

# ゲームプログラミングⅢ

---

## ○評価要件

- ☑ステージの表示
- ☑プレイヤーの表示
- ☑プレイヤーの XZ 平面移動
- ☑プレイヤーの XYZ 軸回転

## ○概要

これから 3D ゲームプログラミングをするにあたって一番最初にやっておきたいことは 3D モデルを画面に表示することです。

3D モデルを描画するためには DirectX や OpenGL などの API を利用してシステムを組む必要がありますが、この課題では 3D モデルを簡単に表示するためのプログラムをあらかじめ用意しています。

DirectX の勉強と 3D モデルを描画する方法については他の課題を通して学習しましょう。ここではゲーム内での 3D オブジェクトの表示や操作方法について学習していきます。

今回の課題では 2 つの 3D オブジェクトを表示していきます。

- ステージ
- プレイヤーキャラクター

まずはソリューションファイルを開き、コンパイルしてゲームを実行しましょう。  
何も表示されない下図の画面が表示されるので、これから 3D オブジェクトを表示していきます。



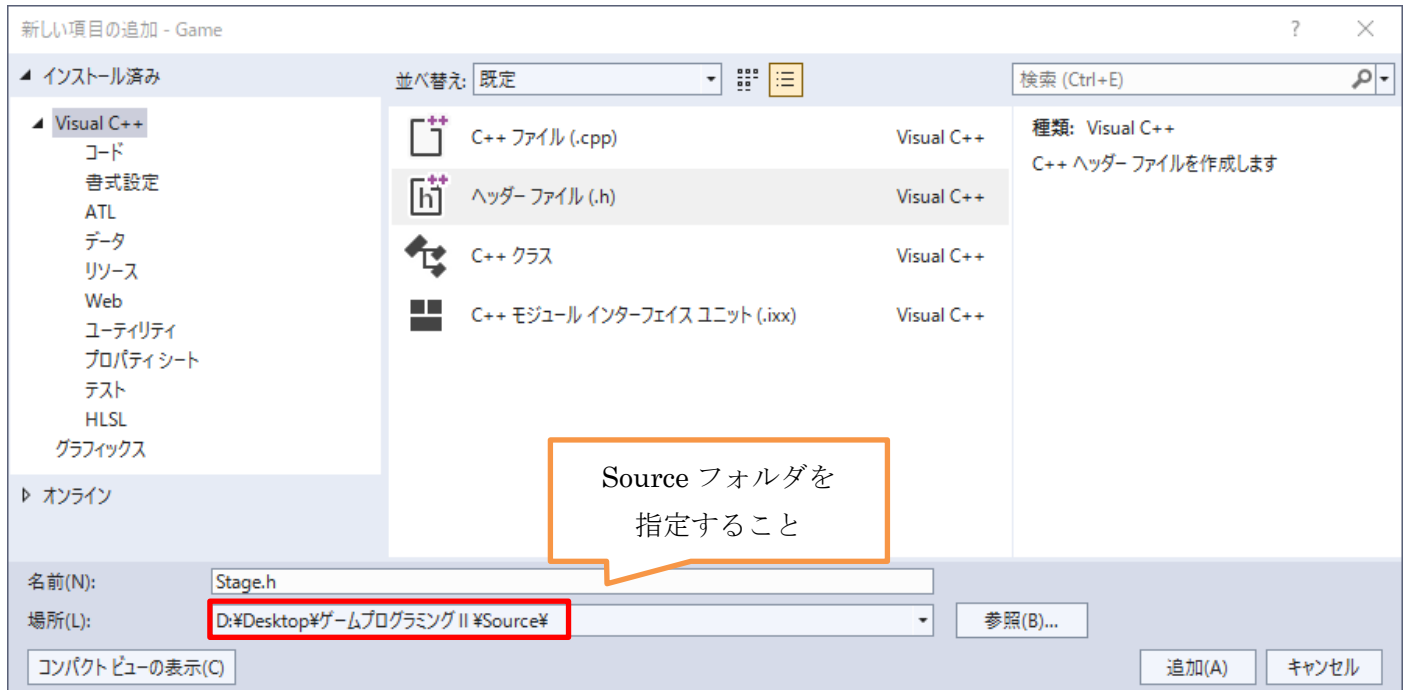
## ゲームプログラミングⅢ

### ○ステージの表示

まずはステージを表示するために Stage クラスを作成します。

Source フォルダ以下に Stage.cpp と Stage.h を作成しましょう。

ソースファイルを作成する場所に気を付けましょう。



下記のプログラムを記述してください。

### Stage.h

```
#pragma once

#include "System/ModelRenderer.h"

// ステージ
class Stage
{
public:
    Stage();
    ~Stage();

    // 更新処理
    void Update(float elapsedTime);

    // 描画処理
    void Render(const RenderContext& rc, ModelRenderer* renderer);

