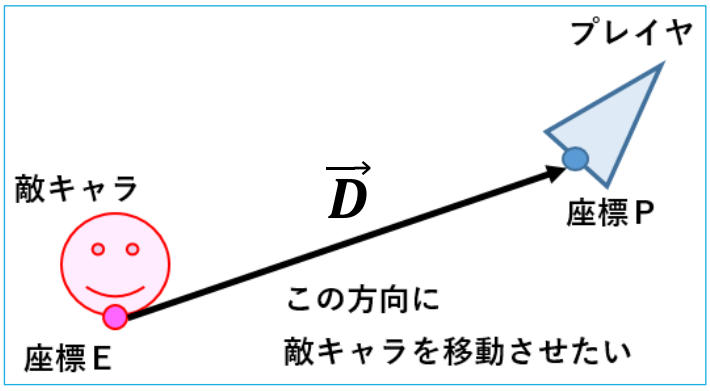
速度ベクトル（移動ベクトル）

■速度ベクトル

あるゲームで「敵キャラ」が「プレイヤ」キャラに向かって接近することを考えます。
（簡単のために 2 次元で考えます）



いま、敵キャラの座標 E の位置ベクトル $\overrightarrow{en} = (en_x, en_y)$,
プレイヤの座標 P の位置ベクトル $\vec{p} = (p_x, p_y)$ とします。

【瞬間移動】

E から P に ***瞬間移動（たとえば 1 フレームで移動）させる*** ベクトル $\vec{D}$ は

次の計算式で求まります。

$$\vec{D} = \boxed{\vec{p}} - \boxed{\overrightarrow{en}}$$

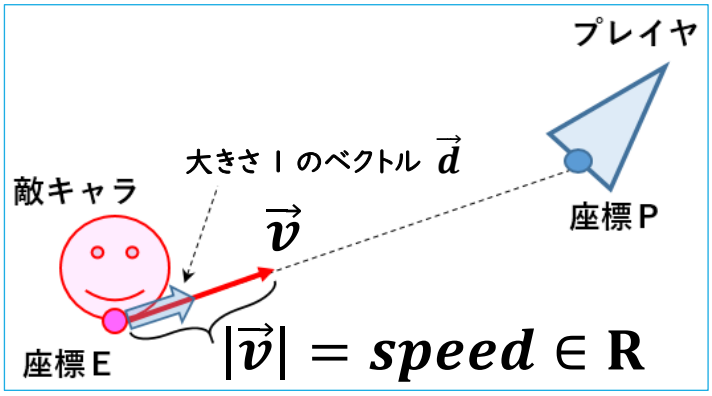
成分表示では $\vec{D} = \left(\boxed{p_x - en_x} , \boxed{p_y - en_y} \right)$

「 ベクトル tips 集 」

岡山情報 IT クリエイター専門学校

【1フレームあたりの移動】

つぎに、E から P に *1 フレームで speed（実数）の距離だけ移動させる* ベクトル $\vec{v}$ を求めたいとき、どのような方法が考えられるか、ディカッションしてみましょう。



※ヒント

- 大きさ 1 のベクトル（単位ベクトル）を実数 $speed$ 倍したベクトルの大きさは $speed$ である
- 移動したい方向は【Step.01】のベクトル $\vec{D}$ の方向である

$$\vec{D} = \boxed{\vec{p}} - \boxed{\vec{en}}$$

$$\vec{d} = \boxed{\vec{D}} \;/\; \boxed{|\vec{D}|} \quad \cdots \text{ ベクトル}\vec{D}\text{方向の単位ベクトル}$$

$$\vec{v} = \boxed{speed} \times \boxed{\vec{d}}$$

◆ まとめ ◆

位置 P から位置 Q へ、1 フレームあたり $speed \in \mathbf{R}$ だけ移動する速度ベクトルの求め方

1. まず位置 P から位置 Q への方向ベクトル $\overrightarrow{PQ}$ を求める　… $\vec{D} = \vec{q} - \vec{p}$
2. 方向ベクトル $\overrightarrow{PQ}$ を「正規化」する　… $\vec{d} = \vec{D}.nomalized$
3. 2. で正規化したベクトルに $speed$　… $\vec{v} = speed \times \vec{d}$

