内容

[**Chapter 1 デカルト座標系** 3](#_Toc84951943)

[**1.1 2Dデカルト座標** 3](#_Toc84951944)

[**1.1.2 原点** 3](#_Toc84951945)

[**1.1.3　軸** 4](#_Toc84951946)

[**1.1.4 デカルト座標系を用いて、場所を指定する** 4](#_Toc84951947)

[**1.1.5 軸の向き** 5](#_Toc84951948)

[**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** 6](#_Toc84951949)

[**1.2 【ハンズオン】星を表示してよう(2D)** 6](#_Toc84951950)

[**1.3 評価テスト(5分)** 8](#_Toc84951951)

[**1.4 3Dデカルト座標** 9](#_Toc84951952)

[**1.4.1 左手座標系、右手座標系** 9](#_Toc84951953)

[**1.4.2 Y-up、Z-up** 9](#_Toc84951954)

[**1.5 【ハンズオン】星を表示してみよう(3D)** 10](#_Toc84951955)

[**1.6 評価テスト(5分)** 11](#_Toc84951956)

[**Chapter 2 ベクトル** 12](#_Toc84951957)

[**2.1 ベクトル　数学的な定義** 12](#_Toc84951958)

[**2.1.1　ベクトルとスカラー** 12](#_Toc84951959)

[**2.1.2　ベクトルの次元** 12](#_Toc84951960)

[**2.1.3 ベクトルの数学的な記法** 12](#_Toc84951961)

[**2.2　ベクトル　幾何学的な定義** 12](#_Toc84951962)

[**2.2.1 ベクトルはどのように見えるか？** 13](#_Toc84951963)

[**2.2.2 【ハンズオン】k2Engineの機能を使って、ベクトルを可視化してみる。** 14](#_Toc84951964)

[**2.2.3 座標とベクトル** 16](#_Toc84951965)

[**2.2.4 問題** 17](#_Toc84951966)

[**Chapter 3 ベクトルの演算** 19](#_Toc84951967)

[**3.1　線形代数 vs 我々が求めるもの** 19](#_Toc84951968)

[**3.1 ベクトルの反転** 19](#_Toc84951969)

[**3.1.1 線形代数の公式** 19](#_Toc84951970)

[**3.1.2 幾何学的解釈** 20](#_Toc84951971)

[**3.1.3 問題** 21](#_Toc84951972)

[**3.1.4 【ハンズオン】ベクトルを反転させてみよう** 22](#_Toc84951973)

[**3.2 ベクトルの大きさ(長さ)** 22](#_Toc84951974)

[**3.2.1　線形代数の公式** 22](#_Toc84951975)

[**3.2.2 幾何学的解釈** 23](#_Toc84951976)

[**3.2.3 問題** 24](#_Toc84951977)

[**3.2.4 【ハンズオン】ベクトルの長さを求めるプログラムを実装する～その１～** 25](#_Toc84951978)

[**3.2.5 【ハンズオン】ベクトルの長さを求める　～その２～** 26](#_Toc84951979)

[**3.3　ベクトルとスカラーの割り算と掛け算** 27](#_Toc84951980)

[**3.3.1 線形代数の公式** 27](#_Toc84951981)

[**3.3.2　幾何学的解釈** 28](#_Toc84951982)

[**3.3.3 問題** 29](#_Toc84951983)

[**3.3.4 【ハンズオン】ベクトルとスカラーの割り算と掛け算** 31](#_Toc84951984)

[**3.4　ベクトルの正規化** 32](#_Toc84951985)

[**3.4.1 線形代数の公式** 32](#_Toc84951986)

[**3.4.2 幾何学的解釈** 33](#_Toc84951987)

[**3.4.3 【ハンズオン】ベクトルを正規化する　～その１～** 35](#_Toc84951988)

[**3.4.4 【ハンズオン】ベクトルを正規化する　～その２～** 36](#_Toc84951989)

[**3.4.5 問題** 36](#_Toc84951990)

[**3.5　ベクトルの足し算** 37](#_Toc84951991)

[**3.5.1 線形代数の公式** 37](#_Toc84951992)

[**3.5.2 幾何学的解釈** 37](#_Toc84951993)

[**3.5.2.1　座標とベクトルの足し算** 37](#_Toc84951994)

[**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。** 38](#_Toc84951995)

[**3.5.2.3　ベクトルとベクトルの足し算** 40](#_Toc84951996)

[**3.5.2.4 Hands-On 風の影響を受けてみよう** 41](#_Toc84951997)

[**3.5.3 問題** 44](#_Toc84951998)

[**3.6 ベクトルの引き算** 46](#_Toc84951999)

[**3.6.1 線形代数の公式** 46](#_Toc84952000)

[**3.6.2 幾何学的解釈** **エラー! ブックマークが定義されていません。**](#_Toc84952001)

[**3.6.2.1 座標と座標の引き算** 46](#_Toc84952002)

[**3.6.2.2 座標と座標の引き算のゲームでの活用法** 48](#_Toc84952003)

[**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。** 50](#_Toc84952004)

[**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。** 53](#_Toc84952005)

[**3.6.3 問題** 54](#_Toc84952006)

[**3.7 練習問題** 55](#_Toc84952007)

[**3.8 章末試験** 58](#_Toc84952008)

# **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。原点は座標系の中心を表し、X=0、Y=0の地点を指します。

### **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。また、これらの軸は必ず垂直に交わっています。

### **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。

ヨッシーー

マリオ

図 1-2



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 【ハンズオン】星を表示してよう(2D)**

　では、デカルト座標系で座標を指定して、複数の星を表示するプログラムを実装していきましょう。Sample\_01\_00/Game.slnを開いてください。

**step-1 ModelRenderクラスのオブジェクトを５つ定義する**

　星を描画するために、まずはModelRenderクラスのオブジェクトを定義しましょう。main.cppを開いて、リスト1-1のプログラムを入力してください。

[リスト1.1 main.cpp]

|  |
| --- |
| // step-1 ModelRenderクラスのオブジェクトを５つ定義する  ModelRender starRender[5]; |

**step-2 星のモデルをロードする**

　続いて、星のモデルデータをロードします。星のモデルデータのロードは、ModelRender::Init()関数を使用することで行うことができます。では、リスト1-2のプログラムを入力して下さい。

[リスト1-2 main.cp]

|  |
| --- |
| // step-2 星のモデルをロードする。  for (int i = 0; i < 5; i++) {  starRender[i].Init("Assets/modelData/star.tkm");  } |

**step-3 デカルト座標系での座標を設定する。**

　step-3がこのハンズオンの本題です。作成した星の描画オブジェクトに、星を表示する位置をデカルト座標系で指定します。リスト1-3のプログラムを入力して下さい。

[リスト1-3 main.cpp]

|  |
| --- |
| // step-3 デカルト座標系での座標を指定する。  starRender[0].SetPosition( -400.0f, 0.0f, 0.0f );  starRender[1].SetPosition( -200.0f, 100.0f, 0.0f );  starRender[2].SetPosition( 0.0f, 200.0f, 0.0f );  starRender[3].SetPosition( 200.0f, 100.0f, 0.0f );  starRender[4].SetPosition( 400.0f, 0.0f, 0.0f ); |

**step-4 描画オブジェクトのアレやコレやを更新する**

　さて、step-4からは今回のテーマからはズレる話となってくるのですが、これも大事なプログラムなので、慣れるために入力しておきましょう。ModelRenderクラスを利用して、絵を表示するためには、ゲームループの中で毎フレーム、ModelRender::Update()関数を呼び出す必要があります。リスト1-4のプログラムを入力して下さい。

[リスト1-4 main.cpp]

|  |
| --- |
| // step-4 星のあれやこれやを更新する。  for (int i = 0; i < 5; i++) {  starRender[i].Update();  } |