private:
    ModelRenderer* model = nullptr;
};
```

## ゲームプログラミングⅢ

### Stage.cpp

```
#include "Stage.h"

// コンストラクタ
Stage::Stage()
{
    // ステージモデルを読み込み
    model = new Model("Data/Model/Stage/ExampleStage.mdl");
}

Stage::~~Stage()
{
    // ステージモデルを破棄
    delete model;
}

// 更新処理
void Stage::Update(float elapsedTime)
{
    // 今は特にやることはない
}

// 描画処理
void Stage::Render(const RenderContext& rc, ModelRenderer* renderer)
{
    DirectX::XMFLOAT4X4 transform;
    DirectX::XMStoreFloat4x4(&transform, DirectX::XMMatrixIdentity());

    // レンダラにモデルを描画してもらう
    renderer->Render(rc, transform, model, ShaderId::Lambert);
}
```

transform は 3D モデルを表示する  
位置や姿勢の情報（後に説明）

ShaderId は見た目の指定  
Lambert は一般的な陰影表現

ステージクラスの作成が完了したのでシーンに描画してみましょう。

SceneGame.h と SceneGame.cpp を開き下記プログラムコードを追記しましょう。

### SceneGame.h

```
#pragma once

#include "Stage.h"

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

private:
    Stage* stage = nullptr;
};
```

