決めてやればいいわけです。

> 二次元でも三次元でも、考え方は同じです



始点と終点がわかれば、相対的な位置関係からベクトルが求められる。

ベクトルが求められさえすれば、これを単位化して単位ベクトルを求めることができる。

> この単位ベクトルこそが進むべき方向を指し示している

021

- 人工衛星は月に向かって移動し続ける
- 月の座標に向かって移動したいので、つまりこれが「終点」であり……
- 人工衛星の現在の位置が「始点」であるとみなすことができる
- 終点 - 始点、という計算で進行方向ベクトルを求める
- 向きだけに注目したいので単位化し人工衛星を平行移動させる

ベクトルの加減算

さてどうでしょうか。少しずつ、ベクトルを使う感覚が、わかってきたでしょうか。

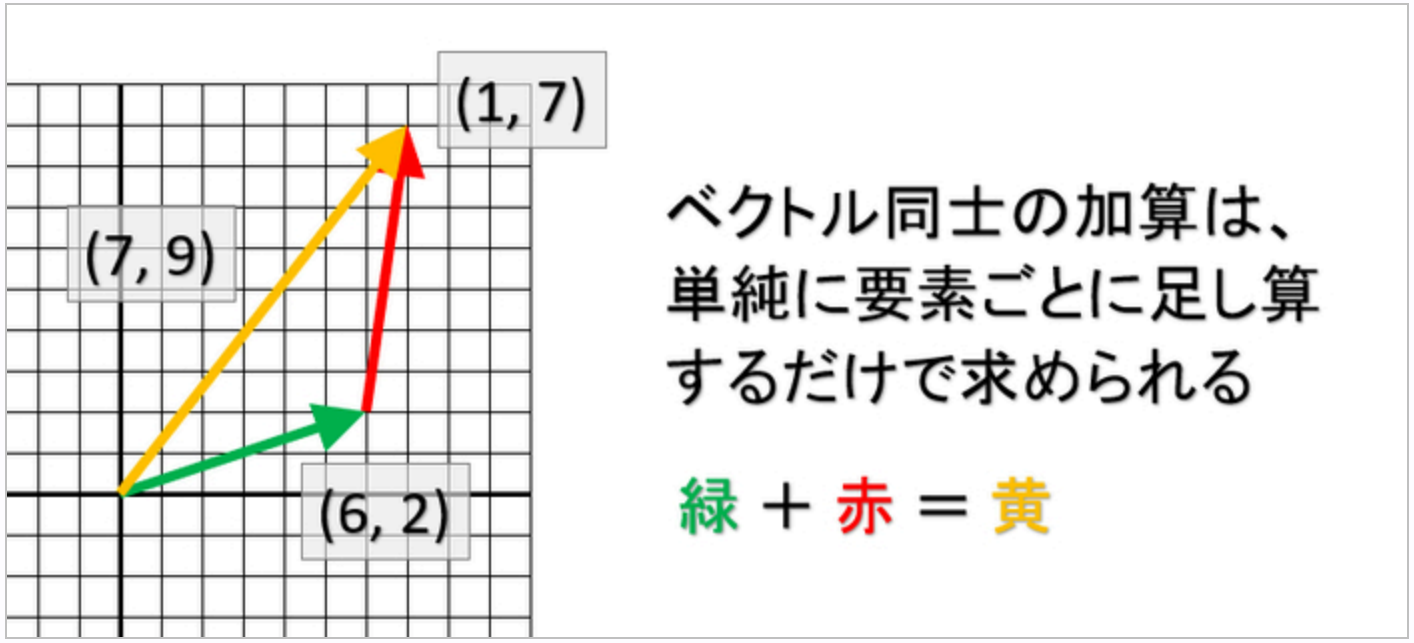
もし、すぐにはパッとイメージが沸かなかったとしても、最初は仕方ない部分もありますから少しずつ慣れていけば大丈夫です。

