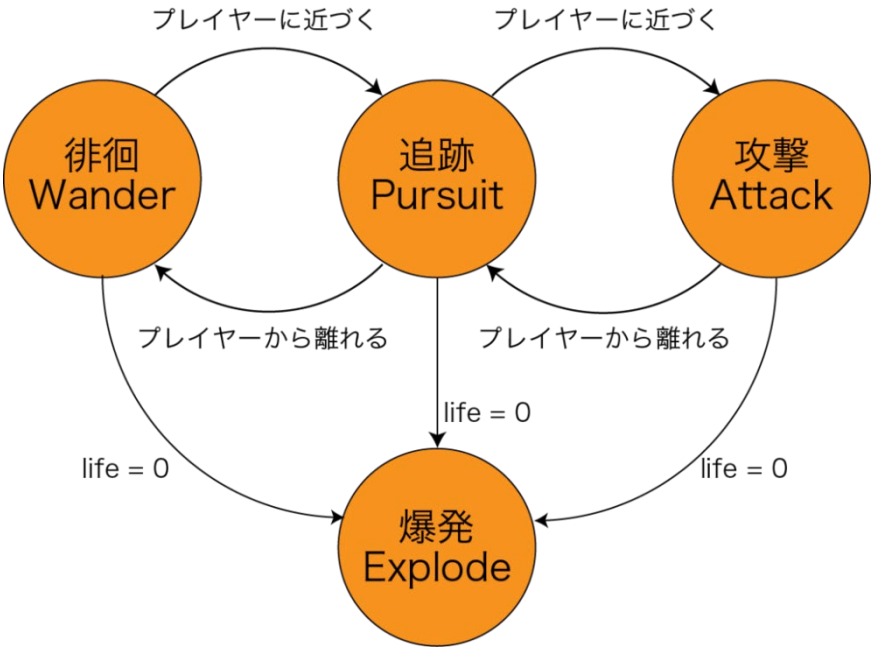
ゲームプログラミングⅢ

○評価要件

☑敵の行動処理

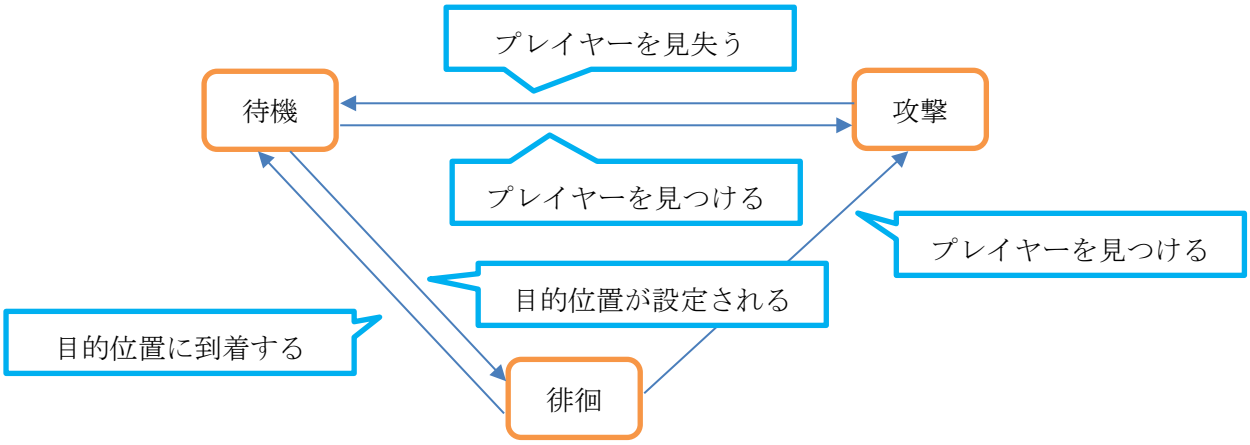
○概要

今回は敵の行動制御を実装します。
今まではプレイヤーの操作を重点的に実装してきましたが、今回は敵の行動を実装していきます。
プレイヤーと違い、自分で操作するのではなく、敵が状況に応じて行動を判断する AI を実装しましょう。