### SceneGame.cpp

```
---省略---

// 初期化
void SceneGame::Initialize()
{
    // ステージ初期化
    stage = new Stage();
}

// 終了化
void SceneGame::Finalize()
{
    // ステージ終了化
    if (stage != nullptr)
    {
        delete stage;
        stage = nullptr;
    }
}

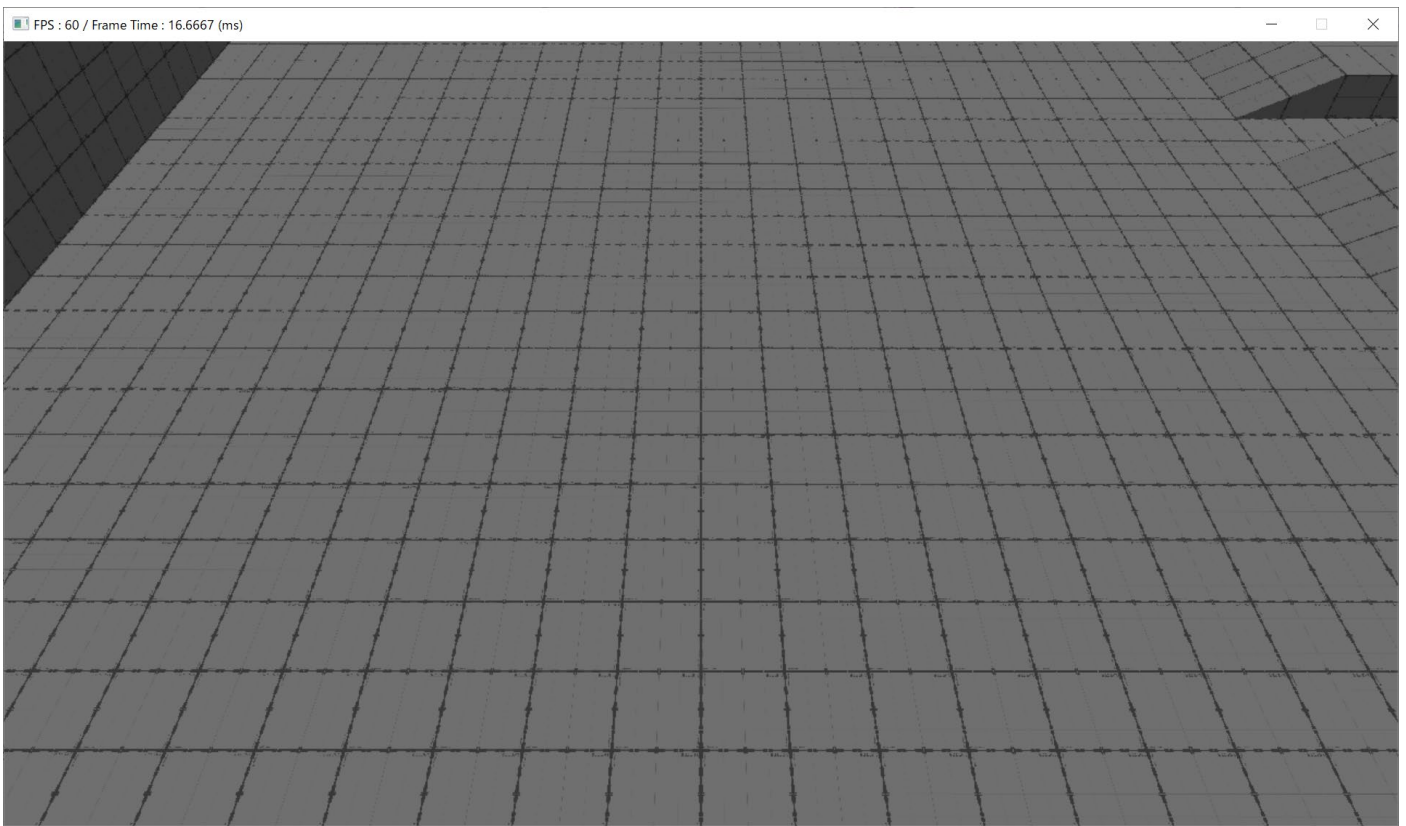
// 更新処理
void SceneGame::Update(float elapsedTime)
{
    // ステージ更新処理
    stage->Update(elapsedTime);
}

// 描画処理
void SceneGame::Render()
{
    ---省略---
    // 3Dモデル描画
    {
        // ステージ描画
        stage->Render(rc, modelRenderer);
    }
    ---省略---
}
```

実行確認をしてみましょう。

下図のようにステージの一部が表示されていれば OK です。

# ゲームプログラミングⅢ



本課題の 3D モデルの表示は **Model** を作成&読み込みをし、**ModelRenderer** に **Render()**してもらうことで表示できます。

## ○キャラクター表示に必要なもの

次はキャラクターを表示してみましょう。

キャラクターはステージと違い、移動したり、回転したりします。

基本的に動くオブジェクトには以下の 3 つの情報が必要です。

- ・位置
- ・回転
- ・スケール

この 3 つの情報を組み合わせることでキャラクターの「姿勢」を表現できます。

そしてこの「姿勢」は「行列」であらわすことになります。

## ゲームプログラミングⅢ

### ○行列について

3D ゲームプログラミングで行列とは 4x4 の 16 個の数値を用いて「位置」「回転」「スケール」を組み合わせた「姿勢」を表現できます。

```
DirectX: XMFLOAT4X4 transform =
```

```
{
```

```
1, 0, 0, 0,
```

```
0, 1, 0, 0,
```

```
0, 0, 1, 0,
```

```
0, 0, 0, 1
```

```
};
```