「 ベクトル tips 集 」

岡山情報 IT クリエイター専門学校

■単位（物理量）のはなし

さて、ここで ***speed*** の「単位」について考えたいと思います。「単位」はゲームプログラミング（Unity を使うときも）するうえで、非常に大事な考え方になります。

2D ゲームを考えた場合、移動距離を測るときは「pixel（ピクセル）」が都合がよいでしょう。

たとえば、1 フレームで 3 ピクセルだけ移動する場合は、3[pixel/frame]（3 ピクセル パー フレーム）といったように記述します（略して 3[p/f]や 3ppf と記述してもよいでしょう）。この [pixel] や [pixel/frame] を「単位」と呼び、ものの量などを測る/計る/量る（はかる）ための基準や尺度です。

また、　　の部分の考え方は、我々の身の回りでは「速さ」と呼んでいます。

よく聞く速さの例としては、たとえば、時速 30 キロ（30[km/h]）などがありますね。
この場合は、「1 時間で 30 キロメートル進む」ということになります。

「速さ」に近い言葉として「速度」がありますが、「速度」は「速さ」と「方向」を合わせた意味となります。

まさに「ベクトル」です。**・・・ これを速度ベクトルと呼びます**

3D の場合は、そのゲーム空間の尺度によります。単純に 1 フレームあたり 3 移動する（単位を考えない）、と考える場合もありますが、実世界に照らし合わせて、「m（メートル）」や「km（キロメートル）」をよく使います。

■「速さ」と「時間」と「距離」

・ 速さ　　… 単位時間あたりに進む距離
　※ ここでいう単位は1秒、1分、1時間のことで、1 ということが重要！
　※ 速さ ＝ 進んだ距離 ÷ 進むのにかかった時間
　※ ある瞬間の速さではなく、ある時間をかけて進んだ距離で計算するので、
　　その時間の間（あいだ）の「平均の速さ」を意味します

【速さの単位の例】
　X[m/s]　　… 秒速 X メートル（1 秒間に X メートル進む）
　X[km/h]　　… 時速 X キロメートル（1 時間に X キロメートル進む）

・ 時間　　… 進むのにかかった時間 ＝ 進んだ距離 ÷ 速さ
・ 距離　　… 進んだ距離 ＝ 速さ × 進むのにかかった時間

【Unity で試してみよう ①】

1. まずは Plane オブジェクトを配置して地面とみなしましょう
2. その Plane 上に2つの Cube オブジェクトを十分な距離をとって配置しよう
3. どちらかの Cube を、もう一方の Cube（ターゲット）に向かって移動させよう
   - インスペクタから「速さ」をパラメータとして指定できるようにすること
   - もう一方の Cube に接触、もしくは追い越したら一時停止させること（Debug.Break 関数）

   【技術調査】
   ※ 球と球の当たり判定
   ※ 移動後の位置からターゲットへの方向ベクトルと速度ベクトルとの内積が…

★発展★

ターゲットをキーボード or ゲームパッドで動かして逃げ回ろう

※ 縦横の入力（アナログスティックが Best）からターゲットの速度（移動）ベクトルを求めよう

※ 斜め入力のときの速さが、一方向入力の場合より速くならないように!!
（どの方向の入力も速さは同じにすること）

※ カメラをターゲットの Child に設定し、カメラの視線は追いかけてくる Cube への方向

※ Cube の2つとも視界に入るようにできれば Best

【任意のフレームレートへの対応】

★ここまでの問題点★

1フレームあたりの速さで制御すると、フレームレートが違うデバイスごとに、見た目の速さ（たとえば 1 秒あたりに進む距離）に違いが生じます。

どういうことか、少し具体的に

60FPS と 120FPS で比較する場合を考えてみます。FPS は「Frame Per Second（フレーム パー セカント）」の略で、1秒間に画面を何回更新するか、という単位です。60FPS の場合、1秒間に画面を 60 回更新する（60 枚の絵を表示する）、ということになります。