　実は、このModelRender::Update()関数の中で、後々勉強するワールド行列というものを計算しています。

**step-5 星を描画するためのドローコールを呼び出す。**

　これで最後です。絵を表示するためには、GPUに対して、毎フレーム描画命令を送る必要があります。この描画命令はModelRender::Draw()関数を利用することで、送ることができます。では、リスト1-5のプログラムを入力してください。

[リスト1-5 main.cpp]

|  |
| --- |
| // step-5 星を描画する。  for (int i = 0; i < 5; i++) {  starRender[i].Draw(renderContext);  } |

入力出来たら実行してみてください。正しく実装できている場合は、図1-4のようなプログラムが実行できます。

図 -4

黒い背景に白い文字がある

低い精度で自動的に生成された説明

## **1.3 評価テスト(5分)**

下記の評価テストを実施しなさい。

<https://docs.google.com/forms/d/e/1FAIpQLScVUDRWzEHuhazdpDaJuTYPUaKR87idRWkeb1e9qlOCDWrTuA/viewform?usp=sf_link>

## **1.4 3Dデカルト座標**

　さて、ここまで２D空間でのデカルト座標についてみてきました。ここからは3D空間でのデカルト座標を見ていきましょう。3Dのデカルト座標系では、X軸とY軸に加えてZ軸という軸が追加されます。そして、3D空間ではX軸、Y軸、Z軸を使って、オブジェクトの座標を表すこととなります。

### **1.4.1 左手座標系、右手座標系**

　３D空間では、必ずしもX軸が右、Y軸が上、Z軸が奥とは限りません。X軸が左、Y軸が上、Z軸が奥の座標系もあります。多くの3Dを扱うアプリケーションでは、この二つのどちらかの座標系が使われています。「X軸が右、Y軸が上、Z軸が奥」の座標系は左手座標系と呼ばれます。一方「X軸が左、Y軸が上、Z軸が奥の座標系」は右手座標系です。サブテキストの図1-12(P13)や図1-13(p14)のように、手の中指をZ軸、人差し指をY軸、親指をX軸としたときに、右手と左手で親指の向きが反転しています。ここから、X軸が右を向いている座標系のことを左手座標系、X軸が左を向いている座標系のことを右手座標系と呼称するようになりました。

　さて、この座標系ですがどのように使い分けるものなのでしょうか？ここまで聞いても、なぜそんなものがあるの？統一すればいいだけじゃない？と思ったのではないでしょうか。実はこの座標系の特別優劣はなく、こういうケースでは左手座標系を使った方がいい、だとか、右手座標系を使った方がいいといったものはありません。ただし、その歴史的な背景により、分野ごとに座標系の好き好みがあります。

例えば、3dsMax、Maya、Blenderなどの代表的な3DCGツールの座標系は右手座標系です。一方UnrealEngineやUnityは左手座標系です。学内のk2Engineも左手座標系です。この違いは重要です。なぜ重要なのかというと、3dsMaxやMayaなどで＋Xの方向に配置したオブジェクトを、なにも工夫せずにUnity、UnrealEngineなどに持ってくると、－Xの方向に表示されるということです。３Dゲームを作る際には、そのデータが作られた環境の座標系(例えば3dsMaxなど)、ゲームエンジンの座標系が右手系なのか、左手系なのか知っておく必要があります。

### **1.4.2 Y-up、Z-up**

3D空間では、左手系、右手系だけではなく、上方向の軸が違う場合もあります。例えば、3dsMaxやUnrealEngineであれば、上方向はZ軸となります。上方向がZ軸となっている環境をZ-upといいます。一方、Unityやk2Engineの上方向はY軸となります。上方向がY軸になっている環境はU-upと呼ばれます。

　表1.1に代表的なツール、エンジンの座標系についてまとめます。

[表1.1]

|  |  |  |
| --- | --- | --- |
| **エンジン/ツール** | **座標系** | **上方向** |
| UnrealEngine | 左手座標系 | Z-up |
| Unity | 左手座標系 | Y-up |
| k2Engine | 左手座標系 | Y-up |
| 3dsMax | 右手座標系 | Z-up |
| Maya | 右手座標系 | Y-up |
| Blender | 右手座標系 | Z-up |

　この座標系の問題により、例えば、3dsMax(右手座標系/Z-up)の座標データを、k2Engine(左手座標系/Y-up)に持ってくる場合は、次のような座標返還を行う必要があります。

|  |
| --- |
| // objectPositionには3dsMaxで配置されたオブジェクトの座標が記憶されている。  Vector3 pos = objectPosition;  // 3dsMaxではZが上方向になっているので、YとZを入れ替える。  float tmp = pos.y;  pos.y = pos.z;  pos.z = tmp;  // さらに右手座標系から左手座標系に変換する。  pos.x \*= -1.0f; |

## **1.5 【ハンズオン】星を表示してみよう(3D)**

では、今度は3D座標系で座標を指定して、複数の星を表示するプログラムを実装していきましょう。Sample\_01\_01/Game.slnを開いてください。

**step-1 3D空間で星の座標を設定する。**

　では、main.cppを開いて下さい。3D空間で座標を設定するといっても、新たに増えたZ軸に座標を設定するだけのことです。では、main.cppを開いて、リスト1-6のプログラムを入力してください。

|  |
| --- |
| // step-1 3D空間で星の座標を指定する。  starRender[0].SetPosition( -400.0f, 100.0f, -200.0f );  starRender[1].SetPosition( -200.0f, 200.0f, 200.0f );  starRender[2].SetPosition( 0.0f, 300.0f, -200.0f );  starRender[3].SetPosition( 200.0f, 200.0f, 200.0f );  starRender[4].SetPosition( 400.0f, 100.0f, -200.0f ); |

　入力出来たら実行してみて下さい。正しく入力出来ていると、図1-5のようなプログラムが実行できます。

**図1-5**

建物 が含まれている画像

自動的に生成された説明

## **1.6 評価テスト(5分)**

下記の評価テストを実施しなさい。

[**https://docs.google.com/forms/d/e/1FAIpQLScOe6yV-\_fqynAncGiXC6dAZxXcIavL6YNSrLdjWgUk\_4nOwQ/viewform?usp=sf\_link**](https://docs.google.com/forms/d/e/1FAIpQLScOe6yV-_fqynAncGiXC6dAZxXcIavL6YNSrLdjWgUk_4nOwQ/viewform?usp=sf_link)

# **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 【ハンズオン】k2Engineの機能を使って、ベクトルを可視化してみる。**

　本校のエンジンのk2Engineにはベクトルを可視化できる機能があります。この機能を使って、ベクトルを可視化してみましょう。Sample\_02\_01/Game/Game.slnを立ち上げてmain.cppを開いてください。

**step-1 ベクトルを定義する**

　まずは、表示するベクトルを定義します。main.cppの該当するコメントの箇所にリスト2.1のプログラムを入力してください。

[リスト2.1]

|  |
| --- |
| // step-1 ベクトルを定義する。  Vector3 testVector;  testVector.x = 500.0f;  testVector.y = 500.0f;  testVector.z = 0.0f; |

**step-2 ベクトルを表示する**

　ベクトルが定義できたら、k2EngineのDrawVector()関数を利用して、ベクトルを表示しましょう。リスト2.2のプログラムを入力してください。

[リスト2.2]

|  |
| --- |
| // step-2 ベクトルを表示する。  \_k2Engine->DrawVector(  testVector, // 第一引数は表示したいベクトル。  g\_vec3Zero // 第二引数はベクトルの基点。  ); |

