内容

[**第一章　プログラムに慣れよう** 4](#_Toc510965410)

[**Lesson\_00\_0 tkEngne2** 4](#_Toc510965411)

[**Lesson\_00\_1 IGameObject** 5](#_Toc510965412)

[**Lesson\_00\_2 クラスを追加してみよう** 5](#_Toc510965413)

[**Lesson\_00\_3 キャラクターを表示しよう** 8](#_Toc510965414)

[**Lesson\_00\_4 まとめ** 10](#_Toc510965415)

[**Lesson\_00\_05 中間テスト** 10](#_Toc510965416)

[**Lesson\_01\_1 キャラクターを動かしてみよう。** 11](#_Toc510965417)

[**Lesson\_01\_2 パッドの入力でキャラを動かしてみよう。** 12](#_Toc510965418)

[**Lesson\_01\_3 キャラクターをジャンプさせてみよう。** 12](#_Toc510965419)

[**Lesson\_01\_4 キャラクターに重力を加えてみよう。** 13](#_Toc510965420)

[**Lesson\_01\_5 中間テスト** 13](#_Toc510965421)

[**Lesson\_02\_1 キャラクターを回転させてみよう。** 14](#_Toc510965422)

[**Lesson\_02\_2 パッドの入力でキャラクターを回転させてみよう。** 15](#_Toc510965423)

[**Lesson\_02\_3 中間テスト** 16](#_Toc510965424)

[**Lesson\_03\_01 走るアニメーションを再生してみよう。** 17](#_Toc510965425)

[**Lesson\_03\_02 ジャンプアニメーションを再生してみよう。** 17](#_Toc510965426)

[**Lesson\_03\_03 中間テスト** 18](#_Toc510965427)

[**Lesson\_04 カメラを動かしてみよう。** 19](#_Toc510965428)

[**Lesson\_04\_01　注視点を動かそう** 19](#_Toc510965429)

[**Lesson\_04\_02　視点を動かそう** 20](#_Toc510965430)

[**Lesson\_04\_03 近平面と遠平面** 21](#_Toc510965431)

[**Lesson\_04\_04　画角** 22](#_Toc510965432)

[**Lesson\_04\_05　まとめ** 23](#_Toc510965433)

[**Lesson\_04\_06　中間テスト** 23](#_Toc510965434)

[**Lesson\_05\_01 背景とキャラクターのあたり判定をとれるようにしてみよう。** 24](#_Toc510965435)

[**Lesson\_05\_01 CPhysicsStaticObject** 24](#_Toc510965436)

[**Lesson\_05\_02 CCharacterController** 25](#_Toc510965437)

[**Lesson\_05\_03 まとめ** 25](#_Toc510965438)

[**Lesson\_05\_04 中間テスト** 26](#_Toc510965439)

[**Lesson\_06\_01 ☆をゲットできるようにしてみよう。** 27](#_Toc510965440)

[**Lesson\_06\_02 ☆を表示してみよう。** 27](#_Toc510965441)

[**Lesson\_06\_03 ☆を回転させてみよう。** 28](#_Toc510965442)

[**Lesson\_06\_04 ☆とプレイヤーの２点間の距離を計算して、☆をゲットしよう。** 28](#_Toc510965443)

[**Lesson\_06\_05 中間テスト** 29](#_Toc510965444)

[**Lesson\_07\_01 BGMを流そう** 30](#_Toc510965445)

[**Lesson\_07\_02 コインを取得したときに音を鳴らそう。** 32](#_Toc510965446)

[**Lesson\_07\_03 中間テスト** 32](#_Toc510965447)

[**Lesson\_08\_00 自機を表示してみよう。** 33](#_Toc510965448)

[**Lesson\_08\_01 wWinMain関数** 33](#_Toc510965449)

[**Lesson\_08\_02 AirPlane.fbxをVisualStudioに追加** 34](#_Toc510965450)

[**Lesson\_08\_03 プレイヤーのクラスの追加** 35](#_Toc510965451)

[**Lesson\_08\_04 プレイヤーのクラスの実装** 37](#_Toc510965452)

[**Lesson\_08\_05 Playerクラスのインスタンス化** 38](#_Toc510965453)

[**Lesson\_08\_06 中間テスト** 39](#_Toc510965454)

[**Lesson\_09\_00 自機を動かしてみよう。** 40](#_Toc510965455)

[**Lesson\_09\_01 Update関数を追加する。** 40](#_Toc510965456)

[**Lesson\_09\_02 Update関数に自機を旋回させる処理を追加する。** 41](#_Toc510965457)

[**Lesson\_09\_3 画面外に移動できないようにしよう。** 42](#_Toc510965458)

[**Lesson\_09\_4 中間テスト** 42](#_Toc510965459)

[**Lesson10\_0 弾を撃てるようにしてみよう** 44](#_Toc510965460)

[**Lesson 10\_1 Bulletクラスを作成する。** 44](#_Toc510965461)

[**Lesson 10\_2 Bulletクラスのインスタンス化** 45](#_Toc510965462)

[**Lesson\_10\_3 弾丸を自機の場所から発生するようにしよう** 46](#_Toc510965463)

[**Lesson\_10\_4 弾丸を動かそう。** 47](#_Toc510965464)

[**Lesson\_10\_5　弾丸の発射にインターバルを設定しよう。** 48](#_Toc510965465)

[**Lesson\_10\_6 弾丸に寿命を設定しよう** 48](#_Toc510965466)

[**Lesson\_10\_7　中間テスト** 49](#_Toc510965467)

[**Lesson\_11\_1 敵機の実装をしよう** 50](#_Toc510965468)

[**Lesson\_11\_1 Enemyクラスを作成する。** 50](#_Toc510965469)

[**Lesson\_11\_2 Enemy生成器を作成しよう** 51](#_Toc510965470)

[**Lesson\_11\_3 敵機の発生場所をランダムにしてみる。** 53](#_Toc510965471)

[**Lesson\_11\_4 中間テスト** 53](#_Toc510965472)

[**Lesson\_12\_1 敵機も弾丸を撃てるようにしてみよう** 54](#_Toc510965473)

[**Lesson\_12\_2 プレイヤーと敵機の弾丸に名前を付けよう。** 55](#_Toc510965474)

[**Lesson\_12\_3　2点間の距離を計算** 55](#_Toc510965475)

[**Lesson 12\_4 中間テスト** 56](#_Toc510965476)

[**Lesson 13\_1 敵機が死亡する時に爆発のＳＥを鳴らそう。** 57](#_Toc510965477)

[**Lesson 13\_2 爆発のエフェクトを発生させよう。** 58](#_Toc510965478)

[**Lesson 13\_3 中間テスト** 60](#_Toc510965479)

[**Lesson 14\_1 GameOverの2Dを表示しよう** 61](#_Toc510965480)

[**Lesson 14\_2 背景を表示してみよう。** 63](#_Toc510965481)

[**Lesson 14\_3 中間テスト** 65](#_Toc510965482)

[**Lesson 15\_1 ゲームで使用されるデザインパターン** 66](#_Toc510965483)

[**Lesson 15\_2 ゲームループ** 66](#_Toc510965484)

[**Lesson 15\_3 画面のリフレッシュレート** 67](#_Toc510965485)

[**Lesson 15\_4 UpdateMethodパターン** 67](#_Toc510965486)

[**Lesson 16\_1 ゲームオブジェクトマネージャー** 70](#_Toc510965487)

[**Lesson 16\_2 Update Method パターン** 70](#_Toc510965488)

[**Lesson 16\_3 IGameObject　インターフェースクラス** 70](#_Toc510965489)

[**Lesson 16\_4 NewGO インスタンスの生成と登録** 70](#_Toc510965490)

[**Lesson 16\_5 FindGO インスタンスの検索** 71](#_Toc510965491)

[**Lesson 16\_6 QueryGOs ゲームオブジェクトに対してクエリ(問い合わせ)を行う。** 71](#_Toc510965492)

[**16\_7 DeleteGO インスタンスの破棄** 72](#_Toc510965493)

[**Lesson 16\_8 中間テスト** 72](#_Toc510965494)

[**Lesson 16\_1 シーン切り替えを行おう。** 73](#_Toc510965495)

[**Lesson 16\_2 スーパーマリオの場合** 73](#_Toc510965496)

[**Lesson 16\_3 安全かつ完璧にシーンを切り替える** 74](#_Toc510965497)

[**Lesson 16\_4 グローバル変数を利用したシーン切り替え** 74](#_Toc510965498)

[**Lesson 16\_5ポリモーフィズムを活用したシーン切り替え** 75](#_Toc510965499)

# **第一章　プログラムに慣れよう**

この章では、短い簡単はハンズオンを繰り返して、プログラムを書くということに慣れていきましょう。

# **Lesson\_00\_0 tkEngne2**

　今、多くのゲームは大きく分けて、下記の３つのエンジンを用いて開発が行われています。

・Unity

　　スマートフォンなどのゲームでよく使われている。

・UnrealEngine4

　　PlayStation4、Nintendo Switchなどのコンシュマーゲームでよく使われている。

**・各ゲーム会社内製のゲームエンジン**

**自社でエンジン開発が行える、技術力と資金力のある会社で使われる。FF15、モンスターハンター、アンチャーテッド、大鷲のトリコ、ダークソウル、Call of dutyなど多くのAAAタイトルで使われている。**

　この授業ではtkEngine2という河原電子ビジネス専門学校の内製のゲームエンジンを用いて勉強を進めていきます。このエンジンはC++で開発することができ、多くのゲーム会社が求めている、よりコンピューターに近い、低レベルな知識に簡単にアクセスできるようになっており、就職作品制作に向けた基礎力を蓄えていくことを目的としています。

# **Lesson\_00\_1 IGameObject**

ゲームはパラパラ漫画のようなもので、１秒間に30枚～60枚の絵を作成して、それを連続して再生することで、キャラクターが動いているように見せています。多くのゲームでは１秒間に30回～60回呼ばれる処理の中にゲームのプログラムを書いていきます。

　tkEngine2ではIGamObjectを継承したクラスを作成して、そのインスタンスを生成することで、下記の機能が手に入ります。

# **Lesson\_00\_2 クラスを追加してみよう**

　では、話ばかりでは退屈だと思いますので、実際に手を動かして、キャラクターを表示するプログラムを書いてみましょう。

Lesson00\_Question/Game/Game.slnをダブルクリックして、VisualStudioを立ち上げてください。

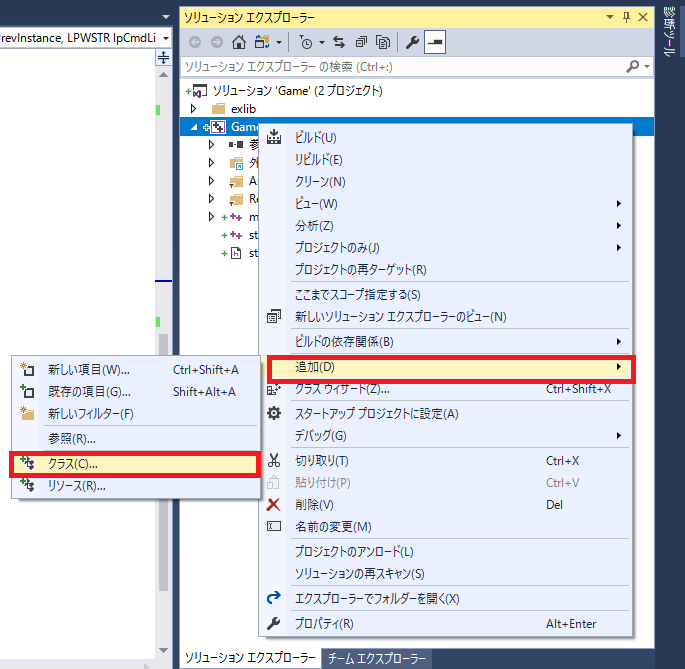


VisualStudioが立ちあがったら、Ctrl+F5を押してゲームを起動してみてください。すると下記のような灰色のウィンドウが起動したと思います。

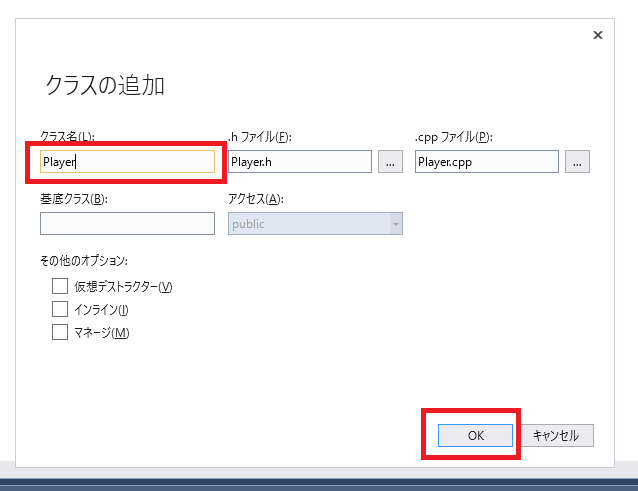


では、プログラムを追加して、キャラクターを表示してみましょう。まずはVisualStudioにPlayerクラスを追加しましょう。

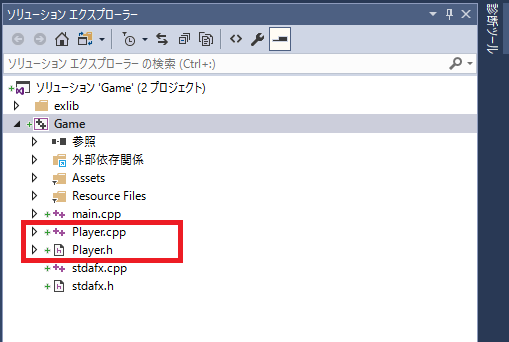
ソリューションエクスプローラーからGameプロジェクトを選択して、右クリックでポップアップするメニューから追加―＞クラスを選択してください。



　すると下記のようなダイアログボックスが表示されるので、クラス名にPlayerを入力して、OKボタンを押してください。



　すると、ソリューションエクスプローラーに下記の二つのファイルが追加されたと思います。



では、追加されたファイルにプログラムを追加していって、キャラクターを表示してみましょう。

# **Lesson\_00\_3 キャラクターを表示しよう**

　Player.hをクリックして、下記のようにコードを記述してください。網影の部分が追加されるコードです。

|  |
| --- |
| #pragma once  class Player **: public IGameObject**  {  public:  Player();  ~Player();  **prefab::CSkinModelRender\* skinModelRenderer;**  }; |

続いて、Player.cppを開いて下記のコードを追加してください。網掛けの部分が追加されたプログラムです。

|  |
| --- |
| #include "stdafx.h"  #include "Player.h"  Player::Player()  {  **//スキンモデルレンダラーを作成する。**  **skinModelRenderer = NewGO<prefab::CSkinModelRender>(0);**  **skinModelRenderer->Init(L"modelData/unityChan.cmo");**  }  Player::~Player()  {  } |

