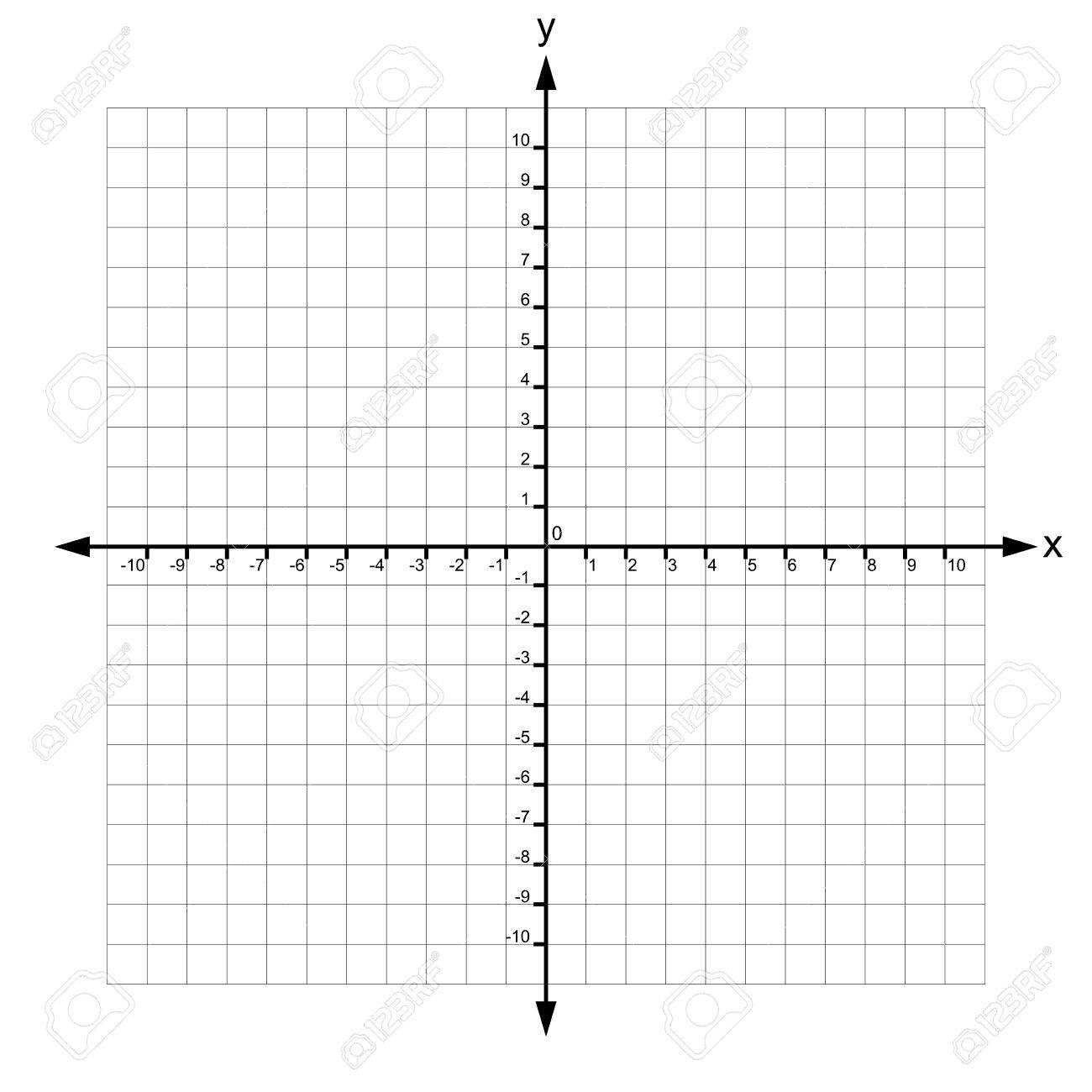
# **Chapter 1 デカルト座標系**

## **1.1 2Dデカルト座標**

　デカルトとは直角を表す言葉です。皆さんが中学校の数学などで見たことがあるであろう、下のような図1-1が2Dデカルト座標系です。

図1-1



## **1.1.2 原点**

　すべての2Dデカルト座標系には、原点と呼ばれる座標系の中心を表す特別な場所を持っています。

## **1.1.3　軸**

　すべての2Dデカルト座標系には、原点を通る２つの直線を持っています。それぞれの線は軸として知られています。その軸は2Dデカルト座標系であれば、x軸、y軸と呼ばれることが多いです。

## **1.1.4 デカルト座標系を用いて、場所を指定する**

　デカルト座標系を使えば、2Dゲームのオブジェクトの表示されている場所を、数字で指定することができます。下記の図1-2を見てみてください。

　図1-2



ヨッシーー

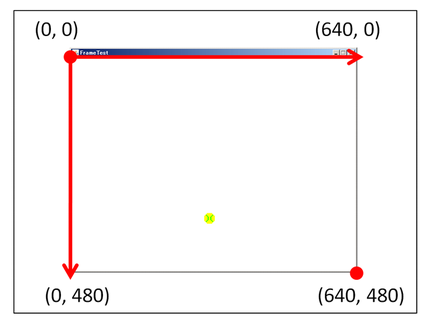
マリオ

この図であればマリオの場所は、X軸上に2.5、Y軸上に0.0で表すことができます。ヨッシーの場所は、X軸上に1.0、Y軸上に2.0あたりでしょうか。

## **1.1.5 軸の向き**

　我々はxの＋は右方向、yの+は上方向と習慣的に覚えているかもしれません。しかし、xの+を左方向、yの+を下方向とすることもできます。例えば、ウィンドウプログラムでは図1-3のように、yの+が下方向になっていることがあります。

