3D アクションゲームプログラミング

○評価要件

☑攻撃ステート処理
☑拳ボーンとの衝突判定
☑衝突判定のタイミングを制御

○概要

今回はボーンとの衝突判定を実装します。
前回で基本的なプレイヤーの行動処理を実装出来ました。
今回はパンチ攻撃処理を実装し、拳ボーンと敵との衝突判定でダメージ処理を実装します。
この時のダメージ衝突判定は攻撃アニメーション時の指定区間の間だけ行うようにします。

○攻撃ステートの実装

まずは攻撃を行うため、攻撃ステートを実装しましょう。
今回、攻撃は待機ステート時と移動ステート時に B ボタンを入力することで攻撃アニメーションを実行し、アニメーション終了後は待機ステートに戻るように実装しましょう。

Player.h

```
---省略---

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

private:
    ---省略---

    // 攻撃入力処理
    bool InputAttack();

    ---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:
    ---省略---

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

    ---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::Attack:
        UpdateAttackState(elapsedTime);
        break;
    }

    ---省略---
}
---省略---

// 攻撃入力処理
bool Player::InputAttack()
{
    GamePad& gamePad = Input::Instance().GetGamePad();

    if (gamePad.GetButtonDown() & GamePad::BTN_B)
    {
        return true;
    }

    return false;
}

---省略---

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

    // 攻撃入力処理
    if (InputAttack())
    {
        // 攻撃ステートへ遷移
        TransitionAttackState();
    }
}

---省略---

// 移動ステート更新処理
void Player::UpdateMoveState(float elapsedTime)
{
    ---省略---

    // 攻撃入力処理
    if (InputAttack())
    {
        // 攻撃ステートへ遷移
        TransitionAttackState();
    }
}

---省略---

// 攻撃ステートへ遷移
void Player::TransitionAttackState()
{
```

```
    state = State::Attack;

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

// 攻撃ステート更新処理
void Player::UpdateAttackState(float elapsedTime)
{
    // 攻撃アニメーション終了後
    if (!model->IsPlayAnimation())
    {
        // 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。
待機時と移動時に B ボタン入力で攻撃アニメーションが再生され、終了後に待機に戻っていれば OK です。

○ボーン情報を取得

ゲームによって攻撃時のダメージ判定方法は様々です。
アクション性を追求しないゲームでは適当にキャラクターの少し前方の位置と衝突判定をしたり、そもそも衝突判定をしない場合もあります。
今回はアクションゲームを想定しているので、ボーンとの衝突判定を実装します。

ボーンと衝突判定をするには、3D モデルから指定のボーンの位置情報を取得する必要があります。
3D モデルには多くのボーンが入っているので、指定のボーンを取得するようにモデルクラスを拡張しましょう。

課題で提供しているモデルクラスではボーンのことを「ノード」と呼称しています。
開発現場によってボーンやノードなど呼称は様々ですが、伝えたいことは同じです。
今回はモデルクラスのノード構造体を操作していきます。

まずはモデルクラスから名前で検索してノードを取得する関数を実装しましょう。

Model.h

```
---省略---
// モデル
class Model
{
public:
    ---省略---
```

```
    // ノード検索
    Node* FindNode(const char* name);