　第一引数は表示したいベクトル、第二引数はベクトルを飛ばす基点となる座標です。今は原点を基点としています。

　入力出来たら実行してください。図2.1のようにベクトルが表示されていたら実装できています。

図2.2

図形 が含まれている画像

自動的に生成された説明

### **2.2.3 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。

### **2.2.4 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～その１～**

　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　]

あ

### **3.1.4 【ハンズオン】ベクトルを反転させてみよう**

　では、実際にベクトルを反転させるプログラムを実装してみましょう。Sample\_03\_01/Game/Game.slnを開いて下さい。

**step-1 コントローラーのAボタンでベクトルを反転する。**

　では、main.cppを改造して、Aボタンの入力でベクトルを反転させるプログラムを実装しましょう。該当するコメントの箇所にリスト3.1のプログラムを入力してください。

[リスト3.1]

|  |
| --- |
| // step-1 コントローラーのAボタンでベクトルを反転する。  if (g\_pad[0]->IsTrigger(enButtonA)) {  // ベクトルに-1を乗算して反転させる。  testVector \*= -1.0f;  } |

入力出来たら、実行してください。コントローラーのAボタンを入力することで、表示されているベクトルが反転するはずです。

## **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>

### **3.2.4 【ハンズオン】ベクトルの長さを求めるプログラムを実装する～その１～**

では、ベクトルの長さを求めるプログラムを実装してみましょう。Sample\_03\_02/Game/Game.slnを開いてください。

**step-1 ベクトルを定義する。**

　まず、適当なベクトルを定義しましょう。main.cppにリスト3.2のプログラムを入力してください。

[リスト3.2]

|  |
| --- |
| // step-1 ベクトルを定義する。  Vector3 testVector;  testVector.x = 5.0f;  testVector.y = 4.0f;  testVector.z = 3.0f; |

**step-2 ベクトルの長さを計算する。**

続いて、ベクトルの長さを求める公式をプログラミングして、ベクトルの長さを計算して、それを変数lengthに記憶しましょう。リスト3.3のプログラムを入力してください。

[リスト3.3]

|  |
| --- |
| // step-2 ベクトルの長さを計算する。  // xの2乗を計算する  float x2 = testVector.x \* testVector.x;  // yの2乗を計算する  float y2 = testVector.y \* testVector.y;  // zの2乗を計算する  float z2 = testVector.z \* testVector.z;  // x^2 + y^2 + z^2を計算する。  float t = x2 + y2 + z2;  // x^2 + y^2 + z^2の平方根を計算して、  // ベクトルの長さを求める。  // C言語には平方根を求めるsqrt()関数がある。  float length = sqrt(t); |

**step-3 ベクトルの長さをメッセージボックスで表示する**

　これで最後です。ベクトルの長さを確認するために、メッセージボックスを利用して、step-2で求めた、length変数の中身を表示しましょう。リスト3.4のプログラムを入力してください。

[リスト3.4]

|  |
| --- |
| // step-3 ベクトルの長さをメッセージボックスで表示する。  char text[256];  sprintf(text, "ベクトルの長さは %f です。\n", length);  MessageBoxA( nullptr, text, "結果通知", MB\_OK); |

　入力出来たら実行してください。正しく実装できていると「ベクトルの長さは 7.071068 です。」と表示されます。

### **3.2.5 【ハンズオン】ベクトルの長さを求める　～その２～**

　さて、ベクトルの長さを求める処理はゲームプログラムでは頻繁に行う処理です。そこで、このような頻繁に行う処理は関数としてまとめておいて、再利用できるようにしておくと便利です。k2EngineのベクトルクラスにはLength()というベクトルの長さを求める関数が用意されています。今回のハンズオンでは、Vector::Length()関数を使って、ベクトルの長さを求めてみましょう。Sample\_03\_03/Game/Game.slnを開いてください。

**step-1 Vector::Length()関数を利用して、ベクトルの長さを求める。**

　では、main.cppの該当するコメントの箇所に、リスト3.5のプログラムを入力してください。

[リスト3.5]

|  |
| --- |
| // step-1 Vector::Length()関数を利用して、ベクトルの長さを求める。  float length = testVector.Length(); |

入力出来たら実行してください。先ほどのハンズオンと同じように「ベクトルの長さは 7.071068 です。」と表示されたら実装できています。

## **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　]



### **3.3.4 【ハンズオン】ベクトルとスカラーの割り算と掛け算**

　では、実際にベクトルとスカラーの割り算と掛け算をプログラミングしてみましょう。Sample\_03\_04/Game/Game.slnを開いて下さい。

**step-1 コントローラーのAボタンの入力でベクトルを倍にする。**

　まずは、コントローラーのAボタンでベクトルの大きさを２倍にするプログラムを実装しましょう。main.cppの該当するコメントの箇所に、リスト3.6のプログラムを入力してください。

[リスト3.6]

|  |
| --- |
| if (g\_pad[0]->IsTrigger(enButtonA)) {  testVector \*= 2.0f;  } |

**step-2 コントローラーのBボタンの入力でベクトルを1/2にする。**

続いて、コントローラーのBボタンの入力で、ベクトルの大きさを半分にするプログラムを入力しましょう。main.cppの該当するコメントの箇所に、リスト3.7のプログラムを入力してください。

[リスト3.7]

|  |
| --- |
| // step-2 コントローラーのBボタンの入力でベクトルを1/2にする。  if (g\_pad[0]->IsTrigger(enButtonB)) {  testVector /= 2.0f;  } |

　入力出来たら実行してみてください。コントローラーの入力で、ベクトルの大きさが変更できるようになっていたら実装できています。

## **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 【ハンズオン】ベクトルを正規化する　～その１～**

では、ベクトルの正規化を実際にプログラミングしてみましょう。Sample\_03\_05/Game/Game.slnを立ち上げてください。

**step-1 ベクトルを定義する。**

　まずは正規化されていないベクトルを定義しましょう。main.cppに次のプログラムを入力してください。

|  |
| --- |
| // step-1 ベクトルを定義する。  Vector3 testVector;  testVector.x = 5.0f;  testVector.y = 5.0f;  testVector.z = 0.0f; |

**step-2 コントローラーのAボタンの入力でベクトルを正規化する。**

続いて、コントローラーのAボタンでベクトルを正規化するプログラムを実装しましょう。ベクトルの正規化は、ベクトルの大きさを求めて、その大きさで各要素を除算することで行えます。では、main.cppに次のプログラムを入力してください。

|  |
| --- |
| // step-2 コントローラーのAボタンの入力でベクトルを正規化する。  if (g\_pad[0]->IsTrigger(enButtonA)) {  // まずはベクトルの長さを求める。  float len = testVector.Length();  // 続いて、ベクトルの各要素を長さで割り算する。  testVector.x /= len;  testVector.y /= len;  testVector.z /= len;  } |

**step-3 ベクトルを表示する。**

　では、最後にベクトルを表示するプログラムを実装しましょう。main.cppに次のプログラムを入力してください。

|  |
| --- |
| // step-3 ベクトルを表示する。  g\_k2Engine->DrawVector(testVector, g\_vec3Zero); |

　入力出来たら実行してください。コントローラーのAボタンを押すと正規化されたベクトルが確認できます。

### **3.4.4 【ハンズオン】ベクトルを正規化する　～その２～**

　さて、ベクトルの正規化も、ゲームを作っていくうえで頻繁に行う計算です。ですので、長さを求める処理と同じように、正規化の処理を再利用できるように、Vector3::Normalize()関数が用意されています。では、今度はVector3::Normalize()関数を利用して、正規化するプログラムを実装してみましょう。Sample\_03\_06/Game/Game.slnを開いてください。

**step-1 Vector3::Normalilze()関数を利用して正規化する。**

　では、main.cppの該当するコメントの箇所に次のプログラムを入力してください。

|  |
| --- |
| // step-1 Vector3::Normalilze()関数を利用して正規化する。  testVector.Normalize(); |