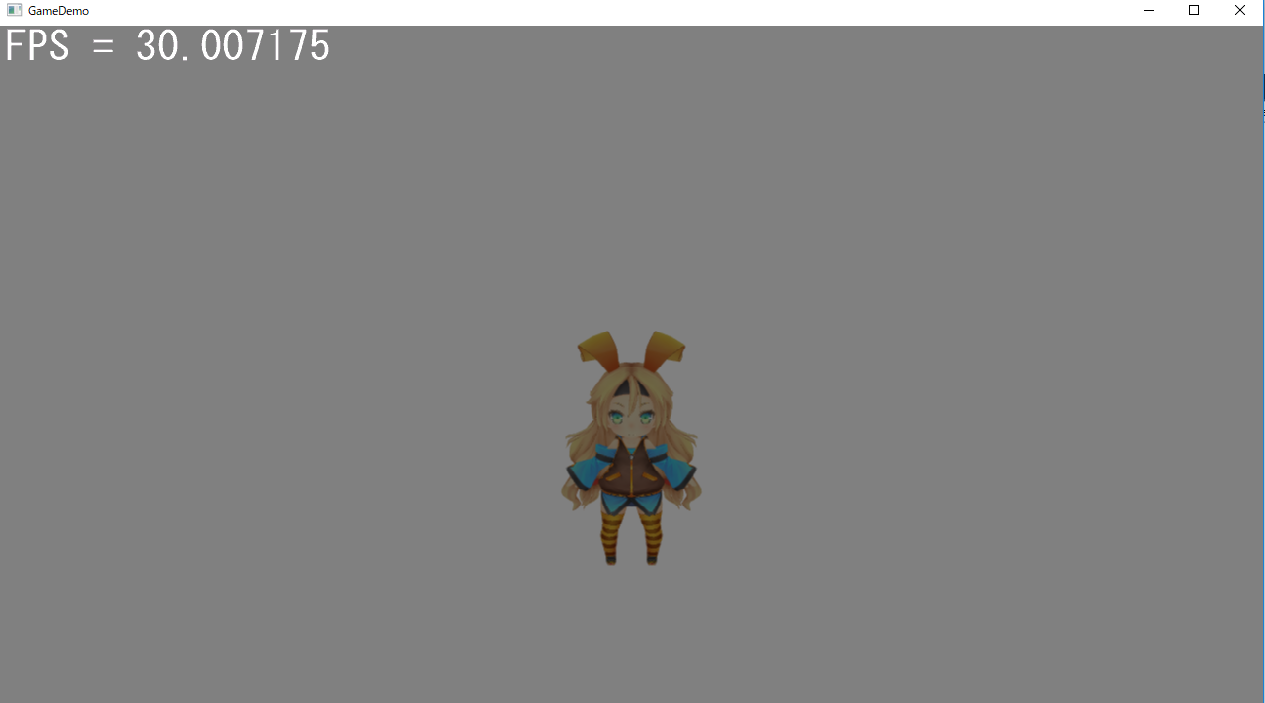
これでプレイヤークラスの作成は完了しました。しかし、クラスは設計図のようなもので、作成しただけでは何の意味もありません。自動車は設計図を用いて、車は製造されますよね？クラスも同じで設計図をもとにインスタンスを作成する必要があります。

では、main.cppの「HandsOn 1 プレイヤークラスの機能を使いたいのでヘッダーファイルをインクルードする」と書かれている箇所と、「HandsOn 2 プレイヤーのインスタンスを作成」と書かれている箇所に、網掛けになっているコードを追加してください。

|  |
| --- |
| /\*!  \*@brief main.cpp  \*/  #include "stdafx.h"  //HnadsOn 1 プレイヤークラスの機能を使いたいのでヘッダーファイルをインクルードする。  **#include "Player.h"** |

|  |
| --- |
| //エンジンを初期化。  if (Engine().Init(initParam) == true) {  //カメラを設定。  MainCamera().SetTarget({ 0.0f, 100.0f, 0.0f });  MainCamera().SetNear(0.1f);  MainCamera().SetFar(3000.0f);  MainCamera().SetPosition({ 0.0f, 100.0f, 300.0f });  MainCamera().Update();  //HandsOn-2 プレイヤーのインスタンスを作成。  **NewGO<Player>(0);**  //ゲームループを実行。  Engine().RunGameLoop();  } |

ここまでできた人はCtrl+F5を押してゲームを起動してみください。すると下記のように女の子のキャラクターが表示されたと思います。



# **Lesson\_00\_4 まとめ**

　いきなり色々な情報が出てきて戸惑ったかもしれませんが、Lesson\_00で押さえておいて欲しいのは下記の２点です。

**①　ゲームは１秒間に３０枚～６０枚の絵を作成して、それを連続して表示することで動いているように見せている。**

**②　IGameObjectとかいうものを使うと、なんかいい感じに①の機能が使える。**

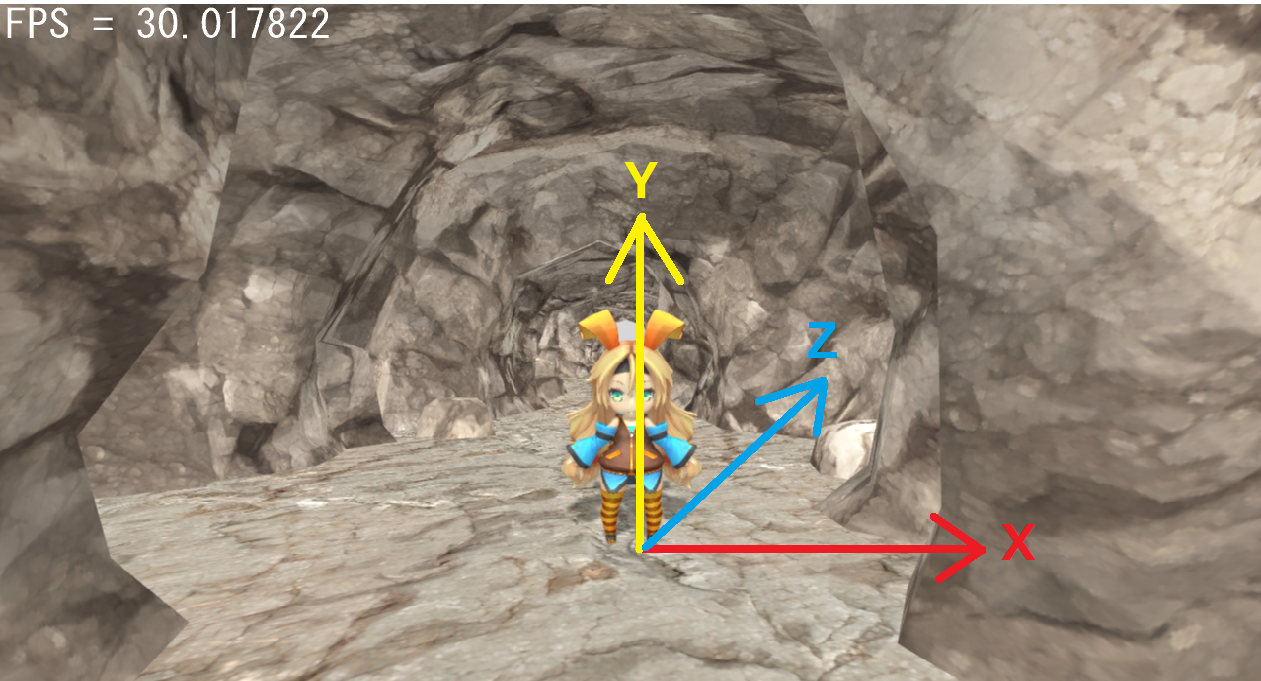
# **Lesson\_00\_05 中間テスト**

　下記のＵＲＬのテストを行いなさい。

<https://goo.gl/forms/bMzSO5NqoUja8Hir2>

# **Lesson\_01\_1 キャラクターを動かしてみよう。**

　3Dゲームでキャラクターの位置は下記の図のように３の軸上の位置で表されます。



この絵のキャラクターはX軸上に0.0、Y軸上に0.0、Z軸上に0.0の場所にいます。では、このキャラクターを少しずつX軸上で動かすプログラムを書いてみましょう。

サンプルプログラムに次のコードを記述してみてください。

Lesson01\_Question/Player.cpp

|  |
| --- |
| void Player::Update()  {  //Question 1 キャラを左右に動かしてみよう。  **position.x += 10.0f;**  //Question 2 キャラを前後に動かしてみよう。  //Question 3 キャラクタをジャンプさせてみよう。    skinModelRender->SetPosition(position);  } |

positionはキャラクターの座標を表す変数です。X軸方向に値を加減算すると、キャラクターはX軸上を移動します。

# **Lesson\_01\_2 パッドの入力でキャラを動かしてみよう。**

Lesson\_01\_1の実装ではキャラクターは常に移動を行っていました。多くのゲームではゲームパッドでの入力によってキャラクターが移動すると思います。このチャプターではゲームパッドの入力によって、キャラクターを動かせるようにしてみましょう。

**2.1 if文**

パッドの入力があった場合だけキャラクターを動かしたい場合はif文を使用します。「Question 1キャラを左右に動かしてみよう。」と記述されている箇所に次のコードを記述してみてください。

Lesson01\_Question/Player.cpp

|  |
| --- |
| //Question 1 キャラを左右に動かしてみよう。  if (Pad(0).IsPress(enButtonRight)) { //もしもゲームパッドの右ボタンが押されていたら。  position.x += 10.0f;  }  if (Pad(0).IsPress(enButtonLeft)) { //もしもゲームパッドの左ボタンが推されていたら。  position.x -= 10.0f;  } |

これでパッドの左右のボタンが入力されるとキャラクターが左右に移動するようになりま

した。

# **Lesson\_01\_3 キャラクターをジャンプさせてみよう。**

　パッドのAボタンが押されたら、キャラクターをジャンプさせてみましょう。ジャンプするということはキャラクターの上方向(y方向)に力が加わるということです。今回はySpeedという変数に上方向の速度を設定してみましょう。「Question 2 キャラクターをジャンプさせてみよう」と書かれているところに、次のコードを記述してください。

Lesson01\_Question/ Player.cpp

|  |
| --- |
| //Question 2 キャラクターをジャンプさせてみよう。  if (Pad(0).IsTrigger(enButtonA)) {  ySpeed = 20.0f; //ｙ方向の速度を設定する。  } |

どうでしょうか、ジャンプできたのではないでしょうか。しかしキャラクターはジャンプしたまま落ちてこなかったのではないでしょうか。きっと、キャラクターはあのまま大気圏を突破して、宇宙の塵となったことでしょう。

# **Lesson\_01\_4 キャラクターに重力を加えてみよう。**

　私たち人間がジャンプをしても大気圏を突破せずに、地面に立つことができるのは地球の重力によって引っ張られているためです。Lesson\_01\_3でキャラクターが大気圏を突破してしまったのは、重力が加えられていなかったためです。「Question 3 重力を加えてみよう。」と書かれているところに、次のコードを記述してみてください。

Lesson01/Question/ Player.cpp

|  |
| --- |
| //Question 3 重力を加えてみよう。  ySpeed -= 1.0f; |

どうでしょうか？これでキャラクターは正しく、地面に落下してきたと思います。

# **Lesson\_01\_5 中間テスト**

下記のURLのテストを行いなさい。

<https://goo.gl/forms/MDpXrUeRdetqEQUq1>

# **Lesson\_02\_1 キャラクターを回転させてみよう。**

　Lesson\_01ではパッドの入力でキャラクターを移動させました。Lesson\_02ではパッドの入力でキャラクターを回転させてみましょう。

1. **クォータニオン(四元数)**

3Dモデルの回転を表現するにはいくつか手法があるのですが、今回は3Dゲームで主流となっているクォータニオンを使用した回転の表現について見ていきましょう。この授業は数学の授業ではないのでクォータニオンの数学的な定義や証明は行いません。クォータニオンをゲームでどのように使用するのかという点に注視して説明を行います。

回転の表現にクォータニオンを使用する理由の大きな理由の一つに任意の軸周りの回転を簡単に扱うことができるというものがあります。では任意の軸周りの回転とはどのようなものなのか見ていきましょう。例えば、下の図のようにキャラクターを回転させたい場合は、Y軸周りに90度回転させることになります。

では次はX軸周りに回転させてみましょう。

では実際にプログラムを記述して、キャラクターを回転させてみましょう。下記のコードをサンプルプログラムに記述してください。

Lesson02\_Question/Player.cpp

|  |
| --- |
| void Player::Rotation()  {  //Question 1 キャラクタを右に向かせてみよう。  　rotation.SetRotationDeg(CVector3::AxisY, 90.0f);  //Question 2 キャラクタを左に向かせてみよう。  //Question 3 キャラクタを奥に向かせてみよう。  //Question 4 キャラクタを手前に向かせてみよう。  } |

キャラクターが右に90度回転したと思います。

# **Lesson\_02\_2 パッドの入力でキャラクターを回転させてみよう。**

　では、続いてパッドの右ボタンが押されたらキャラクターが右を向くようにしてみましょう。Lesson\_01\_2を思い出してください。パッドの右ボタンが押されたら？という処理を書きたい場合はif文を使うのでしたね？では、「Question 1キャラクターを右に向かせてみよう。」と記述されている箇所に次のコードを記述してみてください。

Lesson02\_Question/Player.cpp

|  |
| --- |
| if (Pad(0).IsPress(enButtonRight)) { //もしもゲームパッドの右ボタンが押されていたら  rotation.SetRotationDeg(CVector3::AxisY, 90.0f); //90度回す。  } |

どうでしょう、正しく回転できたでしょうか？

|  |
| --- |
| **Tips**  今まで皆さんが習ってきた角度は0°～180°を使ったものだったと思います。この角度のDegree(デグリー)単位と呼ばれるものなのですが、実はプログラムの世界では、このDegree単位はあまり使われなくて、主にRadian(ラジアン)単位が使用されます。Radian単位は0°～180°を0～π(円周率。3.14……。)で表します。  Degree単位は下記のようにRadian単位に変換されます。  0° 　= 0  90°　= 0.5×π  180° = π  360° = 2×π  今回のキャラクターの角度の指定もRadian単位で行うことができ、下記のようなコードで同じように回転させることができます。  *//Y軸周りに90°回す。*  *rotation.SetRotation(CVector3::Axis, 0.5 \* CMath::PI);*  使用する関数がSetRotationDegだとDegree単位、SetRotationだとRadian単位になります。  なぜプログラムの世界ではRadian単位を使うのか気になると思いますが、詳しいことはここでは説明しませんが、その方が色々な計算がシンプルになり、プログラムが簡単になるからです。 |

# **Lesson\_02\_3 中間テスト**

下記のURLのテストを行いなさい。

<https://goo.gl/forms/b36mqyAqVsTrxilL2>

# **Lesson\_03\_01 走るアニメーションを再生してみよう。**

　Lesson\_02までの実装でキャラクターが移動して、進行方向を向いて回転するようになりました。しかし、キャラクターが立ちのままで移動しているため不自然です。ではプログラムを改造して、走るアニメーションを再生できるようにしてみましょう。「HandsOn走りアニメーションを再生してみよう。」と記述されている箇所に次のコードを記述してみてください。