したがって、60FPS の更新間隔は 1 秒 ÷ 60 回 = 0.01666… [sec] ※sec: second: 秒

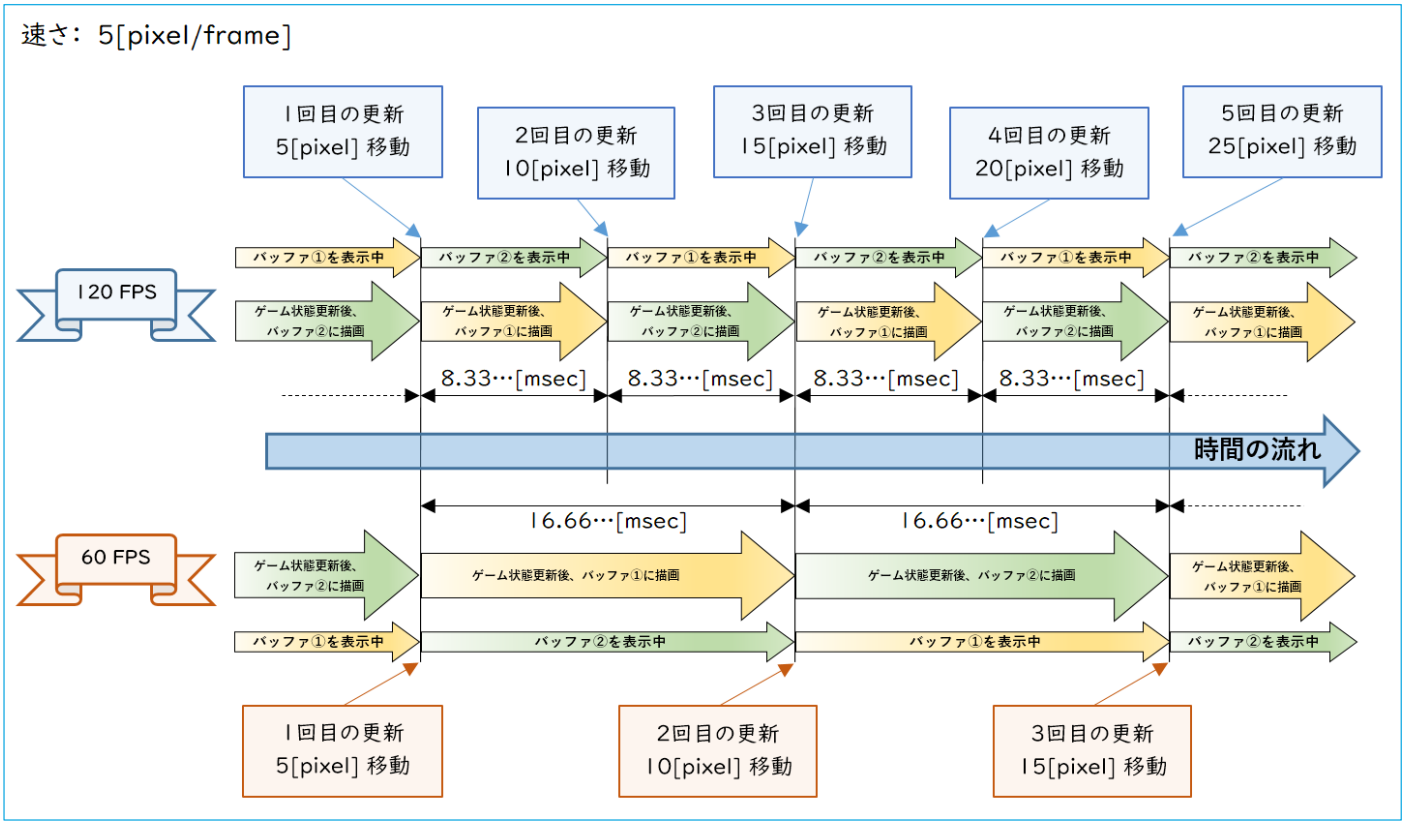
おおよそ 16.7[msec]（16.7 ミリ秒）となります。

つぎに 120FPS の場合では、1 秒 ÷ 120 回 = 0.00833… [sec]、おおよそ 8.33[msec] となります。

ゲームでは画面更新タイミングをきっかけに、つぎの画面更新タイミングで表示するためのゲームの状態を更新しますので、図にすると以下のようになります。

「 ベクトル tips 集 」

岡山情報 IT クリエイター専門学校



このように、1フレームあたりの速さでオブジェクトを移動させると、フレームレートが違うデバイスごとに、見た目の速さ（たとえば 1 秒間で進む距離）に違いが生じることがわかりました。これではゲームをプレイする人から苦情がきますので、どんなフレームレートでも見た目の速さに違いが生じない制御にすることが望ましいですね。