　入力出来たら実行してください。3.4.3のハンズオンと同様に、コントローラーのAボタンを押して、ベクトルが正規化できていたら完成です。

### **3.4.5 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 【ハンズオン】 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Sample\_03\_07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。Sample\_03\_07/Game/ Game.sln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。

スポーツゲーム, 裁判所, グリーン, 男 が含まれている画像

自動的に生成された説明

図3-7

**step-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。

|  |
| --- |
| // step-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。m\_moveVec.x = g\_pad[0]->GetLStickXF();  m\_moveVec.z = g\_pad[0]->GetLStickYF(); |

**step-2 移動ベクトルを10倍にする**

　このままでは、移動ベクトルが小さすぎて、キャラクターの移動速度が遅くなってしまうので、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。

|  |
| --- |
| // step-2 移動ベクトルを10倍にする  m\_moveVec \*= 10.0f; |

**step-3 移動ベクトルと座標を足し算する**

　では、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。

|  |
| --- |
| // step-3 移動ベクトルと座標を足し算する  m\_position += m\_moveVec; |

入力出来たら実行してみてください。ゲームのキャラクターが移動できるようになっているはずです。

#### 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 【ハンズオン】力の合成

　サンプルプログラムのSample\_03\_09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。

**step-1 Aボタンの入力で右方向のベクトルを加える**

**[Player.cpp]**

|  |
| --- |
| // step-1 Aボタンの入力で右方向のベクトルを加える  if (g\_pad[0]->IsTrigger(enButtonA)) {  Vector3 vRight;  vRight.x = 1.0f;  vRight.y = 0.0f;  vRight.z = 0.0f;  m\_moveVec += vRight \* 2.0f;  } |

**step-2 Bボタンの入力で左方向のベクトルを加える。**

**[Player.cpp]**

|  |
| --- |
| // step-2 Bボタンの入力で左方向のベクトルを加える。  if (g\_pad[0]->IsTrigger(enButtonB)) {  Vector3 vLeft;  vLeft.x = -1.0f;  vLeft.y = 0.0f;  vLeft.z = 0.0f;  m\_moveVec += vLeft \* 2.0f;  } |

**step-3 Xボタンの入力で斜め右方向のベクトルを加える。**

**[Player.cpp]**

|  |
| --- |
| // step-3 Xボタンの入力で斜め右方向のベクトルを加える。  if (g\_pad[0]->IsTrigger(enButtonX)) {  Vector3 v;  v.x = 1.0f;  v.y = 0.0f;  v.z = 1.0f;  v.Normalize();  m\_moveVec += v \* 2.0f;  } |

**step-4 Yボタンの入力で斜め右方向のベクトルを加える。**

**[Player.cpp]**

|  |
| --- |
| // step-4 Yボタンの入力で斜め右方向のベクトルを加える。  if (g\_pad[0]->IsTrigger(enButtonY)) {  Vector3 v;  v.x = -1.0f;  v.y = 0.0f;  v.z = 1.0f;  v.Normalize();  m\_moveVec += v \* 2.0f;  } |

### **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 【ハンズオン】プレイヤーがエニメーに近づいたら、発見されるようにしてみよう。

　サンプルプログラムのSample\_03\_09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いたら、プレイヤーを発見した！というメッセージを表示できるようにしてみましょう。

**step-1 プレイヤーのインスタンスのアドレスを検索する。**

　では、敵キャラクターのプログラムを改造していきましょう。Enemy.cppを開いてください。今回は、プレイヤーと敵の距離を計算する必要があるので、Enemyクラスからプレイヤーのインスタンスにアクセスできるようにする必要があります。本校のエンジンには、NewGO()関数で生成された、インスタンスのアドレスを検索するためのFindGO()関数があります。この関数を利用してぷレイヤーにアクセスできるようにしましょう。Enemy.cppの該当するコメントの箇所に、次のプログラムを入力してください。

[Enemy.cpp]

|  |
| --- |
| // step-1 プレイヤーのインスタンスのアドレスを検索する。  // メンバ変数のm\_playerにアドレスを記憶しておく。  m\_player = FindGO<Player>("Player"); |

**step-2 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。**

　続いて、距離を計算するために、プレイヤーからエネミーに向かって伸びるベクトルを計算しましょう。Enemy.cppに次のプログラムを入力してください。

[Enemy.cpp]

|  |
| --- |
| // step-2 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。  Vector3 toPlayer = m\_player->m\_position - m\_position; |

**step-3 エネミーからプレイヤーに向かって伸びるベクトルを表示する。**

　さて、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ことができたので、さっそく距離計算をしたいところですが、座標同士の引き算で求めるベクトルが幾何学的にどのような意味を持つのか、イメージを固めるために、このベクトルを表示するプログラムを実装してみましょう。次のプログラムを入力してください。

|  |
| --- |
| // step-3 エネミーからプレイヤーに向かって伸びるベクトルを表示する。  g\_k2Engine->DrawVector(toPlayer, m\_position); |

　ここまで実装出来たら、一度実行してみてください。すると、図3.8のように敵キャラクタ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ているはずです。

**[図3.8]**



**step-4 プレイヤーとの距離を計算する。**

　では、step-2で求めたベクトルの長さを計算して、プレイヤーとの距離を計算してみましょう。ベクトルの長さはVector3::Length()関数を利用することで求めることができます。次のプログラムを入力してください。

[Enemy.cpp]

|  |
| --- |
| // step-4 プレイヤーとの距離を計算する。  float distToPlayer = toPlayer.Length(); |

**step-5 プレイヤーとの距離が200以下になったら、プレイヤーを発見したというメッセージを表示する。**

これで最後です。step-4で求めた距離が200以下になったら、プレイヤーを発見したというメッセージを表示しましょう。次のプログラムを入力してください。

[Enemy.cpp]

|  |
| --- |
| // step-5 プレイヤーとの距離が200以下になったら、プレイヤーを発見したというメッセージを表示する。  if (distToPlayer < 200) {  MessageBoxA(nullptr, "プレイヤーを見つけた。", "通知", MB\_OK);  } |

　入力出来たら実行してください。プレイヤーが敵キャラクターに近づくと、図3.9のようにメッセージボックスが表示されたら完成です。

**[図3.9]**



#### 3.6.2.4 【ハンズオン】 プレイヤーを追いかけるエネミーを実装してみよう。

　今回はプレイヤーを追いかけるエネミーを実装していきます。プレイヤーを追いかけさせるためには、エネミーからプレイヤーに向かって伸びるベクトルを求める必要があります。つまりエネミーの座標とプレイヤーの座標の引き算を利用する必要があります。では、Sample\_03\_10を立ち上げてください。

**step-1 プレイヤーに向かって伸びるベクトルを正規化する。**

　今回使用するサンプルプログラムは、3.6.4のハンズオンのプログラムを利用したのになります。ですので、プレイヤーからエネミーに向かって伸びるベクトルはすでに計算済みとなります。

　では、まずプレイヤーからエネミーに向かって伸びるベクトルを正規化しましょう。何かを追いかけるプログラムを実装したい場合に必要になるベクトルは、方向のみの情報なので、正規化を行て単位ベクトルに変換します。

|  |
| --- |
| // step-1 プレイヤーに向かって伸びるベクトルを正規化する。  Vector3 toPlayerDir = toPlayer;  toPlayerDir.Normalize(); |

**step-2 step-1で求めたベクトルを利用して、エネミーの座標を動かす。**

　続いて、step-1で求めた、プレイヤーに向かって伸びる単位ベクトルを、スカラー倍して座標を動かします。この掛け算したスカラーの値が移動速度となります。

|  |
| --- |
| // step-2 step-1で求めたベクトルを利用して、エネミーの座標を動かす。  m\_position += toPlayerDir \* 2.0f; |