図1-3



ここで重要なのは、ウィンドウプログラムだとYの方向が違うということを覚えることではありません。次のことをしっかりと頭に入れておいてください。

**「右方向がXの＋、上方向がYの+であるとは限らない！」**

ただし、この後の2Dデカルト座標系での話は、みなさんが分かりやすいように、右が+x、上が+yとして話を進めていきます。

## **1.1.6 プログラムでのオブジェクトの座標の指定の仕方**

　キャラクターを画面に表示するためには、**「どこに表示するのか？」**ということをコンピュータに教えてやる必要があります。多くのゲームでは、この場所に指定にベクトル構造体(もしくはベクトルクラス)を使ってデカルト座標系での位置をコンピュータに教えます。

サンプルコード 1-1

|  |
| --- |
| struct Vector{  float x;  　 float y;  };  int main()  {  Vector marioPos;  marioPos.x = 2.5f;  marioPos.y = 2.0f;  　 //ゲームループ  while(true){  //  }  } |

## **1.2 実習課題**

　GameMath\_1/Chapter\_1/Game.cppの27行目のコメントを読んで、図1-4のような絵が完成するように、各オブジェクトの座標を指定しなさい。

図1-4



# **Chapter 2 ベクトル**

## **2.1 ベクトル　数学的な定義**

　ベクトルは数学的には、ただの数字の配列です。この説明を聞いてもイメージはわかないと思いますが、気にしなくて構いません。

## **2.1.1　ベクトルとスカラー**

　数学者はベクトルとスカラーを区別します。スカラーは普通の数字を表す用語です。皆さんが慣れ親しんでいる数字のことです。今後スカラーというキーワードができてたら、ベクトルじゃなく、普通の数字のことなんだなと思ってください。

## **2.1.2　ベクトルの次元**

ベクトルの次元は、そのベクトルにいくつの数が含まれているのかを指します。主にゲームでは2Ｄ、3Ｄ、そして4Ｄベクトル(後半で)を扱います。

## **2.1.3 ベクトルの数学的な記法**

数学では、ベクトルを下記のように記述します。

もしくは

水平に記述するのは**行ベクトル**、垂直に記述するのは**列ベクトル**です。この違いが意味を持つ場合があるのですが、今は同じものだと思っていて構いません。また、ベクトルの各要素はx，y，z，wで表します。x，yで2D、x，y，zで3D、x，y，z，wで4Dです。4Ｄベクトルはアルファベット順でないことに注意してください。4番目の値はwです。

## **2.2　ベクトル　幾何学的な定義**

　ベクトルは幾何学的には大きさと向きを表す線分です。

・ベクトルの大きさは、ベクトルの長さです。そしてベクトルの長さはスカラーです。

・ベクトルの向きはベクトルが空間内でどこを指しているのかを表します。

上のように表記されているベクトルvはv.x = 2、v.y = 4となります。

## **2.2.1 ベクトルはどのように見えるか？**

　2.2のベクトルvは下記のように図2-1のように図示化することができます。

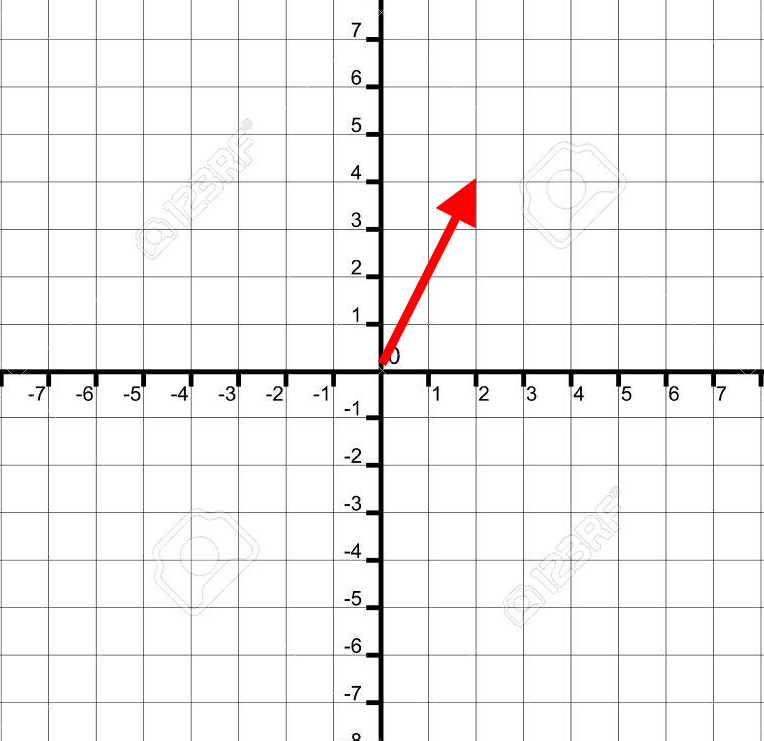


図2-1

## **2.2.2 座標とベクトル**

さて、Chapter1で勉強した、デカルト座標系での位置を表す座標もベクトルを使って表していましたが、実際には座標とベクトルは全くの別物です。

座標は空間の一点を示し、その場所が変わることはありません。しかし、ベクトルは位置情報を持っておらず、大きさと向きのみを保持しています。

点Ｐ＝( ２　４ )　と　ベクトルV = [ 2　４ ]を図示化した図2-2を見てください。



図2-2

このように、赤い矢印はすべて x方向に+2、y方向に+4という大きさを持っているベクトルVとなります。一方点Ｐは赤い丸の一点しか表しません。

## **2.2.3 問題**

問１

　下記の座標P0～P3とベクトルV0～V3を図示化しなさい。

P0 = ( 0　４ )、P1 = ( 1　2 )、P2 = ( －2　6 )、P3 = ( －５　－５ )