スケール&回転

位置

今はまだ深く理解できなくても大丈夫です。

「位置」「回転」「スケール」の情報を組み合わせて最終的に「行列」を作成する必要があると覚えておきましょう。

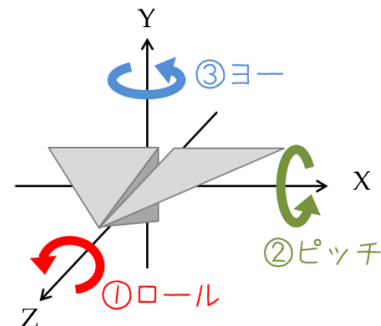
### ○「位置」「回転」「スケール」について

位置は XYZ の座標情報が必要になります。

スケールも XYZ の倍率情報を持ちます。

回転には大きく分けて 2 つの制御方法があります。

- ・オイラー (XYZ)
- ・クォータニオン (XYZW)



今回は直感的で制御しやすい「オイラー」を使っていきます。

オイラーでの XYZ の各要素は軸に対する回転角度 (-3.14~3.14) を設定します。

難しいことは考えず、とりあえずキャラクタークラスを作成してみましょう。

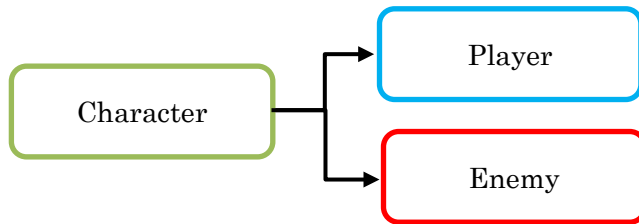
### ○キャラクタークラス

これからキャラクタークラスを作成していくわけですが、実装の前に簡単なプログラム設計をしてみましょう。

この課題では今後、プレイヤーキャラクターと敵キャラクターが出てくる予定です。

なので、C++の「継承」を利用したクラス設計を行います。

## ゲームプログラミングⅢ



キャラクタークラスはプレイヤーや敵の共通の情報を持ち、共通の処理を行うようにしていきます。

では簡単な設計方針もわかったところでプログラムを実装していきます。

Character.cpp と Character.h を作成し、下記プログラムコードを記述しましょう。

### Character.h

```
#pragma once
```

```
#include <DirectXMath.h>
```

```
// キャラクター  
class Character
```

```
{  
public:  
    Character() {}  
    virtual ~Character() {}
```

```
// 行列更新処理
```

```
void UpdateTransform();
```

```
protected:
```

```
    DirectX::XMFLOAT3 position = { 0, 0, 0 };  
    DirectX::XMFLOAT3 angle = { 0, 0, 0 };  
    DirectX::XMFLOAT3 scale = { 1, 1, 1 };  
    DirectX::XMFLOAT4X4 transform = {  
        1, 0, 0, 0,  
        0, 1, 0, 0,  
        0, 0, 1, 0,  
        0, 0, 0, 1  
    };
```

```
};
```

継承する目的のクラスはデストラクタに  
virtual をつけること！  
※継承先のデストラクタが呼ばれなくなってしまう。

継承先でもアクセス  
できるようにする

「位置」「回転」「スケール」を用意  
し、  
これらのパラメータを元に計算する

### Character.cpp

```
#include "Character.h"
```

```
// 行列更新処理
```

```
void Character::UpdateTransform()  
{
```

```
    // スケール行列を作成
```

```
    DirectX::XMATRIX S = DirectX::XMMatrixScaling( );
```

```
    // 回転行列を作成
```

```
    DirectX::XMATRIX R = DirectX::XMMatrixRotationRollPitchYaw( );
```