　入力出来たら実行してみてください。プレイヤーがエネミーに近づいていくと、エネミーが追いかけてくるようになっています。

### **3.6.3 問題**

ベクトルV0～V3を図示化しなさい。

*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

## **3.7 練習問題**

問１　以下の設問を読んで、設問に答えなさい。なお、電卓の使用は許可する。

設問１　下記のa～fベクトルを反転させたベクトルを答えなさい。

a [ 4 5 ] b [ -2 1 ] c [ -4 -2 ] d [ 0 5 ] e [ -5 0 ] f [ 2 3 -1 ]

設問２　下記のa～fのベクトルの大きさを小数点第２位までで答えなさい。

a [ 2 1 ] b [ -3 4 ] c [ -3 -6 ] d [ 0 7 ] e [ -7 0 ] f [ 1 4 -2 ]

(ヒント　ベクトルの大きさは３平方の定理を使えば計算できる)

設問３ 　下記a～fはベクトルとスカラーの乗算の式です。a～fの計算結果を答えなさい。

a [ 3 4] × 2

b [ 6 -2 ] × 3.2

c [ 2 3 ] × -3

d [ -1 -4] × -7

e [ 0 3 ] × 10

f [ 4 0 ] × -8

設問4 下記のa～fはベクトルとスカラーの除算の式です。a～fの計算結果を答えなさい。

a [ 10 8 ] ÷ 2

b [ 9 -6 ] ÷ 3

c [ 6 3 ] ÷ -3

d [ -14 -21 ] ÷ -7

e [ 0 5 ] ÷ 10

f [ 2 0 ] ÷ -8

設問５ 下記のa～dのベクトルを正規化したベクトルを小数点第二位までで答えなさい。

a [ 3 4 ] b [ -4 2 ] c [ -4 -2 ] d [ 3 0 ]

(ヒント　正規化したベクトルは、ベクトルの大きさで、各要素を除算すると求めることができる)

設問６　下記のa～dはベクトルとベクトルの足し算の式です。解答欄のa～dに計算結果を記入しなさい。

a [ 2 4 ] + [ 1 6 ]

b [ -1 5 ] + [ -3 2 ]

c [ -3 4 ] + [ 2 3 ]

d [ -10 -3 ] + [ -2 -4 ]

設問７　下記のa～dはベクトルとベクトルの引き算です。解答欄のa～dに計算結果を記入しなさい。

a [ 1 3 ] - [ 8 4 ]

b [ -2 3 ] - [ -1 4 ]

c [ -2 1 ] - [ 3 4 ]

d [ -6 -2 ] - [ -5 -1 ]

設問８　下記の図の点Pを点P´に移動させたいときに、点Pの座標に足し算するベクトルVを求めなさい。



P( 2 3 )

**P´( -3 5 )**

設問９　下記の図の点Pは毎分ベクトル[ ２　１ ]の速度で移動している。この時に、点PがA地点に到達するのに何分かかるか求めなさい。



**A地点( 6 1 )**

**P( －6 －5 )**

## **3.8 中間試験**

問１　以下の設問を読んで、設問に答えなさい。なお、電卓の使用は許可する。

設問１　下記のa～fベクトルを反転させたベクトルを答えなさい。

a [ 2 4 ] b [ -7 -2 ] c [ 3 -2 ] d [ -2 1 ] e [ 5 0 ] f [ 2 3 -1 ]

設問２　下記のa～fのベクトルの大きさを小数点第２位までで答えなさい。

a [ 3 2 ] b [ -5 2 ] c [ -1 -9 ] d [ 1 7 ] e [ -7 2 ] f [ 1 2 -2 ]

(ヒント　ベクトルの大きさは３平方の定理を使えば計算できる)

設問３ 　下記a～fはベクトルとスカラーの乗算の式です。a～fの計算結果を答えなさい。

a [ 1 5] × 3

b [ 2 -4 ] × 2.1

c [ 6 2 ] × -9

d [ -3 -1] × -1

e [ 1.5 5.6 ] × 10

f [ 7 2 ] × -8

設問4 下記のa～fはベクトルとスカラーの除算の式です。a～fの計算結果を答えなさい。

a [ 2 4 ] ÷ 2

b [ 6 -3 ] ÷ 3

c [ 16 4 ] ÷ -4

d [ -7 -14 ] ÷ -7

e [ 15 12 ] ÷ 10

f [ 4 2 ] ÷ -8

設問５ 下記のa～dのベクトルを正規化したベクトルを小数点第二位までで答えなさい。

a [ 2 1 ] b [ -5 3 ] c [ -6 -1 ] d [ 4 8 ]

(ヒント　正規化したベクトルは、ベクトルの大きさで、各要素を除算すると求めることができる)

設問６　下記のa～dはベクトルとベクトルの足し算の式です。解答欄のa～dに計算結果を記入しなさい。

a [ 1 3 ] + [ 2 2 ]

b [ -5 2 ] + [ -1 3 ]

c [ -6 1 ] + [ 7 2 ]

d [ -12 -7 ] + [ -4 -8 ]

設問７　下記のa～dはベクトルとベクトルの引き算です。解答欄のa～dに計算結果を記入しなさい。

a [ 2 4 ] - [ 1 5 ]

b [ -6 2 ] - [ -3 2 ]

c [ -7 4 ] - [ 2 5 ]

d [ -8 -4 ] - [ -4 -6 ]

設問８　下記の図の点Pを点P´に移動させたいときに、点Pの座標に足し算するベクトルVを求めなさい。



P( -7 5 )

**P´( -3 -2 )**

設問９　下記の図の点Pは毎分ベクトル[ 3 2 ]の速度で移動している。この時に、点PがA地点に到達するのに何分かかるか求めなさい。



**A地点( 2 3 )**

**P( －7 －3 )**

設問10 座標( -2 -4)にある点P、毎分ベクトルＶ[ 4 1 ]の速度で3分間移動して、

点P´に移動した。この点P´の座標を求めて、解答欄に記入しなさい。

設問11　座標( -7 -10)にある点Pが、4分後に座標( 1 6 )の地点に到達するためには、毎分ベクトルＶの速度で移動する必要がある。このベクトルＶを求めて、解答欄に記入しなさい。

# **Chapter 4 ベクトルの演算～その２～**

3.3節でベクトルとスカラーの掛け算を学びました。しかしベクトルはスカラーとの掛け算だけではなく、ベクトル同士の掛け算も定義されています。ベクトル同士の掛け算は「内積」と「外積」の二つが用意されています。ベクトル同士の掛け算はこれまでの演算と異なり、皆さんにとって違和感のあるものだと思います。公式もぱっと覚えることが難しいかもしれません。しかし、ベクトル同士の掛け算はゲームのプログラミングで非常に多用される演算です。特に内積は非常に重要な演算です。

## **4.1 内積**

　内積はベクトル同士の掛け算です。二つのベクトルV1とV2の内積は次のように表記されます。

**V1・V2**

では、内積の詳細についてみていきましょう。

### **4.1.1 線形代数の公式**

　内積はドット積とも呼ばれます。これは、数式を表記する際にV1・V2のようにドット記号を使うことに由来しています。シェ―ダー言語である、HLSLの内積を求めるための関数名がdotになっているのもこれに由来しています。