V0 = [ 1　2 ]、V1 = [ 2　2 ]、V2 = [ －4　3 ]、V3 = [ －2　－１ ]

Y



X

問２

下記の図に記載されているベクトルを求めなさい。



解答欄

a = [ ] b = [ ] c = [ ] d = [ ]

e = [ ] f = [ ] g = [ ]

# **Chapter 3 ベクトルの演算**

　このチャプターではいくつかのベクトルの演算を学び、その演算の幾何学的な意味を考えていこうと思います。

## **3.1　線形代数 vs 我々が求めるもの**

　主にベクトルを扱う数学の分野を線形代数と呼びます。2.1節で述べたように、線形代数において、ベクトルは数字の配列にすぎません。しかし、我々、ゲームプログラマはベクトルの幾何学的な解釈を求めています。線形代数を扱っている教科書では、幾何学的解釈までは十分に扱っていません。この授業では、ベクトルの幾何学的な解釈に焦点を当てて考えていきます。

## **3.1 ベクトルの反転**

　ベクトルの反転は、ベクトルのすべての要素に―１を乗算することで求まります。例えば

あるベクトルV[ 2　５ ]を反転させると、[ －２　－５　]となります。

## **3.1.1 線形代数の公式**

任意の次元のベクトルを反転するには、単純にベクトルのそれぞれの要素の正負を反転するだけです。例えば、３Ｄのベクトル[　ｘ　ｙ　]を反転させた場合、[　－ｘ　－ｙ　]となります。

***式　－[　ｘ　ｙ　]　＝　[　－ｘ　－ｙ　]***

## **3.1.2 幾何学的解釈**

では、ベクトルの反転の幾何学的な意味を見ていきましょう。ベクトルは反転させると、元のベクトルと真逆のベクトルとなります。下記の図3-1を見てみてください。ベクトルVA[ ２　５　]を反転させたベクトルVBは[　－２　－５　]はVAと真逆を向いていることが分かります。



**VB**

**VA**

図3-1

　ベクトルは位置情報を持たずに、方向と大きさのみを表している数字だったことを思い出してください。図3-1のベクトルVAとVBは完全に真逆のベクトルとなっています。

ベクトルを反転させる計算は、ベクトルに対して―１乗算することで求めることができます。ベクトルの乗算に関しては、後程詳しく見ていきます。

## **3.1.3 問題**

問１

下記のベクトルVA～VDを反転したベクトルVA´～VD´を図示化しなさい。

VA[ ３　２ ]　VB[ 1 ７ ] VC[ －２　５　]　VD[　－３　－２　]



## **3.2 ベクトルの大きさ(長さ)**

これまで、見てきたように、ベクトルは大きさと向きを持っています。しかし、ベクトルの中に大きさも向きもはっきりと表されていないことに気が付いたかもしれません。例えば、ベクトル[　３　４　]の大きさは３でも４でもなく、５です。ベクトルの大きさははっきりと表されていないため、計算しなくてはなりません。

## **3.2.1　線形代数の公式**

　線形代数では、ベクトルの大きさはベクトルを挟む２重の垂直な線を用いて記述します。３次元のベクトルＶの大きさを求める式は次のようになります。

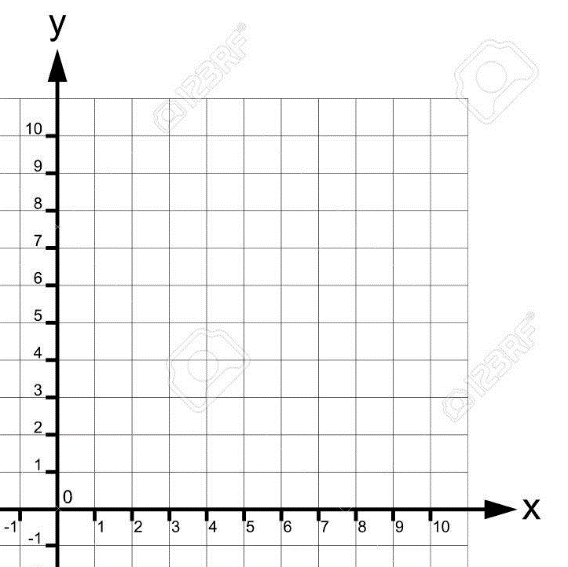
**||Ｖ||＝**

このように、ベクトルの大きさはベクトルの要素の２乗和の平方根となります。

また、ベクトルの大きさは必ず正になります。例えば、ベクトル[　－２　５　－８　]の大きさは下記のように計算されます。

## **3.2.2 幾何学的解釈**

　3.2.1の公式についての理解を幾何学的解釈から深めていきましょう。ベクトルはどんなベクトルであっても、ベクトルvを斜辺として下記の図3-2のように、直角三角形を作ることができます。これは2Dも3Dも同じです。



**|V.ｘ|**

**|V.y|**

**||Ｖ||**

図3-2

ベクトルの大きさ(長さ)の幾何学的な意味は、ベクトルの矢印の長さです。また、図3-2のように、ベクトルｖを斜辺とした直角三角形を作ることができます。また、ピタゴラスの定理より、直角三角形の斜辺の長さは、**残る２辺の長さの２乗和の平方根**となることが分かっているため、次の式が得られます。

