ゲームプログラミング改

○評価要件

☑シーン切り替え
☑ローディング画面

○概要

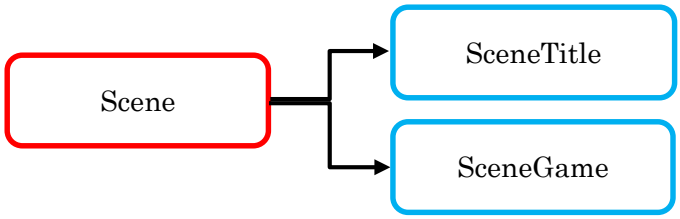
今回はシーン遷移を実装します。
一般的なゲームはタイトル画面から始まってゲーム画面へ遷移してゲームを遊ぶ流れです。
ゲームの仕様にもよりますが、リザルト画面やゲームオーバー画面に遷移した後、タイトル画面へ戻るなど「シーン」という区切りでゲームを構成していることが多いです

今回は簡単なシーン遷移システムを実装します。
現在はあらかじめ用意していたゲームシーンクラスにゲーム内容を実装しています。
タイトルシーンを作り、タイトルシーンからゲームシーンへ遷移するシステムを実装していきましょう。

シーン遷移システムができた人はシーン切り替え時に Now Loading…などのローディング画面の表示に挑戦してみましょう。

○シーン遷移システム

今回は以下のクラス設計でプログラムを実装していきます。



まずはシーン遷移の流れをイメージしましょう。

1.シーンの遷移はシーンを管理するシーンマネージャーが行います。
2.シーンマネージャーはシーンを１つだけ保持します。
3.シーンマネージャーが保持しているシーンの更新処理と描画処理を毎フレーム実行します。
4.シーンを切り替えたいときはシーンマネージャーに新しいシーンを渡します。
5.シーンマネージャーは古いシーンの終了処理を実行します。
6.シーンマネージャーは渡された新しいシーンを保持します。
7.シーンマネージャーは新しいシーンの初期化を行います。

これらを実装すれば簡易的なシーン遷移システムが完成します。

ではまずシーンの基底となるクラスを実装しましょう。
Scene.h を作成し、下記プログラムコードを記述しましょう。

Scene.h

```
#pragma once

// シーン
class Scene
{
public:
    Scene() {}
    virtual ~Scene() {}

    // 初期化
    virtual void Initialize() = 0;

    // 終了化
    virtual void Finalize() = 0;

    // 更新処理
    virtual void Update(float elapsedTime) = 0;

    // 描画処理
    virtual void Render() = 0;
};
```

タイトルシーンを実装しましょう。
SceneTitle.cpp と SceneTitle.h を作成し、下記プログラムコードを記述しましょう。

SceneTitle.h

```
#pragma once

#include "Graphics/Sprite.h"
#include "Scene.h"

// タイトルシーン
class SceneTitle : public Scene
{
public:
    SceneTitle() {}
    ~SceneTitle() override {}

    // 初期化
    void Initialize() override;

    // 終了化
    void Finalize() override;

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

    // 描画処理
    void Render() override;

private:
    Sprite* sprite = nullptr;
};
```

SceneTitle.cpp

```
#include "Graphics/Graphics.h"
#include "SceneTitle.h"

// 初期化
void SceneTitle::Initialize()
{
    // スプライト初期化
    sprite = new Sprite("Data/Sprite/Title.png");
}

// 終了化
void SceneTitle::Finalize()
{
    // スプライト終了化
    if (sprite != nullptr)
    {
        delete sprite;
        sprite = nullptr;
    }
```

```
}

// 更新処理
void SceneTitle::Update(float elapsedTime)
{
}

// 描画処理
void SceneTitle::Render()
{
    Graphics& graphics = Graphics::Instance();
    ID3D11DeviceContext* dc = graphics.GetDeviceContext();
    ID3D11RenderTargetView* rtv = graphics.GetRenderTargetView();
    ID3D11DepthStencilView* dsv = graphics.GetDepthStencilView();

    // 画面クリア＆レンダーターゲット設定
    FLOAT color[] = { 0.0f, 0.0f, 0.5f, 1.0f };    // RGBA(0.0～1.0)
    dc->ClearRenderTargetView(rtv, color);
    dc->ClearDepthStencilView(dsv, D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);
    dc->OMSetRenderTargets(1, &rtv, dsv);

    // 2Dスプライト描画
    {
        float screenWidth = static_cast<float>(graphics.GetScreenWidth());
        float screenHeight = static_cast<float>(graphics.GetScreenHeight());
        float textureWidth = static_cast<float>(sprite->GetTextureWidth());
        float textureHeight = static_cast<float>(sprite->GetTextureHeight());
        // タイトルスプライト描画
        sprite->Render(dc,
            0, 0, screenWidth, screenHeight,
            0, 0, textureWidth, textureHeight,
            0,
            1, 1, 1, 1);
    }
}
```