　二つの３次元ベクトルの内積の線形代数の公式は次のようになります。

**v1.x×v2.x + v1.y×v2.y + v1.z×v1.z**

　内積の結果はスカラーとなります。例えば、v1( 2, 1, 4 )、v2( 4, 5, 2 )の内積は次のように求めることができます。

v1・v2 = 2**×**4 ＋ 1**×**5 ＋ 4**×**2

= 8 + 5 + 8

= 21

#### 4.1.1.1 練習問題

次の二つのベクトルv1,v2の内積を求めなさい

1. v1( 2, 6, 2 )、v2( 3, 5, 1 )
2. v1( 1, 4, 3 )、v2( 2, 1, 5 )
3. v1( -4, 2, -3 )、v2( 1, 3, 4 )
4. v1( -2, 1, -1 )、v2( -2, 2, -5 )
5. v1( -2, 1, -1 )、v2( -2, 2, -5 )

### **4.1.2 幾何学的解釈**

さて、ここまで内積の公式についてみてきました。しかし計算式だけ見てもこれが一体何に使えるのかさっぱり分からなかったのではないかと思います。では、内積の幾何学的解釈を見ていきましょう。

### **4.1.3 幾何学的解釈　～二つのベクトルの相似性を示すことができる～**

単位ベクトル同士で内積を計算すると、その二つのベクトルがどれだけ似ているのか？を-1～1の範囲で示すことができます。単位ベクトル同士の内積の結果は、相似性が高いベクトル同士であれば１、逆に相似性が低ければ-1となります。では、本当にそうなるのか、次のベクトルの内積を計算して確認してみましょう。

1. v1( 1, 0 )、v2( 1, 0 )
2. v1(1, 0 )、v2( -1, 0 )
3. v1( 0.707, 0.707 )、v2( 0.707, 0.707 )
4. v1( 0.707, 0.707 )、v2( -0.707, -0.707 )
5. v1( 1, 0 )、v2( 0.707, 0.707 )
6. v1( 1, 0 )、v2( 0, 1 )

　計算してみると、同じ向きのベクトル同士の内積の１と３の結果がほぼ１になったと思います(3のベクトルは、小数点第４位以降を省略しているので、ちょうど１にはなりませんが)。一方、真逆のベクトル同士の内積になる２と４の結果はほぼ-1になったと思います。

また、少しだけ似ているベクトル同士の内積の５の結果は0.707となります。微妙に似ているので、1より小さな数値となります。

最後に向きが９０度で異なるベクトル同士の内積の結果となる6の結果は0となります。このように、単位ベクトル同士の内積の結果は1～-1の範囲の数値となり、幾何学的には、ベクトルの相似性を表す指標として使えます。

ゲームではこの性質を利用して、様々なプログラムを書いていくのですが、その代表的なものの一つとして、ライティングの計算が挙げられます。有名なランバート拡散反射では、光が当たるサーフェイスの法線とライトの入射方向とで内積を計算して、その結果を利用して、光の強さを求めます。

その外にはフォンの鏡面反射、リムライトなど様々なライトの計算でこの性質が利用されます。

### **4.1.4 【ハンズオン】ランバート拡散反射を実装してみよう**

　では、内積の性質を利用して、拡散反射をシミュレーションするライティングモデルのランバート拡散反射を実装してみましょう。Sample\_04\_01/Game/Game.slnを立ち上げてください。

**step-1 4本のディレクションライトを使ってライトを計算する。**

　今回はHLSLというシェーダープログラムを改造していきます。ランバート拡散反射は、ライトの方向と、ライトが当たるサーフェイスの向きを表す法線を利用して、ライトの強さを求める反射モデルです。では、Assets/shader/model\_4\_1.fxを開いて、該当するコメントの箇所に、次のプログラムを入力してください。

|  |
| --- |
| // step-1 4本のディレクションライトを使ってライトを計算する。  float3 lig = 0.0f;  for(int ligNo = 0; ligNo < 4; ligNo++){      // ライトの方向とサーフェイスの方向で内積を計算する。      float ligPower = dot( normal, light[ligNo].dir );      // ライトの方向とサーフェイスの方向が逆の時に、ライトを強くしたいので、      // -1を乗算する。      ligPower \*= -1.0f;      // 計算したライトの強さを使って、反射光を足し算していく      lig += ligPower \* light[ligNo].color;  } |

　入力出来たら実行してください、。図4.1のようなプログラムが実行出来たら完成です。

**図4.1**

星のマーク

自動的に生成された説明

### **4.1.5 評価テスト**

<https://docs.google.com/forms/d/e/1FAIpQLSd-E6l1dVbgyjanL_RbBKcRYxiew8iiQoWyaKsjWuXclWECFQ/viewform?usp=sf_link>

### **4.1.6 幾何学的解釈　～特定の方向へのパワーを調べる～**

　内積の幾何学的解釈はベクトル同士の相似性を調べるだけではなく、ある方向にどれだけのパワーがあるベクトルなのかを調べることもできます。これは内積を計算する片方のベクトルが単位ベクトルになっている必要があります。

　では、具体的に計算して確認してみましょう。単位ベクトルV1( 1, 0, 0 )とV2( 5, 3, 4 )の内積を計算してみます。

　　V1・V2 = 1 \* 5 + 0 \* 3 + 0 \* 4

= 5

　V1・V2の内積の結果は5となりました。これがどういう意味になるかというと、V2はV1の方向に５のパワーを持っているということなのです。

**練習問題**

　次のベクトルが単位ベクトル( 1, 0, 0 )の方向にどれだけのパワーがあるか、内積を利用して求めなさい。

1. ( 2, 4, 5 )
2. ( 0, 4, 2 )
3. ( 0.5, 2.4, 4.3 )
4. ( -20, 10, -40 )

　この特性もゲームのいたる箇所で利用されるので鵜が、例えば、光をあふれさせるブルームエフェクトなどでも利用されます。

ブルームエフェクトは光がどれくらい特定の色調に向かって、パワーがあるのかを調べて、一定以上パワーがある光をあふれさせるという処理で実現します。この処理の実装の際に利用するのが内積のこの特性です。

### **4.1.7 【ハンズオン】光のあふれ応現を改造する**

　では、この特性を利用して実装されているブルームエフェクトの輝度抽出シェーダーを改造して、変化を確認してみましょう。Sample\_04\_02/Game/Game.slnを開いて実行してみてください。

　すると図4.2のようなプログラムが起動します。

**図4.2**

星のマーク

自動的に生成された説明

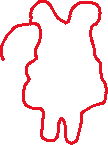
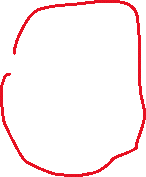


図4.2の赤く囲っている部分が光のあふれエフェクトです。これは、ピクセルのカラーが特定の色調の方向に向かって、一定以上のパワーがあるピクセルから光をあふれ指すことで実現できます。では、ブルームエフェクトの光のパワーを計算している箇所を改造して、このエフェクトの効果を変更してみましょう。

**step-1 赤いパワーを抽出する**

では、Assets/shader/bloom.fxを開いて、該当するコメントの箇所のコードを次のように改造してください。

|  |
| --- |
| // step-1 赤い成分がどれくらい含まれているか計算する  float t = dot(color.xyz, float3(1.0f, 0.0f, 0.0f)); |

　変数colorはピクセルのカラーです。このカラーとfloat3(1, 0, 0)との内積を計算することで、ピクセルカラーがにどれだけ、R成分が含まれているかを計算しています。入力出来たら、実行してください。すると図4.3のようなプログラムが実行できます。

**図4.3**

黒い背景に白い文字がある

低い精度で自動的に生成された説明

いかがでしょうか？図4.2と異なり、赤い成分が強くあふれてきていることが分かるかと思います。

### **4.1.8 評価テスト**

<https://docs.google.com/forms/d/e/1FAIpQLSc4p3fhj2nKVHvUeQPeullWNLiU3bx0DSHz7dMvu6FVY_-tZw/viewform?usp=sf_link>