○敵の行動処理

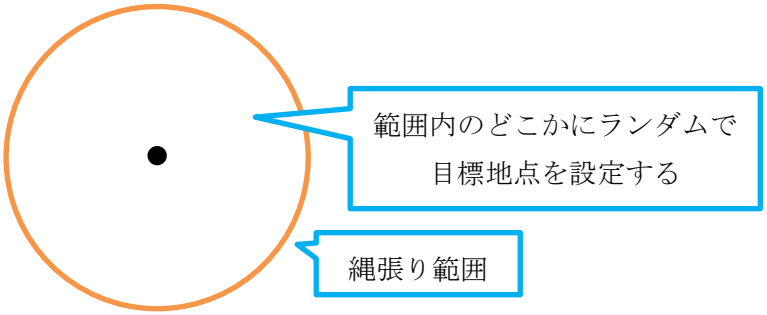
敵はステートマシンで行動を実装していきます。
ステートマシンとは各ステートに設定されている条件を満たすことで別の状態へ遷移するシステムのことです。
簡単な AI としてスライムを下図のような行動をするようにしてみましょう。



○徘徊ステート

まずは徘徊ステートから実装しましょう。
簡易的な敵の行動でよくあるのですが、目的もなく自分の縄張り範囲内を徘徊している状態です。

実装処理としては縄張り範囲内にランダムで目標地点を設定し、目標地点まで移動して到達すると新しい目標地点を設定して移動を繰り返します。



MathUtils.h と MathUtils.cpp を作成し、目標地点計算のため、指定範囲のランダム値を計算する関数を作成します。

MathUtils.h

```
#pragma once

// 浮動小数算術
class MathUtils
{
public:
```

```
    // 指定範囲のランダム値を計算する
    static float RandomRange(float min, float max);
};
```

MathUtils.cpp

```
#include <stdlib.h>
#include "MathUtils.h"

// 指定範囲のランダム値を計算する
float MathUtils::RandomRange(float min, float max)
{
    // 0.0～1.0の間までのランダム値
    float value = static_cast<float>(rand()) / RAND_MAX;

    // min～maxまでのランダム値に変換
    return min + (max - min) * value;
}
```

次は縄張り範囲と目標地点をデバッグ表示し、徘徊処理を実装しましょう。

EnemySlime.h

```
---省略---

// スライム
class EnemySlime : public Enemy
{
public:
    ---省略---

    // デバッグプリミティブ描画
    void RenderDebugPrimitive(const RenderContext& rc, ShapeRenderer* renderer) override;

    // 縄張り設定
    void SetTerritory(const DirectX::XMFLOAT3& origin, float range);

private:
    // ターゲット位置をランダム設定
    void SetRandomTargetPosition();

    // 目標地点へ移動
    void MoveToTarget(float elapsedTime , float moveSpeedRate, float turnSpeedRate);

    // 徘徊ステートへ遷移
    void SetWanderState();

    // 徘徊ステート更新処理
    void UpdateWanderState(float elapsedTime);


private:
    // ステート
```

```
    enum class State
    {
        Wander
    };

private:
    ---省略---
    State                   state = State::Wander;
    DirectX::XMFLOAT3       targetPosition = { 0, 0, 0 };
    DirectX::XMFLOAT3       territoryOrigin = { 0, 0, 0 };
    float                   territoryRange = 10.0f;
    float                   moveSpeed = 2.0f;
    float                   turnSpeed = DirectX::XMConvertToRadians(360);
};
```

EnemySlime.cpp

```
---省略---
#include "MathUtils.h"

// コンストラクタ
EnemySlime::EnemySlime()
{
    ---省略---

    // 徘徊ステートへ遷移
    SetWanderState();
}

---省略---

// 更新処理
void EnemySlime::Update(float elapsedTime)
{
    // ステート毎の更新処理
    switch (state)
    {
    case State::Wander:
        UpdateWanderState(elapsedTime);
        break;
    }

    ---省略---
}

---省略---

// デバッグプリミティブ描画
void EnemySlime::RenderDebugPrimitive(const RenderContext& rc, ShapeRenderer* renderer)
{
    // 基底クラスのデバッグプリミティブ描画
    Enemy::RenderDebugPrimitive(rc, renderer);

    // 縄張り範囲をデバッグ円柱描画
    renderer->RenderCylinder(rc, territoryOrigin, territoryRange, 1.0f,
```

```
                                                    DirectX::XMFLOAT4(0, 1, 0, 1));

    // ターゲット位置をデバッグ球描画
    renderer->RenderSphere(rc, targetPosition, 1.0f, DirectX::XMFLOAT4(1, 1, 0, 1));
}

// 縄張り設定
void EnemySlime::SetTerritory(const DirectX::XMFLOAT3& origin, float range)
{
    territoryOrigin = origin;
    territoryRange = range;
}

// ターゲット位置をランダム設定
void EnemySlime::SetRandomTargetPosition()
{
    float theta = MathUtils::RandomRange(-DirectX::XM_PI, DirectX::XM_PI);
    float range = MathUtils::RandomRange(0.0f, territoryRange);
    targetPosition.x = territoryOrigin.x + sinf(theta) * range;
    targetPosition.y = territoryOrigin.y;
    targetPosition.z = territoryOrigin.z + cosf(theta) * range;
}

// 目標地点へ移動
void EnemySlime::MoveToTarget(float elapsedTime, float moveSpeedRate, float turnSpeedRate)
{
    // ターゲット方向への進行ベクトルを算出
    float vx = targetPosition.x - position.x;
    float vz = targetPosition.z - position.z;
    float dist = sqrtf(vx * vx + vz * vz);
    vx /= dist;
    vz /= dist;

    // 移動処理
    Move(elapsedTime, vx, vz, moveSpeed * moveSpeedRate);
    Turn(elapsedTime, vx, vz, turnSpeed * turnSpeedRate);
}

// 徘徊ステートへ遷移
void EnemySlime::SetWanderState()
{
    state = State::Wander;

    // 目標地点設定
    SetRandomTargetPosition();
}

// 徘徊ステート更新処理
void EnemySlime::UpdateWanderState(float elapsedTime)
{
    // 目標地点までXZ平面での距離判定
    float vx = targetPosition.x - position.x;
    float vz = targetPosition.z - position.z;
    float distSq = vx * vx + vz * vz;
    if (distSq < radius * radius)
    {
        // 次の目標地点設定
```

```
        SetRandomTargetPosition();
    }

    // 目標地点へ移動
    MoveToTarget(elapsedTime, 1.0f, 1.0f);
}
```