このタイトルシーンは画面全体に「TITILE」と書かれたスプライトを表示するだけのシーンです。
まずはシーン管理システムでこのシーンを表示できるようにしましょう。

シーンの管理を行うシーンマネージャーを実装しましょう。
SceneManager.cpp と SceneManager.h を作成し、下記プログラムコードを実装しましょう。

SceneManager.h

```
#pragma once

#include "Scene.h"

// シーンマネージャー
class SceneManager
{
private:
```

```
    SceneManager() {}
    ~SceneManager() {}

public:
    // 唯一のインスタンス取得
    static SceneManager& Instance()
    {
        static SceneManager instance;
        return instance;
    }

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

    // 描画処理
    void Render();

    // シーンクリア
    void Clear();

    // シーン切り替え
    void ChangeScene(Scene* scene);

private:
    Scene* currentScene = nullptr;
};
```

管理しているシーンの
終了処理を行う関数

SceneManager.cpp

```
#include "SceneManager.h"

// 更新処理
void SceneManager::Update(float elapsedTime)
{
    if (currentScene != nullptr)
    {
        currentScene->Update(elapsedTime);
    }
}

// 描画処理
void SceneManager::Render()
{
    if (currentScene != nullptr)
    {
        currentScene->Render();
    }
}

// シーンクリア
void SceneManager::Clear()
{
    if (currentScene != nullptr)
    {
        currentScene->Finalize();
```

```
        delete currentScene;
        currentScene = nullptr;
    }
}

// シーン切り替え
void SceneManager::ChangeScene(Scene* scene)
{
    // 古いシーンを終了処理
    
    // 新しいシーンを設定
    
    // シーン初期化処理
    
}
```

シーンマネージャーの実装が出来たらタイトルシーンを表示てみましょう。

Framework.cpp

```
---省略---
#include "SceneGame.h"
#include "SceneTitle.h"
#include "SceneManager.h"

static SceneGame sceneGame;

---省略---

// コンストラクタ
Framework::Framework(HWND hWnd)
    : hWnd(hWnd)
    , input(hWnd)
    , graphics(hWnd)
{
    ---省略---

    // シーン初期化
    sceneGame.Initialize();
    SceneManager::Instance().ChangeScene(new SceneTitle);
}

// デストラクタ
Framework::~Framework()
{
    // シーン終了化
    sceneGame.Finalize();
    SceneManager::Instance().Clear();

    ---省略---
}
```

```
// 更新処理
void Framework::Update(float elapsedTime/*Elapsed seconds from last frame*/)
{
    ---省略---

    // シーン更新処理
    sceneGame.Update(elapsedTime);
    SceneManager::Instance().Update(elapsedTime);
}

// 描画処理
void Framework::Render(float elapsedTime/*Elapsed seconds from last frame*/)
{
    ---省略---

    // シーン描画処理
    sceneGame.Render();
    SceneManager::Instance().Render();

    ---省略---
}

---省略---
```

実装が終わったら実行確認をしてみましょう。
下図の画面になっていれば OK です。

次はタイトルシーンからゲームシーンへ遷移させましょう。
ゲームシーンを今回作成したシーン遷移システムに対応させましょう。

SceneGame.h

```
---省略---
#include "Scene.h"

// ゲームシーン
class SceneGame
class SceneGame : public Scene
{
public:
    ---省略---
    ~SceneGame() {}
    ~SceneGame() override {}

    // 初期化
    void Initialize();
    void Initialize() override;

    // 終了化
    void Finalize();
    void Finalize() override;

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

    // 描画処理
    void Render();
    void Render() override;

    ---省略---
};
```

