# 【ハンズオン】エネミーに視野角を実装してみる。

## step-1 敵兵の前方方向(enemyForward)を求める。

Sample\_04/Sample\_04.slnをダブルクリックして、プロジェクトを起動してEnemy.cppを開いて、リスト－1のプログラムを入力してください。

リスト－１

|  |
| --- |
| //step-1 敵兵の前方方向(enemyForward)を求める。  //前方方向は{0, 0, 1}のベクトルをm\_rotationで回して求める。  CVector3 enemyForward = CVector3::AxisZ();  m\_rotation.Multiply(enemyForward); |

　このサンプルの敵キャラクターは、全く回転していないときはX軸の方向を向いています。ですので、敵キャラの回転クォータニオンを使って、X軸のベクトルを回してやれば、現在の敵キャラクターの前ベクトルを求めることができます。

## step-2 敵兵からプレイヤーに伸びるベクトル(toPlayerDir)を求める。

　続いて、敵兵からプレイヤーに向かって伸びるベクトルを計算します。リスト－２のプログラムを入力してください。

リスト－２

|  |
| --- |
| //step-2 敵兵からプレイヤーに伸びるベクトル(toPlayerDir)を求める。  CVector3 toPlayerDir = m\_player->GetPosition() - m\_position;  //正規化を行う前に、プレイヤーまでの距離を求めておく。  float toPlayerLen = toPlayerDir.Length();  //正規化！  toPlayerDir.Normalize(); |

　step-3で内積を利用して、敵兵とプレイヤーの角度を調べるプログラムを作成するのですが、この時に必要なベクトルは単位ベクトルなので、正規化を行っています。また、step-5で敵兵とプレイヤーの距離も必要になるので、正規化を行う前に距離計算も行っています。

## step-3 enemyForwardとtoPlayerDirの内積を計算する。

　続いて、敵兵の前方ベクトルと、プレイヤーに向かって伸びるベクトルで内積を計算します。リスト３のプログラムを入力してください。

リスト－３

|  |
| --- |
| //step-3 enemyForwardとtoPlayerDirの内積を計算する。  float d = enemyForward.Dot(toPlayerDir); |

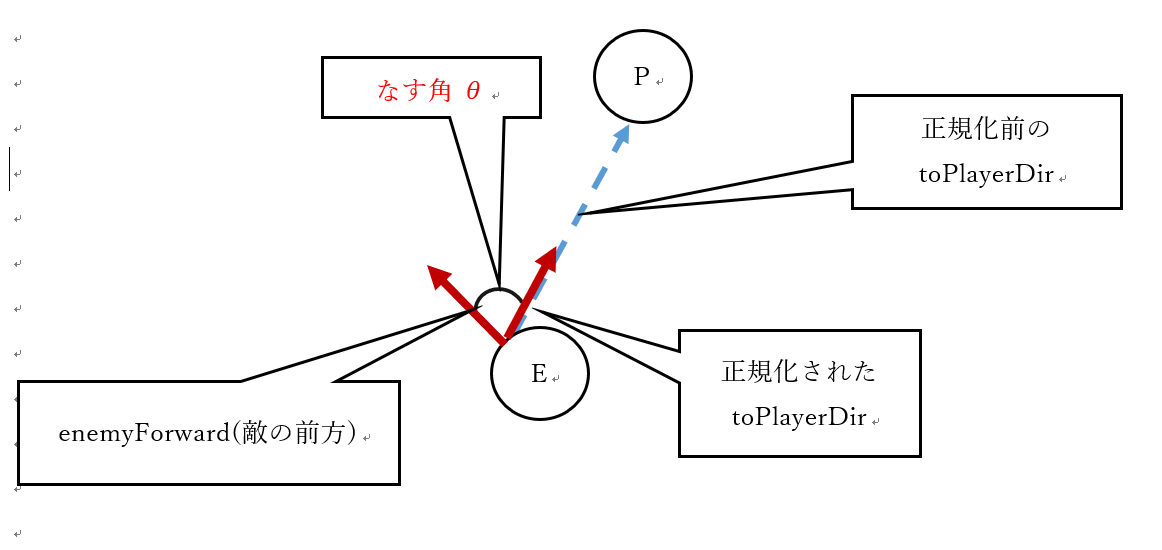
## Step-4 内積の結果をacos関数に渡して、enemyForwardとtoPlayerDirのなす角度を求める。

　内積の結果は二つのベクトルのなす角度のcosθとなっています。この値を逆三角関数に渡して、角度を求めましょう。リスト－４のプログラムを入力してください。

リスト－４

|  |
| --- |
| //step-4 内積の結果をacos関数に渡して、enemyForwardとtoPlayerDirのなす角度を求める。  float angle = acos(d); |