実装出来たら敵配置時に縄張り設定をするようにしましょう。

SceneGame.cpp

```
---省略---

// 初期化
void SceneGame::Initialize()
{
    ---省略---

    // エネミー初期化
    EnemyManager& enemyManager = EnemyManager::Instance();
    for (int i = 0; i < 2; ++i)
    {
        EnemySlime* slime = new EnemySlime();
        slime->SetPosition(DirectX::XMFLOAT3(i * 2.0f, 0, 5));
        slime->SetTerritory(slime->GetPosition(), 10.0f);
        enemyManager.Register(slime);
    }
    ---省略---
}
```

実装出来たら実行確認をしましょう。
スライムが目標地点に向かって移動し、目標地点に到達したら新しい目標地点に向かって移動をはじめれば OK です。

○待機ステート

徘徊ステートを実装しましたが、目標地点へ到達するとすぐに新しい目標地点に向かって移動をはじめるため、動きが忙しい感じになっています。
少し落ち着かせるために待機ステートを挟むことにしましょう。

待機ステートは特に移動することもなく、数秒間その場で待機してから徘徊ステートへ遷移するようにしましょう。

EnemySlime.h

```
---省略---

// スライム
```

```
class EnemySlime : public Enemy
{
public:
    ---省略---

private:
    ---省略---

    // 待機ステートへ遷移
    void SetIdleState();

    // 待機ステート更新処理
    void UpdateIdleState(float elapsedTime);

private:
    // ステート
    enum class State
    {
        ---省略---
        Idle
    };
    ---省略---

private:
    ---省略---
    float                   stateTimer = 0.0f;
};
```

EnemySlime.cpp

```
---省略---

// 更新処理
void EnemySlime::Update(float elapsedTime)
{
    // ステート毎の更新処理
    switch (state)
    {
    ---省略---

    case State::Idle:
        UpdateIdleState(elapsedTime);
        break;
    }

    ---省略---
}

---省略---

// 徘徊ステート更新処理
void EnemySlime::UpdateWanderState(float elapsedTime)
{
    // 目標地点までXZ平面での距離判定
    ---省略---
```

```
    if (distSq < radius * radius)
    {
        // 次の目標地点設定
        SetRandomTargetPosition();
        // 待機ステートへ遷移
        SetIdleState();
    }

    ---省略---
}

// 待機ステートへ遷移
void EnemySlime::SetIdleState()
{
    state = State::Idle;

    // タイマーをランダム設定
    stateTimer = MathUtils::RandomRange(3.0f, 5.0f);
}

// 待機ステート更新処理
void EnemySlime::UpdateIdleState(float elapsedTime)
{
    // タイマー処理
    stateTimer -= elapsedTime;
    if (stateTimer < 0.0f)
    {
        // 徘徊ステートへ遷移
        SetWanderState();
    }
}
```

実装出来たら実行確認をしてみましょう。
徘徊ステートで目標地点まで移動したら待機し、数秒間経ったら徘徊をはじめれば OK です。

○プレイヤーの情報取得

スライムがプレイヤーに対して攻撃を行うプログラムを組むために、プレイヤーの情報を取得できるようにしましょう。
一番簡単な方法として、現状のプレイヤークラスを改造してシングルトンでプレイヤーのインスタンスを取得できるようにします。

