Unityはじめるよ

～3Dフィールド上の敵のAIを考えてみる～

統合開発環境を内蔵したゲームエンジン
http://japan.unity3d.com/

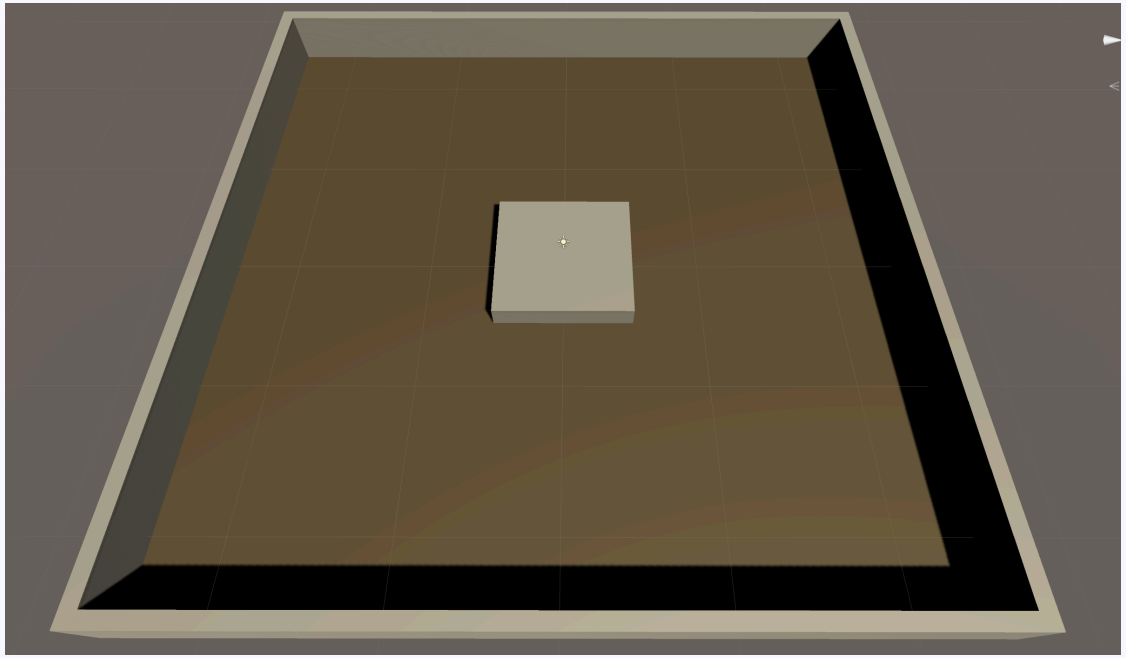
※いろんな職業の方が見る資料なので説明を簡単にしてある部分があります。正確には本来の意味と違いますが上記理由のためです。ご了承ください。
この資料内の一部の画像、一部の文章はUnity公式サイトから引用しています。

今回は敵のAIについて考えてみた。

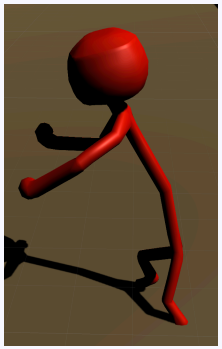
AIというと大げさで、
・敵の動きをどうやって管理しようかな
・こんな方法はどうだろ？
という感じの内容。