ここからはベクトルの加減算についてもう少し深掘りして考えてみましょう。

これもそんなに難しい話じゃないので、落ち着いて考えてみてください。

これまでベクトルの単位化や、２点間を結ぶベクトルの定義の仕方などについて見てきましたが、ベクトルは通常の数値と同じように 加減算 が行えます。

足したり、引いたり、といったことができるわけです。



要素ごとにそれぞれ足したり引いたりすればいいだけ

このようなベクトルの加算処理を応用して作られているのがサンプル 022 です。

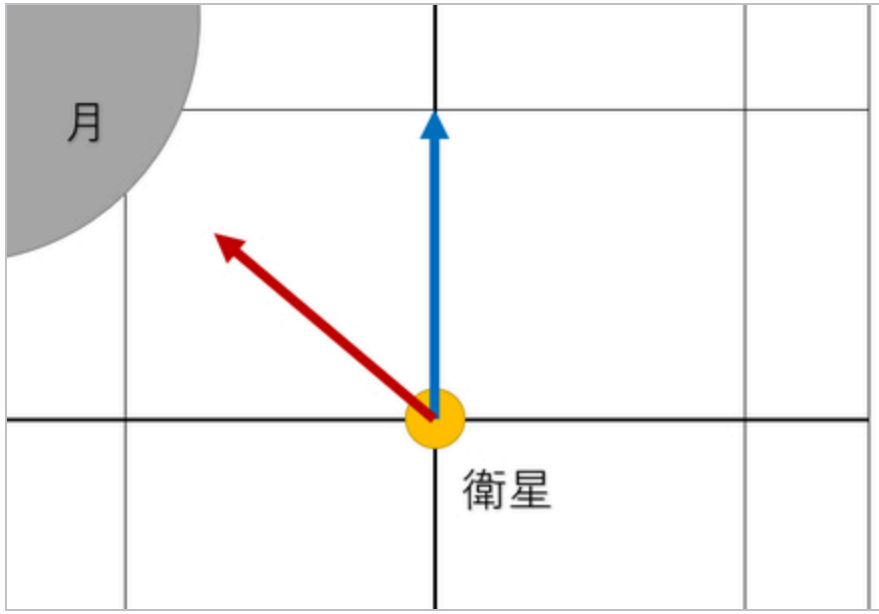
前述の 021 と比べると、人工衛星の軌道がより滑らかに、自然な振る舞いになっていることがわかるはずです。

サンプル 021 の人工衛星は、自分自身の位置と月の位置から進行方向を決めて一直線に目的地に向かうような挙動になっていました。

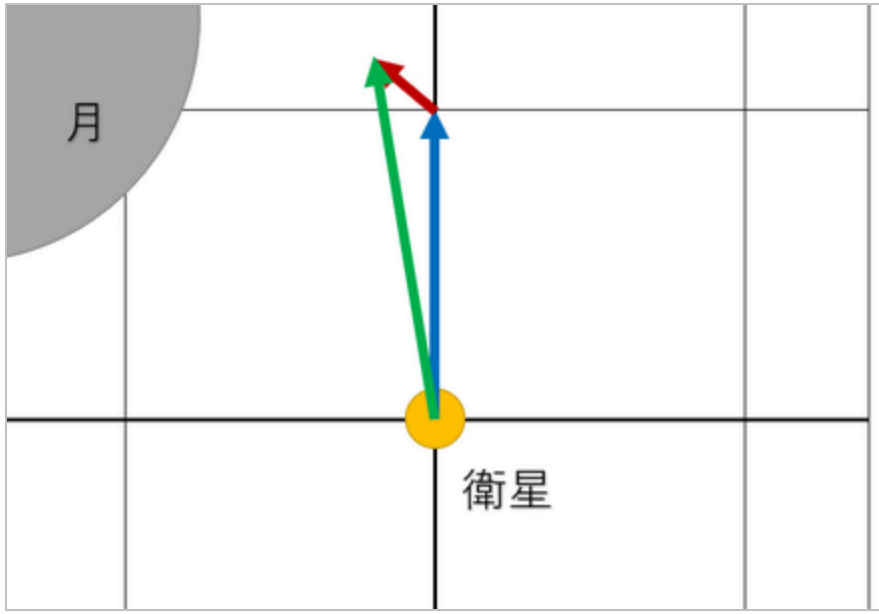
しかしサンプル 022 では、人工衛星は 自分自身の現在の進行方向 をプロパティとして持っています。

人工衛星が持っている自分自身の進行方向に、影響力を小さめにした「月へ向かうベクトル」を加算してから単位化することで……

ほんの少しずつ月に向かって進行方向を補正していくような、そんな挙動を実現しています。



衛星がもともと持っている自分自身の進行方向と、衛星と月との相対的な位置関係を表すベクトル（衛星から見て月がどっちにあるかを表すベクトル）がある状態。



衛星から月へ向かうベクトルの影響力を小さく（短く）してから加算してやると、進行方向がずれた新しいベクトルが得られる。

※ただし長さが変わってしまうので、単位化してから使う

ベクトルを加算すると、先程の図を見ながら考えてみればわかるかと思いますが、ベクトルの長さが変わりますよね。ですから、月の進行方向として加算後のベクトルを使う場合は、忘れずに単位化を行うようにしましょう。