Player.h

```
---省略---

// プレイヤー
class Player : public Character
{
public:
private:
    Player();
```

コンストラクタとデストラクタを
プライベート化し、中身を空にする

```
    Player() {};
    ~Player() override;
    ~Player() override {};

public:
    // インスタンス取得
    static Player& Instance()
    {
        static Player instance;
        return instance;
    }

    // 初期化
    void Initialize();

    // 終了化
    void Finalize();

    ---省略---
};
```

コンストラクタとデストラクタの実装内容を Initialize と Finalize に置き換える

Player.cpp

```
---省略---

// コンストラクタ
// 初期化
Player::Player()
void Player::Initialize()
{
    ---省略---
}

// デストラクタ
// 終了化
Player::~Player()
void Player::Finalize()
{
    ---省略---
}
```

SceneGame のプレイヤーのアクセスをシングルトンに置き換えましょう。

SceneGame.h

```
---省略---
#include "Player.h"

// ゲームシーン
class SceneGame : public Scene
{
    ---省略---
```

```
private:
    ---省略---
    Player* player = nullptr;
};
```

Scene.cpp

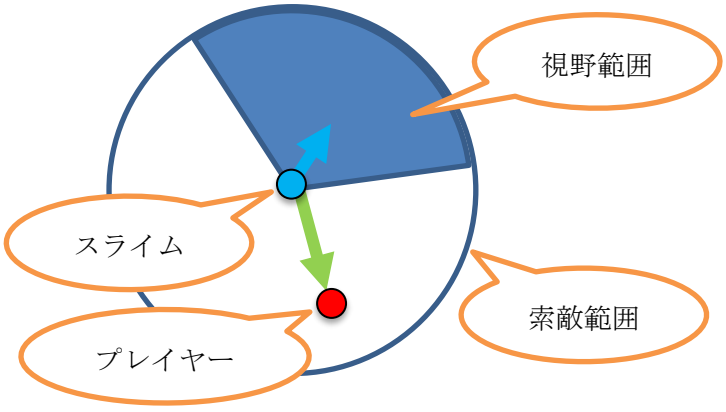
```
---省略---
#include "Player.h"

// 初期化
void SceneGame::Initialize()
{
    ---省略---

    // プレイヤー初期化
    player = new Player();
    Player::Instance().Initialize();

    ---省略---
}

// 終了化
void SceneGame::Finalize()
{
    ---省略---

    // プレイヤー終了化
    if (player != nullptr)
    {
        delete player;
        player = nullptr;
    }
    Player::Instance().Finalize();

    ---省略---
}

// 更新処理
void SceneGame::Update(float elapsedTime)
{
    // カメラコントローラー更新処理
    DirectX::XMFLOAT3 target = player->GetPosition();
    DirectX::XMFLOAT3 target = Player::Instance().GetPosition();

    ---省略---

    // プレイヤー更新処理
    player->Update(elapsedTime);
    Player::Instance().Update(elapsedTime);

    ---省略---
}

// 描画処理
```

```
void SceneGame::Render()
{
    ---省略---

    // 3Dモデル描画
    {
        ---省略---

        // プレイヤー描画
        player->Render(rc, modelRenderer);
        Player::Instance().Render(rc, modelRenderer);

        ---省略---
    }

    // 3Dデバッグ描画
    {
        // プレイヤーデバッグプリミティブ描画
        player->RenderDebugPrimitive(rc, shapeRenderer);
        Player::Instance().RenderDebugPrimitive(rc, shapeRenderer);

        ---省略---
    }
    ---省略---
}

// GUI描画
void SceneGame::DrawGUI()
{
    // プレイヤーデバッグ描画
    player->DrawDebugGUI();
    Player::Instance().DrawDebugGUI();
}
```

プレイヤーの情報を取得できるようになったので攻撃ステートの実装を行います

○攻撃ステート

次はプレイヤーを発見したら攻撃するステートを実装しましょう。
プレイヤーの索敵には範囲と視野判定を行います。



- プレイヤーが索敵範囲の内側にいる
  （距離で判定）
- プレイヤーが視野範囲内にいる
  （内積で判定）

EnemySlime.h

```
---省略---
#include "ProjectileManager.h"

// スライム
class EnemySlime : public Enemy
{
public:
    ---省略---

private:
    ---省略---

    // プレイヤー索敵
    bool SearchPlayer();

    // 攻撃ステートへ遷移
    void SetAttackState();

    // 攻撃ステート更新処理
    void UpdateAttackState(float elapsedTime);

private:
    // ステート
    enum class State
    {
        ---省略---
        Attack
    };

    ---省略---

private:
    ---省略---
    float                   searchRange = 5.0f;
    ProjectileManager       projectileManager;
};
```