初めて見る関数や構造体は  
右クリック→「定義へ移動」  
を  
選択して、どういう内容かを

## ゲームプログラミングⅢ

```
// 位置行列を作成
DirectX::XMATRIX T = DirectX::XMMatrixTranslation();
// 3つの行列を組み合わせ、ワールド行列を作成
DirectX::XMATRIX W = S * R * T;
// 計算したワールド行列を取り出す
DirectX::XMStoreFloat4x4(&transform, W);
}
```

行列は乗算することで  
合体できる

UpdateTransform()関数では位置、回転、スケールの情報を組み合わせて行列を作成する処理を実装しました。

DirectXMath という DirectX が用意してくれている高速計算関数を利用しています。

少し使い方にクセがありますが、今後も DirectXMath はどんどん出てくるので慣れていきましょう。

### ○プレイヤークラス

続いてプレイヤークラスを作成しましょう。

プレイヤークラスではモデルを読み込み、簡単な移動処理や回転処理などをしてみましょう。

Player.cpp と Player.h を作成し、下記プログラムコードを記述しましょう。

#### Player.h

```
#pragma once

#include "System/ModelRenderer.h"
#include "Character.h"

// プレイヤー
class Player : public Character
{
public:
    Player();
    ~Player() override;

    // 更新処理
    void Update(float elapsedTime);

    // 描画処理
    void Render(const RenderContext& rc, ModelRenderer* renderer);

private:
    Model* model = nullptr;
};
```

#### Player.cpp

```
#include "Player.h"

// コンストラクタ
```



## ゲームプログラミングⅢ

```
Player::Player ()
{
    model = new Model ("Data/Model/Mr. Incredible/Mr. Incredible.mdl");

    // モデルが大きいのでスケーリング
    scale.x = scale.y = scale.z = 0.01f;
}

// デストラクタ
Player::~~Player ()
{
    delete model;
}

// 更新処理
void Player::Update(float elapsedTime)
{
    // オブジェクト行列を更新
    UpdateTransform();

    // モデル行列更新
    model->UpdateTransform();
}

// 描画処理
void Player::Render(const RenderContext& rc, ModelRenderer* renderer)
{
    renderer->Render(rc, transform, model, ShaderId::Lambert);
}
```

描画の際に行列を渡すことで  
オブジェクトの位置やスケールを指定する

ひとまずプレイヤークラスの実装はここまでにしておきましょう。

SceneGame.h と SceneGame.cpp にステージの時と同じ要領でプレイヤーをシーンに表示させてみましょう。

### SceneGame.h

```
---省略---
#include "Player.h"


// ゲームシーン
class SceneGame
{
    ---省略---
private:
    ---省略---
    Player* player = nullptr;
};
```