対応できたらタイトルシーンで何かボタンを押したらゲームシーンへ遷移するプログラムを実装しましょう。

SceneTitle.cpp

```
---省略---
#include "SceneGame.h"
#include "SceneManager.h"
#include "Input/Input.h"

---省略---

// 更新処理
void SceneTitle::Update(float elapsedTime)
{
    GamePad& gamePad = Input::Instance().GetGamePad();

    // なにかボタンを押したらゲームシーンへ切り替え
    const GamePadButton anyButton =
          GamePad::BTN_A
        | GamePad::BTN_B
        | GamePad::BTN_X
        | GamePad::BTN_Y
```

```
        ;
    
}
```

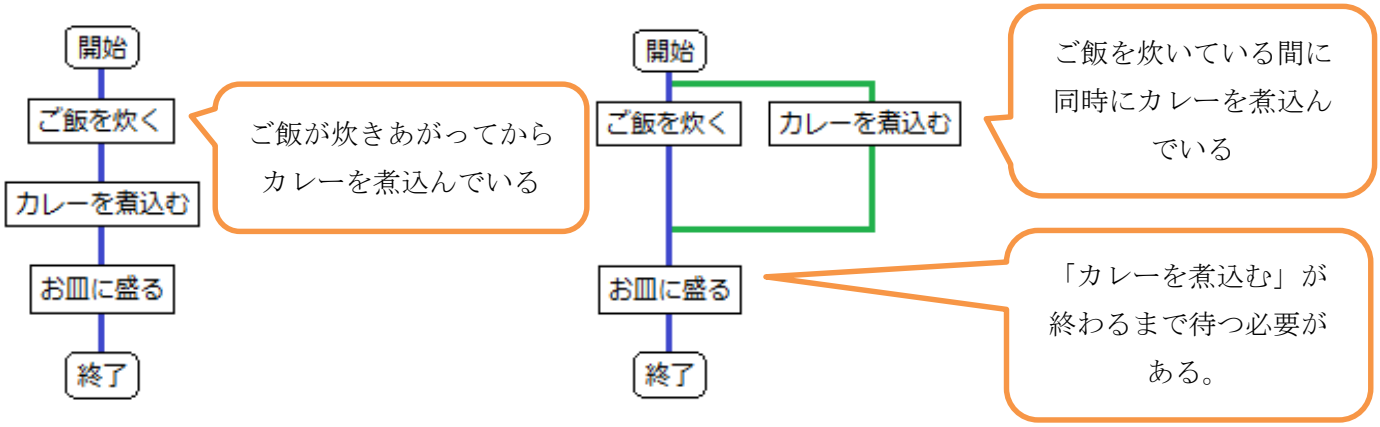
ゲームを起動してタイトルシーンからゲームシーンへ遷移することができれば OK です。

○ローディング画面

タイトルからシーンに切り替わす際に新しいシーンの初期化処理をします。
シーンの初期化処理でモデルやエフェクトなどのリソースを読み込みますが、この読み込み処理が長いと画面が止まってしまいます。
ゲームでは画面が止まって見えることはできるだけ避けたいので、リソースの読み込みをしている間は Now Loading…などの演出を入れることが多いです。
この Now Loading…演出を入れるにはマルチスレッドによる並列処理が必要になってきます。

○マルチスレッド

マルチスレッドとは並列でプログラムを処理することによって処理時間を短縮するための機能です。
皆さんはマルチスレッドを使っておらず、シングルスレッドでのプログラムをしています。
イメージをつけるためにプログラムを「カレーを料理する」場合に置き換えて考えてみましょう。