EnemySlime.cpp

```
---省略---
#include "Player.h"
#include "ProjectileStraight.h"

---省略---

// 更新処理
void EnemySlime::Update(float elapsedTime)
{
    // ステート毎の更新処理
    switch (state)
    {
    ---省略---
```

```
    case State::Attack:
        UpdateAttackState(elapsedTime);
        break;
    }

    ---省略---

    // 速力処理更新
    ---省略---

    // 弾丸更新処理
    projectileManager.Update(elapsedTime);

    ---省略---
}

// 描画処理
void EnemySlime::Render(const RenderContext& rc, ModelRenderer* renderer)
{
    ---省略---

    // 弾丸描画処理
    projectileManager.Render(rc, renderer);
}

---省略---

// デバッグプリミティブ描画
void EnemySlime::RenderDebugPrimitive(const RenderContext& rc, ShapeRenderer* renderer)
{
    ---省略---

    // 索敵範囲をデバッグ円柱描画
    renderer->RenderCylinder(rc, position, searchRange, 1.0f, DirectX::XMFLOAT4(1, 0, 0, 1));
}

---省略---

// プレイヤー索敵
bool EnemySlime::SearchPlayer()
{
    // プレイヤーとの高低差を考慮して3Dでの距離判定をする
    const DirectX::XMFLOAT3& playerPosition = Player::Instance().GetPosition();
    float vx = playerPosition.x - position.x;
    float vy = playerPosition.y - position.y;
    float vz = playerPosition.z - position.z;
    float dist = sqrtf(vx * vx + vy * vy + vz * vz);
    if (dist < searchRange)
    {
        float distXZ = sqrtf(vx * vx + vz * vz);
        // 単位ベクトル化
        vx /= distXZ;
        vz /= distXZ;
        // 前方ベクトル
        float frontX = sinf(angle.y);
        float frontZ = cosf(angle.y);
        // ２つのベクトルの内積値で前後判定
```

```
        float dot = (frontX * vx) + (frontZ * vz);
        if (dot > 0.0f)
        {
            return true;
        }
    }
    return false;
}

// 徘徊ステート更新処理
void EnemySlime::UpdateWanderState(float elapsedTime)
{
    ---省略---

    // プレイヤー索敵
    if (SearchPlayer())
    {
        // 見つかったら攻撃ステートへ遷移
        SetAttackState();
    }
}

---省略---

// 待機ステート更新処理
void EnemySlime::UpdateIdleState(float elapsedTime)
{
    ---省略---

    // プレイヤー索敵
    if (SearchPlayer())
    {
        // 見つかったら攻撃ステートへ遷移
        SetAttackState();
    }
}

// 攻撃ステートへ遷移
void EnemySlime::SetAttackState()
{
    state = State::Attack;

    stateTimer = 0.0f;
}

// 追跡ステート更新処理
void EnemySlime::UpdateAttackState(float elapsedTime)
{
    // 目標地点をプレイヤー位置に設定
    targetPosition = Player::Instance().GetPosition();

    // 目標地点へ移動
    MoveToTarget(elapsedTime, 0.0f, 1.0f);

    // タイマー処理
    stateTimer -= elapsedTime;
    if (stateTimer < 0.0f)
```

```
    {
        // 前方向
        DirectX::XMFLOAT3 dir;
        dir.x = sinf(angle.y);
        dir.y = 0.0f;
        dir.z = cosf(angle.y);
        // 発射位置（プレイヤーの腰あたり）
        DirectX::XMFLOAT3 pos;
        pos.x = position.x;
        pos.y = position.y + height * 0.5f;
        pos.z = position.z;
        // 発射
        ProjectileStraight* projectile = new ProjectileStraight(&projectileManager);
        projectile->Launch(dir, pos);

        stateTimer = 2.0f;
    }

    // プレイヤーを見失ったら
    if (!SearchPlayer())
    {
        // 待機ステートへ遷移
        SetIdleState();
    }
}
```

実装できたら実行確認をしてみましょう。
プレイヤーが索敵範囲内に侵入するとプレイヤーを攻撃しはじめれば OK です。
攻撃が当たった時の判定やダメージ処理なども実装しておきましょう。

今回は「待機」、「徘徊」、「攻撃」の３種類のステートを状況に応じて切り替える簡単な処理を実装しました。
今後は必要に応じてステートを増やして新たな実装やステートを切り替える条件などを考えて実装していきましょう。

お疲れさまでした。