Lesson03\_Question /Player.cpp

|  |
| --- |
| //HandsOn 1 走りアニメーションを再生してみよう。  if (Pad(0).IsPress(enButtonUp)) {  //ゲームパッドの上ボタンが押されているなら。  //走るアニメーションを再生する。  skinModelRender->PlayAnimation(enAnimationClip\_run);  }  else if (Pad(0).IsPress(enButtonDown)) {  //ゲームパッドの下ボタンが押されているなら。  //走るアニメーションを再生する。  skinModelRender->PlayAnimation(enAnimationClip\_run);  }  else {  //何も入力されていなければ立ちアニメーションを再生する。  skinModelRender->PlayAnimation(enAnimationClip\_idle);  } |

# **Lesson\_03\_02 ジャンプアニメーションを再生してみよう。**

　今度はＡボタンが押されたら、ジャンプアニメーションを再生できるようにしてみましょう。「Question 2ジャンプアニメーションを再生してみよう。」と記述されている箇所に次のコードを記述してみてください。

Lesson03\_Question/Player.cpp

|  |
| --- |
| //HandsOn 2 ジャンプアニメーションを再生してみよう。  if (Pad(0).IsTrigger(enButtonA)) {  skinModelRender->PlayAnimation(enAnimationClip\_jump);  } |

どうでしょうか？正しくジャンプできましたか？キャラクターが一瞬ピクッと動きはしますが、ジャンプアニメーションは流れていないと思います。では、なぜこのようなことが起きてしまっているのか見ていきましょう。

|  |
| --- |
| //HandsOn 1 走りアニメーションを再生してみよう。  if (Pad(0).IsPress(enButtonUp)) {  //ゲームパッドの上ボタンが押されているなら。  //走るアニメーションを再生する。  skinModelRender->PlayAnimation(enAnimationClip\_run);  }  else if (Pad(0).IsPress(enButtonDown)) {  //ゲームパッドの下ボタンが押されているなら。  //走るアニメーションを再生する。  skinModelRender->PlayAnimation(enAnimationClip\_run);  }  else {  //何も入力されていなければ立ちアニメーションを再生する。  skinModelRender->PlayAnimation(enAnimationClip\_idle);  }  //HandsOn 2 ジャンプアニメーションを再生してみよう。  10フレーム目にＡボタンが押されてジャンプアニメーションが再生される。  if (Pad(0).IsTrigger(enButtonA)) {  skinModelRender->PlayAnimation(enAnimationClip\_jump);  } |

**11フレーム目にこのプログラムが実行されるので立ちアニメーションが再生されてしまう！！！**

ゲームというのはパラパラアニメのようなものであるため、このプログラムはゲームが実行されている間、定期的に実行されます。そのため、10フレーム目にAボタンが押されてジャンプアニメーションを流したとしても、次のフレームではskinModelRender->PlayAnimation(enAnimationClip\_idle);が実行されてしまい、立ちアニメーションに戻ってしまいます。この問題を解決するためにはジャンプ中は立アニメーションを再生できないようにしてやる必要があります。ではプログラムを次のように改造してみてください。

|  |
| --- |
| void Player::AnimationControl()  {  //Question 1　走りアニメーションを再生してみよう。  **if (isJump == false) { //ジャンプ中でなければ**  if (Pad(0).IsPress(enButtonUp)) { //ゲームパッドの上ボタンが押されているなら。  PlayAnimation(AnimationRun); //走るアニメーションを再生する。  }  else {  //何も押されていなければ立ちアニメーションを再生する。  PlayAnimation(AnimationStand);  }  }  //Question 2 ジャンプアニメーションを再生してみよう。  if (Pad(0).IsTrigger(enButtonA)) {  PlayAnimation(AnimationJump);  **isJump = true; //ジャンプ中のフラグを立てる。**  }  } |

今回はフラグを利用した実装を行いました。isJumpはジャンプ中かどうかを調べるためのものです。isJumpはbool型のメンバ変数でtrueとfalseの２値を取ります。isJumpがtrueになったらジャンプ中だと判断しています。

# **Lesson\_03\_03 中間テスト**

下記のURLのテストを行いなさい。

<https://goo.gl/forms/gFSxocgsHJLA8sUt2>

# **Lesson\_04 カメラを動かしてみよう。**

3Dゲームにはカメラという概念があります。テレビ番組のカメラマンをイメージすると分かりやすいかと思います。Lesson\_01の授業でプレイヤーが移動できるようになりましたが、移動し続けるとプレイヤーが画面外などに出て行ってしまい消えてしまっていたと思います。これはカメラがプレイヤーを追いかけていないためです。Lesson\_05ではプレイヤーを追いかけるカメラを実装していきましょう。

# **Lesson\_04\_01　注視点を動かそう**

　カメラには注視点というものがあります。これはカメラがどこを見ているのかというものです。例えば、プレイヤー追従のカメラであれば、カメラはプレイヤーを見ている必要がありますよね？プレイヤーのゲーム空間上での位置はX,Y,Zの3軸の座標で表現されていたことを思い出して下さい。つまり、カメラにその座標を教えてやれば上手くいきそうな気がします。Lesson04\_Question/Game/GameCamera.cppを開いて「HandsOn 1 注視点を動かす。」と記述されている箇所に次のプログラムを追加してください。

|  |
| --- |
| //HandsOn 1 注視点を動かす。  CVector3 cameraTarget;  cameraTarget.x = pl->position.x;  cameraTarget.y = pl->position.y + 100.0f; //プレイヤーのちょっと上にする。  cameraTarget.z = pl->position.z;  //注視点をカメラに伝える。  MainCamera().SetTarget(cameraTarget); |

常にカメラがプレイヤーを中心に収めるようになったと思います。カメラの注視点のY座標がプレイヤーの座標のちょっと上になっていることに注意してください。プレイヤーの座標は足元になっているため、Y座標をちょっと上げてやらないと窮屈な見え方になってしまいます。

# **Lesson\_04\_02　視点を動かそう**

　注視点を設定したことで、常にカメラの中心にプレイヤーが入るようになりました。しかしプレイヤーが奥に進んでいくと、プレイヤーがどんどん小さくなっていってしまいます。これはカメラがプレイヤーを見ているだけで、プレイヤーを追いかけていないためです。カメラには視点というものがあります。テレビ番組のカメラマンの座標だと思ってください。

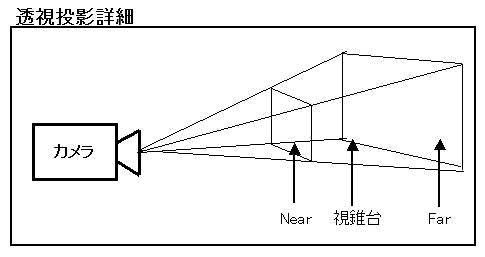
では「HandsOn 2 視点を動かす。」と記述されている箇所に次のプログラムを記述してください。

|  |
| --- |
| //HandsOn 2 視点を動かす。  //視点は注視点からＹ方向に＋４００、Ｚ方向に‐６００の場所にする。  CVector3 cameraPos;  cameraPos = cameraTarget;  cameraPos.y += 400.0f;  cameraPos.z -= 600.0f;  //視点をカメラに伝える。  MainCamera().SetPosition(cameraPos); |

カメラがプレイヤーを追従するようになったと思います。

# **Lesson\_04\_03 近平面と遠平面**

　3Dゲームのカメラには近平面と、遠平面という概念があります。これはカメラ空間のどこから、どこまでを画面に映すかを決めるためのパラメータです。



コンピュータのメモリ、パフォーマンスは有限であるため、画面に映す範囲を制限する必要があります。そのため、近平面と遠平面というパラメータが必要になります。

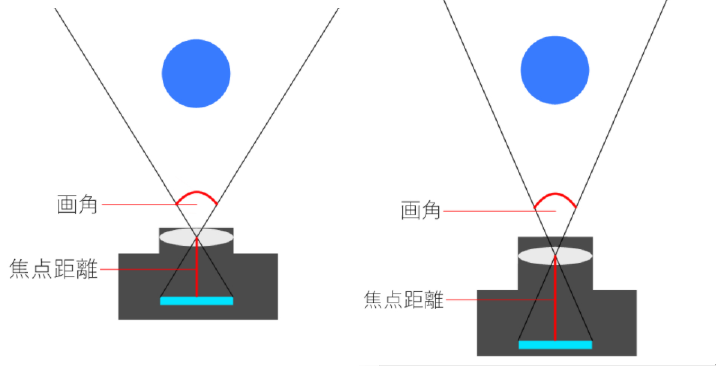
近平面(ニアクリップ)から遠平面(ファークリップ)の間にある、オブジェクトがスクリーンに表示されます。

では、「HandsOn 3 遠平面を設定する。」と書かれている箇所に下記のコードを追加して、公園が全て表示されるようにしてみましょう。

|  |
| --- |
| //HandsOn 3 遠平面を設定する。  MainCamera().SetFar(30000.0f); |

# **Lesson\_04\_04　画角**

　画角も近平面と遠平面と同様に、下記の図のようにカメラが映す範囲を指定するものです。



画角を狭めると、オブジェクトが大きく見えるため、ズームアップしているような効果が得られます。逆に広げるとズームアウトしているように見えると思います。

では、「HandsOn 4 画角を変更してみる」と書かれている箇所に下記のコードを追加してみましょう。

|  |
| --- |
| //HandsOn 4 画角を変更してみる。  //画角を120度にしてみる。  MainCamera().SetViewAngle(CMath::DegToRad(120.0f)); |

# **Lesson\_04\_05　まとめ**

　カメラの注視点、視点の関係は下の図のようになっています。

注視点

視点

# **Lesson\_04\_06　中間テスト**

下記のURLのテストを行いなさい。

<https://goo.gl/forms/SnVr53xf8BPXwNGe2>

# **Lesson\_05\_01 背景とキャラクターのあたり判定をとれるようにしてみよう。**

　Lesson\_05では、キャラクターと背景とあたり判定をとれるようにしてみようと思います。tkEngine2で背景とあたり判定をとれるようにするためには下記の２点を実装する必要があります。

① 背景のあたり判定データを物理世界に登録するためにCPhysicsStaticObjectを使用する。

② キャラクターの移動にCCharacterControllerを使用するようにする。

# **Lesson\_05\_01 CPhysicsStaticObject**

キャラクターと背景のあたり判定を行うには、背景がどのような形状なのか？ということを物理エンジンに教えてやる必要があります。これを行ってくれるクラスがCPhysicsStaticObjectです。このクラスは３Ｄモデルデータから、あたり判定用のデータ(以下Colliderと呼称します)を作成してくれます。

では、Lesson05\_Question/Game/BackGround.hの「HandsOn 1 BackgroudクラスにCPhysicsStaticObjectのインスタンスを保持させる」とコメントが書かれている箇所に、下記のコードを追加してください。

|  |
| --- |
| //HandsOn 1 BackgroudクラスにCPhysicsStaticObjectのインスタンスを保持させる。  CPhysicsStaticObject m\_physicsStaticObject; //静的物理オブジェクト。 |

これでBackGroundクラスにCPhysicsStaticObjectのインスタンスを保持させることができるようになりました。続いて、このインスタンスを３Ｄモデルデータを使って初期化します。「Lesson05\_Question/Game/BackGround.cppのHandsOn 2 CPhysicsStaticObjectの初期化」と書かれている箇所に下記のコードを追加してください。

|  |
| --- |
| //HandsOn 2 CPhysicsStaticObjectの初期化  m\_physicsStaticObject.CreateMeshObject(skinModelRender, pos, qRot); |

# **Lesson\_05\_02 CCharacterController**

Lesson\_05\_01で背景のColliderを物理ワールドに登録しました。あとはキャラクターの移動処理にCCharacterControllerを使うようにすると、あたり判定を取ることができます。

Lesson05\_Question/Game/Player.hの「HandsOn 3 PlayerクラスにCCharacterControllerのインスタンスを保持させる。」と書かれている箇所に下記のコードを追加してください。

|  |
| --- |
| //HandsOn 3 PlayerクラスにCCharacterControllerのインスタンスを保持させる。  CCharacterController charaCon; //キャラクターコントローラー。 |

続いて、キャラクターコントローラーの初期化を行います。

Lesson05\_Question/Game/Player.cppの「HandsOn 4 CCharacterControllerのインスタンスを初期化する。」と書かれている箇所に下記のコードを追加してください。

|  |
| --- |
| //HandsOn 4 CCharacterControllerのインスタンスを初期化する。  charaCon.Init(  20.0f, //キャラクターの半径。  68.0f, //キャラクターの高さ。  position //キャラクターの初期座標。  ); |

第一引数はキャラクターの半径、第二引数は高さです。キャラクターコントローはこの二つの引数を使って、キャラクターを内包するCapsuleColliderを作成します。

最後に、キャラクターコントロラーを使用して、キャラクターを移動させます。

Lesson05\_Question/Game/Player.cppの「HandsOn 5 CCharacterControllerを使って、キャラクターを移動させる。」と書かれている箇所に下記のコードを追加してください。

|  |
| --- |
| //HandsOn 5 CCharacterControllerを使って、キャラクターを移動させる。  position = charaCon.Execute(1.0f, moveSpeed); |

キャラクターコントローラーは移動速度を渡すことで、Execute関数内で、キャラクターを移動させます。移動させた結果、背景にめり込んでいた場合、そのめり込みを解決した新しいキャラクターの座標を返してきます。プレイヤークラスでは、その座標を受け取って3Dモデルに座標を反映させています。

# **Lesson\_05\_03 まとめ**

　キャラクターコントローラーを使った、あたり判定を行う方法をまとめます。

　・あたり判定を取りたいオブジェクト(背景など)にCPhysicsStaticObjectのインスタンスを保持、初期化して、物理ワールドにColliderを登録する。

　・移動処理を行うキャラクターにCCharacterControllerのインスタンスを保持、初期化して、Execute関数に移動させたい量を渡してやって、その結果を３Ｄモデルに反映させる。

# **Lesson\_05\_04 中間テスト**

下記のURLのテストを行いなさい。

<https://goo.gl/forms/YJ7X2R7xRygSgN5L2>

# **Lesson\_06\_01 ☆をゲットできるようにしてみよう。**

　Lesson\_06ではスターを取得する処理を実装していきます。Lesson06/Question/Game/Game.slnを起動して、Star.cppを開いてください。

# **Lesson\_06\_02 ☆を表示してみよう。**

　まずは☆を表示できるようにしてみましょう。これは、Lesson\_00\_3の復習です。

Lesson06\_Question/Game/Player.hを開いて、「HandsOn 1 prefab::CSkinModelRender\*型のメンバ変数を追加。」と書かれている箇所に下記のコードを追加してください。

|  |
| --- |
| //HandsOn 1 prefab::CSkinModelRender\*型のメンバ変数を追加。  prefab::CSkinModelRender\* skinModelRender = nullptr; |