### SceneGame.cpp

```
---省略---
```

## ゲームプログラミングⅢ


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

    // プレイヤー初期化
    
}

// 終了化
void SceneGame::Finalize()
{
    // プレイヤー終了化
    


    ---省略---
}

// 更新処理
void SceneGame::Update(float elapsedTime)
{
    ---省略---

    // プレイヤー更新処理
    
}

// 描画処理
void SceneGame::Render()
{
    ---省略---

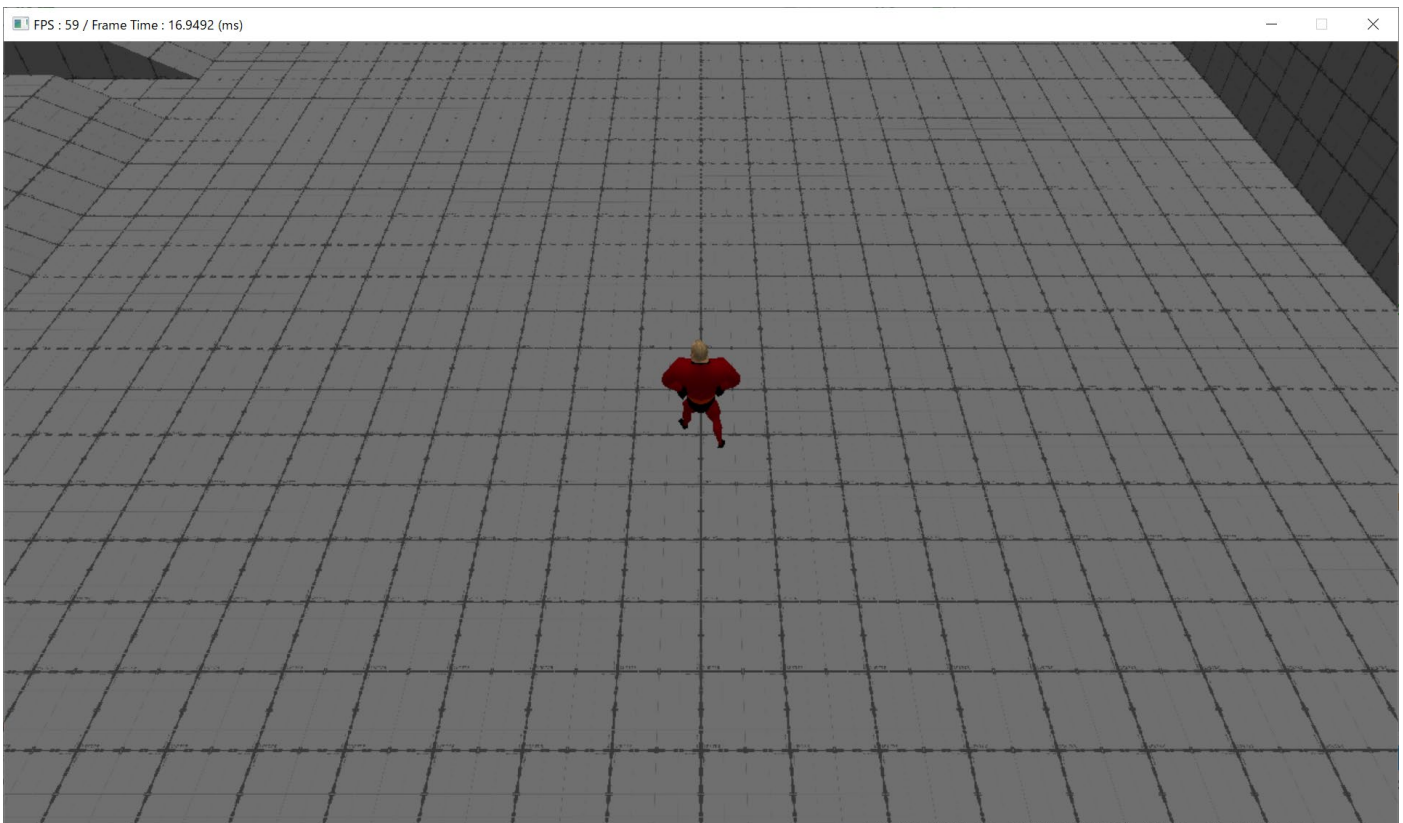
    // 3Dモデル描画
    {
        ---省略---

        // プレイヤー描画
        
    }

    ---省略---
}
```

実行確認してみて下図のようにキャラクターが表示されれば OK です。

# ゲームプログラミングⅢ



## ○移動、回転操作

プレイヤークラスの Update()関数内に簡単な移動処理と回転処理を実装しましょう。  
ゲームパッドの左スティックを入力することでプレイヤーが XZ 平面上に移動するようにしてみましょう。ゲームパッドの A ボタンを押すことでキャラクターY 軸回転するようにしましょう。

### Player.cpp

```
---省略---
#include "System/Input.h"
---省略---

// 更新処理
void Player::Update(float elapsedTime)
{
    // 入力情報を取得
    GamePad& gamePad = Input::Instance().GetGamePad();
    float ax = gamePad.GetAxisLX();
    float ay = gamePad.GetAxisLY();