言葉で書くとこれまたややこしいのですが、実際に図を描いてみたりしながら、コードを読み解いてみてください。

022

- 人工衛星は自分自身の進行方向をあらかじめ持っている
- 月との相対的な位置関係から進むべき方向を求めてやり……
- その影響を小さくしてから進行方向ベクトルに足し込む
- 単位化した後、それを使って人工衛星を移動させる

クォータニオン（四元数）

さて、数学的な話が続いていますが…… 実際のところ、実は意外と簡単なんだな～と感じる場面も、もしかしたらあったのではないでしょうか。

ここからは最後に、ちょっとだけ難しいテーマも扱ってみましょう。

サンプルのコメントにも書いたのですが、もしも今から解説するサンプルやその内容に関して、一見して意味がわからないものなどが出てきても今の段階で過度にそれに対する劣等感や絶望感を抱く必要はまったくありません。

あくまでも、参考程度に見てもらえればと思います。

> 数学には私も苦手意識があるのですが、そんな私でもこれくらいまでならなんとか自作できます……

さてまずはベクトルについてもう少しだけ踏み込んでおきたいと思います。

先ほど、ベクトルの加減算の話が少し出てきたのですが、加算、減算、と来たら、掛け算や割り算はどうなってるんだろう？ と感じたひともいらっしゃるかもしれません。

ベクトルには、厳密には「普通の掛け算」というのはありません。

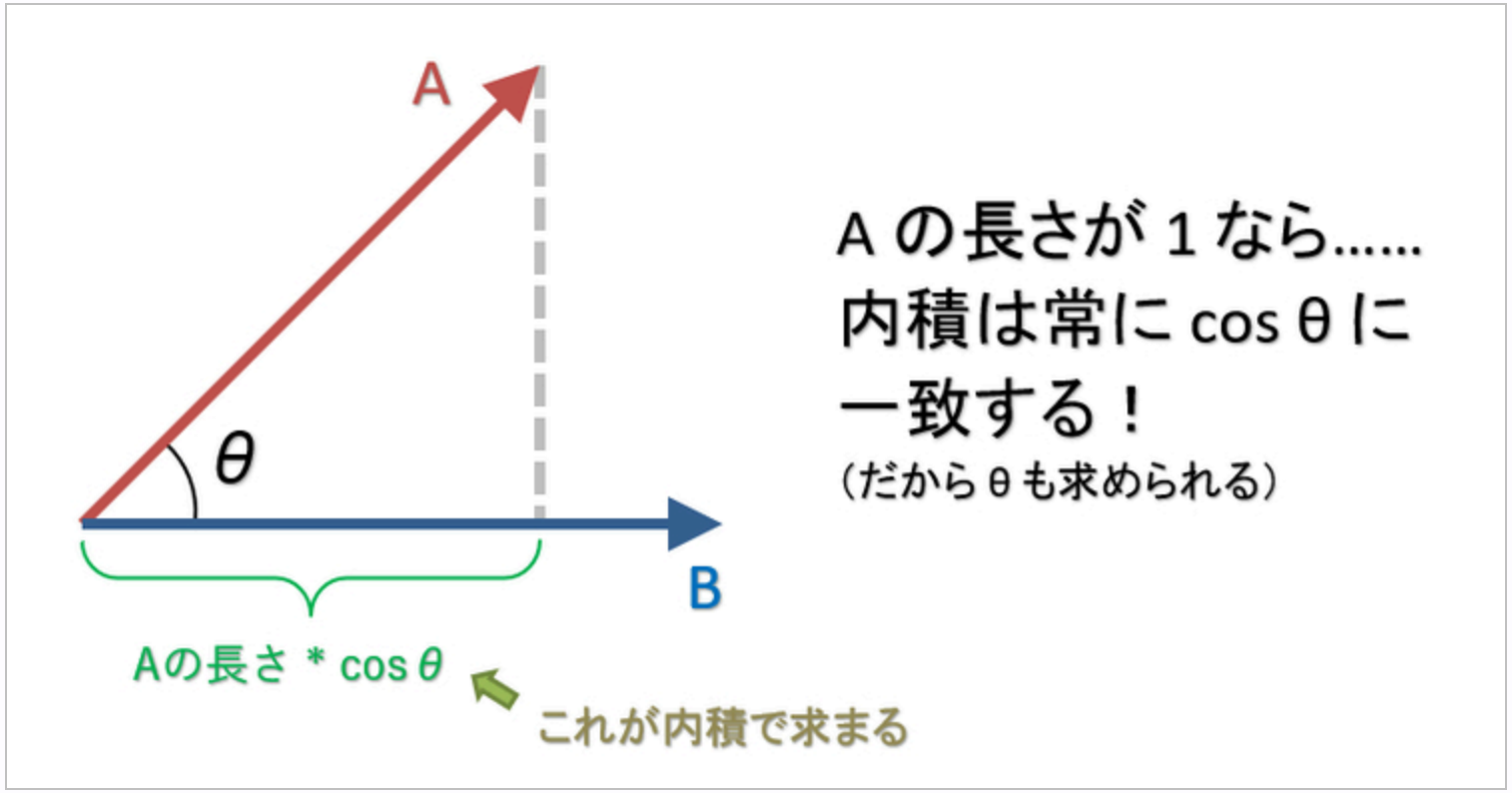
その代わりと言ってはなんですが、掛け算、つまり積の求め方としてふたつのやり方があり、それらは 内積 及び 外積 と呼ばれています。