　このプログラムで下記の図のような角度を求めることができました。



## step-5 視野角判定

　では、最後のハンズオンです。ここまでで求めた結果を利用して視野角判定を行います。リスト－５のプログラムを入力してください。

リスト－５

|  |
| --- |
| //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、絶対値にする。  if ( fabsf(angle) < CMath::DegToRad(45.0f) //角度の判定  && toPlayerLen < 200.0f //距離の判定  ) {  //状態を追跡状態にする。  m\_state = eState\_Tracking;  } |

　視野角判定は角度だけではなく、距離の判定も必要になります。

## step-6 実行

　ここまで入力出来たら、プログラムを実行してみてください。正しく入力出来ていると敵兵がプレイヤーを発見すると追いかけてくるようになります。

【ハンズオン】エネミーを進行方向に向ける。

## step-1　atan2関数を使って、進行方向に向けるための角度を求める。

ここでは、敵の進行方向を利用して、回転させる角度を求めていきます。この角度を求めるためにatan2関数を利用します。  
　リスト－１のプログラムを入力してください。

リスト－１

|  |
| --- |
| //step-1 atan2を使って、進行方向に向けるための角度を求める。  float angle = atan2(m\_moveSpeed.x, m\_moveSpeed.z); |

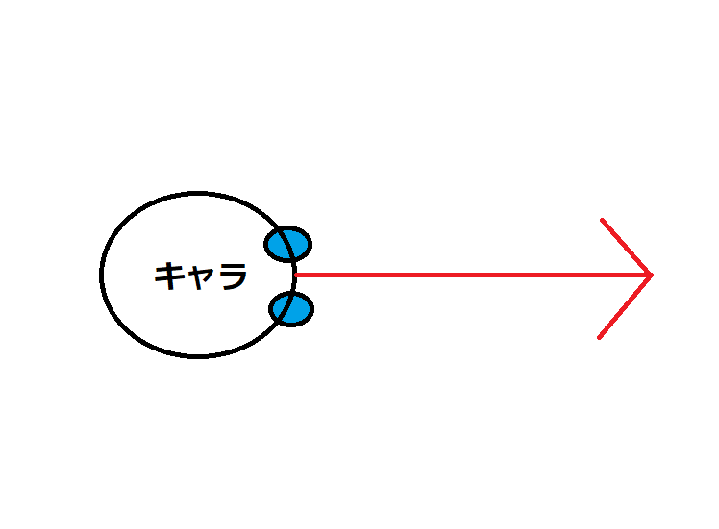
## step-2　求めた角度を使って、回転クォータニオンを求める。

回転させる角度を求めることができたら、あとは回転クォータニオンを作成するだけです。リスト－２のプログラムを入力してください。

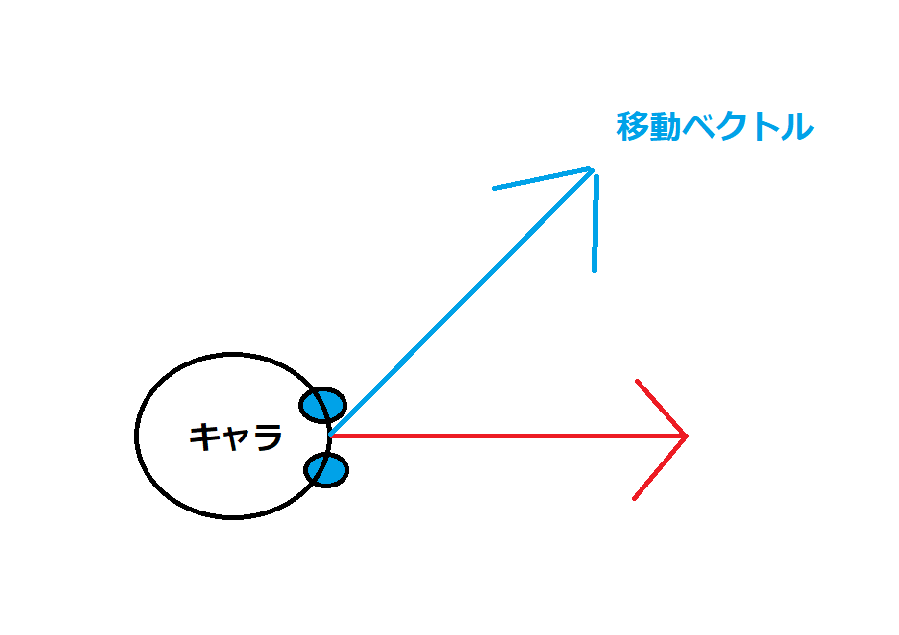
リスト－２

|  |
| --- |
| //step-2 求めた角度を使って、回転クォータニオンを求める。  m\_rotation.SetRotation(  CVector3::AxisY(),  angle  ); |