    ---省略---
};
```

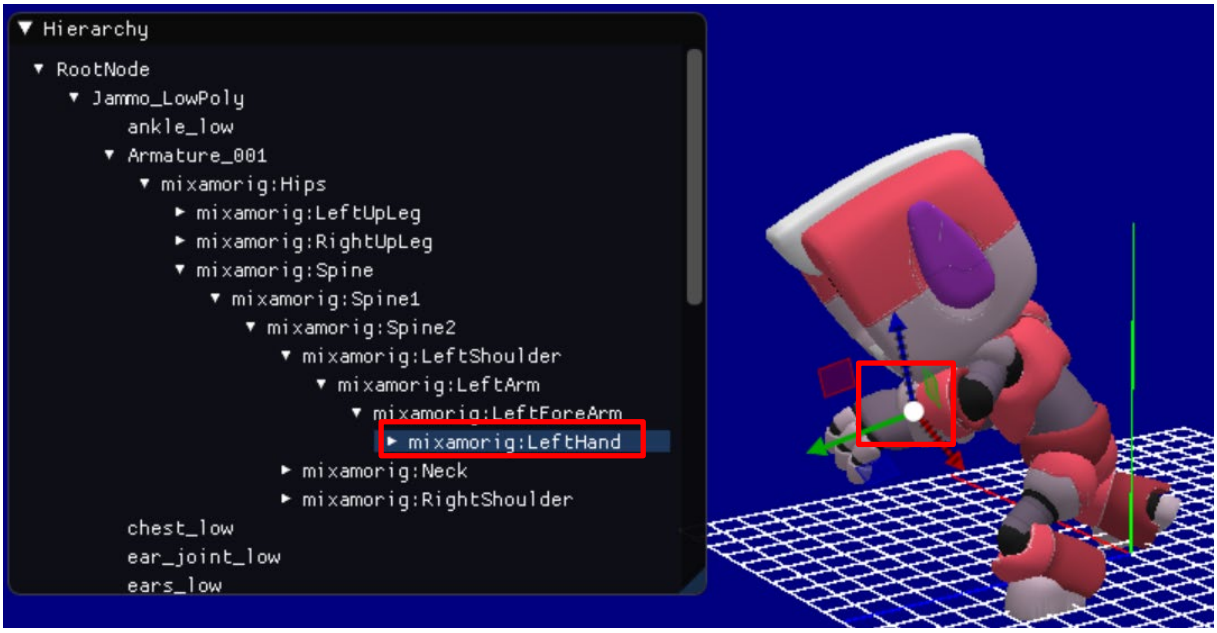
Model.cpp

```
---省略---

// ノード検索
Model::Node* Model::FindNode(const char* name)
{
    // 全てのノードを総当たりで名前比較する
    for (Node& node :  nodes)
    {
        if (::strcmp(node.name, name) == 0)
        {
            return &node;
        }
    }

    // 見つからなかった
    return nullptr;
}
```

ノードを検索する関数を実装しました。
後はボーンの名前さえわかればノードを取得することができます。
ここでモデルエディタを開き、ノード名を確認しましょう。
このキャラクターの攻撃は左拳によるパンチです。
左拳のノード名を調べます。



ノード名が確認できたのでボーンを取得し、ノードと敵の衝突判定を実装しましょう。

○ノードとエネミーの衝突判定

ノードとエネミーの衝突判定は、ノードを球形状、エネミーを円柱形状で行います。
攻撃して左拳ノードとエネミーが衝突したら敵を吹き飛ばし、ヒットエフェクトを表示するようにしましょう。

Player.h

```
---省略---

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

private:
    ---省略---

    // ノードとエネミーの衝突処理
    void CollisionNodeVsEnemies(const char* nodeName, float nodeRadius);

    ---省略---
private:
    ---省略---
    float               leftHandRadius = 0.4f;
};
```

Player.cpp

```
---省略---

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

    // 攻撃衝突用の左手ノードのデバッグ球を描画
    Model::Node* leftHandBone = model->FindNode("mixamorig:LeftHand");
    debugRenderer->DrawSphere(DirectX::XMFLOAT3(
        leftHandBone->worldTransform._41,
        leftHandBone->worldTransform._42,
        leftHandBone->worldTransform._43),
        leftHandRadius,
        DirectX::XMFLOAT4(1, 0, 0, 1)
    );
}

---省略---

// ノードと敵の衝突処理
void Player::CollisionNodeVsEnemies(const char* nodeName, float nodeRadius)
{
    // ノード取得
```

```
    Model::Node* node = model->FindNode(nodeName);

    // ノード位置取得
    DirectX::XMFLOAT3 nodePosition;
    nodePosition.x = node->worldTransform._41;
    nodePosition.y = node->worldTransform._42;
    nodePosition.z = node->worldTransform._43;

    // 指定のノードと全ての敵を総当たりで衝突処理
    EnemyManager& enemyManager = EnemyManager::Instance();
    int enemyCount = enemyManager.GetEnemyCount();
    for (int i = 0; i < enemyCount; ++i)
    {
        Enemy* enemy = enemyManager.GetEnemy(i);

        // 衝突処理
        DirectX::XMFLOAT3 outPosition;
        if (Collision::IntersectSphereVsCylinder(
            nodePosition,
            nodeRadius,
            enemy->GetPosition(),
            enemy->GetRadius(),
            enemy->GetHeight(),
            outPosition))
        {
            // ダメージを与える
            if (enemy->ApplyDamage(1, 0.5f))
            {
                // 吹っ飛ばす
                {
                    const float power = 10.0f;
                    const DirectX::XMFLOAT3& e = enemy->GetPosition();
                    float vx = e.x - nodePosition.x;
                    float vz = e.z - nodePosition.z;
                    float lengthXZ = sqrtf(vx * vx + vz * vz);
                    vx /= lengthXZ;
                    vz /= lengthXZ;

                    DirectX::XMFLOAT3 impulse;
                    impulse.x = vx * power;
                    impulse.y = power * 0.5f;
                    impulse.z = vz * power;

                    enemy->AddImpulse(impulse);
                }

                // ヒットエフェクト再生
                {
                    DirectX::XMFLOAT3 e = enemy->GetPosition();
                    e.y += enemy->GetHeight() * 0.5f;
                    hitEffect->Play(e);
                }
            }
        }
    }
}
```