> ちなみに割り算に相当するものはベクトルにはありません

内積

内積は、二次元ベクトルでも三次元ベクトルでも、計算方法や考え方が同じなので割とわかりやすいです。

用途としては ベクトル同士のなす角 の角度を求めたりするのに使われます。

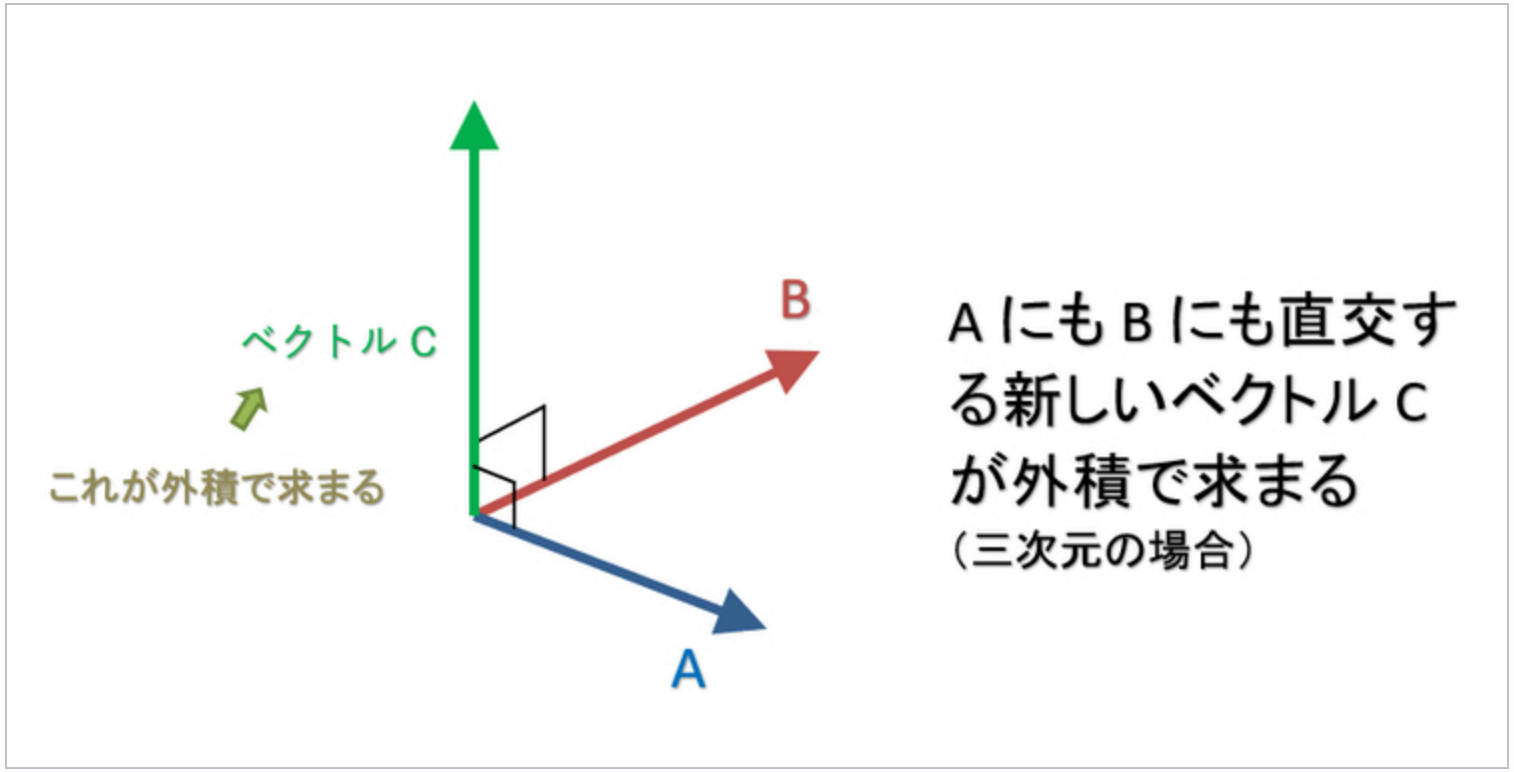


内積の結果はスカラー（ベクトルではない単体の値）になる

外積

外積のほうは、二次元ベクトルと三次元ベクトルで計算結果が若干異なる性質になるのですが..... 二次元ベクトル同士の外積はサインの結果とみなすことができるスカラーの値になります。一方で、三次元ベクトル同士の外積の結果は、三次元ベクトルとして結果が返されます。

そして、この返されたベクトルは もとになったふたつのベクトルに直交するベクトルになる という性質があります。



どちらとも直交（垂直に交わる）ベクトルが出てくる

前提としてのベクトルの話

- 内積や外積はベクトルの積を表すふたつの形
- 単位化したベクトル同士の内積はコサインに等しくなる
- すなわち、内積はふたつのベクトルのなす角を求めることなどに使える
- 単位化したベクトル同士の外積はサインに等しくなる
- 三次元における外積は、ふたつのベクトルに直行するベクトルを求めることなどに使える

以下のリンクは、three.js の `THREE.Vector2` や `Vector3` のソースコードです。

内積や外積は、実は計算方法自体はめっちゃ簡単です。内積の計算結果がスカラーになるというのも、計算方法を見てみれば簡単に理解できますよね。

- three.js/Vector2
- three.js/Vector3

正直なところ、これだけを見せられてもまったくもってチンプンカンプンじゃないかなと思います。

実際にこれをどうやって使うのかとか、どういう場面で便利なのか、ということを一緒に考えないとまったく頭に入ってきません。

そこで、サンプル 023 を見ながら実際に内積や外積が使われているところを見てみましょう。

とは言え、最初はかなり難しいのでキツそうだったら無理に詰め込む必要はありません。

サンプル 023 では、人工衛星が球体からコーンのジオメトリに変わっています。その挙動とイメージを一致させるために 023 では人工衛星ではなくロケットという呼称にしています。