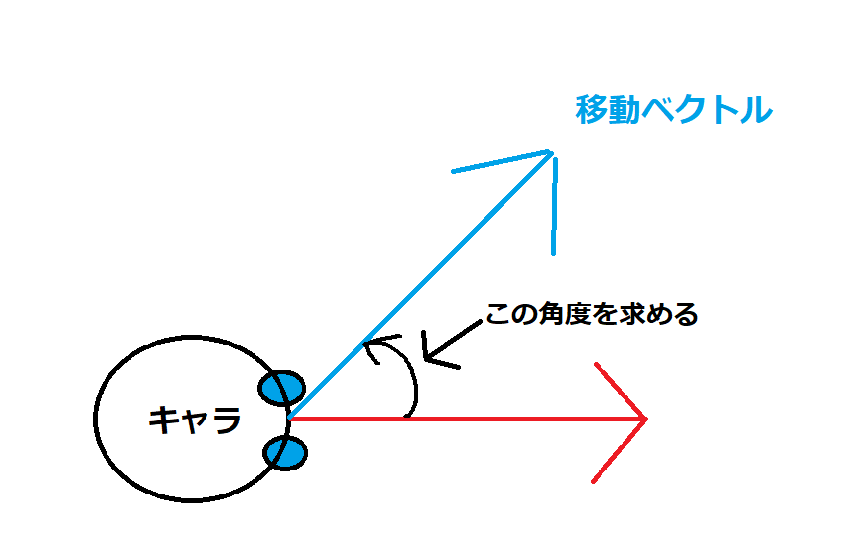
　では、なぜこのプログラムでキャラが進行方向を向くのか考えてみましょう。例えばキャラクターが回転していないときはX軸を向いている場合を考えます。



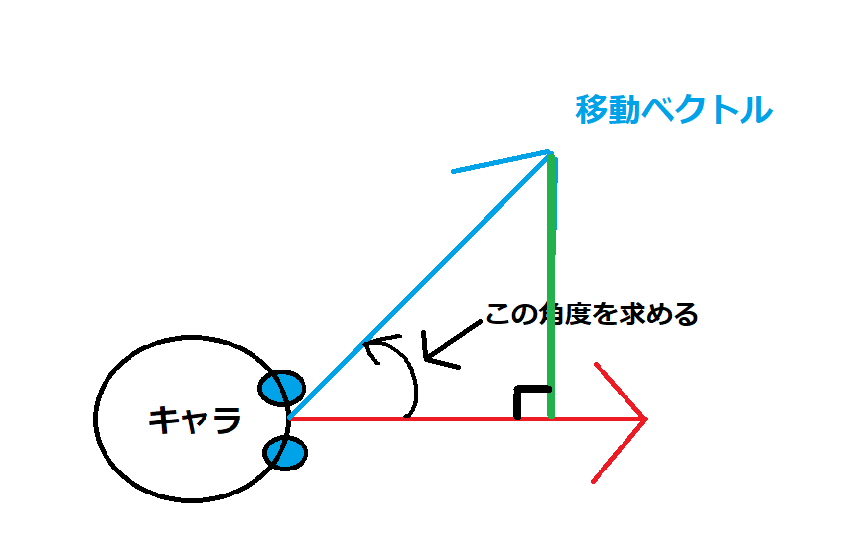
　このキャラを青い矢印の方向に進ませるケースを考えます。



　この時、キャラを進行方向に向けるためには赤い矢印を青い矢印に回すための回転角度を求めればいいことになります。



　ここで補助線を引いてみると、赤い矢印と青い矢印とで直角三角形が作れることが分かります。



　この直角三角形の２辺の長さは、ちょうど移動ベクトルのX成分とZ成分の値になります。直角三角形の２辺の長さが求まれば、この直角三角形のcosθ、sinθ、tanθのどれかの値を求めることができます。今回はZ ÷ Xを計算することで、tanθを求めることができます。tanθが求まれば、その値をatan関数に渡してやれば、角度を求めることができます。あとは、この角度を使って、Y軸周りの回転クォータニオンを作成するだけです。

## 【実習】プレイヤーを進行方向に向かせる

　Sample\_04/Player.cppを改造して、プレイヤーを進行方向に向かせるプログラムを実装しなさい。

# 処理落ち対策(時間ベース)

時間ベースの処理落ち対策を考えていきましょう。ここで説明する時間ベースの処理落ち対策は、教科書のp193のものと同じなのですが、教科書の場合少々回りくどいことをしていますので、違うアプローチで説明していきます。