続いて、CSkinModelRenderのインスタンスを作成して、モデルをロードしましょう。Lesson06\_Question/Game/Player.hを開いて、「HandsOn 2 スターのモデルを読み込んで、SkinModelRenderを作成。」と書かれている箇所に下記のコードを追加してください。

|  |
| --- |
| //HandsOn 2 スターのモデルを読み込んで、SkinModelRenderを作成。  skinModelRender = NewGO<prefab::CSkinModelRender>(0);  skinModelRender->Init(L"modelData/star.cmo");  skinModelRender->SetPosition(position);  //スターのモデルが小さいので20倍くらいにしておく。  CVector3 scale = { 20.0f, 20.0f, 20.0f };  skinModelRender->SetScale(scale); |

Starクラスのインスタンスは、すでにGame.cppで作成されています。Game.cppの42行目～53行目のコードを見てみてください。

|  |
| --- |
| //星を配置。  CLocData locData;  locData.Load(L"loc/star.tks"); //配置情報をロード。  for (int i = 0; i < locData.GetNumObject(); i++) {  //星を一つづつ作成していく。  CQuaternion qRot;  qRot.SetRotationDeg(CVector3::AxisY, 180.0f);  auto star = NewGO<Star>(0);  star->position = locData.GetObjectPosition(i);  qRot.Multiply(star->position);  m\_starList.push\_back(star);  } |

ではCtrl+F5でゲームを起動して、星が正しく配置されているか確認してみてください。

# **Lesson\_06\_03 ☆を回転させてみよう。**

　星がただ表示されているだけでは、味気ないので、☆を回転させてみようと思います。

これはLesson\_02\_1の復習です。

Lesson06\_Question/Game/Star.cppの「HandsOn 3 スターを回転させる。」と書かれている箇所に下記のコードを追加してください。

|  |
| --- |
| //HandsOn 3 スターを回転させる。  //スターを回転させる。  angle += 3.0f; //毎フレーム3°ずつ回転を加算していく。  CQuaternion qRot;  qRot.SetRotationDeg(CVector3::AxisY, angle); //回転クォータニオンを作成。  skinModelRender->SetRotation(qRot); |

# **Lesson\_06\_04 ☆とプレイヤーの２点間の距離を計算して、☆をゲットしよう。**

今回はプレイヤーと☆の距離が130cm以下になったら、☆を削除するプログラムを記述します。このプログラムを記述するためにはプレイヤーと☆の距離を計算する必要があります。プレイヤーや☆の座標はx,y,zの3要素の**ベクトル**で表されます。プレイヤーの座標をP、☆の座標をＳとしたとき、この２点間の距離Lは下記のようになります。

Ｓ

どこかで見たことがある計算式ではないでしょうか？これは中学校で習う**三平方の定理**を使用した計算式になります。数式がでてきたので嫌になる人もいるかもしれませんが、安心してください。今回の実習で使うプログラムには、簡単に距離を求めることができる処理

を用意しています。

距離を求めるサンプルコード

|  |
| --- |
| //プレイヤーの座標を取得。  CVector3 diff = playerPos - starPos;  float L = v.Length(); //Lに２点間距離が入る。 |

では、これを参考にしてstar.cppの「HandsOn 4 プレイヤーとの距離を計算して、130cm以下なら星を削除する。」と記述されている箇所に下記のコードを記入してください。

|  |
| --- |
| //HandsOn 4 プレイヤーとの距離を計算して、130cm以下なら星を削除する。  CVector3 diff = player->position - position;  if (diff.Length() < 130.0f) {  DeleteGO(this);  } |

DeleteGOはIGameObjectを継承したインスタンスを破棄する命令です。今回はDeleteGOにthisポインタを渡しているので、自分自身が削除されています。

# **Lesson\_06\_05 中間テスト**

　下記のURLのテストを行いなさい。

<https://goo.gl/forms/tzSDwQ5eidiS66Wg2>

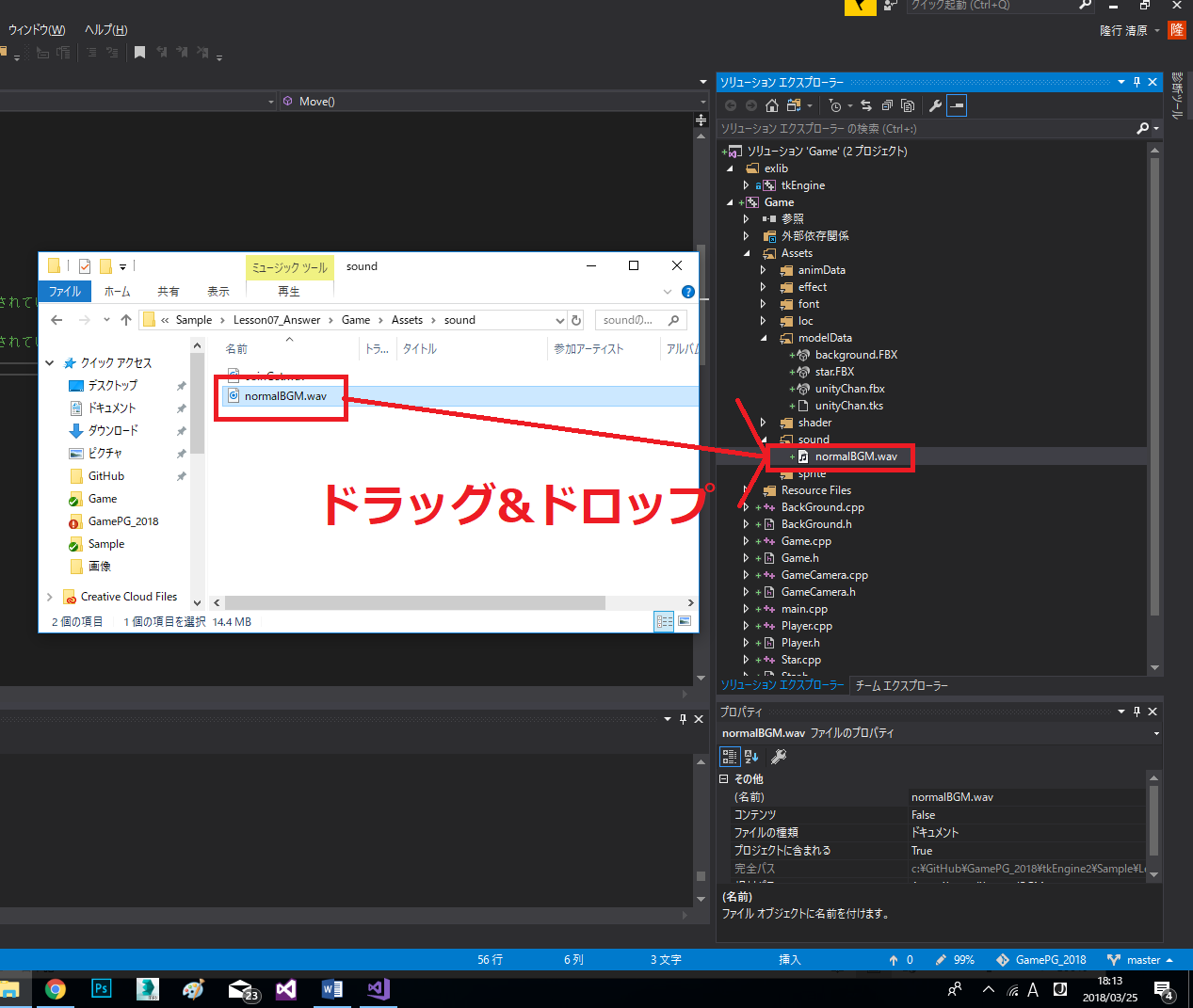
# **Lesson\_07\_01 BGMを流そう**

　ゲームには音楽が欠かすことができません。tkEngine2でサポートしている音楽タのフォーマットはwaveファイルのみです。Lesson\_07では音楽データをVisualSutdioに追加する方法と、その再生方法について見ていきましょう。

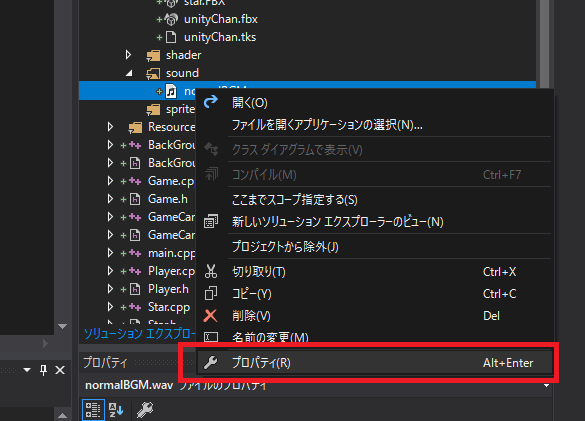
音を再生するためには、まずファイルをVisualStudioに追加する必要があります。

では、Lesson07\_QuestionにBGMデータを追加してみましょう。

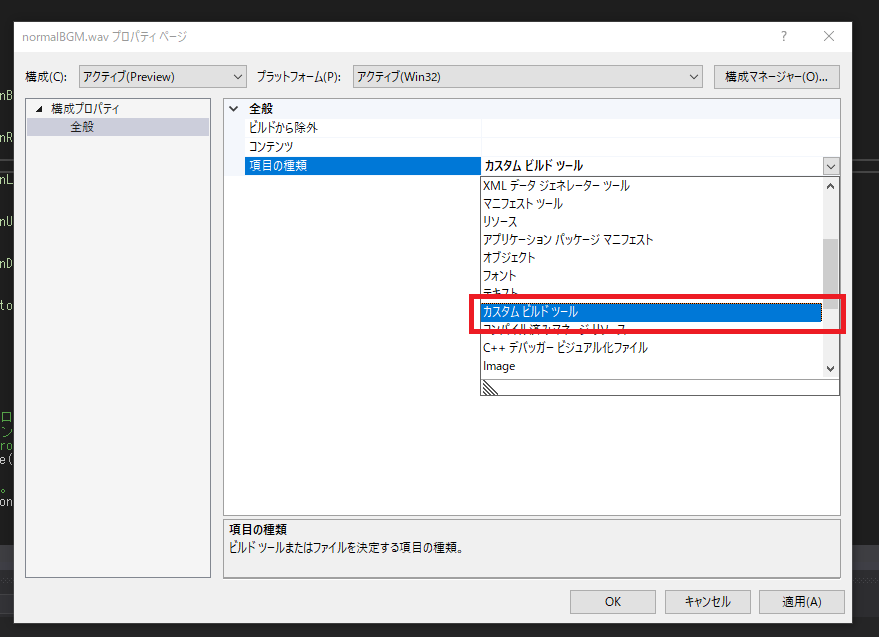
**Step 1 waveファイルをVisusalStudioに追加。**



**Step 2 追加したファイルを選択して右クリック→プロパティ**



**Step3 ビルドツールにカスタムビルドツールを指定する。**



これで音楽ファイルをVisualStudioに追加することができました。

では、続いて、この音楽ファイルを再生できるようにプログラミングしましょう。Game.cppを開いて、「HandsOn 1 BGMを再生しよう。」と書かれている箇所に下記のコードを追加してください。

|  |
| --- |
| //HandsOn 1 BGMを再生しよう。  m\_bgmSource = NewGO<prefab::CSoundSource>(0); //CSoundSourceのインスタンスを作成。  m\_bgmSource->Init("sound/normalBGM.wav"); //音声ファイルをロード。  m\_bgmSource->Play(true); //再生。 |

Play関数の第一引数は、音楽をループ再生するかどうかのフラグです。

# **Lesson\_07\_02 コインを取得したときに音を鳴らそう。**

　 では、続いてコインを取得したときに音を鳴らしてみましょう。まずはLesson07\_01を参考にして、VisualStudioにAssets/sound/coinGet.wavを追加してください。

追加できたら、「HandsOn 2 コインを取得したときに音を鳴らそう」と書かれている箇所に下記のコードを追加してください。

|  |
| --- |
| //HandsOn 2 コインを取得したときに音を鳴らそう。  prefab::CSoundSource\* ss = NewGO<prefab::CSoundSource>(0);  ss->Init("sound/coinGet.wav");  ss->Play(false); |

コインの取得音はループ再生ではないので、Play関数の第一引数がfalseになっていることに注意してください。

# **Lesson\_07\_03 中間テスト**

　下記のURLのテストを行いなさい。

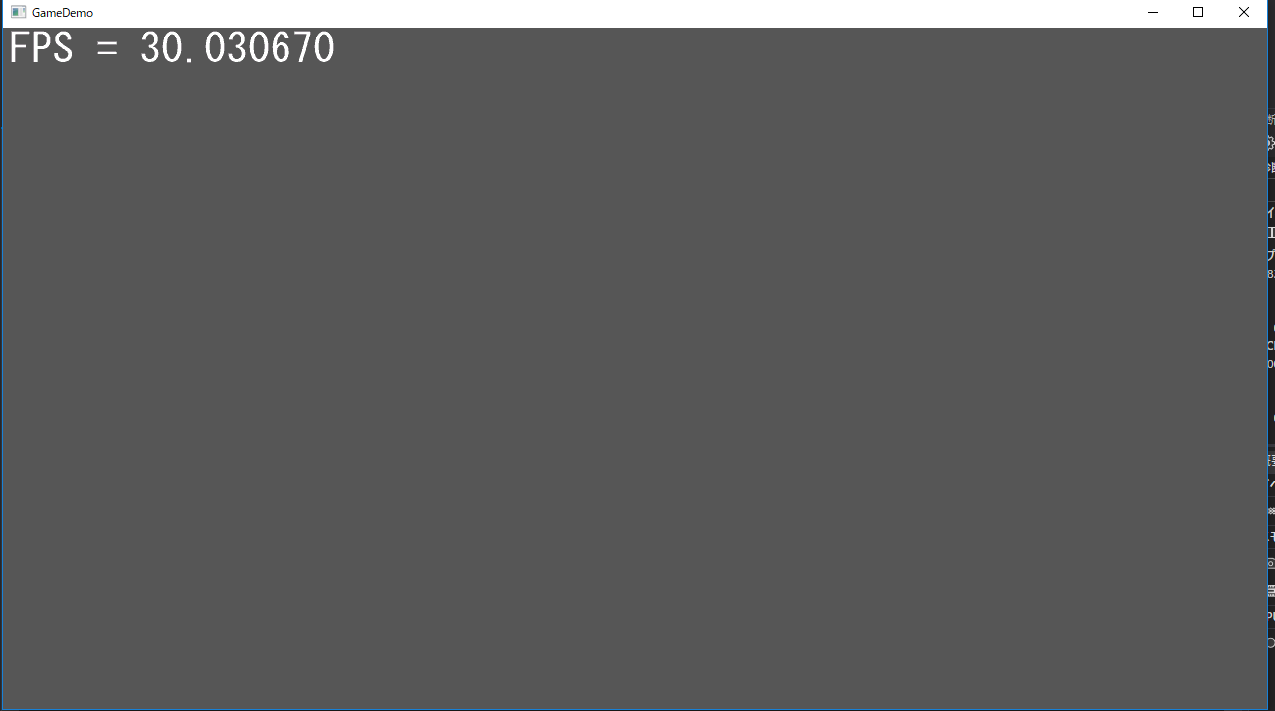
<https://docs.google.com/forms/d/e/1FAIpQLSe3zKoMamLBbwbaQ2f-OzP34SG0rbivuliziKiLsOINUxmgaw/viewform?usp=sf_link>