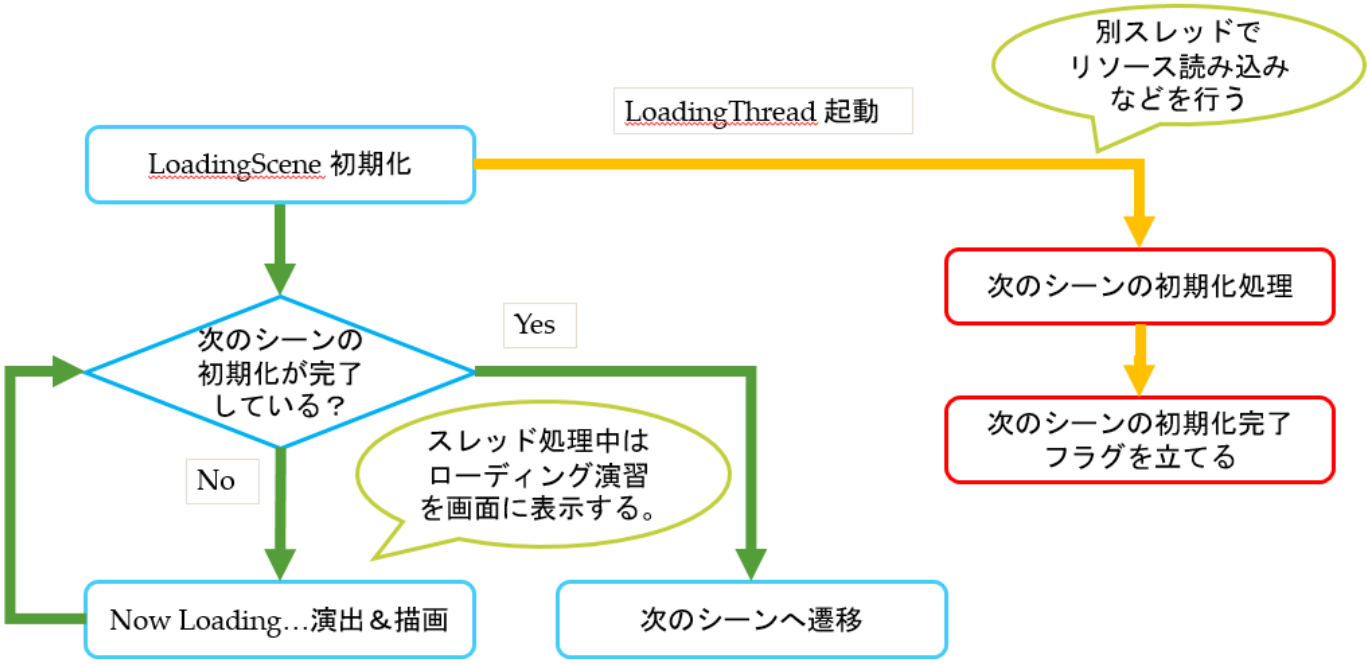
シングルスレッド　　マルチスレッド

カレーを料理する場合、「ご飯を炊く」、「カレーを煮込む」「お皿に盛る」の３工程が必要とします。
この場合、「ご飯を炊く」と「カレーを煮込む」は同時にできそうです。
「お皿に盛る」は「ご飯を炊く」と「カレーを煮込む」が完了していないとできないです。

シングルスレッドでも処理はうまくいくのですが時間が長くかかるので時間短縮なためにマルチスレッドで同時作業をしようという考えです。

○ローディング処理

今回はマルチスレッドを利用してローディングシーンを作成します。
ローディングシーンの処理の流れは下図のような感じです。



1.タイトルシーンなどと同じ手順でローディングシーンを実装します。
2.ローディングシーンはコンストラクタで次に遷移したいシーンを受け取ります。
3.シーンマネージャーにローディングシーンを渡し、シーンを切り替えます。
4.ローディングシーン初期化時に新しいスレッドを立ち上げ、次のシーンの初期化をします。
5.ローディングシーンは次のシーンの初期化が終わるまで Now Loading を表示し続けます。
6.次のシーンの初期化が終わるとローディングシーンは新しいシーンへ切り替えます。

この流れで実装すると今のシーン遷移システムをほぼ変更することなく実現できそうです。

○ローディングシーン

マルチスレッドとローディング処理の流れを理解したところでローディングシーンを実装しましょう。
SceneLoading.cpp と SceneLoading.h を作成し、下記プログラムコードを記述しましょう。

SceneLoading.h

```
#pragma once

#include "Graphics/Sprite.h"
#include "Scene.h"

// ローディングシーン
```

```
class SceneLoading : public Scene
{
public:
    SceneLoading() {}
    ~SceneLoading() override {}

    // 初期化
    void Initialize() override;

    // 終了化
    void Finalize() override;

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

    // 描画処理
    void Render() override;

private:
    Sprite* sprite = nullptr;
    float   angle = 0.0f;
};
```

