3D アクションゲームプログラミング

○評価要件

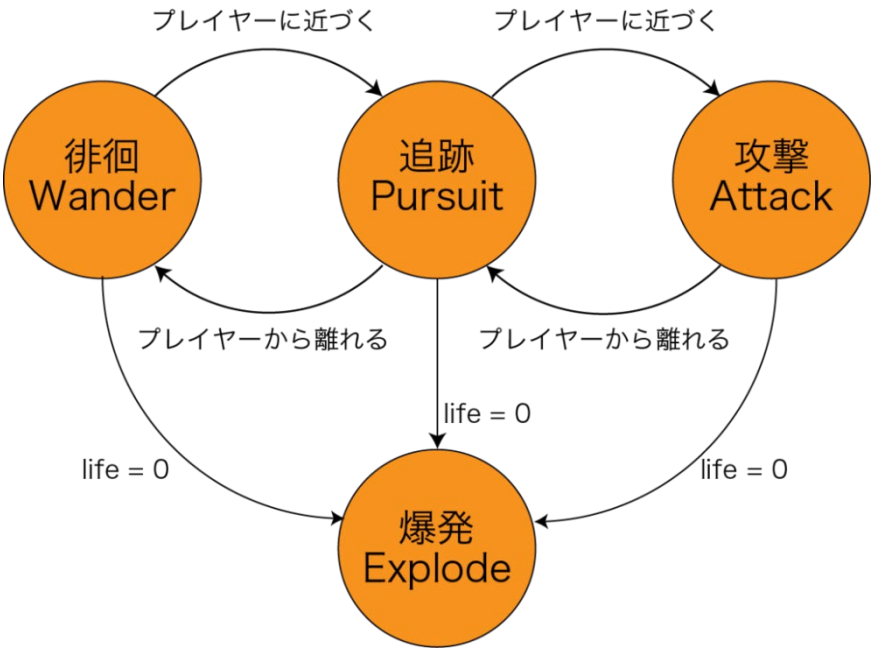
☑敵の行動処理
☑敵とプレイヤーのダメージ&死亡リアクション

○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。
プレイヤーと同じく、簡単なステートマシンによる制御を行いますが、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

敵の行動もプレイヤーと同じく、ステートマシンで行動を実装していきます。
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。

以前の課題で Mathf.h と Mathf.cpp を作成していない人はここで作成しましょう。
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。

Mathf.h

```
---省略---

// 浮動小数算術
class Mathf
{
public:
    ---省略---
    // 指定範囲のランダム値を計算する
    static float RandomRange(float min, float max);
};
```

Mathf.cpp

```
#include <stdlib.h>
---省略---

// 指定範囲のランダム値を計算する
float Mathf::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 DrawDebugPrimitive() 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 speedRate);

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

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


private:
    // ステート
    enum class State
    {
        Wander
    };

    // アニメーション
```

```
    enum Animation
    {
        Anim_IdleNormal,
        Anim_IdleBattle,
        Anim_Attack1,
        Anim_Attack2,
        Anim_WalkFWD,
        Anim_WalkBWD,
        Anim_WalkLeft,
        Anim_WalkRight,
        Anim_RunFWD,
        Anim_SenseSomthingST,
        Anim_SenseSomthingPRT,
        Anim_Taunt,
        Anim_Victory,
        Anim_GetHit,
        Anim_Dizzy,
        Anim_Die
    };

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 = 3.0f;
    float                   turnSpeed = DirectX::XMConvertToRadians(360);
};
```

EnemySlime.cpp

```
---省略---
#include "Graphics/Graphics.h"
#include "Mathf.h"

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

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

---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---

    // モデルアニメーション更新
    model->UpdateAnimation(elapsedTime);

    // モデル行列更新
    ---省略---
}

---省略---

// デバッグプリミティブ描画
void EnemySlime::DrawDebugPrimitive()
{
    // 基底クラスのデバッグプリミティブ描画
    Enemy::DrawDebugPrimitive();

    DebugRenderer* debugRenderer = Graphics::Instance().GetDebugRenderer();

    // 縄張り範囲をデバッグ円柱描画
    debugRenderer->DrawCylinder(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
    debugRenderer->DrawSphere(targetPosition, radius, 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 = Mathf::RandomRange(-DirectX::XM_PI, DirectX::XM_PI);
    float range = Mathf::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 speedRate)
{
    // 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(vx, vz, moveSpeed * speedRate);
    Turn(elapsedTime, vx, vz, turnSpeed * speedRate);
}

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

    // 目標地点設定
    SetRandomTargetPosition();

    // 歩きアニメーション再生
    model->PlayAnimation(Anim_WalkFWD, true);
}

// 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, 0.5f);
}
```