・・・・・・・・２Ｄベクトルの場合

　・・・・３Ｄベクトルの場合

これは、まさに3.2.1で見た線形代数の公式と同じになります。

tips

|  |
| --- |
| ベクトルの長さは、２点間の距離の計算を行うときなどに使われる演算で、ゲームで非常によく使われる演算です。２点間の距離の計算を理解するには、ベクトルの減算も学ぶ必要があります。 |

## **3.2.3 問題**

問１

下記のベクトルa~gの長さを小数点第２位までで求めなさい。また、電卓を使っていいものとする。

解答例　a = 10.25



a =

b =

c =

d =

e =

f =

g =

電卓の使い方の動画

<https://www.youtube.com/watch?v=vhPbA0E-8FA&feature=youtu.be>

## **3.3　ベクトルとスカラーの割り算と掛け算**

　ベクトルとスカラーを足すことはできませんが、ベクトルとスカラーの割り算と掛け算はできます。結果は、元のベクトルと平行なベクトルになります。ただし、長さが変わったり、向きが反対だったりします。

## **3.3.1 線形代数の公式**

　ベクトルとスカラーの掛け算と割り算は簡単です。ベクトルの要素それぞれにスカラーを掛けるor割るだけです。スカラーＫと３ＤベクトルＶの掛け算は下記のようになります。

**[　V.x V.y V.z ]×Ｋ**

**＝[　KV.x　KV.y KV.z ]**

例をいくつかあげます。

[ 1　2　3　]×２　＝　[ 2　4　6　]

[ －5　0　0.4　]×－３ = [ 15 0 －1.2 ]

続いて、割り算を見ていきましょう。スカラーＫと3DベクトルVの割り算は下記のようになります。

**[　V.x V.y V.z ]÷Ｋ**

**＝[　V.x/K　V.y/K kV.z/Ｋ ]**

例をいくつかあげます。

[ 4.7 －6 8 ] ÷ 2　=　[ 2.35 －３ 8 ]

[ 10 －30 45 ] ÷ 5　=　[ 2 －6 9 ]

## **3.3.2　幾何学的解釈**

　幾何学的には、ベクトルにスカラーＫを掛け算or割り算するのは、ベクトルの長さをK倍スケーリングする効果を持ちます。(割り算の場合は1/K倍)

例えば、ベクトルの長さを２倍にするには、ベクトルに２をかけます。K＜０なら、ベクトルの向きは反転します。図3-3はベクトルV[ 2 4 ]にいくつかの異なるスカラーを掛けたベクトルを図示化しています。



**－2V**

**V/2**

**2V**

**V**

図3-3

## **3.3.3 問題**

問１

下記のベクトルVA～VDにスカラー３を掛け、ベクトルVA´～VD´を図示化しなさい。

VA[ ３　２ ]　VB[ 1 2 ] VC[ －２　4　]　VD[　－３　－２　]



問2

下記のベクトルVA～VDをスカラ2で割った、ベクトルVA´～VD´を図示化しなさい。

VA[ 4　２ ]　VB[ 6 4 ] VC[ －２　4　]　VD[　－4　－4　]



## **3.4　ベクトルの正規化**

　ベクトルを扱う場合、我々は多くの場合で向きと大きさに関心があります。ベクトルの大きさは、3.2節で勉強した三平方の定理を使えば求まります。

　そして、ベクトルの向きを扱うときは、**単位ベクトル**を扱うのが便利です。単位ベクトルとは大きさが１のベクトルです。単位ベクトルを求めることを、ベクトルを正規化するといいます。

　単位ベクトルを扱うのが便利なのかは、このチャプターの最後に行う実習を通して説明を行います。

## **3.4.1 線形代数の公式**

　どのようなベクトルについても(ゼロベクトルは除く)、ベクトルvと同じ向きを指す単位ベクトルである、vNormを計算することができます。単位ベクトルを求める処理はベクトルの正規化として知られています。ベクトルを正規化するには、ベクトルをその大きさで割ります。

例えば、2Dのベクトル[12 —５]を正規化するには、次のようにします。

## **3.4.2 幾何学的解釈**

では、ベクトルの正規化の幾何学的解釈を見ていきましょう。図3-4を見てください。



**7**

**4**

**V[ 4 7 ]**

図3-4

このベクトルVの大きさは約8.06になります。正規化は、ベクトルの各要素をベクトルの大きさで除算することで求まります。図3-5を見て下さい。

**V[ 4 7 ]**

図3-5



これが正規化されたベクトルvNorm[ 0.49 0.868 ]

**4÷8.06**

**≒0.49**

**7÷8.06**

**≒0.868**

　ベクトルVを斜辺とする直角三角形の2辺の長さを、斜辺の長さで除算しているので、ベクトルVの大きさは１となります。

## **3.4.3 問題**

　下記のURLの問題を解きなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSdVPZmTAxmqlLdhmAHP_P7IjZPk9ZnCAoQZQ5vEG8-M6YFYng/viewform?usp=sf_link>

## **3.5　ベクトルの足し算**

　二つのベクトルが同じ次元なら、それらを足し算することができます。足し算したベクトルは、元のベクトルと同じ次元のベクトルとなります。

## **3.5.1 線形代数の公式**

　ベクトルの足し算の線形代数の規則は単純です。２つのベクトルを足すには、対応する要素を足し算するだけです。

例えば、ベクトルv1[ 2 5 1]とv2[4 7 3]の足し算は下記のようになります。

　ベクトルの足し算は交換法則が成り立ちます。つまり、ベクトルv1とv2の足し算は**v1+v2 = v2 + v1**が成り立ちます。

## **3.5.2 幾何学的解釈**

　では、ベクトルの足し算の幾何学的解釈を見ていきましょう。ベクトルの足し算は線形代数の公式の通り、ベクトルの各要素を足し算するだけですが、足し算の考え方で解釈が変わってきます。計算の結果は変わらないのに、解釈が変わってくるのです。そしてこれはとても重要です。