Loading.cpp

```
#include "Graphics/Graphics.h"
#include "Input/Input.h"
#include "SceneLoading.h"
#include "SceneManager.h"

// 初期化
void SceneLoading::Initialize()
{
    // スプライト初期化
    sprite = new Sprite("Data/Sprite/LoadingIcon.png");
}

// 終了化
void SceneLoading::Finalize()
{
    // スプライト終了化
    if (sprite != nullptr)
    {
        delete sprite;
        sprite = nullptr;
    }
}

// 更新処理
void SceneLoading::Update(float elapsedTime)
{
    constexpr float speed = 180;
    angle += speed * elapsedTime;
}
```

```
// 描画処理
void SceneLoading::Render()
{
    Graphics& graphics = Graphics::Instance();
    ID3D11DeviceContext* dc = graphics.GetDeviceContext();
    ID3D11RenderTargetView* rtv = graphics.GetRenderTargetView();
    ID3D11DepthStencilView* dsv = graphics.GetDepthStencilView();

    // 画面クリア＆レンダーターゲット設定
    FLOAT color[] = { 0.0f, 0.0f, 0.5f, 1.0f };    // RGBA(0.0～1.0)
    dc->ClearRenderTargetView(rtv, color);
    dc->ClearDepthStencilView(dsv, D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);
    dc->OMSetRenderTargets(1, &rtv, dsv);

    // 2Dスプライト描画
    {
        // 画面右下にローディングアイコンを描画
        float screenWidth = static_cast<float>(graphics.GetScreenWidth());
        float screenHeight = static_cast<float>(graphics.GetScreenHeight());
        float textureWidth = static_cast<float>(sprite->GetTextureWidth());
        float textureHeight = static_cast<float>(sprite->GetTextureHeight());
        float positionX = screenWidth - textureWidth;
        float positionY = screenHeight - textureHeight;

        sprite->Render(dc,
            positionX, positionY, textureWidth, textureHeight,
            0, 0, textureWidth, textureHeight,
            angle,
            1, 1, 1, 1);
    }
}
```

画面右下にローディングアイコンが表示され、アイコンが回転しているだけのシーンです。
まず、このシーンが正しく表示されるか確認しましょう。
タイトル画面でボタンを押した際、ゲームシーンに切り替える代わりにローディングシーンに切り替えましょう。

SceneTitle.cpp

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

---省略---

// 更新処理
void SceneTitle::Update(float elapsedTime)
{
    GamePad& gamePad = Input::Instance().GetGamePad();

    // なにかボタンを押したらローディングシーンへ切り替え
    ---省略---
    if (gamePad.GetButtonDown() & anyButton)
    {
```

```
        SceneManager::Instance().ChangeScene(new SceneLoading);
    }
}

---省略---
```

実装出来たら実行確認をしてみましょう。
右下にローディングアイコンが表示され、ぐるぐる回っていれば OK です。

次はローディングシーンからゲームシーンへ遷移する処理を実装しましょう。
まず、ローディングシーン初期化処理時に新しくスレッドを作成します。
スレッドの作成には std::thread クラスを使用します。

先ほどのご飯とカレーを例にして、一般的なスレッドの使い方を見てみましょう。

```
#include <thread>

void CookingRiceThread()
{
    printf("ごはんを炊く");
}

void CookingCurryThread()
{
    printf("野菜を切る");
    printf("肉を炒める");
    printf("カレーを煮込む");
}

void main()
{
    // ご飯をつくるスレッドを立ち上げる
    std::thread cookingRiceThread(CookingRiceThread);
    // カレーをつくるスレッドを立ち上げる
    std::thread cookingCurryThread(CookingCurryThread);

    // ご飯が出来上がるまで待つ
    cookingRiceThread.join();
    // カレーが出来があるまで待つ
    cookingCurryThread.join();

    // ご飯とカレーが出来たので盛り付ける
    printf("盛り付けをする");
}
```

スレッド内で処理される関数。
関数の処理が終了すると
スレッドが消える

スレッドを立ち上げる時は
std::thread のコンストラクタに
スレッド内で処理する関数を渡す。

join()関数でスレッドが
終了するまで待つ

使い方がなんとなくわかったでしょうか。
上記の方法は main()関数内で別スレッド処理が終わるまで待っていますが、今回は毎フレーム Now Loading のアニメーションは更新し続けたいのでメインスレッドで join()関数を使って待つわけにはいきませんので今回は違った方法で実装していきます。
Scene.h を開き、下記プログラムコードを追記しましょう。

Scene.h