    // 移動操作
    float moveSpeed = 5.0f * elapsedTime;
    {
        // 左スティックの入力情報をもとにXZ平面への移動処理
    }
}
```

一秒間に 5.0 移動する速度

## ゲームプログラミングⅢ

```
}

// 回転操作
float rotateSpeed = DirectX::XMConvertToRadians(360) * elapsedTime;
if (gamePad.GetButton() & GamePad::BTN_A)
{
    // X軸回転操作
    
}
if (gamePad.GetButton() & GamePad::BTN_B)
{
    // Y軸回転操作
    
}
if (gamePad.GetButton() & GamePad::BTN_X)
{
    // Z軸回転操作
    
}

// オブジェクト行列を更新
---省略---
```

一秒間に 360 度回転する速度

GamePad クラスは実際のゲームコントローラー以外にもキーボード割り当てもされています。  
左スティック入力値は「W」「A」「S」「D」キー、または「↑」「←」「↓」「→」キーに割り当て、  
A ボタンは Z キー  
B ボタンは X キー  
X ボタンは C キー  
Y ボタンは V キーに割り当てられています。

実装できたら実行確認してみましょう。

キャラクターが自由に移動でき、A、B、X ボタンを押し続けることで各軸に対して回転出来ていれば OK です。

### ○デバッグメニュー描画

今後の開発を円滑に進めるためにデバッグメニューを導入しましょう。

デバッグメニューがあるとゲーム中に移動速度の調整をしたり、座標の設定をしたりなど大変便利です。

ImGui という非常に優れたライブラリを使ってデバッグメニューを表示しましょう。

本来、ネットからライブラリをダウンロードして自身にプログラムに導入するものですが、このプロジェクトではあらかじめ導入しているのでこの手順を省きます。

Player.h と Player.cpp を開き、下記プログラムコードを追記しましょう。

## ゲームプログラミングⅢ

### Player.h

```
---省略---

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

    // デバッグ用GUI描画
    void DrawDebugGUI();

    ---省略---
};
```

### Player.cpp

```
#include <imgui.h>
---省略---

// デバッグ用GUI描画
void Player::DrawDebugGUI()
{
    ImVec2 pos = ImGui::GetMainViewport()->GetWorkPos();
    ImGui::SetNextWindowPos(ImVec2(pos.x + 10, pos.y + 10), ImGuiCond_Once);
    ImGui::SetNextWindowSize(ImVec2(300, 300), ImGuiCond_FirstUseEver);