## **3.5.2.1　座標とベクトルの足し算**

　座標とベクトルの足し算は、幾何学的には座標をベクトル方向に動かすことを意味します。では、次の計算を考えてみましょう。

　座標P( 2 3 )とベクトルV[ 3 5 ] を足すと、座標はベクトルと考えてＯＫです、結果はPをベクトルV方向に移動させた座標P´( 5 8 )となります。

　では、これを幾何学的に考えてみましょう。図3-6を見てください。



Y方向に+5

X方向に+3

**P´( 5 8 )**

**V[ 3 5 ]**

**P( 2 3 )**

図3-6

　このように、座標とベクトルの足し算は、座標をベクトル方向に移動させます。

ゲームにおいて、キャラクターを移動させることは全て座標とベクトルの足し算であるとも言えます。

## **3.5.2.2 Hands-On ベクトルを使ってキャラクターを移動させる。**

　サンプルプログラムのChapter\_3\_1を使って、キャラクターをベクトルで移動させるプログラムを書いてみましょう。Chapter\_3\_1を立ち上げて、実行してください。図3-7のような画面が表示されると思います。



図3-7

HandsOn-1 ゲームパッドの左スティックの入力量からキャラクターの移動ベクトルを作成する。

では、ゲームコントローラ―の左スティックの入力から移動ベクトルを作成するプログラムを書いてみましょう。Player.cppを開いて、次のコードを記入してみてください。

|  |
| --- |
| /HandsOn-1 ゲームパッドの左スティックの入力量からキャラクタの移動ベクトルを作成する。  //GetLStickXF関数は左スティックのX方向の入力具合によって-1.0～1.0の値を返してくる。  //GetLStickYF関数は左スティックのY方向の入力具合によって-1.0～1.0の値を返してくる。  vMove.x = pad.GetLStickXF();  vMove.y = 0.0f;  vMove.z = pad.GetLStickYF(); |

これでパッドの入力をもとに移動ベクトルが作成されました。図3-8のように移動ベクトルが可視化されていたら完成です。



HandsOn-2 移動ベクトルと座標を足し算する

　ハンズオン1で移動ベクトルを作成しました。では、その移動ベクトルとキャラの座標とで足し算を行って、キャラを動かしてみましょう。

Player.cppに下記のコードを記述してください。

|  |
| --- |
| //HandsOn-2 移動ベクトルと座標を足し算する。  m\_position.x = m\_position.x + vMove.x;  m\_position.y = m\_position.y + vMove.y;  m\_position.z = m\_position.z + vMove.z; |

HandsOn-3 移動ベクトルを10倍にする

　このままでは、移動速度が遅すぎるので、移動ベクトルの大きさを10倍にしてみましょう。次のコードを記述してください。

|  |
| --- |
| //HandsOn-3 移動速度が遅すぎるので、ベクトルを10倍にする。  vMove.x \*= 10.0f;  vMove.y \*= 10.0f;  vMove.z \*= 10.0f; |

## **3.5.2.3　ベクトルとベクトルの足し算**

　ベクトルとベクトルの足し算は、幾何学的にはベクトルの合成を意味します。では、次の計算を考えてみましょう。

　ベクトルV1[ 2 3 ]とベクトルV2[ 6 2 ]を足すと、結果はV3[ 8 5 ]となります。当然、計算の仕方は【3.5.2.1座標とベクトルの足し算】と同じですが、この計算をベクトル同士の足し算として解釈する場合は、これはベクトルV1にベクトルV2を加えて新しいベクトルV3を作成することを意味します。では、この計算を図示化した図3-9を見てください。



**V1[ 2 3 ]**

**V3[ 8 5 ]**

**V2[ 6 2 ]**

図3-9

これがベクトルの合成です。あなたが真っすぐ進んでいるときに、横から強烈な風が吹いてきて、その力が加わって斜めに進んでしまうと考えるとわかりやすいかもしれません。

## **3.5.2.4 Hands-On 風の影響を受けてみよう**

　サンプルプログラムのChapter\_3\_2を使って、キャラクターをベクトルで移動させるプログラムを書いてみましょう。Chapter\_3\_2を立ち上げて、実行してください。図3-10のような画面が表示されると思います。



図3-10

このサンプルでは、砂嵐ギミックが実装されていて、画面の中央あたりで砂嵐が発生しています。しかし、砂嵐に巻き込まれても、キャラクターはその影響を受けていません。そこで、プログラムを改造して、キャラクターの移動ベクトルに、砂嵐の吹き飛ばしベクトルを加算して、キャラクターに砂嵐の影響を与えてみましょう。

　Player.cppを開いて、下記のコードの網掛けの部分を追加してみましょう。

Player.cpp(50行目)

|  |
| --- |
| //Hands-On 砂嵐と衝突しているか調べて、衝突している場合は吹き飛ばしベクトルを  // キャラの移動速度に合成する。  //砂嵐ギミックのインスタンスを取得。  Wind\* wind = FindGO<Wind>("砂嵐");  //風と衝突しているか調べる。  if (wind->IsHit(m\_charaCon) == true) {  **//砂嵐と衝突しているので、吹き飛ばしベクトルを速度に加える。**  **CVector3 windPower = { 8.0f, 0.0f, -5.0f };**  **vMove += windPower; //移動ベクトルに吹き飛ばしベクトルを合成する。**  } |