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 TransitionIdleState();

    // 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

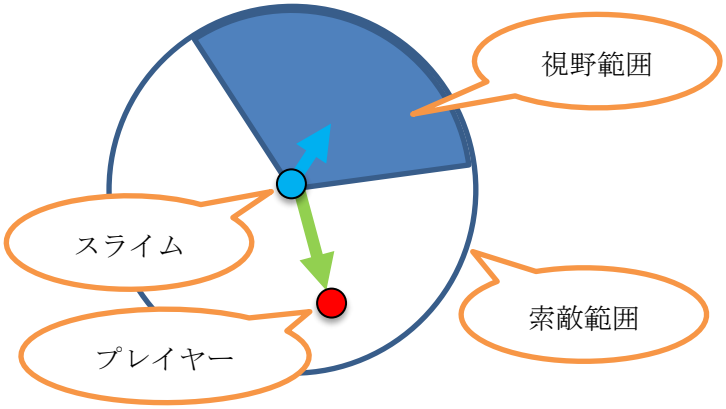
```
---省略---

// 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
        TransitionIdleState();
    }

    ---省略---
}

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

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

    // 待機アニメーション再生
    model->PlayAnimation(Anim_IdleNormal, true);
}

// 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
        TransitionWanderState();
    }
}
```

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。

○追跡ステート

次はプレイヤーを発見したら追跡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）

このステートを実装するにはプレイヤーの情報が必要ですが、現状ではプレイヤーの情報を取得する手段がありません。
今回はプレイヤークラスを拡張してシングルトン風の簡易的な実装をしてプレイヤー情報を取得するようにします。
きちんとプログラムを組む人はプレイヤーマネージャーなど自分で考えてみましょう。

Player.h

```
---省略---

// プレイヤー
class Player : public Character
{
public:
    ---省略---

    // インスタンス取得
    static Player& Instance();

    ---省略---
};
```

Player.cpp

```
---省略---

static Player* instance = nullptr;

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

// コンストラクタ
Player::Player()
{
    // インスタンスポインタ設定
    instance = this;

    ---省略---
}
```

プレイヤーの情報を取得できるようになったのでプレイヤーの索敵処理を実装して、追跡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---

private:
    ---省略---

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

    ---省略---

    // 追跡ステートへ遷移
    void TransitionPursuitState();

    // 追跡ステート更新処理
    void UpdatePursuitState(float elapsedTime);

private:
    // ステート
    enum class State
    {
        ---省略---
```

```
        Pursuit
    };

    ---省略---

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

EnemySlime.cpp

```
---省略---
#include "Player.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::Pursuit:
        UpdatePursuitState(elapsedTime);
        break;
    }

    ---省略---
}

---省略---

// デバッグプリミティブ描画
void EnemySlime::DrawDebugPrimitive()
{
    ---省略---

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

---省略---

// 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())
    {
        // 見つかったら追跡ステートへ遷移
        TransitionPursuitState();
    }
}

---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())
    {
        // 見つかったら追跡ステートへ遷移
        TransitionPursuitState();
    }
}

// 追跡ステートへ遷移
void EnemySlime::TransitionPursuitState()
{
    state = State::Pursuit;

    // 数秒間追跡するタイマーをランダム設定
    stateTimer = Mathf::RandomRange(3.0f, 5.0f);
```

```
    // 歩きアニメーション再生
    model->PlayAnimation(Anim_RunFWD, true);
}

// 追跡ステート更新処理
void EnemySlime::UpdatePursuitState(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, 1.0f);

    // タイマー処理
    stateTimer -= elapsedTime;
    if (stateTimer < 0.0f)
    {
        // 待機ステートへ遷移
        TransitionIdleState();
    }
}
```

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追跡しはじめれば OK です。

○攻撃ステート

プレイヤーを追跡することができるようになったので攻撃範囲まで近づくと攻撃するように実装しましょう。

EnemySlime.h

```
---省略---

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

private:
    ---省略---

    // ノードとプレイヤーの衝突処理
    void CollisionNodeVsPlayer(const char* nodeName, float boneRadius);

    ---省略---

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

    // 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                   attackRange = 1.5f;
};
```