**２章　シューティングゲームを作ろう。**

　 ここからはShooting\_00を使用して、簡単なシューティングゲームを作っていきます。

# **Lesson\_08\_00 自機を表示してみよう。**

　Shooting\_00のプログラムを実行すると、何も表示されていないウィンドウが表示されたと思います。ではこのゲームにシューティングゲームの自機を表示するプログラムを実装しましょう。



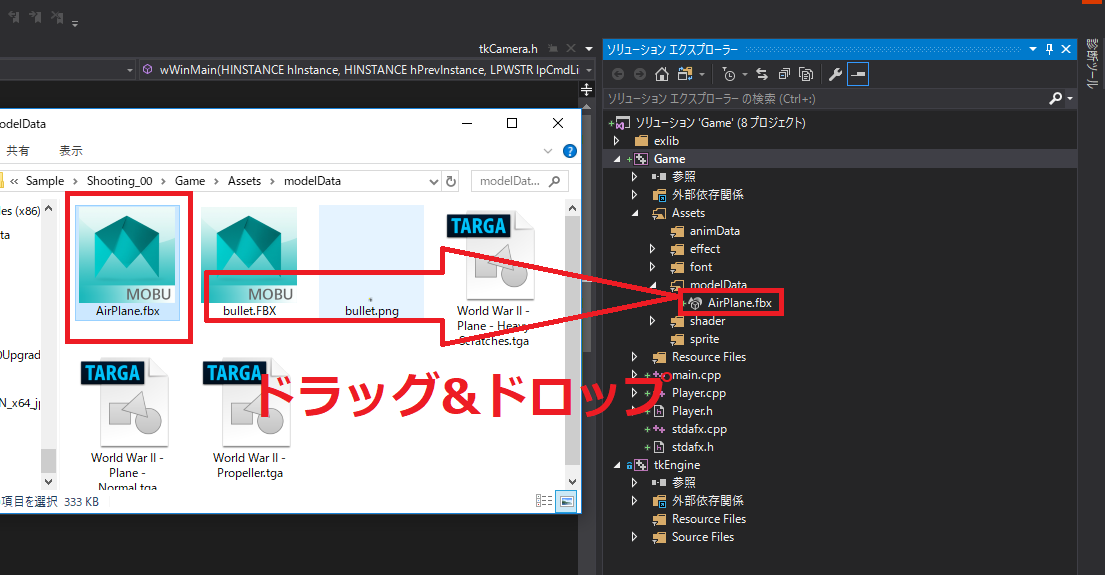
# **Lesson\_08\_01 wWinMain関数**

　main.cppの7行目のwWinMain関数を見てみてください。実はこの関数はWindowsプログラムのメイン関数と呼ばれるもので，C言語のmain関数と同じでようなものになります。プログラムが起動するとこの関数が実行され、Engine().RunGameLoop();というプログラムが実行されると、ゲームループと呼ばれる一定間隔でループする処理が動き出します。

|  |
| --- |
| #include "stdafx.h"  int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nCmdShow)  {  UNREFERENCED\_PARAMETER(hPrevInstance);  UNREFERENCED\_PARAMETER(lpCmdLine);  srand((unsigned)time(NULL));  //tkEngine2の初期化パラメータを設定する。  SInitParam initParam;  initParam.nCmdShow = nCmdShow;  initParam.hInstance = hInstance;  initParam.screenWidth = 1280;  initParam.screenHeight = 720;  initParam.frameBufferWidth = 1280;  initParam.frameBufferHeight = 720;  //影の設定。  initParam.graphicsConfing.shadowRenderConfig.isEnable = true;  initParam.graphicsConfing.shadowRenderConfig.shadowMapWidth = 1024;  initParam.graphicsConfing.shadowRenderConfig.shadowMapHeight = 1024;  initParam.graphicsConfing.shadowRenderConfig.softShadowLevel = EnSoftShadowQualityLevel::eSSSS\_PCF;  //アンチ  initParam.graphicsConfing.aaConfig.isEnable = true;  //Bloom  initParam.graphicsConfing.bloomConfig.isEnable = true;  //tonemap  initParam.graphicsConfing.tonemapConfig.isEnable = true;  GraphicsEngine().GetShadowMap().SetFar(1000.0f);  GraphicsEngine().GetShadowMap().SetNear(50.0f);  //エンジンを初期化。  if (Engine().Init(initParam) == true) {  //カメラを初期化。  MainCamera().SetTarget({ 0.0f, 0.0f, 0.0f });  MainCamera().SetNear(0.1f);  MainCamera().SetFar(15000.0f);  MainCamera().SetPosition({ 0.0f, 0.0f, -10000.0f });  MainCamera().Update();  //ゲームループを実行。  Engine().RunGameLoop();  }  //エンジンの終了処理。  Engine().Final();  return 0;  } |

# **Lesson\_08\_02 AirPlane.fbxをVisualStudioに追加**

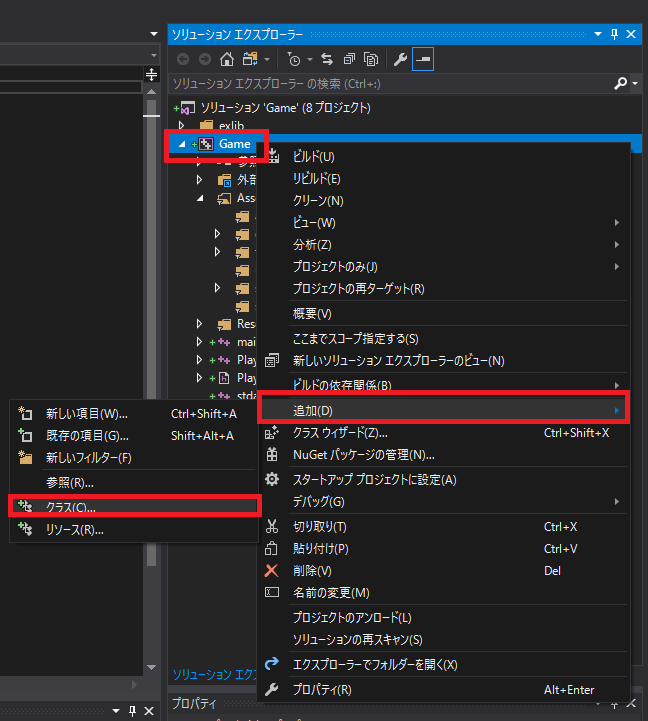
　まずは、自機のモデルとなる、AirPlane.fbxをVisualStudioに追加しましょう。



# **Lesson\_08\_03 プレイヤーのクラスの追加**

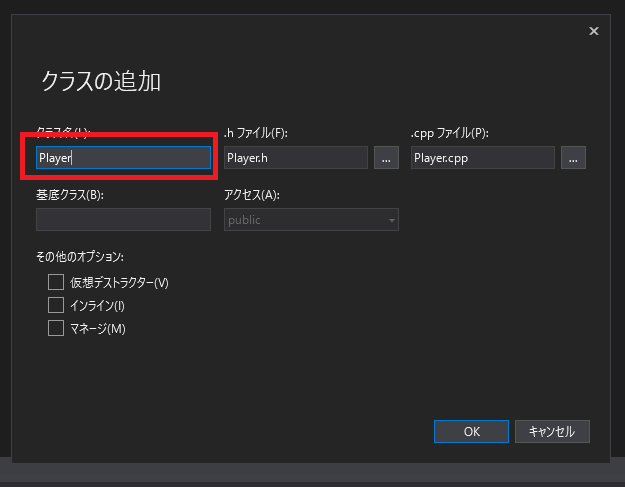
続いて、自機(プレイヤー)のプログラムを記述するためにプレイヤーのクラスを作りましょう。ソリ

ューションエクスプローラーのGameプロジェクトを選択して右クリック/追加/クラスを選択してください。

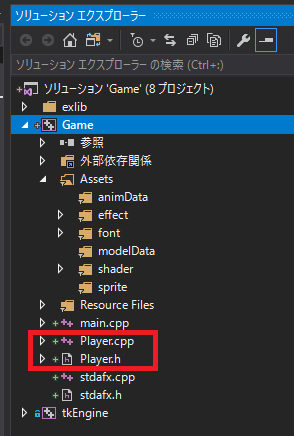
****

選択すると次のウィンドウが開くので追加を選択してください。

選択すると、次のウィンドウが開くので、class nameの欄にPlayerと記入してください。



するとGameプロジェクトにPlayer.hとPlayer.cppというファイルが追加されます。



# **Lesson\_08\_04 プレイヤーのクラスの実装**

　ではPlayer.hを下記のように書き換えてみてください。

|  |
| --- |
| #pragma once  class Player : public IGameObject  {  public:  Player();  ~Player();  bool Start();  //////////////////////////////  //　ここからメンバ変数。  //////////////////////////////  prefab::CSkinModelRender\* m\_skinModelRenderer = nullptr; //スキンモデルレンダラー。  }; |

拡張子がhのファイルはヘッダーファイルと呼ばれるものです。ヘッダーファイルにはPlayerのクラスがどのようなものなのかという**クラス宣言**を記述しました。続いて、Playerクラスの実装を記述していきましょう。Player.cppを下記のように書き換えてください。

|  |
| --- |
| #include "stdafx.h"  #include "Player.h"  //////////////////////////////////////////////////////////  // コンストラクタ。インスタンスが生成されるときに呼ばれる処理。  //////////////////////////////////////////////////////////  Player::Player()  {  }  //////////////////////////////////////////////////////////  // デストラクタ。インスタンスが破棄されるときに呼ばれる処理。  //////////////////////////////////////////////////////////  Player::~Player()  {  //スキンモデルレンダラーを破棄  DeleteGO(m\_skinModelRenderer);  }  //////////////////////////////////////////////////////////  // 初めてUpdate関数が呼ばれる直前に、一度だけ呼ばれる関数。  //////////////////////////////////////////////////////////  bool Player::Start()  {  //スキンモデルレンダラーを作成。  m\_skinModelRenderer = NewGO<prefab::CSkinModelRender>(0);  m\_skinModelRenderer->Init(L"modelData/AirPlane.cmo");  //横を向かせたいので、Y軸周りに-90°回す。  CQuaternion qRot;  qRot.SetRotationDeg(CVector3::AxisY, -90.0f);  m\_skinModelRenderer->SetRotation(qRot);  return true;  } |

これでPlayerクラスが作成されました。Player::Start関数は初めてUpdate関数が呼ばれる直前に、一度だけ呼ばれる関数です。この関数で、モデルデータをロードして、prefab::CSkinModelRenderのインスタンスを生成しています。

# **Lesson\_08\_05 Playerクラスのインスタンス化**

さて、クラスを作っただけではプレイヤーは表示されません。クラスは設計図のようなものです。設計図だけ作成してもビルは建ちませんよね？ではクラスはインスタンス化(実体化)することによって初めて使うことができます。ではPlayerクラスをインスタンス化してみましょう。(注意：今回のインスタンス化の仕方はtkEngine2独自のものです。C++の機能ではないのでご注意ください。)

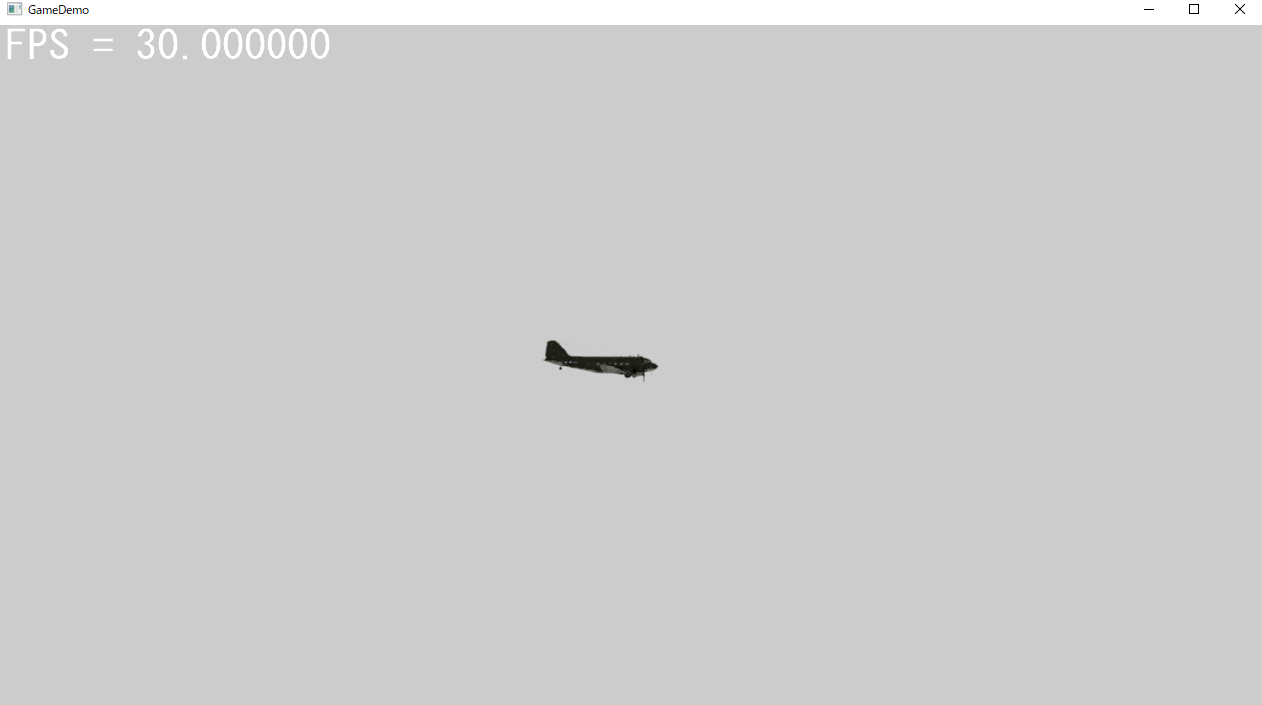
wWinMain関数に下記の網掛けになっているコードを追加してみてください。

|  |
| --- |
| int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nCmdShow)  {  UNREFERENCED\_PARAMETER(hPrevInstance);  UNREFERENCED\_PARAMETER(lpCmdLine);  srand((unsigned)time(NULL));  //tkEngine2の初期化パラメータを設定する。  SInitParam initParam;  initParam.nCmdShow = nCmdShow;  initParam.hInstance = hInstance;  initParam.screenWidth = 1280;  initParam.screenHeight = 720;  initParam.frameBufferWidth = 1280;  initParam.frameBufferHeight = 720;  //影の設定。  initParam.graphicsConfing.shadowRenderConfig.isEnable = true;  initParam.graphicsConfing.shadowRenderConfig.shadowMapWidth = 1024;  initParam.graphicsConfing.shadowRenderConfig.shadowMapHeight = 1024;  initParam.graphicsConfing.shadowRenderConfig.softShadowLevel = EnSoftShadowQualityLevel::eSSSS\_PCF;  //アンチ  initParam.graphicsConfing.aaConfig.isEnable = true;  //Bloom  initParam.graphicsConfing.bloomConfig.isEnable = true;  //tonemap  initParam.graphicsConfing.tonemapConfig.isEnable = true;  GraphicsEngine().GetShadowMap().SetFar(1000.0f);  GraphicsEngine().GetShadowMap().SetNear(50.0f);  //エンジンを初期化。  if (Engine().Init(initParam) == true) {  //カメラを初期化。  MainCamera().SetTarget({ 0.0f, 0.0f, 0.0f });  MainCamera().SetNear(0.1f);  MainCamera().SetFar(15000.0f);  MainCamera().SetPosition({ 0.0f, 0.0f, -10000.0f });  MainCamera().SetUpdateProjMatrixFunc(CCamera::enUpdateProjMatrixFunc\_Ortho);  MainCamera().SetWidth(20000.0f);  MainCamera().SetHeight(20000.0f \* 9.0f / 16.0f);  MainCamera().Update();  //プレイヤーのインスタンスを作成。  NewGO<Player>(0);  //ゲームループを実行。  Engine().RunGameLoop();  }  //エンジンの終了処理。  Engine().Final();  return 0;  } |