### **4.1.8 幾何学的解釈　～数式から読み解く～**

　ここまで内積の幾何学的解釈に数式を無視してみてきました。この節では、数式を使って、内積の性質の理解を深めていきます。

３次元のベクトルV1とV2があるとき、V1とV2の内積は下記の計算で求めることができることを学びました。

**V1・V2 = V1.x × V2.x + V1.y × V2.y + V1.z × V2.z ・・・・・・・・①**

この式は余弦定理を使うと、下記の関係になることが証明されています。

**V1・V2 = ||V1||×||V2||×cosθ ・・・・・・・・・・・・・・・・・・・・②**

||V1||はV1のベクトルの大きさ、||V2||はV2のベクトルの大きさ、cosθのθはV1とV2のなす角度です(図4.4)。

**図4.4**

**ダイアグラム

自動的に生成された説明**

ゲームで重要になってくるのは、この２番の公式です。実はこれまで見てきた、幾何学的解釈も、すべてこの公式で説明できるものになります。

#### 4.1.8.1 射影

これまで見てきた幾何学的解釈は、先ほどの公式を利用したものです。また、この性質は名前がついていて、射影と呼ばれます。では、「特定方向へのパワーを求める」ことができるという性質を、数式を使って考えていきましょう。

　例えばVAが大きさ１の正規化されたベクトルで、VBが向きと大きさを持っているベクトル(非正規化)だとします。このとき、VAとVBの内積は下記のような計算になります。

**1×||VB||cosθ = ||VB||cosθ**

これはつまり、VBのベクトルをVAの無限線分上に垂線を落として射影した長さということになります。

大きさ１ベクトルVA

大きさと向きを持つベクトルVB

この長さが求まる

つまり、内積を計算することによって、VBがVA方向にどれだけの力があるのかを求めることができるわけです。

　単位ベクトル同士の内積を計算すると相似性を求めることができるという特性も、この射影の性質を利用したものです。考え方は先ほどと何も変わりません。例えば、単位ベクトルVAと単位ベクトルVBの内積の結果は次のようになります。

大きさ1のベクトルVB

大きさ１のベクトルVA

この長さが求まる

　このように、VBをVA上に射影した長さを求めることができるため、相似性が高ければ、内積の結果は1になり、相似性が低くなると、内積の結果が小さくなるのです。

### **4.1.9 内積の応用事例**

　さて、前節で内積の重要な公式である、**V1・V2 = ||V1||×||V2||×cosθ**を学びました。　この節では、この公式を応用して、敵キャラクターの視野角をプログラミングしてみましょう。

　この公式の||V1||と||V2||はベクトルの大きさを表しています。ここで、V1とV2が正規化(Normalize)された大きさ１のベクトルであるとします。するとV1とV2の内積は下記のような計算となります。

**V1・V2 = １×１×cosθ**

　つまり、単位ベクトル同士の内積の結果は、二つのベクトルがなす角θのcosθとなるのです。

ダイアグラム

自動的に生成された説明

　また、C言語にはacos()関数というcosθをθに戻す関数が存在ます。つまり、次のようなプログラムを書くことで、二つのベクトル間の角度を求めることができるわけです。

|  |
| --- |
| // v1とv2は正規化されているものとする。  // v1とv2の内積を計算してcosθの値を求める。  float t = v1.x \* v2.x + v1.y \* v2.y + v1.z \* v2.z;  // acos()関数を利用して、v1とv2の間の角度を求める。  float angle = acos( t ); |

さて、ではこれを利用して、メタルギアソリッド1の敵兵のような挙動をプロラミングしてみましょう。

メタルギアソリッド1の敵兵は視野角というデータを持っていて、プレイヤーがその視野角の中に入るとプレイヤーを発見して追いかけてくる思考になっています。

この視野角の判定は内積を使用すれば簡単に行うことができます。ではサンプルコードを見てみましょう。

このサンプルコードでは、プレイヤーの座標をplayerPos、敵の座標をenemyPos、敵の前方方向をenemyDirとします。

|  |
| --- |
| //敵からプレイヤーに向かうベクトルを計算する。①  Vector3 toPlayer = playerPos – enemyPos;  //プレイヤーに向かうベクトルを正規化する(大きさ1にする)。②  toPlayer.Normalize();  //敵の前方方向と、プレイヤーへの向きベクトルの内積を計算する。  float angle = toPlayer.Dot((enemyDir);  //内積の結果はcosθになるため、なす角θを求めるためにacosを実行する。③  angle = acos(angle);  //これでangleにはラジアン単位の角度が入ったため、視野角の判定を行える。  if (fabsf(angle) < Math::DegToRad(45.0f)) {  //視野角90度以内に入った。  } |

この計算を図示すると下記のようになります。

③なす角θ

1. 正規化前のtoEnemy

②　正規化されたtoEnemy

enemyDir(敵の前方)

4.1.10 【ハンズオン】視野角をプログラミングする

　では、実際に視野角を持ったAIをプログラミングしてみましょう。Sample\_04\_03/Game/Game.slnを開いてください。

**step-1 視野角45度以内にプレイヤーが入ってきたら追いかける。**

サンプルが立ち上がったら、Enemy.cppの該当するコメントの箇所に次のプログラムを入力してください。

|  |
| --- |
| // step-1 視野角45度以内にプレイヤーが入ってきたら追いかける。  // 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。  Vector3 toPlayer = m\_player->m\_position - m\_position;  // プレイヤーとの距離を計算する。  float distToPlayer = toPlayer.Length();  // プレイヤーとの距離が400以下だったら追いかける。  if (distToPlayer < 400) {  // ここから視野角判定  // プレイヤーに向かって伸びるベクトルを正規化する。  Vector3 toPlayerDir = toPlayer;  toPlayerDir.Normalize();  // エネミーの前方方向とtoPlayerDirとの内積を計算する。  float t = toPlayerDir.Dot(m\_enemyFowrad);  // 内積の結果をacos関数に渡して、m\_enemyFowradとtoPlayerDirのなす角度を求める。  float angle = acos(t);  // 視野角判定  // fabsfは絶対値を求める関数！  // 角度はマイナスが存在するから、絶対値で判定する。  if (fabsf(angle) < Math::DegToRad(45.0f)) {  // プレイヤーを発見したので、追いかける。  // エネミーの座標を動かす。  m\_position += toPlayerDir \* 2.0f;    // 敵キャラの前方方向を更新する。  m\_enemyFowrad = toPlayerDir;  }  } |

　入力出来たら、実行してみてください。

### **4.1.10 評価テスト**

<https://docs.google.com/forms/d/e/1FAIpQLSdBZzT-6DdVkUFQVScc-F0nvQgNkIetkIIa1EMj4H-d86omgw/viewform?usp=sf_link>

## **4.2 外積**

　もう一つの掛け算は外積です。ふたつのベクトルV1、V2の外積は次のように表記されます。

**V1 × V2**

　外積はベクトルとスカラーの掛け算と同じクロス記号を利用します。しかし、スカラーとの掛け算とは異なり、クロス記号を省略することはできません。外積は内積と異なり、３次元ベクトルのみで定義されており、２次元ベクトルでは定義されていません。

　外積はクロス積とも呼ばれます。多くの数学ライブラリで、外積を利用する関数名はこれに由来しています。例えばhlslでは、外積を求めるcross()関数が用意されています。