学生に相談されたのがキッカケ。

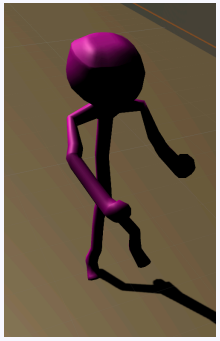
サンプルプロジェクトの画面



ステージ



プレイヤー



敵

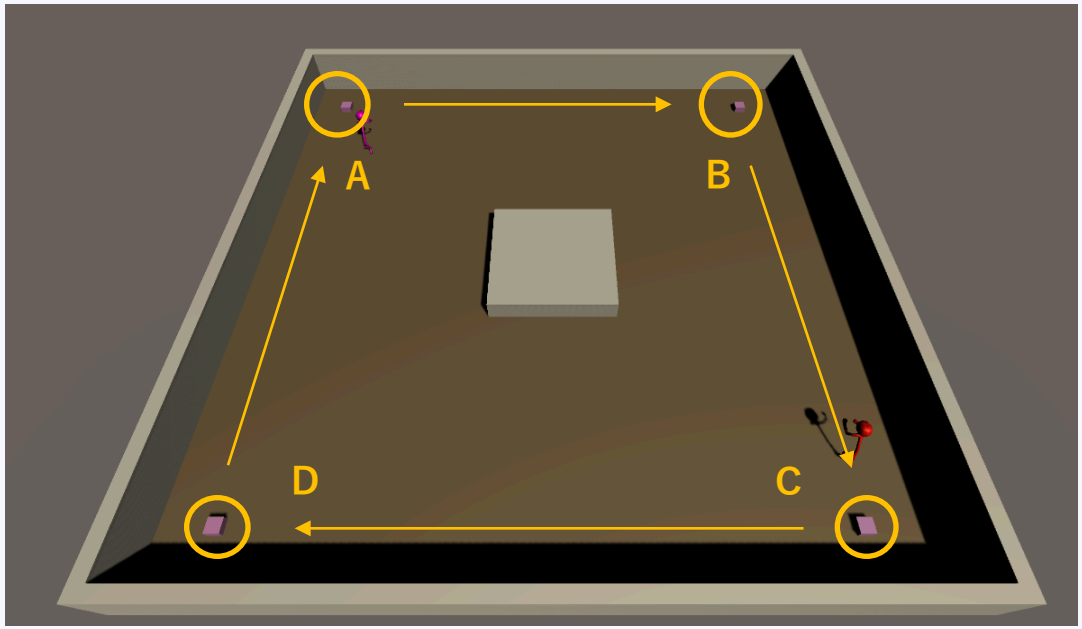
敵にどんな行動をさせたいのか

まずは、
敵にどんな行動をさせたいのかを決める。

今回は、
・ルート巡回
・追跡
・攻撃
の３つ。

ルート巡回

プレイヤーを探し回るフェーズ。
指定した地点(オレンジの丸の位置)を巡回させる。



ルート巡回

NavMeshを使えば簡単に実現できる。

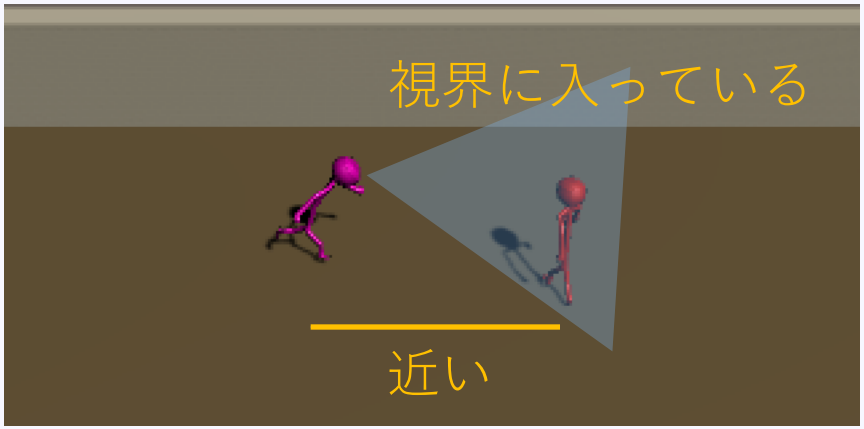
A地点に近づいたら目的地をB地点に、
B地点に近づいたら目的地をC地点に・・・

と繰り返していくだけ。
ルートの追加も容易。

Vector3.Distance関数を使えば、
自分の位置と目的地の距離を求められる

ルート巡回

プレイヤーが近くにいて、かつ、視界に入ったら
追跡フェーズに移行する。



Vector3.Distance関数でプレイヤーとの距離を調べ、
Vector3.Angle関数で視界に入っているか調べる。

追跡

プレイヤーを追いかけるフェーズ。
プレイヤーとの距離が縮まれば**攻撃フェーズ**へ、
離れれば**ルート巡回**フェーズに。

追跡

これもNavMeshを使えば簡単に実現できる。

NavMeshAgentの目的地をプレイヤーにするだけ。

ルート巡回の時と同じように、
Vector3.Distance関数でプレイヤーとの距離を調べ、
攻撃に行くか、ルート巡回に行くか判断する。

攻撃

プレイヤーを攻撃するフェーズ。
プレイヤーの方を向き攻撃を繰り出す。
距離が離れたら追跡フェーズへ。

攻撃

プレイヤーに向かって攻撃をさせるために、
Transform.LookAt 関数を使ってプレイヤーの方に向かせるのがキモ。

サンプルプログラムでは、
いきなりプレイヤーの方を向かせているけど、
自然に見せるにはlerp関数などを使って
滑らかに回転させた方がよい。

全体の管理方法

ルート巡回、追跡、攻撃をどうやって管理するか。

今回はシンプルに
- enumでフェーズを定義
- switch文で分岐させる
- 各フェーズ一つの関数にまとめる
- フェーズの切り替え処理も関数化

とした。

全体の管理方法

各フェーズでは、
　・**フェーズを抜ける理由**
だけを明確にしておく。
次にどのフェーズに移行するかを判断するのは、
別関数で行う。
そうすることで、
AIのアルゴリズム変更も容易となる。

フェーズを抜けた理由を保持する変数を用意しておく

```
/// <summary>
/// 行動を変更した理由.
/// </summary>
enum ReasonForChange
{
    Near,         // 近い
    Far,         // 遠い
    Damage,         // ダメージを受けた
}
ReasonForChange    m_reasonForChange = ReasonForChange.Far;
```

enumでフェーズを定義

```
/// <summary>
/// 敵の行動の種類.
/// </summary>
public enum RoutineType
{
    Patrol,         // パトロール
    Chase,          // 追跡中
    Attack,         // 攻撃
}
RoutineType    m_routine = RoutineType.Patrol;
```

switch文で分岐させる

```
switch (m_routine) {
case RoutineType.Patrol:    // 巡回
    {
        Patrol ();
    }
    break;

case RoutineType.Chase: // 追跡
    {
        Chase ();
    }
    break;

case RoutineType.Attack: // 攻撃
    {
        Attack ();
    }
    break;
}
```

全体の管理方法

各フェーズ一つの関数にまとめる

関数にまとめることで呼び出し側がすっきりして見やすいコードになる

```
/// <summary>
/// 攻撃処理