これだけだと、Playerが定義されていないというコンパイルエラーが起きていると思います。このエラーを修正するためにはmain.cppにPlayer.hをインクルードして、Playerクラスのことについて教えてやる必要があります。main.cppに網掛けになっている箇所を追加してください。

|  |
| --- |
| /\*!  \*@brief main.cpp  \*/  #include "stdafx.h"  #include "Player.h" |

これでPlayerクラスがインスタンス化されて、自機が表示されるようになったと思います。



# **Lesson\_08\_06 中間テスト**

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLScFEa9nu9wWT1X6yBUDXgeaHpPzXEK3kJnqoLAUJ4qRWSWu8g/viewform?usp=sf_link>

*参考プロジェクト*

*Lesson\_08の内容がすべて実装されたサンプルプログラム。*

*Lesson08\_Answer*

# **Lesson\_09\_00 自機を動かしてみよう。**

では自機を動かす処理を記述してみましょう。まずは左右に移動させてみましょう。

# **Lesson\_09\_01 Update関数を追加する。**

Game/Player.hを開いて、PlayerクラスにUpdate関数と自機の座標を表すメンバ変数のm\_positionを追加してください。

|  |
| --- |
| #pragma once  class Player : public IGameObject  {  public:  Player();  ~Player();  bool Start();  void Update();  //////////////////////////////  //　ここからメンバ変数。  //////////////////////////////  prefab::CSkinModelRender\* m\_skinModelRenderer = nullptr; //スキンモデルレンダラー。  CVector3 m\_position = CVector3::Zero; //座標。  }; |

続いてGame/Player.cppを開いてPlayerクラスのUpdate関数の実装を行います。

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Player::Update()  {  if (Pad(0).IsPress(enButtonRight)) {  m\_position.x += 200.0f;  }  if (Pad(0).IsPress(enButtonLeft)) {  m\_position.x -= 200.0f;  }  if (Pad(0).IsPress(enButtonUp)) {  m\_position.y += 200.0f;  }  if (Pad(0).IsPress(enButtonDown)) {  m\_position.y -= 200.0f;  }  //座標をスキンモデルレンダラーに反映させる。  m\_skinModelRenderer->SetPosition(m\_position);  } |

Update関数はIGameObjectの機能で、インスタンスが生成されると一定間隔で実行される関数です。(注意：C++の機能ではなく、tkEngine2の機能なので注意！)

これで上下左右のキーを押すことによって、自機が移動するようになりました。

# **Lesson\_09\_02 Update関数に自機を旋回させる処理を追加する。**

　Game/Player.cppのPlayer::Update関数に下記の太字のコードを追加して下さい。

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Player::Update()  {  **CQuaternion qRot = CQuaternion::Identity;**  if (Pad(0).IsPress(enButtonRight)) {  m\_position.x += 200.0f;  }  if (Pad(0).IsPress(enButtonLeft)) {  m\_position.x -= 200.0f;  }  if (Pad(0).IsPress(enButtonUp)) {  m\_position.y += 200.0f;  **//Z軸周りの回転クォータニオンを作成。**  **qRot.SetRotationDeg(CVector3::AxisZ, -10.0f);**  }  if (Pad(0).IsPress(enButtonDown)) {  m\_position.y -= 200.0f;  **//Z軸周りの回転クォータニオンを作成。**  **qRot.SetRotationDeg(CVector3::AxisZ, 10.0f);**  }  **//Y軸周りに-90°回す回転クォータニオンを作成する。**  **CQuaternion qRot2;**  **qRot2.SetRotationDeg(CVector3::AxisY, -90.0f);**  **m\_skinModelRenderer->SetRotation(qRot);**  **//qRotとqRot2のクォータニオンを合成。**  **qRot.Multiply(qRot2);**  //座標をスキンモデルレンダラーに反映させる。  m\_skinModelRenderer->SetPosition(m\_position);  **//回転をスキンモデルレンダラーに反映させる。**  **m\_skinModelRenderer->SetRotation(qRot);**  } |

旋回するようになったと思います。回転を行うためには回転軸と回転角度が必要になるのを思い出して下さい。

# **Lesson\_09\_3 画面外に移動できないようにしよう。**

　今の状態では左右に移動すると自機が画面外に出てしまいます。そこで、自機のx座標を調べて画面外に行けないようにしてみましょう。ここではif文を活用してみます。

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Player::Update()  {  CQuaternion qRot = CQuaternion::Identity;  if (Pad(0).IsPress(enButtonRight)) {  m\_position.x += 200.0f;  }  if (Pad(0).IsPress(enButtonLeft)) {  m\_position.x -= 200.0f;  }  if (Pad(0).IsPress(enButtonUp)) {  m\_position.y += 200.0f;  //Z軸周りの回転クォータニオンを作成。  qRot.SetRotationDeg(CVector3::AxisZ, -10.0f);  }  if (Pad(0).IsPress(enButtonDown)) {  m\_position.y -= 200.0f;  //Z軸周りの回転クォータニオンを作成。  qRot.SetRotationDeg(CVector3::AxisZ, 10.0f);  }  //Y軸周りに-90°回す回転クォータニオンを作成する。  CQuaternion qRot2;  qRot2.SetRotationDeg(CVector3::AxisY, -90.0f);  m\_skinModelRenderer->SetRotation(qRot);  //qRotとqRot2のクォータニオンを合成。  qRot.Multiply(qRot2);  **//もしもm\_position.xが10000よりも大きくなったら。**  **if (m\_position.x > 10000.0f) {**  **//10000.0fを代入する。**  **m\_position.x = 10000.0f;**  **}**  **//もしもm\_position.xが-10000よりも小さくなったら。**  **if (m\_position.x < -10000.0f) {**  **//-10000.0fを代入する。**  **m\_position.x = -10000.0f;**  **}**  //座標をスキンモデルレンダラーに反映させる。  m\_skinModelRenderer->SetPosition(m\_position);  //回転をスキンモデルレンダラーに反映させる。  m\_skinModelRenderer->SetRotation(qRot);  } |

# **Lesson\_09\_4 中間テスト**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSeNwxSfms9Ki0wobhKi-3U14I2mpsCIA6akKukzM4PsPl4How/viewform?usp=sf_link>

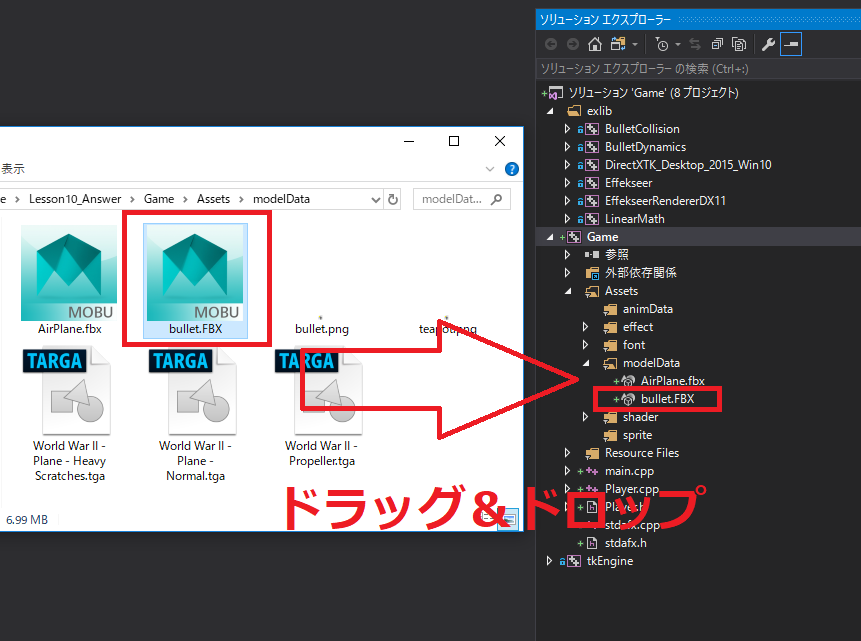
*参考プロジェクト*

*Lesson\_09の内容がすべて実装されたサンプルプログラム。*

*Lesson\_09\_Answer*

# **Lesson10\_0 弾を撃てるようにしてみよう**

この節ではAボタンを押すことで弾丸を発射できるようにしてみましょう。まずはVisualStudioに弾丸のモデルを追加しましょう。



## **Lesson 10\_1 Bulletクラスを作成する。**

Lesson 8の復習です。弾丸の処理を記述するBulletクラスを作成しましょう。ソリューションエクスプローラーでGameプロジェクトを選択して、右クリック。ポップアップメニューから追加/クラスを選択してBulletクラスを作成してください。Bullet.cppとBullet.hが作成できたら、次のコードをBullet.hとBullet.cppに記述してください。

Bullet.h

|  |
| --- |
| #pragma once  class Bullet : public IGameObject  {  public:  Bullet();  ~Bullet();  bool Start();  /////////////////////////////////  // ここからメンバ変数。  /////////////////////////////////  prefab::CSkinModelRender\* m\_skinModelRender = nullptr; //スキンモデルレンダー  }; |

Bullet.cpp

|  |
| --- |
| #include "stdafx.h"  #include "Bullet.h"  Bullet::Bullet()  {  }  Bullet::~Bullet()  {  DeleteGO(m\_skinModelRender);  }  //////////////////////////////////////////////////////////  // 初めてUpdate関数が呼ばれる直前に、一度だけ呼ばれる関数。  //////////////////////////////////////////////////////////  bool Bullet::Start()  {  m\_skinModelRender = NewGO<prefab::CSkinModelRender>(0);  m\_skinModelRender->Init(L"modelData/bullet.cmo");  return true;  } |

これでBulletクラスの作成はひとまず完了です。

# **Lesson 10\_2 Bulletクラスのインスタンス化**

　クラスは設計図のようなものです。クラスを作っただけで弾丸が勝手に発射されたりはしません。弾丸を発射するためにはBulletクラスをインスタンス化する必要があります。Player.cppに次の網掛けになっているコードを追加してください。

Player.cpp

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Player::Update()  {  MoveAndRotation(); //プレイヤーの移動と回転処理はMove関数にまとめたよ。  if (Pad(0).IsPress(enButtonA)) {  Bullet\* bullet = NewGO<Bullet>(0);  }  } |

このままだと、まだ下記のようなコンパイルエラーが起きているのではないでしょうか。



このエラーは「Bulletなんていうクラス知らないよ？」というエラーです。Bulletクラスを使うためにはPlayer.cppにBullet.hをインクルードして、Bulletクラスのことを教えてやる必要があります。Player.cppの網掛けになっているコードを追加してください。

|  |
| --- |
| #include "stdafx.h"  #include "Player.h"  **#include "Bullet.h"** |

ではゲームをビルドして、Aボタンを押して弾丸を発射してみてください。

# **Lesson\_10\_3 弾丸を自機の場所から発生するようにしよう**

　Lesson\_10\_2の実装で、弾丸を生成することはできましたが、すべての弾丸が画面の中央に固まっています。では、まず弾丸を自機の場所から発生するようにしてみようと思います。Bullet.hにUpdate関数と弾丸の座標を表すm\_positionを追加してください。

|  |
| --- |
| #pragma once  class Bullet : public IGameObject  {  public:  Bullet();  ~Bullet();  bool Start();  **void Update();**  /////////////////////////////////  // ここからメンバ変数。  /////////////////////////////////  prefab::CSkinModelRender\* m\_skinModelRender = nullptr; //スキンモデルレンダー  **CVector3 m\_position = CVector3::Zero; //座標。**  }; |

　続いて、Bullet.cppにUpdate関数の定義を実装します。

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Bullet::Update()  {  //スキンモデルレンダーに座標を伝える。  m\_skinModelRender->SetPosition(m\_position);  } |

　最後にBulletのインスタンスを生成するときに、プレイヤーの座標を教えてやるようにしましょう。Player.cppに下記のコードを追加してください。

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Player::Update()  {  MoveAndRotation(); //プレイヤーの移動と回転処理はMove関数にまとめたよ。  if (Pad(0).IsPress(enButtonA)) {  Bullet\* bullet = NewGO<Bullet>(0);  **//弾丸の座標にプレイヤーの座標を代入する。**  **bullet->m\_position = m\_position;**  }  } |

# **Lesson\_10\_4 弾丸を動かそう。**

　Lesson\_10\_3の実装で弾丸が表示されるようになりました。しかし、弾丸は自機の場所に表示されるだけで前に飛んでいきません。これを飛ぶようにしてみましょう。今回は外部から弾丸の移動速度を設定して飛ばすという方法で実装してみたいと思います。Bullet.hとBullet.cppに網掛けになっているコードを追加してください。

Bullet.h

|  |
| --- |
| #pragma once  class Bullet : public IGameObject  {  public:  Bullet();  ~Bullet();  bool Start();  void Update();  /////////////////////////////////  // ここからメンバ変数。  /////////////////////////////////  prefab::CSkinModelRender\* m\_skinModelRender = nullptr; //スキンモデルレンダー  CVector3 m\_position = CVector3::Zero;　　 //座標。  **CVector3 m\_moveSpeed = CVector3::Zero; //移動速度。**  }; |

Bullet.cpp

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Bullet::Update()  {  //弾丸を移動させる。  m\_position += m\_moveSpeed;  //スキンモデルレンダーに座標を伝える。  m\_skinModelRender->SetPosition(m\_position);  } |

では続いてBulletのインスタンス化の時に移動速度を設定するようにしてみましょう。Player.cppに網掛けになっているコードを追加してください。

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Player::Update()  {  MoveAndRotation(); //プレイヤーの移動と回転処理はMove関数にまとめたよ。  if (Pad(0).IsPress(enButtonA)) {  Bullet\* bullet = NewGO<Bullet>(0);  //弾丸の座標にプレイヤーの座標を代入する。  bullet->m\_position = m\_position;  **//X軸方向に400の速度を設定する。**  **bullet->m\_moveSpeed.x = 400.0f;**  }  } |