EnemySlime.cpp

```
---省略---
#include "Collision.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---
}

---省略---

// デバッグプリミティブ描画
void EnemySlime::DrawDebugPrimitive()
{
    ---省略---

    // 攻撃範囲をデバッグ円柱描画
    debugRenderer->DrawCylinder(position, attackRange, 1.0f, DirectX::XMFLOAT4(1, 0, 0, 1));
}

---省略---

// ノードとプレイヤーの衝突処理
void EnemySlime::CollisionNodeVsPlayer(const char* nodeName, float nodeRadius)
{
```

```
    // ノードの位置と当たり判定を行う
    Model::Node* node = model->FindNode(nodeName);
    if (node != nullptr)
    {
        // ノードのワールド座標
        DirectX::XMFLOAT3 nodePosition(
            node->worldTransform._41,
            node->worldTransform._42,
            node->worldTransform._43
        );

        // 当たり判定表示
        Graphics::Instance().GetDebugRenderer()->DrawSphere(
            nodePosition, nodeRadius, DirectX::XMFLOAT4(0, 0, 1, 1)
        );

        // プレイヤーと当たり判定
        Player& player = Player::Instance();
        DirectX::XMFLOAT3 outPosition;
        if (Collision::IntersectSphereVsCylinder(
            nodePosition,
            nodeRadius,
            player.GetPosition(),
            player.GetRadius(),
            player.GetHeight(),
            outPosition))
        {
            // ダメージを与える
            if (player.ApplyDamage(1, 0.5f))
            {
                // 敵を吹っ飛ばすベクトルを算出
                DirectX::XMFLOAT3 vec;
                vec.x = outPosition.x - nodePosition.x;
                vec.z = outPosition.z - nodePosition.z;
                float length = sqrtf(vec.x * vec.x + vec.z * vec.z);
                vec.x /= length;
                vec.z /= length;

                // XZ平面に吹っ飛ばす力をかける
                float power = 10.0f;
                vec.x *= power;
                vec.z *= power;
                // Y方向にも力をかける
                vec.y = 5.0f;

                // 吹っ飛ばす
                player.AddImpulse(vec);
            }
        }
    }
}

---省略---

// 追跡ステート更新処理
void EnemySlime::UpdatePursuitState(float elapsedTime)
{
```

```
    ---省略---

    // プレイヤーの近くづくと攻撃ステートへ遷移
    float vx = targetPosition.x - position.x;
    float vy = targetPosition.y - position.y;
    float vz = targetPosition.z - position.z;
    float dist = sqrtf(vx * vx + vy * vy + vz * vz);
    if (dist < attackRange)
    {
        // 攻撃ステートへ遷移
        TransitionAttackState();
    }
}

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

    // 攻撃アニメーション再生
    model->PlayAnimation(Anim_Attack1, false);
}

// 攻撃ステート更新処理
void EnemySlime::UpdateAttackState(float elapsedTime)
{
    // 任意のアニメーション再生区間でのみ衝突判定処理をする
    float animationTime = model->GetCurrentAnimationSeconds();
    if (animationTime >= 0.1f && animationTime <= 0.35f)
    {
        // 目玉ノードとプレイヤーの衝突処理
        CollisionNodeVsPlayer("EyeBall", 0.2f);
    }
}
```

実行確認をしてみましょう。
プレイヤーに近づき、攻撃をしてダメージを与えることができれば OK です。
次は攻撃後にどうするかを考えましょう。

○戦闘待機ステート

攻撃範囲内にいるときに常に攻撃を繰り返していると不自然に見えますし、遊びにくいです。
攻撃をした後は少し様子を見るようなステートを実装してみましょう。

EnemySlime.h

```
---省略---

// スライム
class EnemySlime : public Enemy
{
public:
```

```
    ---省略---

private:
    ---省略---

    // 戦闘待機ステートへ遷移
    void TransitionIdleBattleState();

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

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

    ---省略---
};
```

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::IdleBattle:
        UpdateIdleBattleState(elapsedTime);
        break;
    }

    ---省略---
}

---省略---

// 攻撃ステート更新処理
void EnemySlime::UpdateAttackState(float elapsedTime)
{
    ---省略---

    // 攻撃アニメーションが終わったら戦闘待機ステートへ遷移
    if (!model->IsPlayAnimation())
    {
        TransitionIdleBattleState();
    }
}
```