そして、このコーンメッシュはロケット自身が持つ進行方向ベクトルに応じて 常に姿勢が変わる ようになっています。

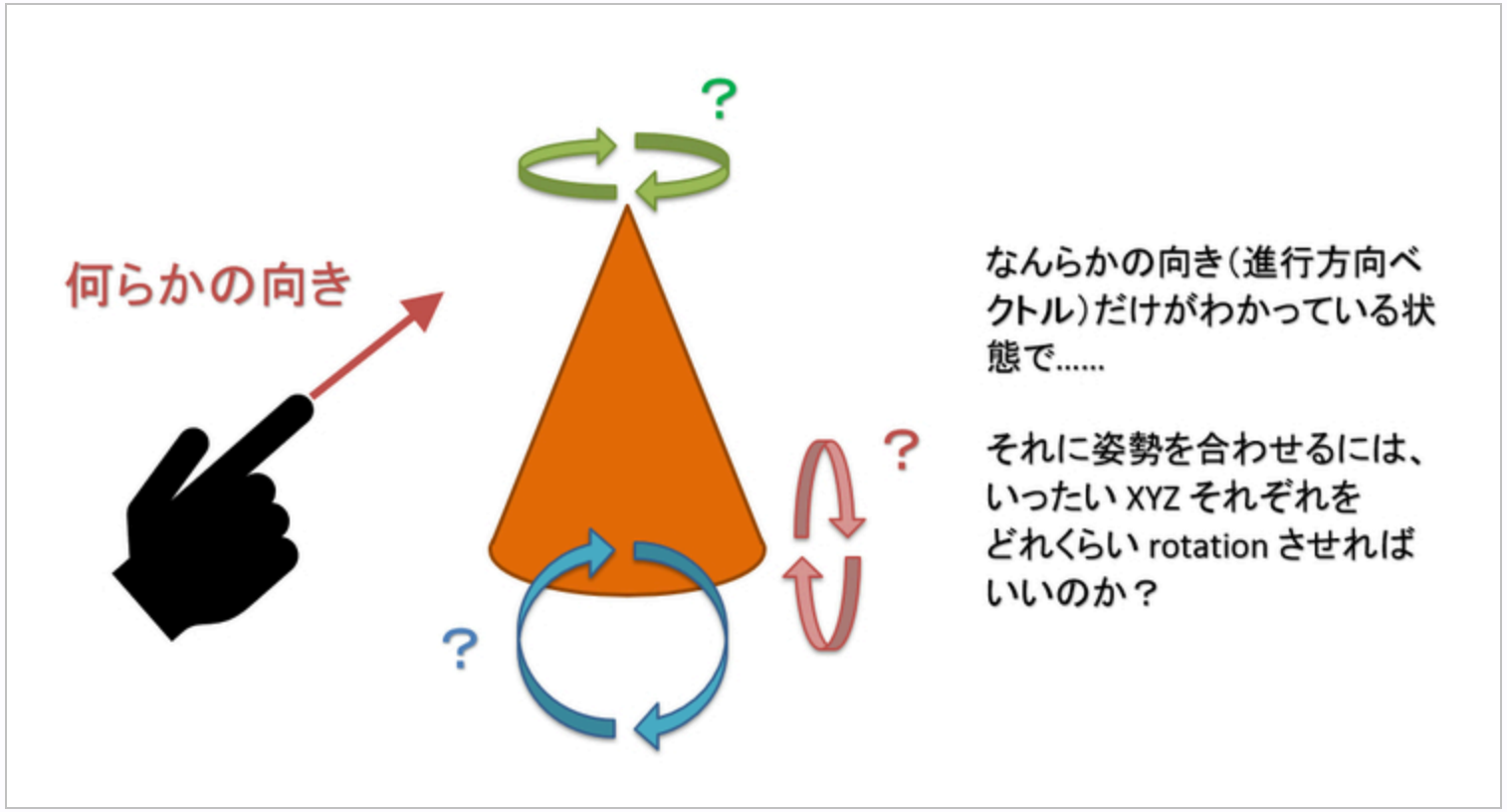
3D プログラミングなどで「姿勢」というキーワードが出てきたとき、これは主に回転を表していると考えるとよいでしょう。

今回の例で言えばコーンメッシュの 姿勢を正しく実現するため には、コーンメッシュに どのような回転を加えたらいいのか ということを考えるわけですね。

> なかなか計算が難しそうですね……

たとえば、こんなふうに考えてみましょう。今、この瞬間に適当で構わないので自由にどこか指を差して、その指差した方向に沿った姿勢にコーンを回転させたいとします。

このとき、今まで three.js のサンプルでよく見かけた `rotation` に、いったいどんな数値を指定したらいいでしょうか？ またその数値ってどうやって計算したらいいんでしょうか？



なんらかの向き（進行方向ベクトル）だけがわかっている状態で……

それに姿勢を合わせるには、いったい XYZ それぞれをどれくらい rotation させればいいのか？

three.js の `rotation` のように、XYZ の各軸に対して回転量を指定する回転の表現方法（これまでのサンプルで使っていた回転表現）のことを、オイラー角、またはオイラー角による回転表現などと言います。

オイラー角は非常に直感的で、人間に優しい回転表現です。Y 軸に対してこれくらい回転すれば……といったように、頭のなかでも回転を非常にイメージしやすいのがオイラー角の特徴です。

現にこれまでのサンプルでは、回転はオイラー角で（つまり `rotation` プロパティを使って）表現していました。

ただし一方で、今回のケースのように「特定の姿勢を得るための回転」などを考える場合、オイラー角では非常に話がややこしくなってしまう場合があります。これは前述のとおり、姿勢だけを見て、そのような姿勢を得るためにどういったオイラー角の指定が必要なのかは、なかなか求めるのが難しい場合が多いからです。

そこで出てくる回転表現の救世主こそが、**クォータニオン** です。日本語では四元数と呼ばれます。

オイラー角が人に優しい回転表現であるのに対して、クォータニオンはその中身を見ても、どんな姿勢を表しているのかはパッと見た感じではほぼわからないです。そういった意味で、直感的ではありません。

クォータニオンはプログラム的に言うと長さが４の配列になるのですが、その中身を見ても、実際どんな回転（あるいは回転後の姿勢）になるのかは、人の目では判別しにくいです。