# **Lesson\_10\_5　弾丸の発射にインターバルを設定しよう。**

　Lesson\_10\_4の実装では、Aボタンを押し続けているとBulletのインスタンスが毎フレーム生成されるようになっています。そこでPlayerクラスを改造して、Bulletのインスタンスの生成に最低でも5フレームは間隔をあけるようにしてみましょう。

　まず、Player.hにm\_timerというメンバ変数を追加してください。

|  |
| --- |
| #pragma once  class Player : public IGameObject  {  public:  Player();  ~Player();  bool Start();  void Update();  void MoveAndRotation();  //////////////////////////////  //　ここからメンバ変数。  //////////////////////////////  prefab::CSkinModelRender\* m\_skinModelRenderer = nullptr; //スキンモデルレンダラー。  CVector3 m\_position = CVector3::Zero; //座標。  **int m\_timer = 0; //タイマー。**  }; |

続いて、Player.cppに下記のコードを追加します。

Player.cppのUpdate関数

|  |
| --- |
| //タイマーをインクリメント。  m\_timer++;  //Aボタンが押された、かつm\_timerが2以上なら、  //弾丸を生成する。  if (Pad(0).IsPress(enButtonA) && m\_timer >= 2 ) {  //直進する弾丸を作成。  Bullet\* bullet = NewGO<Bullet>(0);  //弾丸の座標にプレイヤーの座標を代入する。  bullet->m\_position = m\_position;  //X軸方向に400の速度を設定する。  bullet->m\_moveSpeed.x = 400.0f;  m\_timer = 0;  } |

# **Lesson\_10\_6 弾丸に寿命を設定しよう**

Lesson\_10\_5の実装で弾を連続して打つことができるようになりました。しかし、弾を撃ち続けるとどんどんゲームが遅くなっていったのではないでしょうか？実は弾丸はインスタンスの削除を行っていないため、遠くまで飛んで行って画面に映らなくなっても処理が行われています。そのため、どんどん処理が重くなっていきました。Lesson\_10\_5ではNewGOで生成したインスタンスを破棄する方法を勉強しましょう。

インスタンスはDeleteGOという関数を使用すれば破棄することができます。では弾丸に寿命を設定して、寿命が尽きたらインスタンスを破棄するようにプログラムを改造してみましょう。

　Bullet.hにm\_timerというメンバ変数を追加してください。

|  |
| --- |
| #pragma once  class Bullet : public IGameObject  {  public:  Bullet();  ~Bullet();  bool Start();  void Update();  /////////////////////////////////  // ここからメンバ変数。  /////////////////////////////////  prefab::CSkinModelRender\* m\_skinModelRender = nullptr; //スキンモデルレンダー  CVector3 m\_position = CVector3::Zero; //座標。  CVector3 m\_moveSpeed = CVector3::Zero; //移動速度。  **int m\_timer = 0; //タイマー。**  }; |

Bullet.cppのUpdate関数に網掛けになっているコードを追加してください。

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Bullet::Update()  {  //弾丸を移動させる。  m\_position += m\_moveSpeed;  //スキンモデルレンダーに座標を伝える。  m\_skinModelRender->SetPosition(m\_position);  **//タイマーを加算する。**  **m\_timer++;**  **if (m\_timer == 50) {**  **//タイマーが50になったらインスタンスを削除する。**  **DeleteGO(this);**  **}**  } |

DeleteGO関数に削除したいインスタンスを渡すことで、インスタンスを破棄することができます。thisというキーワードはC++で用意されている特殊なもので、自分自身のインスタンスを表します。つまりDeleteGO(this)というコードで自分自身を破棄しているのです。

# **Lesson\_10\_7　中間テスト**

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSejw0nDDyxXdu6tklhRe_MOdXlduTg6Is5SJPDfH36VwK2Hmg/viewform?usp=sf_link>

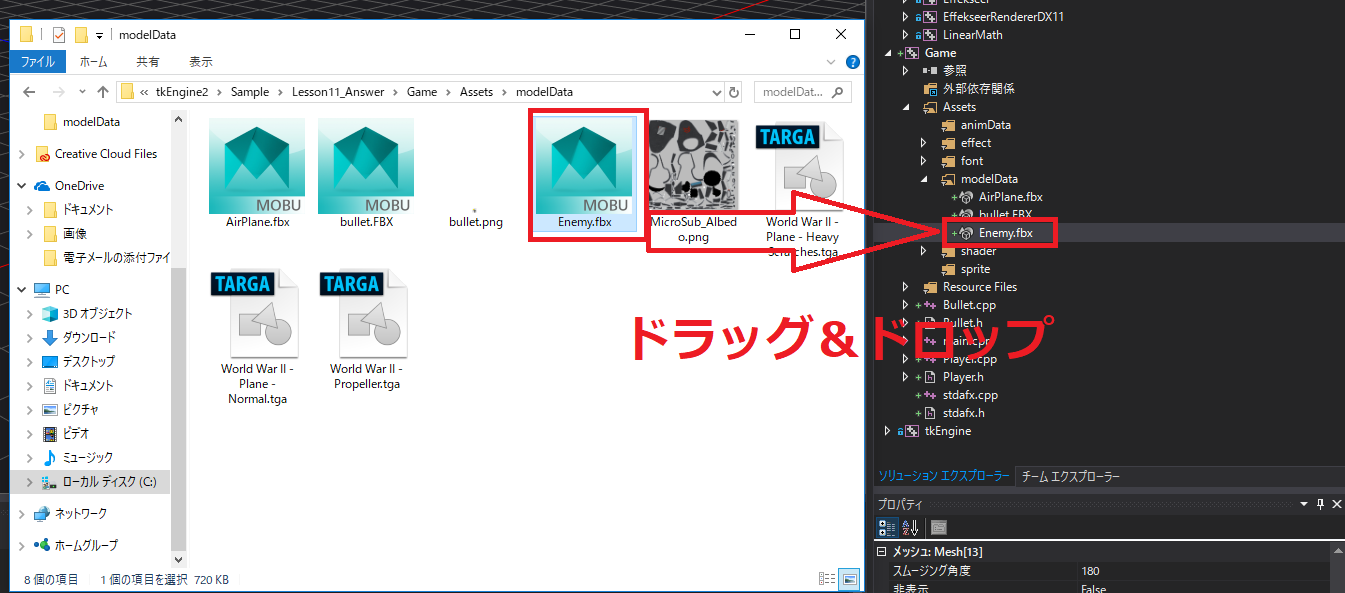
*参考プロジェクト*

*Lesson\_10の内容がすべて実装されたサンプルプログラム。*

*Lesson\_10\_Answer*

# **Lesson\_11\_1 敵機の実装をしよう**

　Lesson\_11では敵機を実装していこうと思います。では、例のごとく敵機のモデルをVisualStudioに追加しましょう。



# **Lesson\_11\_1 Enemyクラスを作成する。**

　これも今まで通りです。まずはEnemyクラスを作成しましょう。Enemyクラスの作成が出来たら、Enemy.hとEnemy.cppに下記のようなコードを書いてください。

Enemy.h

|  |
| --- |
| #pragma once  class Enemy  {  public:  Enemy();  ~Enemy();  bool Start();  void Update();  ///////////////////////////////  // ここからメンバ変数  ///////////////////////////////  prefab::CSkinModelRender\* m\_skinModelRender = nullptr; //スキンモデルレンダー。  CVector3 m\_position = CVector3::Zero; //座標。  }; |

Enemy.cpp

|  |
| --- |
| #include "stdafx.h"  #include "Enemy.h"  Enemy::Enemy()  {  }  Enemy::~Enemy()  {  DeleteGO(m\_skinModelRender);  }  //////////////////////////////////////////////////////////  // 初めてUpdate関数が呼ばれる直前に、一度だけ呼ばれる関数。  //////////////////////////////////////////////////////////  bool Enemy::Start()  {  m\_skinModelRender = NewGO<prefab::CSkinModelRender>(0);  m\_skinModelRender->Init(L"modelData/enemy.cmo", nullptr, 0, CSkinModel::enFbxUpAxisY);  //モデルが小さいので１０倍する。  CVector3 scale;  scale.x = 10.0f;  scale.y = 10.0f;  scale.z = 10.0f;  m\_skinModelRender->SetScale(scale);  //横を向かせたいのでY軸周りに90°回転させる。  CQuaternion qRot;  qRot.SetRotationDeg(CVector3::AxisY, 90.0f);  //スキンモデルに回転クォータニオンを設定する。  m\_skinModelRender->SetRotation(qRot);  return true;  }  //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Enemy::Update()  {  //スキンモデルレンダーに座標を伝える。  m\_skinModelRender->SetPosition(m\_position);  } |

これでひとまずEnemyクラスの作成は完了です。

# **Lesson\_11\_2 Enemy生成器を作成しよう**

　Lesson\_11\_1でEnemyクラスを作成することができました。しかし、クラスを作成しただけではダメでしたね？敵機を出すためにはEnemyクラスのインスタンスを作成する必要があります。シューティングゲームで敵機を作成する方法はいくつかありますが、今回は一定間隔で敵機を作成するようにしてみましょう。では、敵機を作成するためのクラスのEnemyGeneratorクラスを作成しましょう。

EnemyGenerator.h

|  |
| --- |
| #pragma once  class EnemyGenerator : public IGameObject  {  public:  EnemyGenerator();  ~EnemyGenerator();  void Update();  ///////////////////////////////  // ここからメンバ変数  ///////////////////////////////  int m\_timer = 0; //タイマー。  }; |

EnemyGenerator.cpp

|  |
| --- |
| #include "stdafx.h"  #include "EnemyGenerator.h"  #include "Enemy.h"  EnemyGenerator::EnemyGenerator()  {  }  EnemyGenerator::~EnemyGenerator()  {  }  void EnemyGenerator::Update()  {  m\_timer++;  if (m\_timer == 60) {  m\_timer = 0; //タイマーをリセット。  //60フレームごとに敵機を作成する。  Enemy\* enemy = NewGO<Enemy>(0);  }  } |

60フレームごとに敵機を作成する敵機生成クラスが作成できました。では、EnemyGeneratorのインスタンスを作成しましょう。main.cppに下記の網掛けになっているコードを追加してください。

main.cpp(38行目)

|  |
| --- |
| //エンジンを初期化。  if (Engine().Init(initParam) == true) {  //カメラを初期化。  MainCamera().SetTarget({ 0.0f, 0.0f, 0.0f });  MainCamera().SetNear(0.1f);  MainCamera().SetFar(15000.0f);  MainCamera().SetPosition({ 0.0f, 0.0f, -10000.0f });  MainCamera().SetUpdateProjMatrixFunc(CCamera::enUpdateProjMatrixFunc\_Ortho);  MainCamera().SetWidth(20000.0f);  MainCamera().SetHeight(20000.0f \* 9.0f / 16.0f);  MainCamera().Update();  **//敵機生成器のインスタンスを作成する。**  **NewGO<EnemyGenerator>(0);**  //プレイヤーのインスタンスを作成。  NewGO<Player>(0);  //ゲームループを実行。  Engine().RunGameLoop();  } |

EnemyGenerator.hをインクルードするのを忘れないように注意してください。

main.cpp(4行目)

|  |
| --- |
| #include "stdafx.h"  #include "Player.h"  **#include "EnemyGenerator.h"** |

# **Lesson\_11\_3 敵機の発生場所をランダムにしてみる。**

　これで敵機は生成できるようになりましたが、敵機の生成場所が一か所になっているため、重なって表示されてしまっています。そこで、敵機の発生場所をランダムにしてみましょう。EnemyGenerator.cppに下記の網掛けになっているコードを追加してください。

|  |
| --- |
| void EnemyGenerator::Update()  {  m\_timer++;  if (m\_timer == 60) {  m\_timer = 0;  //60フレームごとに敵機を作成する。  Enemy\* enemy = NewGO<Enemy>(0);  **//X座標は画面際にする。**  **enemy->m\_position.x = 9000.0f;**  **//Y座標をランダムにする。**  **//GetRandDoubleは0.0～1.をランダムに返してくる関数。**  **float t = Random().GetRandDouble();**  **//CMath::Leap関数は二つの値の間を線形補完する関数。**  **enemy->m\_position.y = CMath::Lerp(t, -5625.0f, 5625.0f);**      }  } |

Random().GetRandDouble()はtkEngine2の乱数生成機能で、0.0～1.0の値をランダムに返してきます。

# **Lesson\_11\_4 中間テスト**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLScAv3TT45XuyGnEzWpDkozz1kKhQkCAFjKTV1EwbMWSb21cvg/viewform?usp=sf_link>

*参考プロジェクト*

*Lesson\_11の内容がすべて実装されたサンプルプログラム。*

*Lesson\_11\_Answer*

# **Lesson\_12\_1 敵機も弾丸を撃てるようにしてみよう**

　では、敵機も弾丸を撃てるようにしてみましょう。弾丸の処理はプレイヤーと同じくBulletクラスを使用して実装します。まず、プレイヤーと同じように、弾丸の生成にはインターバルを設定したいので、Enemyクラスにm\_timerというメンバ変数を追加します。Lesson\_10\_5の復習です。

Enemy.h

|  |
| --- |
| #pragma once  class Enemy : public IGameObject  {  public:  Enemy();  ~Enemy();  bool Start();  void Update();  ///////////////////////////////  // ここからメンバ変数  ///////////////////////////////  prefab::CSkinModelRender\* m\_skinModelRender = nullptr; //スキンモデルレンダー。  CVector3 m\_position = CVector3::Zero; //座標。  CVector3 m\_moveSpeed = CVector3::Zero; //移動速度。  **int m\_timer = 0; //タイマー。**  }; |

続いて、Enemy::Update関数に弾丸を生成するコードを追加しましょう。

Enemy.cpp(33行目)

|  |
| --- |
| /////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Enemy::Update()  {  m\_position += m\_moveSpeed;  if (m\_position.x < 7000.0f) {  //移動方向を変更する。  m\_moveSpeed.x = 50.0f;  }  if (m\_position.x > 9000.0f) {  //移動方向を変更する。  m\_moveSpeed.x = -50.0f;  }  //スキンモデルレンダーに座標を伝える。  m\_skinModelRender->SetPosition(m\_position);  **//タイマーをインクリメント。**  **m\_timer++;**  **if (m\_timer == 10) {**  **//タイマーが10になったので、弾丸のインスタンスを生成する。**  **Bullet\* bullet = NewGO<Bullet>(0, "EnemyBullet");**  **bullet->m\_position = m\_position;**  **bullet->m\_moveSpeed.x = -400.0f;**  **//タイマーをリセット。**  **m\_timer = 0.0f;**  **}**  } |

# **Lesson\_12\_2 プレイヤーと敵機の弾丸に名前を付けよう。**

　さて、ここからはプレイヤーと敵機が弾丸に衝突したら爆発する処理を実装していきたいと思います。そのためには、まず、敵機が発射した弾丸なのか、プレイヤーが発射した弾丸なのかを区別できるようにする必要があります。自分自身が発射した弾丸に当たってしまってはダメですよね。

　tkEngineのNewGOには、生成したインスタンスに名前を付ける機能があります。Playerが生成した弾丸に、”PlayerBullet”、敵機が生成した弾丸にはEnemyBulletという名前を付けてやりましょう。