```
// 戦闘待機ステートへ遷移
void EnemySlime::TransitionIdleBattleState()
{
    state = State::IdleBattle;

    // 数秒間待機するタイマーをランダム設定
    stateTimer = Mathf::RandomRange(2.0f, 3.0f);

    // 戦闘待機アニメーション再生
    model->PlayAnimation(Anim_IdleBattle, true);
}

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

    // タイマー処理
    stateTimer -= elapsedTime;
    if (stateTimer < 0.0f)
    {
        // プレイヤーが攻撃範囲にいた場合は攻撃ステートへ遷移
        float vx = targetPosition.x - position.x;
        float vy = targetPosition.y - position.y;
        float vz = targetPosition.z - position.z;
        float dist = sqrtf(vx * vx + vy * vy + vz * vz);
        if (dist < attackRange)
        {
            // 攻撃ステートへ遷移
            TransitionAttackState();
        }
        else
        {
            // 徘徊ステートへ遷移
            TransitionWanderState();
        }
    }

    MoveToTarget(elapsedTime, 0.0f);
}
```

実行確認をしてみましょう。
プレイヤーを攻撃後に待機状態に入り、数秒待ってからプレイヤーが攻撃範囲内にいた場合は攻撃し、範囲外だった場合は徘徊するようになっていれば OK です。

○ダメージステートと死亡ステート

敵の行動処理については一通り実装しました。
あとはダメージを受けた時と死亡した時のリアクションを強化しましょう。

現状ではダメージを受けたときに吹き飛ばされ、死亡した瞬間に消滅するという処理になっていましたが、ダメージを受けたときはダメージアニメーションを再生し、死亡するときは死亡アニメーションの再生が終わってから消滅するようにします。

EnemySlime.h

```
---省略---

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

protected:
    // ダメージを受けた時に呼ばれる
    void OnDamaged() override;

private:
    ---省略---

    // ダメージステートへ遷移
    void TransitionDamageState();

    // ダメージステート更新処理
    void UpdateDamageState(float elapsedTime);

    // 死亡ステートへ遷移
    void TransitionDeathState();

    // 死亡ステート更新処理
    void UpdateDeathState(float elapsedTime);

private:
    // ステート
    enum class State
    {
        ---省略---
        Damage,
        Death
    };

    ---省略---
};
```

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::Damage:
        UpdateDamageState(elapsedTime);
        break;

    case State::Death:
        UpdateDeathState(elapsedTime);
        break;
    }

    ---省略---
}

---省略---

// ダメージを受けた時に呼ばれる
void EnemySlime::OnDamaged()
{
    // ダメージステートへ遷移
    TransitionDamageState();
}

// 死亡した時に呼ばれる
void EnemySlime::OnDead()
{
    // 自身を破棄
    Destroy();

    // 死亡ステートへ遷移
    TransitionDeathState();
}

---省略---

// ダメージステートへ遷移
void EnemySlime::TransitionDamageState()
{
    state = State::Damage;

    // ダメージアニメーション再生
    model->PlayAnimation(Anim_GetHit, false);
}

// ダメージステート更新処理
void EnemySlime::UpdateDamageState(float elapsedTime)
{
    // ダメージアニメーションが終わったら戦闘待機ステートへ遷移
    if (!model->IsPlayAnimation())
    {
        TransitionIdleBattleState();
    }
}

// 死亡ステートへ遷移
```

```
void EnemySlime::TransitionDeathState()
{
    state = State::Death;

    // ダメージアニメーション再生
    model->PlayAnimation(Anim_Die, false);
}

// 死亡ステート更新処理
void EnemySlime::UpdateDeathState(float elapsedTime)
{
    // ダメージアニメーションが終わったら自分を破棄
    if (!model->IsPlayAnimation())
    {
        Destroy();
    }
}
```

実装出来たら実行確認してみましょう。
プレイヤーが敵に攻撃を行い、ダメージを与えてダメージアニメーションが再生され、死亡時には死亡アニメーション後に消滅できていれば OK です。

○プレイヤーのダメージ＆死亡処理

予定していた敵の実装はすべて完了しました。
ただ、敵がプレイヤーに攻撃を当てたときにプレイヤーのリアクションが実装されていません。
敵と同じ方法で実装できるので実装しておきましょう。