ただ、プログラム的にはメリットがいろいろとあり、わかりやすいところで言えば計算量がかなり節約できたり、回転が破綻しないなどのメリットがあります。

また、クォータニオンはなんとこれひとつで ありとあらゆる全ての姿勢を表現できる という性質があります。

ちょっとした傾きも、大きな傾きも、半回転も全回転も、とにかくどんな回転もたったひとつのクォータニオンだけで表現できます。一見そんな馬鹿なと思うかもしれませんが、これはほんとなんです。

> ありとあらゆる全ての姿勢はひとつのクォータニオンで表現できてしまう

元の状態

ローカル座標での
原点はこのあたり

とある回転後の姿勢 A

とある回転後の姿勢 B

ありとあらゆる姿勢は、たった１つのクォータニオンで表現できる

クォータニオンの考え方のベースは、回転軸と回転量 というふたつのポイントを押さえることです。

先ほど、クォータニオンは配列で言うと長さが４、と書きましたが、この４つの要素はそれぞれ XYZ で表した軸ベクトル と 回転量 です。それで、長さが４になるんですね。

```
// よくあるクォータニオンのプログラム的定義
q = [x, y, z, 回転量];
```

クォータニオンは配列なら長さ４で表現でき、その内訳は、軸ベクトルと回転量からなる

> とてもシンプルに全ての姿勢を表現できる形をしている

さて、かなり話がややこしくなってきたので、ここで一度整理しましょう。まず、いま実現しようとしていること、つまり最終的な目的はこうです。

進行方向ベクトルがわかっている状態で、そのベクトルに平行な姿勢を取らせる回転を得たい。

そして、そのような回転を表現するにあたって、オイラー角を用いると…… XYZ の各軸に対する回転量を求めて姿勢制御するのが難しい（または面倒） という問題があります。

そこで、こういった場合はクォータニオンを使います。

クォータニオンは……

それ単体であらゆる姿勢を表現できる という特徴を持ち、長さ４の配列で表現できるなどプログラミング上のメリットが多い という特性があります。

クォータニオンを構成する要素（長さ４の配列の内訳）は XYZ で表される軸ベクトル と 回転する量 の合計４つ。

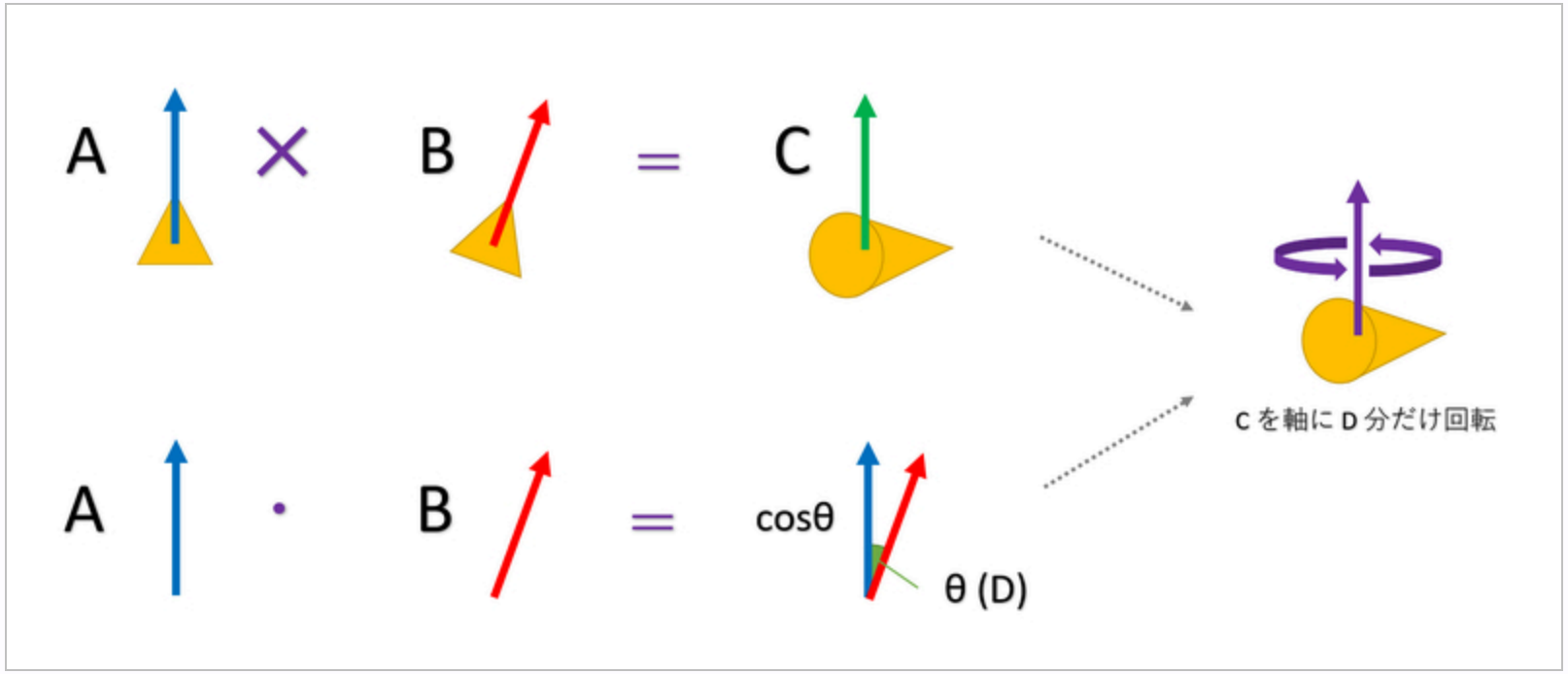
つまり **どの軸を使って、どのくらい回すか** ということをクォータニオンは表すことができるため、ありとあらゆるすべての姿勢を表現できるのです。