　では、外積の詳細を見ていきましょう。

### **4.2.1 線形代数の公式**

二つのベクトルV1とV2の外積の公式は次のようになります。

× =

外積の結果は新しいベクトルとなります。例えば、v1( 2, 1, 4 )、v2( 4, 5, 2 )の外積は次のように求めることができます。

× =

=

外積はこれまで見てきた公式の中では、覚えにくい公式ですが、覚える上でのポイントがあります。では、そのポイントを見ていきましょう。

1. 二つのベクトルの外積の各成分を求めたいときは、該当成分以外のたすき掛けを行う

　例えば、外積の結果のx成分を求めたい場合は、x成分以外のy, z成分でたすき掛けを行います。

x成分を求める場合

× =

同様に、y成分、z成分を求める場合も、該当成分以外のたすき掛けを行います。

y成分を求める場合

× =

z成分を求める場合

× =

1. 左辺と右辺どっち？？？

　私も外積の公式を覚える際に、１番のポイントはすぐに覚えられたのですが、この２番の点で苦労しました。

　１のポイントを覚えることで、外積のx成分を求める際に、その他の成分でたすき掛けをすればいいことは分かりました。

　x成分を求める場合

× =

さて、このたすき掛けですが、V1.y \* V2.z －　V1.z \* V2.y でしょうか？それとも、V1.z \* V2.y －　V1.y \* V2.zでしょうか？引き算は演算の順番によって、結果が変わるため、この順番は重要です。これを覚えるには左辺のV1の項のみ覚えたらOKです。

× =

外積の結果の左辺のV1の項は上からy,z,xとなります。この順番だけ覚えておけば、後は１番のポイントに当てはめれば計算すればOKです。

#### 4.2.1.1 練習問題

次の二つのベクトルv1,v2の外積を求めなさい

1. v1( 2, 6, 2 )、v2( 3, 5, 1 )
2. v1( 1, 4, 3 )、v2( 2, 1, 5 )
3. v1( -4, 2, -3 )、v2( 1, 3, 4 )
4. v1( -2, 1, -1 )、v2( -2, 2, -5 )
5. v1( -3, 1, -2 )、v2( -1, 2, -4 )

### **4.2.2 幾何学的解釈　～垂直なベクトルを求める～**

　では、外積の幾何学的解釈を見ていきましょう。外積で求まるベクトルは、元の二つのベクトルに垂直なベクトルとなります(図4.5)。

**図4.5**

文字が書かれている

低い精度で自動的に生成された説明

　この性質がゲームで最も利用される外積の性質です。もっともよく使われるのは回転軸の計算でしょう。その他には、ポリゴンの法線の計算でも外積が利用されます。

### **4.2.3【ハンズオン】ポリゴンの法線を求める。**

　ポリゴンの法線というのは、ポリゴンに垂直なベクトルとなります。法線は、ポリゴンを構成する３頂点が分かっていれば求めることができます。図4.6を見てください。

ダイアグラム

低い精度で自動的に生成された説明

　では、Sample\_04\_04/Game/Game.slnを立ち上げて実行してみてください。すると、図4.7のような何も表示されない、真っ黒なプログラムが実行されます。

**図4.7**

図形, 四角形

自動的に生成された説明

実はこのプログラムはユニティちゃんを表示していて、ライトもあてています。実はこのプログラムは、ライトをあてているのに、ユニティちゃんが真っ黒で表示されているのです。なぜユニティちゃんが真っ黒になってしまっているかというと、このプログラムでは、ユニティちゃんのポリゴンの法線を計算するプログラムをオフにしているからです。ライティングの計算では、ライトの方向とポリゴンの法線を利用していました。しかし、法線が計算されておらず、０ベクトルになってしまっているため、ライトの計算が正しく行えていないのです。

　では、ハンズオンでポリゴンの法線を計算するプログラムを実装しましょう。

**step-1 0番目の頂点から１番目の頂点に向かって伸びるベクトルv1を計算する。**

法線を求めるためには、ポリゴンの０番目の頂点から１番、２番の頂点に向かって伸びるベクトルを二つ求める必要があります。では、まずは０番目の頂点から１番目の頂点に向かって伸びるベクトルv1を求めましょう。HandsOn.cppを開いて、該当するコメントの箇所に次のプログラムを入力してください。

|  |
| --- |
| // step-1 0番目の頂点から１番目の頂点に向かって伸びるベクトルv1を計算する。  Vector3 v1 = p1 - p0; |

**step-2 0番目の頂点から２番目の頂点に向かって伸びるベクトルv2を計算する。**

　続いて、０番目の頂点から２番目の頂点に向かって伸びるベクトルv2を計算します。該当するコメントの箇所に次のプログラムを入力してください。

|  |
| --- |
| // step-2 0番目の頂点から２番目の頂点に向かって伸びるベクトルv2を計算する。  Vector3 v2 = p2 - p0; |

**step-3 v1とv2に垂直なベクトルを求める。**

　０番目の頂点から１番、２番の頂点に向かって伸びる二つのベクトルを求めることができたので、次は外積を使って、この二つのベクトルに垂直なベクトルを求めます。本校のエンジンには、ベクトルの外積を求めることができるCross()関数がありますので、今回はそれを利用します。

　では、該当するコメントの箇所に次のプログラムを入力してください。

|  |
| --- |
| // step-3 v1とv2に垂直なベクトルを求める。  normal = Cross(v1, v2); |

**step-4 外積で求めたベクトルを正規化する。**

最後のステップはまだ説明をしていないのですが、外積の結果は単位ベクトルになっているとは限らないので、正規化を行います。該当するコメントの箇所に次のプログラムを入力してください。

|  |
| --- |
| // step-4 外積で求めたベクトルを正規化する。  normal.Normalize(); |

　入力出来たら実行してください。すると図4.8のように、ユニティちゃんが表示されます。

**図4.8**

ゲームのキャラクター

低い精度で自動的に生成された説明

### **4.2.4 幾何学的解釈　～垂直なベクトルの向きは？～**

　さて、外積を計算することで、元の二つのベクトルに垂直なベクトルを求めることができることが分かりましたが、では、垂直なベクトルの向きはどうなるのでしょうか？図4.9を見てみてください。

**図4.9**

折れ線グラフ

中程度の精度で自動的に生成された説明

　このように、V1とV2に垂直なベクトルは２本あることが分かります。V1×V2を計算すると、どちらのベクトルが求まるのでしょうか？

　結論を言いますと、この図の場合、V1に対してV2が時計回り方向にあるため、①のベクトルが求まります(図4.10)。

**図4.10**

ダイアグラム

自動的に生成された説明

逆に、V2がV1に対して、反時計回りの場合は、２番のベクトルになります。このように、外積で求まるベクトルの方向は、元のベクトルの方向によって異なりますので注意してください。

### **4.2.5 幾何学的解釈　～平行四辺形の面積を求める～**

　外積の最後のおまけのような内容なのですが、外積で求めたベクトルの大きさは、元の二つのベクトルを辺とする平行四辺形の面積になります(図4.11)。

**図4.11**

図形, 多角形

自動的に生成された説明

外積の大きさは次のような関係になることが証明されています。

**||V1×V２|| = ||V1||×||V2||×sinθ**

　ここでのθはV1とV2のなす角のθです。この計算式を紐解くと、外積の結果で求めることができるベクトルの大きさが、平行四辺形の面積になることが分かります(図4.12)。

**図4.12**

グラフィカル ユーザー インターフェイス が含まれている画像

自動的に生成された説明

この特性はめったに利用することはありません。著者は昔、三角形ポリゴンの面積を求める際などに利用したことがあります(平行四辺形の面積を半分にしたら三角形の面積になりますね)。めったには使いませんが、知っておくと便利なのこともあるので、紹介しておきました。

　ただ、ここで知っておいてほしいのは、外積の結果のベクトルの大きさは単位ベクトルになるとは限らないということです。これは注意しておいてください。(もちろんたまたま単位ベクトルになることはあります。)

### **4.2.6 評価テスト**

<https://docs.google.com/forms/d/e/1FAIpQLSdaaXfVER6fvSZ3LwxY8JrJEZWm4mH9uY5w4JMCfjhSrugMHA/viewform?usp=sf_link>

**Chapter 5 行列入門**

# **Chapter 6 行列と線形変換**