時間ベースでの処理落ち対策を行うためには、これまでのフレームベースでの考え方から下記のように変更する必要があります。

**1フレームに5移動する　→　1秒間に３００移動する。**

１秒間に３００移動すると考えた時に、じゃぁ、1フレームではどれだけ動けばいいの？というと、下記の表のように1フレームにかかった秒数に応じて移動量を計算する必要があります。

|  |  |  |
| --- | --- | --- |
| 1フレームかかった時間 | 計算 | 1フレームの移動量 |
| 0.1秒 | 300×0.1秒 | 30 |
| 0.2秒 | 300×0.2秒 | 60 |
| 0.5秒 | 300×0.5秒 | 150 |
| 0.016秒 | 300×0.016秒 | 4.8 |

時間ベースで考えることで、1フレームにかかった時間が長くなれば、移動量も大きくなりますし、時間が短ければ移動量も少なくなります。そして、結果として1秒間に移動する量は30fpsで動作していようが、60fpsで動作していようが、300移動することになります。

時間ベースでの処理落ち対策を実施するためには、1フレームの時間を計測する必要があります。この後Sample\_05を使用して、ハンズオンで処理落ち対策を実装していきます。このサンプルには時間計測するためのStopwatchクラスと、1フレームの時間を記憶するためのGameTimeクラスが用意されています。自身のゲームで処理落ち対策が必要であれば、参考にしてみてください。

# 【ハンズオン】処理落ち対策(時間ベース)

## Step-1 動作確認

　では、まずSample\_05を立ち上げて実行して、処理落ちしていない状態での動作を確認してみてください。

## Step-2 16ミリ秒スリープさせて、わざと処理落ちさせてみる

　続いて、わざと処理落ちを発生させて、フレームレートを落としてみましょう。Main.cppの55行目を下記のように書き換えてみて下さい。

|  |
| --- |
| //step-2 16ミリ秒スリープさせて、わざと処理落ちさせてみる。  Sleep(16); |

Sleep関数はマイクロソフトが用意している関数で、指定された時間(ミリ秒単)プログラムを停止させることができます。

## Step-3 動作確認

　では、Sample\_05を実行して、処理落ちが発生していることを確認してください。

## Step-4 プレイヤーの移動を時間ベースに変更する。

　処理落ちが発生していることが確認出来たら、プレイヤーの移動処理をフレームベースから時間ベースに変更して、処理落ち対策を実施しましょう。Player.cppを開いて36行目のプログラムを下記のように変更してください。

|  |
| --- |
| //Step - 4 プレイヤーの移動を時間ベースに変更する。  move \*= 300.0f \* g\_gameTime.GetFrameDeltaTime(); |

## Step-5 動作確認

　Step-4が入力出来たら、実行して動作を確認してください。プレイヤーの移動速度が30fpsと60fpsで変わらなくなっています。

## Step-6 アニメーションの再生もフレームレートの変更に対応する。

　実は、処理落ちが発生して問題が起きているのは、移動速度だけではありません。アニメーションの再生も遅くなっています。アニメーションはUpdate関数を呼び出すことで、進んでいくのですが、実はこの関数の第一引数で指定しているのは、アニメーションを進める秒数です。現在は1/60秒を指定しているので、60fpsで動作しているときは問題ないのですが、処理落ちが発生して、30fpsになってしまうと、アニメーションの再生速度が低下します。では、プログラムを下記のように改造して、アニメーションもフレームレートの変更に対応できるようにしてみましょう。

|  |
| --- |
| //Step-6 アニメーションの再生もフレームレートの変更に対応する。  m\_animation.Update(g\_gameTime.GetFrameDeltaTime()); |

## Step-7 動作確認

　では、動作確認をしてください。プレイヤーのアニメーションも可変フレームレートに対応ができています。

# 積分誤差

　今回のような移動処理はオイラー積分と呼ばれる積分法なのですが、実は加速度を加えて、速度が変化していくような環境では、計算に誤差が生まれます。下記の表は物体が1秒間落下したときの座標の変位を示しています。

***ケース１****１フレームに0.2秒かかっている場合 (1秒経過するまでに５フレームかかる)。*

|  |  |  |
| --- | --- | --- |
| フレーム数 | 落下速度(m/s) | 座標 |
| 1 | -1.96 | -0.392 |
| 2 | -3.92 | -1.176 |
| 3 | -5.88 | -2.352 |
| 4 | -7.84 | -3.92 |
| 5 | -9.8 | -5.88 |

***ケース２****１フレームに0.5秒かかっている場合(１秒経過するまでに２フレームかかる)。*

|  |  |  |
| --- | --- | --- |
| フレーム数 | 落下速度(m/s) | 座標 |
| 1 | -0.49 | -0.245 |
| 2 | -0.98 | -0.735 |

　このように、1フレームにかかっている時間によって、誤差が生じてしまうため、厳密な同期が必要なネットワークゲームなどでは、時間ベースの可変フレーム対応を行うことは困難になります。