Player.h

```
---省略---

// プレイヤー
class Player : public Character
{
public:
    ---省略---

protected:
    ---省略---

    // ダメージを受けた時に呼ばれる
    void OnDamaged() override;

    // 死亡した時に呼ばれる
    void OnDead() override;

private:
    ---省略---

    // ダメージステートへ遷移
    void TransitionDamageState();
```

```
    // ダメージステート更新処理
    void UpdateDamageState(float elapsedTime);

    // 死亡ステートへ遷移
    void TransitionDeathState();

    // 死亡ステート更新処理
    void UpdateDeathState(float elapsedTime);

private:
    ---省略---

    // ステート
    enum class State
    {
        ---省略---
        Damage,
        Death
    };

    ---省略---
};
```

Player.cpp

```
---省略---

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

    case State::Damage:
        UpdateDamageState(elapsedTime);
        break;

    case State::Death:
        UpdateDeathState(elapsedTime);
        break;
    }

    // 速力処理更新
    ---省略---

    // 無敵時間更新
    UpdateInvincibleTimer(elapsedTime);

    ---省略---
}

---省略---
```

```
// 着地した時に呼ばれる
void Player::OnLanding()
{
    ---省略---

    // 着地ステートへ遷移
    TransitionLandState();

    // ダメージ、死亡ステート時は着地した時にステート遷移しないようにする
    if (state != State::Damage && state != State::Death)
    {
        // 着地ステートへ遷移
        TransitionLandState();
    }
}

// ダメージを受けた時に呼ばれる
void Player::OnDamaged()
{
    // ダメージステートへ遷移
    TransitionDamageState();
}

// 死亡した時に呼ばれる
void Player::OnDead()
{
    // 死亡ステートへ遷移
    TransitionDeathState();
}

---省略---

// ダメージステートへ遷移
void Player::TransitionDamageState()
{
    state = State::Damage;

    // ダメージアニメーション再生
    model->PlayAnimation(Anim_GetHit1, false);
}

// ダメージステート更新処理
void Player::UpdateDamageState(float elapsedTime)
{
    // ダメージアニメーションが終わったら待機ステートへ遷移
    if (!model->IsPlayAnimation())
    {
        TransitionIdleState();
    }
}

// 死亡ステートへ遷移
void Player::TransitionDeathState()
{
    state = State::Death;
```

```
    // 死亡アニメーション再生
    model->PlayAnimation(Anim_Death, false);
}

// 死亡ステート更新処理
void Player::UpdateDeathState(float elapsedTime)
{

}
```

実行確認してみましょう。
プレイヤーが敵からダメージを受けた時にダメージアニメーションが再生され、死亡時に死亡アニメーションが再生されれば OK です。

○プレイヤーの復活処理

現状ではプレイヤーが死亡後は何も操作することができません。
死亡後の処理をどうするかはゲームによって異なりますが、今回は何かボタンを押すことで復活させることにしましょう。

Player..h

```
---省略---

// プレイヤー
class Player : public Character
{
    ---省略---

private:
    ---省略---

    // 復活ステートへ遷移
    void TransitionReviveState();

    // 復活ステート更新処理
    void UpdateReviveState(float elapsedTime);

private:
    ---省略---

    // ステート
    enum class State
    {
        ---省略---
        Revive
    };

    ---省略---
};
```

Player.cpp

```
---省略---

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

    case State::Revive:
        UpdateReviveState(elapsedTime);
        break;
    }

    ---省略---
}

---省略---

// 死亡ステート更新処理
void Player::UpdateDeathState(float elapsedTime)
{
    if (!model->IsPlayAnimation())
    {
        // ボタンを押したら復活ステートへ遷移
        GamePad& gamePad = Input::Instance().GetGamePad();
        if (gamePad.GetButtonDown() & GamePad::BTN_A)
        {
            TransitionReviveState();
        }
    }
}

// 復活ステートへ遷移
void Player::TransitionReviveState()
{
    state = State::Revive;

    // 体力回復
    health = maxHealth;

    // 復活アニメーション再生
    model->PlayAnimation(Anim_Revive, false);
}

// 復活ステート更新処理
void Player::UpdateReviveState(float elapsedTime)
{
    // 復活アニメーション終了後に待機ステートへ遷移
    if (!model->IsPlayAnimation())
    {
        TransitionIdleState();
    }
```

```
}
```

実行確認してみましょう。
プレイヤーが死亡後にボタンを押して復活できれば OK です。

お疲れさまでした。