```
---省略---

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

    // 左手ノードとエネミーの衝突処理
    CollisionNodeVsEnemies("mixamorig:LeftHand", leftHandRadius);
}
```

実装出来たら実行確認してみましょう。
敵の近くへ移動し、攻撃をして敵に当たったら OK です。

○当たり判定のタイミングを制御

攻撃が敵に当たるようになりましたが、現状では拳と敵が密着した状態で攻撃すると、アニメーションが再生された瞬間に攻撃が当たってしまします。
拳を振り上げている状態で攻撃が当たってしまうのは見た目的な説得力が低いので拳を振り下ろしている間に当たり判定が発生するようにしましょう。

任意のタイミングで当たり判定を発生させるにはアニメーションの再生時間で判定するのが一般的です。
アニメーションの再生時間を取得し、任意のタイミングでのみ衝突処理を実行します。
まずはアニメーション時間を取得できるようにしましょう。

Model.h

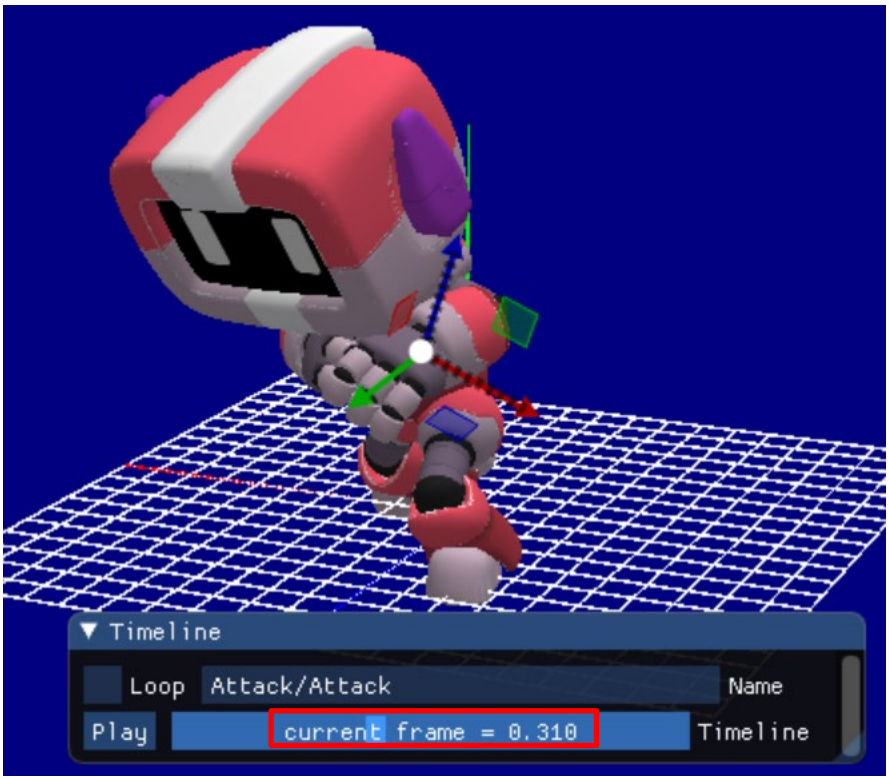
```
---省略---

// モデル
class Model
{
public:
    ---省略---

    // 現在のアニメーション再生時間取得
    float GetCurrentAnimationSeconds() const { return currentAnimationSeconds; }

    ---省略---
};
```

再生中のアニメーション時間を取得できるようになったので、当たり判定を発生させる区間を確認しましょう。
モデルエディタを開き、攻撃アニメーションを再生し、衝突判定を発生させたい時間を調べましょう。



このアニメーションでは 0.3～0.4 秒くらいの再生時間に衝突判定をさせればよさそうです。
アニメーションの再生時間を取得し、衝突判定を発生させる時間を制限しましょう。

Player.h

```
---省略---

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

private:
    ---省略---
    bool                attackCollisionFlag = false;
};
```

Player.cpp

```
---省略---

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

    // 攻撃衝突用の左手ノードのデバッグ球を描画
    Model::Node* leftHandBone = model->FindNode("mixamorig:LeftHand");
    debugRenderer->DrawSphere(---省略---);
```

```
    if (attackCollisionFlag)
    {
        Model::Node* leftHandBone = model->FindNode("mixamorig:LeftHand");
        debugRenderer->DrawSphere(---省略---);
    }
}

---省略---

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

    // 左手ノードとエネミーの衝突処理
    CollisionNodeVsEnemies("mixamorig:LeftHand", leftHandRadius);

    // 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();
    attackCollisionFlag = animationTime >= 0.3f && animationTime <= 0.4f;
    if (attackCollisionFlag)
    {
        // 左手ノードとエネミーの衝突処理
        CollisionNodeVsEnemies("mixamorig:LeftHand", leftHandRadius);
    }
}
```

実装が完了したら実行確認してみましょう。
左拳の攻撃衝突判定用のデバッグ球が攻撃時の衝突判定が有効な時だけ表示され、敵との衝突判定ができていれば OK です。

お疲れさまでした。