    if (ImGui::Begin("Player", nullptr, ImGuiWindowFlags_None))
    {
        // トランスフォーム
        if (ImGui::CollapsingHeader("Transform", ImGuiTreeNodeFlags_DefaultOpen))
        {
            // 位置
            ImGui::InputFloat3("Position", &position.x);
            // 回転
            DirectX::XMFLOAT3 a;
            a.x = DirectX::XMConvertToDegrees(angle.x);
            a.y = DirectX::XMConvertToDegrees(angle.y);
            a.z = DirectX::XMConvertToDegrees(angle.z);
            ImGui::InputFloat3("Angle", &a.x);
            angle.x = DirectX::XMConvertToRadians(a.x);
            angle.y = DirectX::XMConvertToRadians(a.y);
            angle.z = DirectX::XMConvertToRadians(a.z);
            // スケール
            ImGui::InputFloat3("Scale", &scale.x);
        }
    }
    ImGui::End();
}
```

デバッグメニューをシーンに描画します。

## ゲームプログラミングⅢ

SceneGame.cpp に下記プログラムコードを追記しましょう。

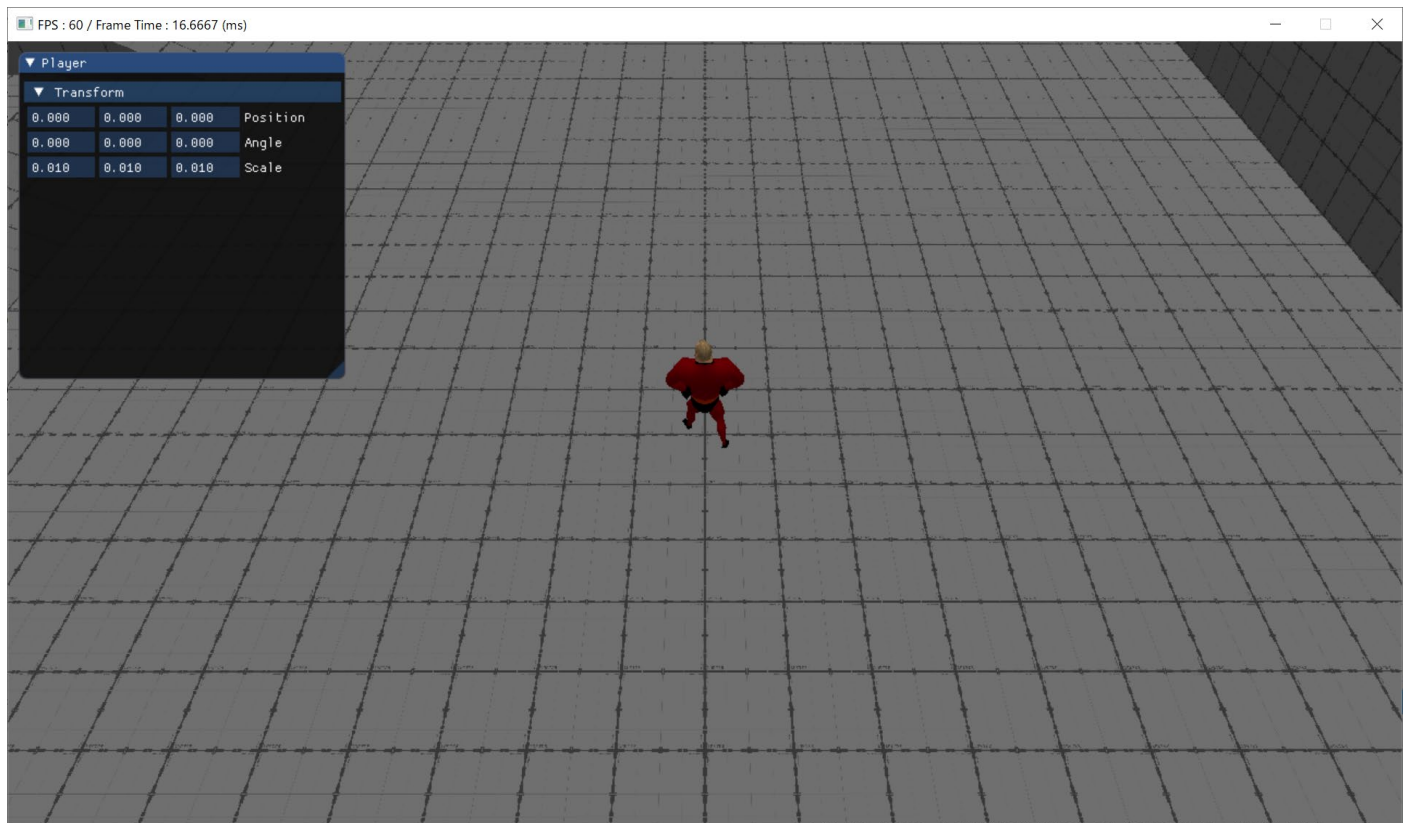
SceneGame.cpp

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

下図のようにデバッグメニューが表示されていれば OK です。

ImGui についての説明は今後することはありません。

自分でいろいろ試してみて使いやすいデバッグメニューにしていきましょう。



課題はこれで完了です。

お疲れさまでした。