では、どうしたらよいでしょうか？

「 ベクトル tips 集 」

岡山情報 IT クリエイター専門学校

■デルタタイム（delta time / dt / 微小時間）の導入

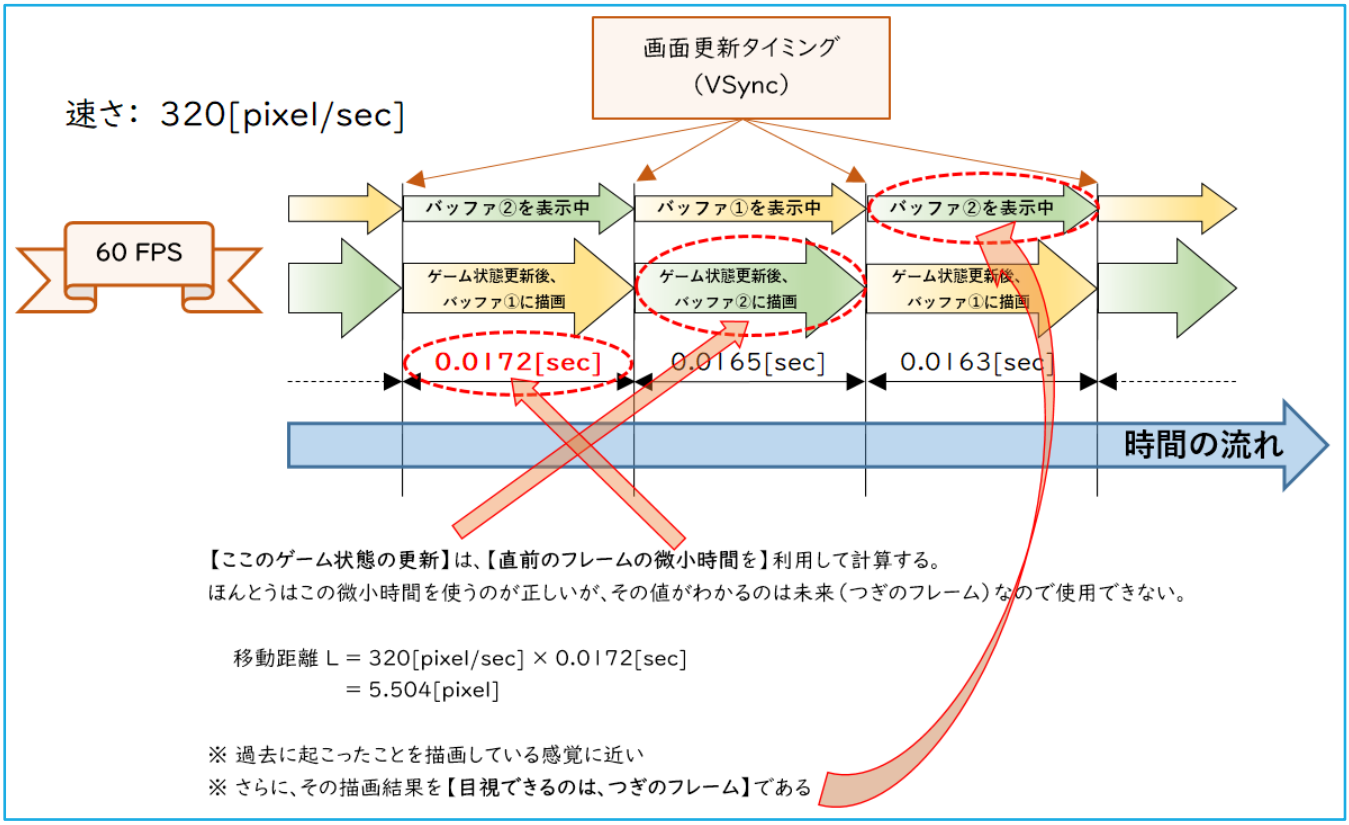
結論からいいます。速さを「秒速」や「時速」など、我々が見た目の動きを想像しやすいものにして計算します。秒速 320 ピクセル（1920×1080 の画面を 6 秒間で横断するイメージ）を例として考えてみます。