## **3.5.3 問題**

問１　下記の計算で求まる結果を、例題を参考に図示化しなさい。

*例題*

*座標Z0 = 座標( 1 2 ) + ベクトル[ 3 1 ]*

座標P0 = 座標( 2 4 ) + ベクトル[ 2 3 ]

座標P1 = 座標( －３ １ ) + ベクトル[ 1 5 ]

座標P2 = 座標( 2 －4 ) + ベクトル[ －４ 5 ]

座標P3 = 座標( 7 2 ) + ベクトル[ －１０ －5 ]



**Z0**

問２ 下記の計算で求まる結果を、例題を参考に図示化しなさい。

*例題*

*ベクトルZ0 = ベクトル[ 1 2 ] + ベクトル[ 3 1 ]*

ベクトルV0 = ベクトル[ 1 3 ] + ベクトル[ 3 2 ]

ベクトルV1 = ベクトル[ －1 3 ] + ベクトル[ 3 1 ]

ベクトルV2 = ベクトル[ 3 －2 ] + ベクトル[ －４ 4 ]

ベクトルV3 = ベクトル[ 5 2 ] + ベクトル[ －１０ －5 ]



**Z0**

## **3.6 ベクトルの引き算**

　足し算と同様に、二つのベクトルが同じ次元なら、それらを引き算することができます。引き算したベクトルは、元のベクトルと同じ次元のベクトルとなります。

## **3.6.1 線形代数の公式**

　ベクトルの引き算の線形代数の規則は足し算と同様です。２つのベクトルを減算するには、対応する要素を引き算するだけです。

例えば、ベクトルv1[ 2 5 1]とv2[4 1 2]の引き算は下記のようになります。

　ベクトルの引き算は交換法則が成り立ちません。先ほどのv1－v2の左辺と右辺を入れ替えて計算してみましょう。

答えが違う！

つまり、ベクトルv1とv2の引き算は**v1 – v2 ≠ v2 – v1**となります。これはベクトルの引き算を幾何学的に見ていくときに非常に重要なこととなりますので、覚えておいてください。

## **3.6.2 幾何学的解釈**

　では、ベクトルの引き算の幾何学的解釈を見ていきましょう。ベクトルの引き算も足し算と同様に考え方で解釈が変わってきます。

## **3.6.2.1 座標と座標の引き算**

　座標と座標の引き算が、おそらくベクトルの引き算で最も使われる計算でしょう。ゲームであれば、例えば**プレイヤーの座標 ― モンスターの座標**といった形で使われます。では、次の計算を考えてみましょう。

　プレイヤーの座標P( 4 2 ) とモンスターの座標M( 8 5 )があるときに、P－Mを計算すると、答えは[ －4 －3 ]というベクトルVが求まります。では、この計算を図示化してみます。図3-11を見てください。



**V[ －4 －3 ]**

Y方向に－3

X方向に－4

**M( 8 5 )**

**P( 4 2 )**

図3-11

このように、**P－Mの結果となるベクトルVはモンスターの座標から、プレイヤーの座標に向かって伸びるベクトル**となります。

では、次は、引き算の順番を変えてM－Ｐを計算してみましょう。M－Pは[ ４ ３ ]というベクトルVが求まります。では、このベクトルを図示化してみましょう。図3-12を見てください。



**V[ 4 3 ]**

Y方向に＋3

X方向に＋4

**M( 8 5 )**

**P( 4 2 )**

図3-12

このように、**M－Pの結果となるベクトルVはプレイヤーの座標から、モンスターの座標に向かって伸びるベクトル**となって、先ほど求めたベクトルとは逆向きになることが分かります。

## **3.6.2.2 座標と座標の引き算のゲームでの活用法**

　では、座標と座標の引き算のゲームでの活用法をいくつか見ていきましょう。

①　2点間の距離の計算

　まず、真っ先に思いつくのは、2点間の距離の計算です。先ほどのプレイヤーとモンスターの座標の引き算で、求まったベクトルの長さを求めてやれば、プレイヤーとモンスターの距離が分かります。

　例えば、ゲームであれば、「**モンスターはプレイヤーが一定の距離以内に入ってきたら、攻撃する」**といったプログラムを書きたいときなどに活用できます。

*プレイヤーとモンスターの距離を調べる疑似コード*

|  |
| --- |
| //【プレイヤーの座標】－【モンスターの座標】を計算する。  Vector v;  v.x = playerPosition.x - monsterPosition.x;  v.y = playerPosition.y - monsterPosition.y;  v.z = playerPosition.y - monsterPosition.y;  //ベクトルVの大きさを求めて、プレイヤーとモンスター間の  //距離を計算する。距離の計算は三平方の定理を使う。  //① ベクトルvの各要素の２乗の総和を求めて、変数tに代入する。  float t = v.x \* v.x + v.y \* v.y + v.z \* v.z;  //② 平方根を求める、C言語の標準関数のsqrtを使って、距離を求める。  float len = sqrt( t );  //③ 例えば、プレイヤーとモンスターの距離が一定値以下なら～をする  // みたいな処理を書くならこんな感じのif文を書く。  if (len < 30.0f) {  ・  　　・  　　・  　　省略  　　・  　　・  　　・  } |