/// </summary>
void Attack()
{
    // プレイヤーとの距離
    float distance = Vector3.Distance (transform.position, Player.position);

    // プレイヤーとの距離が離れたなら
    if (distance > AttackRange)
    {
        // パトロールに戻す
        m_reasonForChange = ReasonForChange.Far;

        // 次のRoutineを決める
        m_routine = ChangeRoutine ();
    }
    // アニメーションが終了したら もう一度攻撃
    else if ((m_animator.GetCurrentAnimatorStateInfo (0).normalizedTime > 1.0) &&
            (m_animator.GetCurrentAnimatorStateInfo (0).IsName ("Attack") == true))
    {
        // アニメーションを攻撃に変更
        m_animator.SetTrigger (kAnimatorPramAttack);

        // プレイヤーの方を向く
        transform.LookAt (Player);

    }
}
```

全体の管理方法

フェーズの切り替え処理も関数化

ここがAIのキモとなる部分。ここで次にどのフェーズに進めるかを決める。

```
/// <summary>
/// 敵の行動を変更.
/// </summary>
/// <returns>The routine.</returns>
public RoutineType ChangeRoutine()
{
    RoutineType ret = m_routine;

    switch (m_routine) {
    case RoutineType.Patrol:    // 巡回中に
        {
            // 敵を発見したので追跡に
            if (m_reasonForChange == ReasonForChange.Near) {
                ret = RoutineType.Chase;
            }
        }
        break;

    case RoutineType.Chase:         // 追跡中に
        {
            // 距離が離れたので巡回へ
            if (m_reasonForChange == ReasonForChange.Far) {
                ret = RoutineType.Patrol;
            }
            // 距離が近づいたの攻撃
            else if (m_reasonForChange == ReasonForChange.Near) {
                ret = RoutineType.Attack;
            }
        }
        break;

    case RoutineType.Attack:    // 攻撃中に
        {
            // 距離が離れたので追跡へ
            if (m_reasonForChange == ReasonForChange.Far) {
                ret = RoutineType.Chase;
            }
        }
        break;

    }


    // 次の行動を開始
    StartRoutine (ret);

    return ret;
}
```

まとめ

ゲーム制作の勉強を始めて半年の学生でも
わかるような作りとしました。
今回はベース部分だけなので、HPや時間、ランダム要素などの条件を付け足していくことで、より自然なAIとなるでしょう。

ただこの方法だと、行動パターンが増えてくると、一つのスクリプトが膨れ上がってしまう問題もあります。なので行動パターンごとスクリプトを分けるのもありです。

ご清聴ありがとうございました