前回の更新タイミングから今回の更新タイミングまでの微小時間を deltaTime とします。この微小時間は、画面更新タイミングの時刻を記憶しておき、前回時刻からの経過時間を毎フレーム計算して求めます（じつは毎回おなじ間隔にはなりません）。

60FPS の場合は、deltaTime = 0.0166…[sec]が理論値ですが、今回の deltaTime は 0.0172[sec]だったと仮定しましょう。その場合の移動距離はつぎのように計算します。

$$\text{移動距離 } L = 320[\text{pixel/sec}] \times 0.0172[\text{sec}]$$

$$= 5.504[\text{pixel}]$$



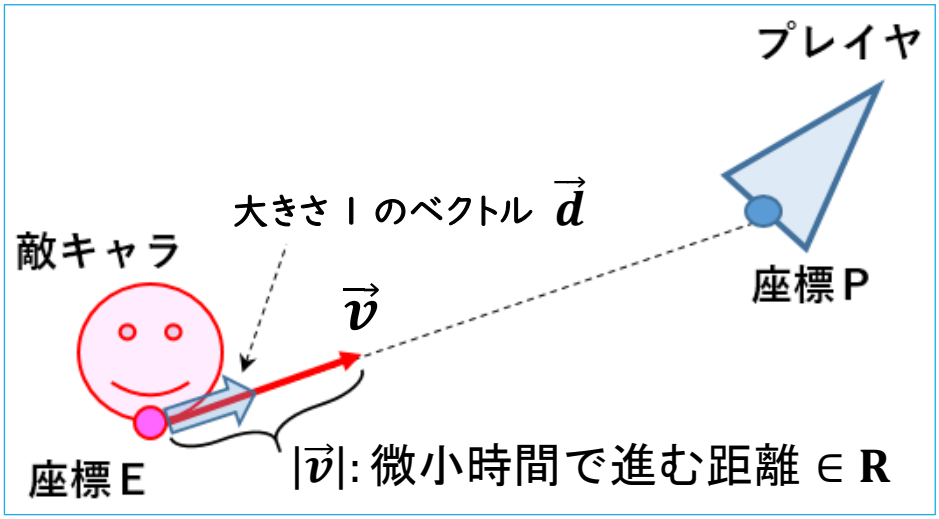
計算結果は、このように実数になりますので、オブジェクトの位置を保存しておく変数も実数型にしておく必要があります。また、2D ゲームの描画時のフレームバッファは整数座標なので、小数点以下は四捨五入（0.5 を足して整数型にキャスト）したり、切り捨てしたりして使用します。

「 ベクトル tips 集 」

岡山情報 IT クリエイター専門学校

【速度ベクトルではどうなる!?】

【任意のフレームレートへの対応】では、「速さ」だけに着目しました。さて、2D ゲームを例にあげた下図を思い出してください。実際のゲームでは、オブジェクトは 2 次元方向、あるいは 3 次元方向に移動するので、ベクトルを使って移動後の座標を計算します。



敵キャラ座標 E の位置ベクトル $\overrightarrow{en}$ プレイヤ座標 P の位置ベクトル $\vec{p}$ とする。
いま、敵キャラが $speed[m/s] \in \mathbf{R}$（秒速 $speed$ メートル）の速さでプレイヤに向かって移動するとき、移動後の敵キャラ座標 E'の位置ベクトル $\overrightarrow{en}'$ はつぎのように計算できます。ただし、任意のフレームレートに対応した計算をおこない、フレーム間の微小時間 $deltaTime$[sec] は既知とする。

Step1. $\vec{d} = Normalize(\ \vec{p} - \overrightarrow{en}\ )$

Step2. $\vec{v} =$ $speed$ × $deltaTime$ × $\vec{d}$

Step3. $\overrightarrow{en}' =$ $\overrightarrow{en}$ + $\vec{v}$

【Unity で試してみよう ②】

【Unity で試してみよう ①】で作成したスクリプトを deletaTime を使って任意のフレームレートに対応せよ。
【技術調査】Unity における deltaTime の取得方法