　距離の計算には、引き算の順番は重要ではないことに注意してください。

**「モンスターからプレイヤーに伸びるベクトル」**であろうが**「プレイヤーからモンスターに伸びるベクトル」**であろうが、ベクトルの大きさは変わりません。

②　プレイヤーを追いかける

　続いて、プレイヤーを追いかける処理を見ていきましょう。ここではベクトルの引き算の順番が重要になってきます。では、アルゴリズムの流れを見ていきましょう。

**1.　プレイヤーの座標 － モンスターの座標を計算して、モンスターからプレイヤーに向かって伸びるベクトルVを求める。**

**2. 1で求めたベクトルVを正規化して、大きさ1にする。**

**3.　正規化されたベクトルVを、いい感じのスカラーで乗算して、ベクトルを大きくする。**

**4.　いい感じの大きさになったベクトルVをモンスターの座標に足し算する。**

では、サンプルコードを見てみましょう。

*プレイヤーを追いかける疑似コード*

|  |
| --- |
| //① プレイヤーの座標 － モンスターの座標を計算して、  //　 モンスターからプレイヤーに向かって伸びるベクトルVを求める。  Vector v;  v.x = playerPosition.x - monsterPosition.x;  v.y = playerPosition.y - monsterPosition.y;  v.z = playerPosition.y - monsterPosition.y;  //②　1で求めたベクトルVを正規化して、大きさ1にする。  // 正規化するためには、まず、ベクトルの大きさを求める。  float t = v.x \* v.x + v.y \* v.y + v.z \* v.z;  float len = sqrt( t );  // 求めた大きさでベクトルVの各要素を除算する。  v.x /= len;  v.y /= len;  v.z /= len;    //③ 正規化されたベクトルVを、いい感じのスカラーで乗算して、ベクトルを大きくする。  float s = 3.0f; //いい感じのスカラー。ゲームでいい感じに調整してください。  v.x \*= s;  v.y \*= s;  v.z \*= s;  //④ いい感じの大きさになったベクトルVをモンスターの座標に足し算する。  float s = 3.0f; //いい感じのスカラー。ゲームでいい感じに調整してください。  monsterPosition.x = monsterPosition.x + v.x;  monsterPosition.y = monsterPosition.y + v.y;  monsterPosition.z = monsterPosition.z + v.z; |

モンスターからプレイヤーに向かって伸びているベクトルを正規化して、ベクトルの大きさを１にしています。このようにすることによって、モンスターの追いかける速度の調整が容易になります。

## **3.6.2.3 Hands-On エネミーにプレイヤーが近づいたら、発見されるようにしてみよう。**

　サンプルプログラムのChapter\_3\_3を立ち上げて実行してください。このサンプルには、プレイヤーと警備兵となるエネミーが存在します。しかし、警備兵はプレイヤーが近づいても何も反応しません。そこで、プログラムを改造して、プレイヤーが警備兵に近づきすぎたらゲームオーバーになる処理を実装してみましょう。

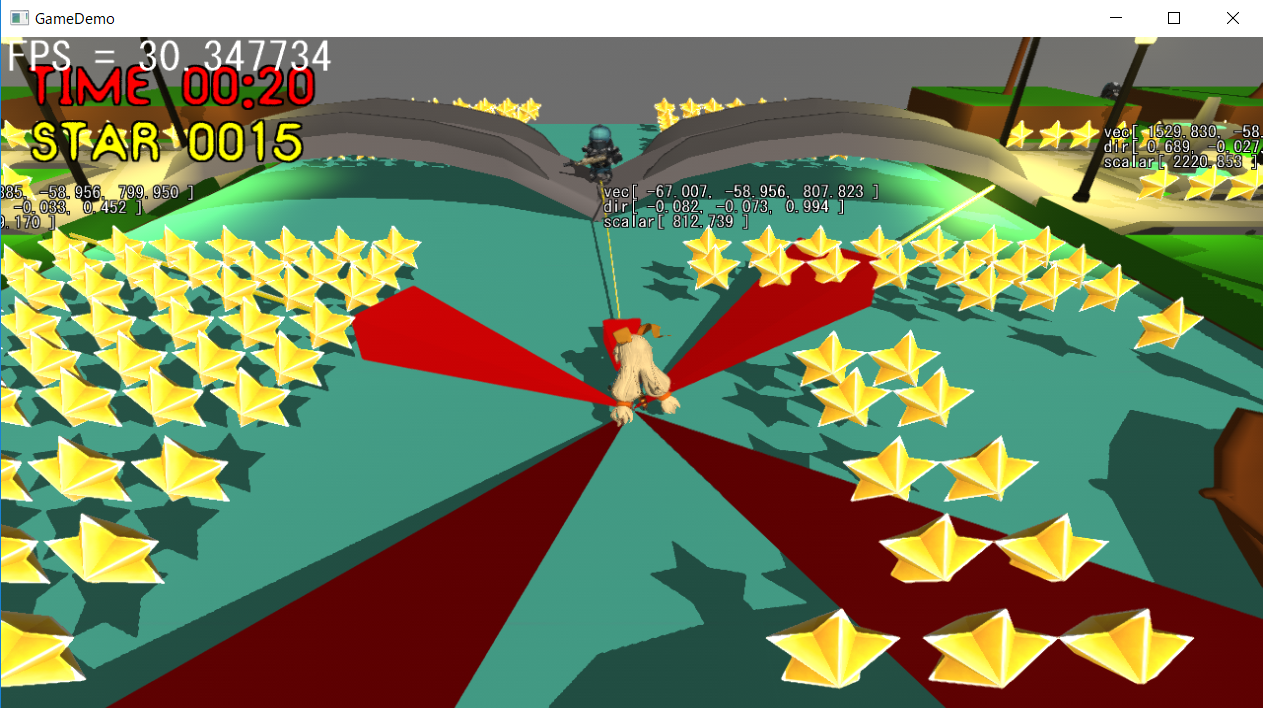
Hands-On-1 エネミーからプレイヤーに向かって伸びるベクトルを計算する。