```
---省略---

// シーン
class Scene
{
public:
    ---省略---

    // 準備完了しているか
    bool IsReady() const { return ready; }

    // 準備完了設定
    void SetReady() { ready = true; }

private:
    bool    ready = false;
};
```

SceneLoading.h

```
---省略---

// ローディングシーン
class SceneLoading : public Scene
{
public:
    SceneLoading(Scene* nextScene) : nextScene(nextScene) {}

    ---省略---

private:
    // ローディングスレッド
    static void LoadingThread(SceneLoading* scene);

private:
    ---省略---
    Scene*  nextScene = nullptr;
};
```

コンストラクタの
初期化子リストで
次のシーンを設定

SceneLoading.cpp

```
#include <thread>
---省略---

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

    // スレッド開始
    std::thread thread(LoadingThread, this);
```

スレッドを立ち上げたあと
thread 変数が破棄されるまでに、
（今回では Initialize()関数が終了するとき）
join()関数でスレッドの終了を待つか、
detach()でスレッドの管理を放棄しなければならない。
上記のことをしないとエラーで止まります。
今回はメインスレッドを止めたくないので
デタッチでスレッドの管理を放棄しています。

```
    // スレッドの管理を放棄
    thread.detach();
}

---省略---

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

    // 次のシーンの準備が完了したらシーンを切り替える
    
}

---省略---

// ローディングスレッド
void SceneLoading::LoadingThread(SceneLoading* scene)
{
    // COM関連の初期化でスレッド毎に呼ぶ必要がある
    CoInitialize(nullptr);

    // 次のシーンの初期化を行う
    

    // スレッドが終わる前にCOM関連の終了化
    CoUninitialize();

    // 次のシーンの準備完了設定
    

}
```

今回は std::thread の join()と detach()を理解してもらうために detach()関数を使用しました。
detach()関数を使わずとも std::thread をポインタ変数で保持してポインタ変数が破棄されるタイミングで join()をする方法でも実現できます。

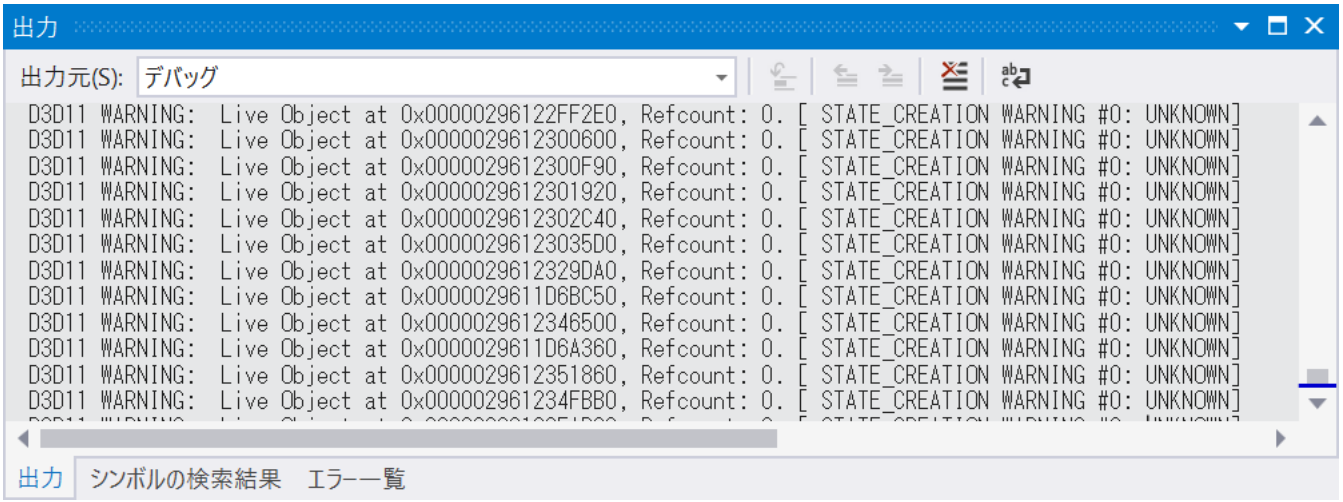
マルチスレッドでのローディングシーン切り替えの実装が完了したので、タイトルシーンからローディングシーン→ゲームシーンへ遷移するようにしましょう。

SceneTitle.cpp