クォータニオンを使って姿勢制御

さて、これまでの内容を総合的に考えると……

次のページに書いたような手順を順番に行ってやれば目的を達成できます。

- ひとつ前のフレームの進行方向ベクトル（A）と、現在の進行方向ベクトル（B）で外積を取る
- この外積の結果のベクトルは、A と B に直交するベクトル（C）である
- さらに A と B で内積を取ることでコサインが求まる
- このコサインからラジアンを逆算できるので、A と B のなす角（D）がわかる
- C を回転の軸として、D 分だけ回転したクォータニオンを定義する
- このクォータニオンをコーンメッシュの回転量に利用する



箇条書きされた内容（A ～ D）と見比べながら考えてみよう

クォータニオンは、計算の量が非常に少なくて済む上に、データ構造的には長さ４の配列に過ぎないため、メモリの消費も抑えられます。さらに、オイラー角では避けられないジンバルロック現象も起こりませんし、こと回転を表すことにかけては最強です。

ただ、ご覧のとおりかなり扱うのが難しいというか……必要な前提知識が高いレベルになるので、こればっかりは地道に勉強したり慣れたりしていくしかありません。

クォータニオンが今の時点でまったく理解できなくても、最初に書いたとおり落ち込む必要はまったくありません。

サンプルはみなさんのお手元にずっとあるわけですから、少しずつ、焦らず取り組んでいきましょう。

023

- ベクトルの内積や外積を駆使する
- クォータニオンの回転軸は外積を使って……
- 回転の量については内積を使って求める
- `Object3D.quaternion` にクォータニオンを乗算して姿勢を制御する

さいごに

さて、深遠なる 3D 数学の一端を垣間見た、そんな第三回講義となりましたがいかがでしたでしょうか。

正直なところ、クォータニオンとかはかなり難しい部類の知識になるので、今すぐに「完全に理解する」必要はなく過度に心配する必要はありません。

むしろ、重要なのはラジアンやそれを活用したサイン・コサインの求め方、さらにはそれらとベクトルとの関係です。

特に、XY 平面においてサイン・コサインが 半径 1 の円における円周上 XY 座標と一致する ということはかなり重要で、本当に様々な場面で活用できますし、これを理解していないとそもそも読み解くのが難しい概念も多くあります。

またベクトルについても、その長さを求める方法、さらには長さを使ってベクトルを単位化する方法などは、言葉を聞いただけでどのようなことを言っているのかスムーズに想起できるようにしておきましょう。

慣れるまでは、どうしてもベクトルってなんとなくわかりにくいのですが、慣れてくると「このベクトルは単位化したから方向を表していて……」みたいに、頭の中で処理が思い浮かぶようになってきます。少しずつでいいですから、地道に取り組んでみてください。

最後に今回の課題ですが、せっかくサイン・コサインを覚えたので、それを活用したテーマとして……

地球上を飛ぶ旅客機（を模した Box や Plane 等で可）の動きを実現してみましょう。課題の実装ポイントは「旅客機のような見た目ではなく」、あくまでも「旅客機のような動き」です。

余裕があれば地球上に建築されたビルや、山脈などの自然物なんかも追加してみてもいいかもしれません。

繰り返しますが、あくまでも 動きを再現する というのが課題の意図です。グループ機能や、サイン、コサインなど、何を使うのも自由です。これまでに登場した概念を駆使して実現してみましょう。