　Enemy.cppのHands-On-1「エネミーからプレイヤーに向かって伸びるベクトルを計算する。」とコメントされている箇所に下記のコードを入力してください。

Enemy.cpp

|  |
| --- |
| //Hands-On-1 エネミーからプレイヤーに向かって伸びるベクトルを計算する。  toPlayer.x = pl->m\_position.x - m\_position.x;  toPlayer.y = pl->m\_position.y - m\_position.y;  toPlayer.z = pl->m\_position.z - m\_position.z; |

入力ができたら実行してみてください。図3-13のように、エネミーからプレイヤーに向かって伸びているベクトルが表示されたら完成です。

図3-13

　また、プログラムで使用されているCVector3クラスは引き算演算子をオーバーロードしているため、下記のように記述することもできます。

|  |
| --- |
| //Hands-On-1 エネミーからプレイヤーに向かって伸びるベクトルを計算する。  toPlayer = pl->m\_position - m\_position; |

Hands-On-2 toPlayerの距離を計算する。

　次は、1で求めたベクトルtoPlayerの大きさを3平方の定理を活用して計算しましょう。下記のコードを入力してください。

Enemy.cpp

|  |
| --- |
| //Hands-On-2 toPlayerの距離を計算する。  float t = toPlayer.x \* toPlayer.x + toPlayer.y \* toPlayer.y + toPlayer.z \* toPlayer.z;  float len = sqrt(t); |

　また、CVector3にはベクトルの大きさを求めるLengthメンバ関数が用意されているので、このコードは下記のように記述することもできます。

|  |
| --- |
| //Hands-On-2 toPlayerの距離を計算する。  float len = toPlayer.Length(); |

Hands-On-3 ゲームオーバー判定。

では、最後に求めた距離を使って、ゲームオーバー判定を実装してみましょう。今回のサンプルでは、GameクラスのインすんタスのNotifyGameOverメンバ関数を呼び出したら、ゲームオーバーになるように作っていますので、その関数を呼び出してみましょ

下記のコードを入力してください。

Enemy.cpp

|  |
| --- |
| //Hands-On-3 ゲームオーバー判定。  if (len < 400.0f) {  //ゲームクラスにゲームオーバーを通知。  Game\* game = FindGO<Game>("Game");  game->NotifyGameOver();  } |

## **3.6.2.4 Hands-On 星がプレイヤーを追いかけるようにしてみよう。**

　Chapter\_3\_3を立ち上げて実行してください。このチャプターでは、プレイヤーが星に近づいたときに、パッと消えるのではなく、星を吸い込んでいるような処理を追加してみましょう。

HandsOn-1 星からプレイヤーに向かって伸びるベクトルvを計算する。

　Star.cppを開いて「HandsOn-1 星からプレイヤーに向かって伸びるベクトルvを計算する。」とコメントされている箇所に下記のコードを追加してください。

|  |
| --- |
| //HandsOn-1 星からプレイヤーに向かって伸びるベクトルvを計算する。  Player\* pl = FindGO<Player>("ぷれいや～");  CVector3 v = pl->m\_position - m\_position;  //ちゃんと計算できているか確認のためにベクトルを可視化してみる。  dbg::DrawVector(v, m\_position); |

これは、計算がちゃんとできているか確認するためのコードです。このようなコードは開発用のコードで製品に残すべきではないため、このようなコードはデバッグ用のコードと呼ばれて、リリースビルド(製品版)では、無効になるコードです。

また、星が消えてしまうと、計算の確認ができなくなってしまうので、DeleteGO(this);のコードを削除しておいてください。

HandsOn-2 vを正規化して、いい感じにスカラー倍して座標に加算する。

　続いて、プレイヤーに向かって伸びるベクトルvを使って、星を動かしてみようと思います。Start.cppの「HandsOn-2 vを正規化して、いい感じにスカラー倍して座標に加算する。」と書かれている箇所に下記のコードを追加してください。(これはまだ完成ではありません。)

|  |
| --- |
| //HandsOn-2 vを正規化して、いい感じにスカラー倍して座標に加算する。  m\_position += v; |

いかがでしょうか？星からプレイヤーまでのベクトルvと星の座標m\_positionを足し算しているので、星が一瞬でプレイヤーの足元に移動しています。これは求めていたものではありません。では、先ほどの箇所に、下記の網掛けになっているコードを入力してください。

|  |
| --- |
| //HandsOn-2 vを正規化して、いい感じにスカラー倍して座標に加算する。  v.Normalize(); //これは正規化を行ってくれるCVector3のメンバ関数。  v \*= 20.0f; //vをいい感じにスカラー倍。  m\_position += v; |

HandsOn-3 星とプレイヤーの距離が一定値以下になったら星を消す。

　では、最後に星を削除するコードを追加しましょう。下記のコードを「星とプレイヤーの距離が一定値以下になったら星を消す。」とコメントされている箇所に追加してください。

|  |
| --- |
| if (v.Length() < 20.0f) {  DeleteGO(this);  } |

## **3.6.3 問題**

*例題*

*ベクトルZ0 = 座標( 5 2 ) － 座標( 3 1 )*

ベクトルV0 = 座標( 2 4 ) － 座標(2 1 )

ベクトルV1 = 座標( －３ １ ) － 座標(1 5 )

ベクトルV2 = 座標( 2 －4 ) － 座標(－４ 5 )

ベクトルV3 = 座標( 7 2 ) + 座標(－１０ －5 )



Z0