```
// 更新処理
void SceneTitle::Update(float elapsedTime)
{
    GamePad& gamePad = Input::Instance().GetGamePad();
```

```
    // なにかボタンを押したらローディングシーンを挟んでゲームシーンへ切り替え
    ---省略---
    
}
```

実行してローディング画面を挟んだ後、ゲームシーンへ遷移していれば OK です。
しかし、ゲーム終了後、Visual Studio の出力ウインドウを見てみましょう。
下図のようにメモリリークが起きているはずです。



これはゲームシーンが意図せず２回初期化されているためです。
ローディングスレッドで１回とローディングシーンからゲームシーンへのシーン切り替え時に１回で計２回初期化されてしまっています。
シーン切り替え時に重複して初期化をしてしまわないように対応しましょう。

SceneManager.cpp

```
// シーン切り替え
void SceneManager::ChangeScene(Scene* scene)
{
    // 古いシーンを終了処理
    Clear();

    // 新しいシーンを設定
    currentScene = scene;

    // 未初期化の場合は初期化処理
    
}
```

実行してメモリリークがなくなれば OK です。

しかしまだ問題が残っています。何回もゲームを起動してローディングシーンを挟んでゲームシーンへのシーン遷移ををしていると稀にエラーでプログラムが止まることがあります。
これは排他制御ができていないからです。

○スレッド間の排他制御

マルチスレッドは複数のスレッドが同時に処理を実行することで処理時間を短縮するものと学習しました。
マルチスレッドでは必ず注意しなければならないことがあります。それが排他制御です。
マルチスレッドでは同時に同じ変数をアクセスするとエラーが起きることがあります。

例えば料理人 A と B がいたとします。
A と B はそれぞれ別の料理を作りますが、包丁は一つしかありません。
もし、A と B が同じ時に包丁を使おうとすると喧嘩が起きてエラーになるわけです。
なので A が包丁を使っている間は B は包丁を使ってはいけないというルールをつくる必要があります。
スレッド間の排他制御をするということはそのルールを作るということです。

今回、ローディングスレッドでゲームシーンの初期化を行っているわけですが、排他制御すべきものはないように思えるのですが、Effekseer を実装した人はエラーが発生する場合があります。

画面の表示の際、毎フレーム D3D11DeviceContext という変数を使っているのですが、Effekseer の初期化の際にどうやら同じ D3D11DeviceContext を同時アクセスしてしまう場合があります。
そのため、D3D11DeviceContext を同時にアクセスしないようにルールを作りましょう。
この排他制御をするためには std::mutex というクラスを使用します。

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

Graphics.h

```
#pragma once

#include <mutex>
---省略---

// グラフィックス
class Graphics
{
public:
    ---省略---

    // ミューテックス取得
    std::mutex& GetMutex() { return mutex; }

private:
```

DeviceContext を
同時アクセスさせないための
排他制御用オブジェクト

```
    ---省略---
    std::mutex              mutex;
};
```

Framework.cpp

```
---省略---

// 描画処理
void Framework::Render(float elapsedTime/*Elapsed seconds from last frame*/)
{
    // 別スレッド中にデバイスコンテキストが使われていた場合に
    // 同時アクセスしないように排他制御する
    std::lock_guard<std::mutex> lock(graphics.GetMutex());

    ---省略---
}
```

std::lock_guard 変数の
コンストラクタにミューテックス
を渡すことで、信号待ちをする

Effect.cpp

```
---省略---

// コンストラクタ
Effect::Effect(const char* filename)
{
    // エフェクトを読み込みする前にロックする
    // ※マルチスレッドでEffectを作成するとDeviceContextを同時アクセスして
    //   フリーズする可能性があるので排他制御する
    std::lock_guard<std::mutex> lock(Graphics::Instance().GetMutex());

    ---省略---
}
```

Framework::Render()が実行されている間は DeviceContext が使わています。
Effect のコンストラクタ内での Effekseer のリソース読み込みで DeviceContext が使われます。
それぞれの関数のはじめにミューテックスによる鍵をかけることで、片方の関数が実行中の間は処理が同時に実行されないように待つというルールを作りました。

これで現時点での排他制御ができました。
今後もしスレッド間で同じ変数をアクセスしなければならないという事態が発生した場合は今回のように対応しましょう。

お疲れさまでした。