■斜め入力のときの速さが一方向入力の場合より速くなってしまうあなたへ

以下のような計算をしていませんか？

（例） 2D シューティングゲームで考えてみる。

・ 自機の位置 Vector2 pos;
・ 自機の速さは float speed[pixel/frame];
※簡単のためにフレームあたりの速さで考える
・ キーボードの WASD キーで自機を動かす
… W: 画面上方向（Y 軸プラス方向とする）
… A: 画面左方向（X 軸マイナス方向とする）
… S: 画面下方向（Y 軸マイナス方向とする）
… D: 画面右方向（X 軸プラス方向とする）

パターン① W のみ押された

```
pos.y += speed;
```

たしかに Y 方向に speed 分だけ移動する … 問題なし

パターン② D のみ押された

```
pos.x += speed;
```

たしかに X 方向に speed 分だけ移動する … 問題なし

パターン③ W と D の同時押し

```
pos.x += speed;
pos.y += speed;
```

プログラム的には↓こうしちゃってる。

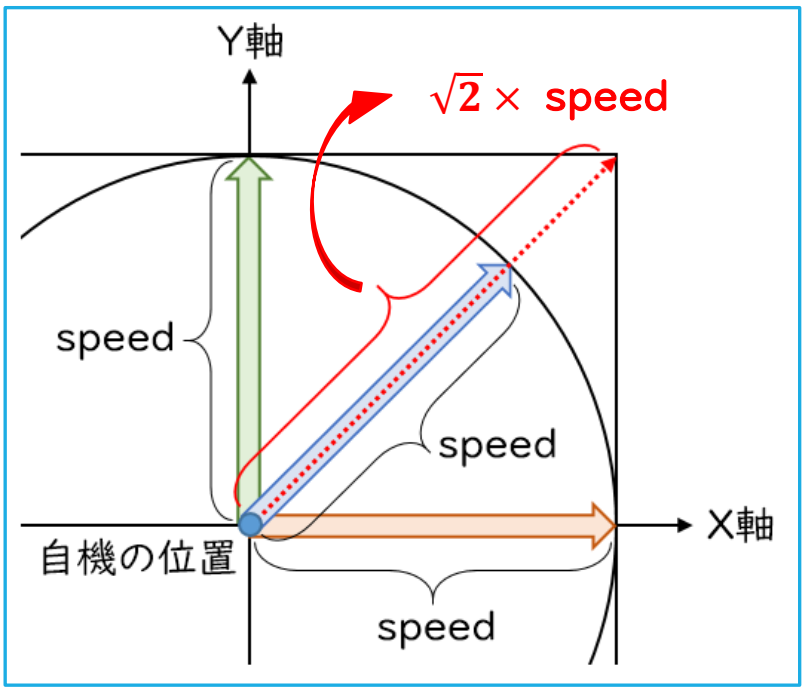
```
{
    if( key == 'W' ) pos.y += speed;
    if( key == 'A' ) pos.x -= speed;
    if( key == 'S' ) pos.y -= speed;
    if( key == 'D' ) pos.x += speed;
}
```

両成分にそのまま speed を足しちゃダメ！

以下のような状態になっている。

「 ベクトル tips 集 」

岡山情報 IT クリエイター専門学校



三平方の定理から **speed** が$\sqrt{2}$倍された大きさになってしまっている。
斜め方向の速さも **speed** になる方法はつぎのとおり（すでに道具はそろっている）。

1. まずは方向ベクトル $\vec{D}$ を得る

   → キー入力に応じて、2 次元ベクトルのそれぞれの成分に −1, 0, +1 の値を設定する
   ※アナログスティック入力の場合は、−1.0 ～ 1.0 の実数になり、
   垂直方向入力値、水平方向入力値をそのまま使う

2. 1.で作った方向ベクトルを正規化し、ベクトル $\vec{d}$ とする

   ※アナログスティック入力の場合、すでに正規化された状態の場合がある

3. 求める速度ベクトル $\vec{v}$ は $\vec{v} = \boldsymbol{speed} \times \vec{d}$

4. 移動後の位置 $\vec{p}' = \vec{p} + \vec{v}$