Player.cpp(95行目)

|  |
| --- |
| //直進する弾丸を作成。  Bullet\* bullet = NewGO<Bullet>(0**, "PlayerBullet"**); |

Enemy.cpp(53行目)

|  |
| --- |
| Bullet\* bullet = NewGO<Bullet>(0**, "EnemyBullet"**); |

# **Lesson\_12\_3　2点間の距離を計算**

　弾丸に名前を付けることができました。次は弾丸とのあたり判定を実装する必要があります。あたり判定は弾丸との距離を計算して、その距離が一定値以下なら当たっているという処理で行うことにします。

例えば、敵機や弾丸の座標はx,y,zの3要素のベクトルで表されます。敵機の座標をE、弾丸の座標をBとしたとき、この２点間の距離Lは下記のようになります。

どこかで見たことがある計算式ではないでしょうか？これは中学校で習う三平方の定理を使用した計算式になります。数式がでてきたので嫌になる人もいるかもしれませんが、安心してください。今回の実習で使うプログラムには、簡単に距離を求めることができる処理

を用意しています。(この処理はもっと詳しく後期のDirectXⅠで教えます。)

距離を求めるサンプルコード

|  |
| --- |
| CVector3 diff = enemyPos - bulletPos;  float L = v.Length(); //Lに２点間距離が入る。 |

では、Enemy.cppにプレイヤーの弾丸とのあたり判定を書いてみましょう。

|  |
| --- |
| //////////////////////////////////////////////////////////  QueryGOsは指定された名前のゲームオブジェクトへの問い合わせを行う処理。  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Enemy::Update()  {  　　　　　　・  　　　　　　・  　　　　　　・  　　　　　省略  　　　　　　・  　　　　　　・  　　　　　　・  //タイマーをインクリメント。  m\_timer++;  if (m\_timer == 10) {  //タイマーが10になったので、弾丸のインスタンスを生成する。  Bullet\* bullet = NewGO<Bullet>(0, "EnemyBullet");  bullet->m\_position = m\_position;  bullet->m\_moveSpeed.x = -400.0f;  //タイマーをリセット。  m\_timer = 0.0f;  }  **//PlayerBulletという名前のゲームオブジェクトに対してクエリ(問い合わせ)を行う。**  **QueryGOs<Bullet>("PlayerBullet", [&](Bullet\* bullet) {**  波かっこの中身が問い合わせの内容。  **//２点間の距離を計算する。**  **CVector3 diff = bullet->m\_position - m\_position;**  **if (diff.Length() < 2000.0f) { //距離が2000以下になったら。**  **//死亡。**  **DeleteGO(this);**  **}**  **});**  } |

QueryGOsは指定された名前のゲームオブジェクトに対してクエリ(問い合わせ)を行う関数で、問い合わせの内容が記述されているのが、波かっこのブロックです。

# **Lesson 12\_4 中間テスト**

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSe3f5KqDpyOgrGUYaIfxw44pgnZQHcVues5wOfYpF4m05XFpw/viewform?usp=sf_link>

*参考プロジェクト*

*Lesson\_12の内容がすべて実装されたサンプルプログラム。*

*Lesson\_12\_Answer*

# **Lesson 13\_1 敵機が死亡する時に爆発のＳＥを鳴らそう。**

　Lesson 13では、敵機と自機が死亡する時の演出を実装していこうと思います。まずは敵機が死亡する時に爆発のＳＥを再生できるようにしてみましょう。Assets/sound/explosion.wavをVisualStudioに追加しましょう。追加の仕方はLesson\_07\_01を参照してください。

追加ができたら、Enemy.cppに下記の網掛けになっているコードを追加してください。

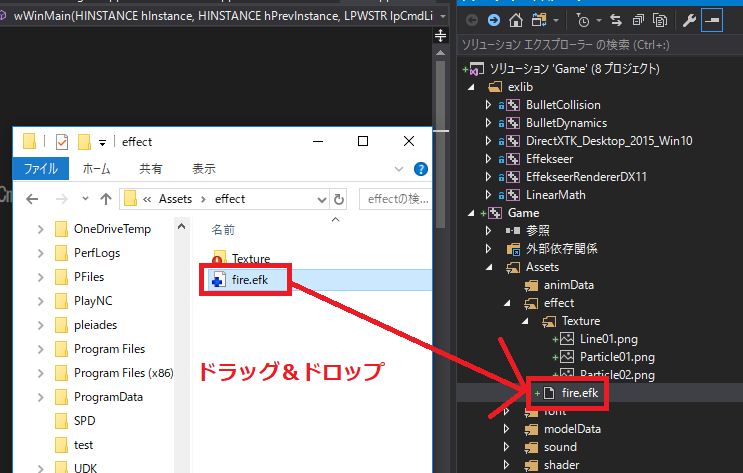
Enemy.cpp(59行目)

|  |
| --- |
| //PlayerBulletという名前のゲームオブジェクトに対してクエリ(問い合わせ)を行う。  QueryGOs<Bullet>("PlayerBullet", [&](Bullet\* bullet)->bool {  //２点間の距離を計算する。  CVector3 diff = bullet->m\_position - m\_position;  if (diff.Length() < 500.0f) { //距離が500以下になったら。  //死亡。  **prefab::CSoundSource\* ss = NewGO<prefab::CSoundSource>(0);**  **ss->Init("sound/explosion.wav"); //explosion.wavをロード。**  **ss->SetVolume(0.05f); //うるさいので音を小さくする。**  **ss->Play(false); //ワンショット再生。**  DeleteGO(this); //自分自身を死亡させる。  //falseを返したらクエリは終了。  return false;  }  //trueを返したらクエリは継続。  return true;  }); |

# **Lesson 13\_2 爆発のエフェクトを発生させよう。**

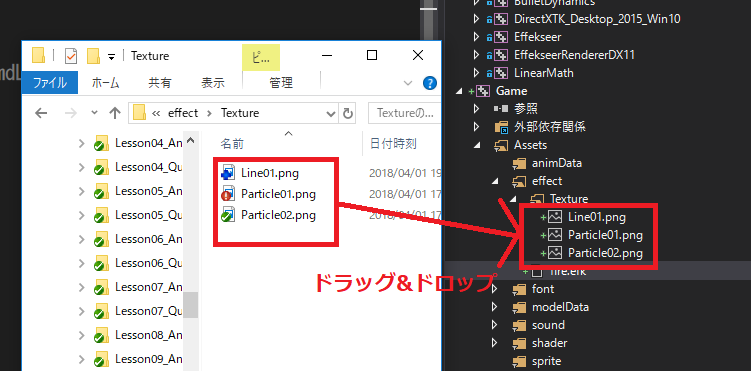
　では、続いて爆発のエフェクトを再生させてみましょう。エフェクトのデータはAssets/effectフォルダの中にあります。エフェクトをゲーム中で再生するためには、エフェクト再生用のデータの.efkファイルと、そのエフェクトで使用されているテクスチャ(画像データ)をVisualStudioに追加する必要があります。今回はfire.efkというファイルを追加します。

*fire.efxを追加*

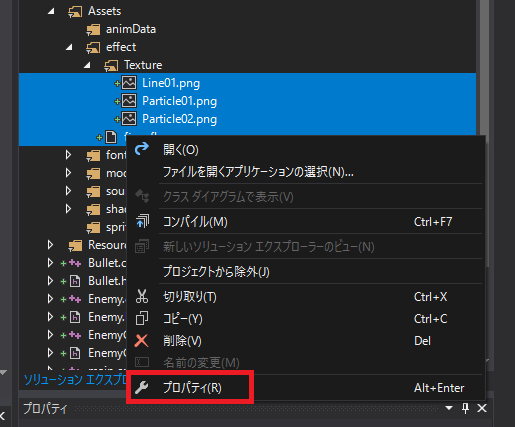


fire.efkを追加出来たら、次はテクスチャを追加しましょう。

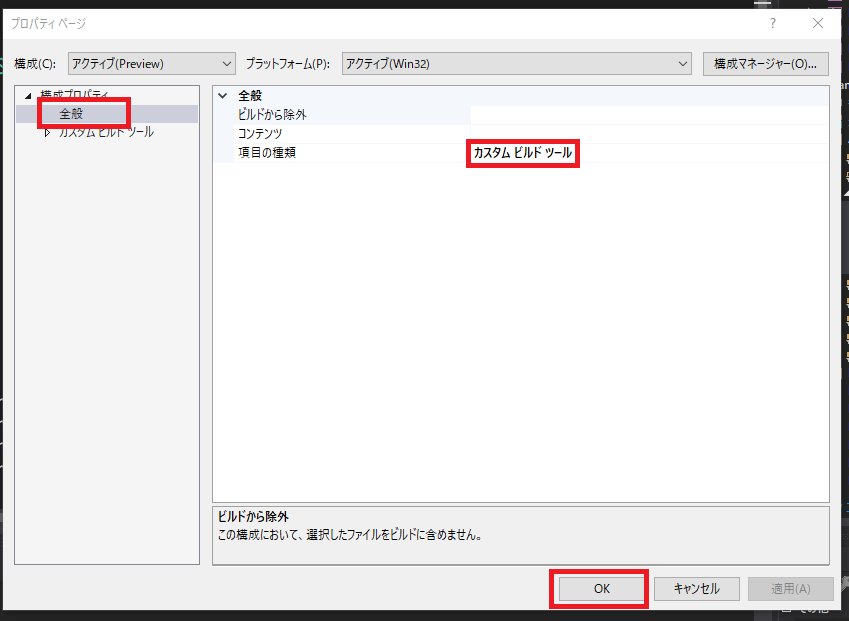
*テクスチャを追加。*



追加出来たら、これらのファイルを選択して、右クリック→プロパティを選択。



下記の図のようなダイアログボックスを開くことができたら、全般→項目の種類でカスタムビルドツールを選択してOKを押してください。



これでエフェクトデータをVisualStudioに追加することができました。あとは、敵機が死亡する時に下記の網掛けのプログラムを追加して、エフェクトを再生できるようにしましょう。

Enemy.cpp

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Enemy::Update()  {  　　　　　・  　　　　　・  　　　　　・  　　　　 省略  　　　　　・  　　　　　・  　　　　　・  //PlayerBulletという名前のゲームオブジェクトに対してクエリ(問い合わせ)を行う。  QueryGOs<Bullet>("PlayerBullet", [&](Bullet\* bullet)->bool {  //２点間の距離を計算する。  CVector3 diff = bullet->m\_position - m\_position;  if (diff.Length() < 500.0f) { //距離が500以下になったら。  //死亡。  prefab::CSoundSource\* ss = NewGO<prefab::CSoundSource>(0);  ss->Init("sound/explosion.wav"); //explosion.wavをロード。  ss->SetVolume(0.05f); //うるさいので音を小さくする。  ss->Play(false); //ワンショット再生。  **//エフェクトのインスタンスの作成。**  **prefab::CEffect\* effect = NewGO<prefab::CEffect>(0);**  **effect->Play(L"effect/fire.efk"); //fire.efkを再生。**  **effect->SetPosition(m\_position); //エフェクトの発生位置として敵機の座標を渡す。**    DeleteGO(this); //自分自身を死亡させる。  //falseを返したらクエリは終了。  return false;  }  //trueを返したらクエリは継続。  return true;  });  } |

これで、エフェクトが再生できるようになっているはずです。

# **Lesson 13\_3 中間テスト**

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSefHPvVwfEVieAHu-kF6CJqiKFPo9IjTUBUCNfapI4t3PyZjA/viewform?usp=sf_link>

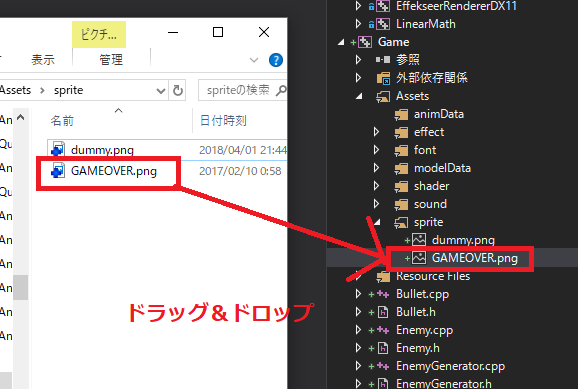
*参考プロジェクト*

*Lesson\_13の内容がすべて実装されたサンプルプログラム。*

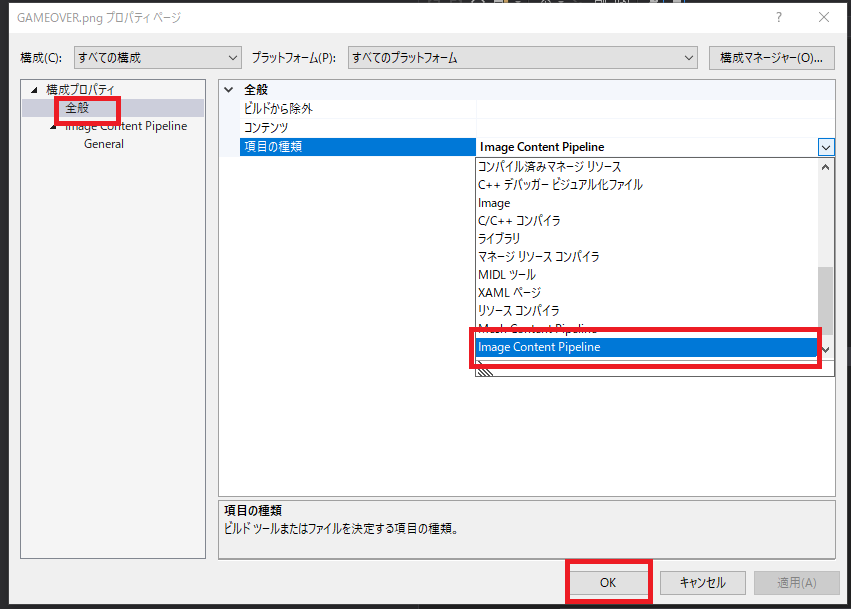
*Lesson\_13\_Answer*

# **Lesson 14\_1 GameOverの2Dを表示しよう**

　まず、GAMEOVER.pngをVisualStudioに追加しましょう。



続いて、GAMEOVER.pngを選択して右クリック→プロパティを選択して、項目の種類をImage Content Pipelineに変更してください。



これで２Ｄ画像を出す準備ができました。では、VisualStudioにGameOver2Dというクラスを追加しましょう。

追加出来たら、GameOver2D.hとGameOver2D.cppに下記のようなコードを追加してください。

GameOver2D.h

|  |
| --- |
| #pragma once  class GameOver2D **: public IGameObject**  {  public:  GameOver2D();  ~GameOver2D();  **bool Start();**  **prefab::CSpriteRender\* m\_spriteRender = nullptr; //Spriteレンダラー。**  }; |

GameOver2D.cpp

|  |
| --- |
| #include "stdafx.h"  **#include "GameOver2D.h"**  GameOver2D::GameOver2D()  {  }  GameOver2D::~GameOver2D()  {  **//スプライトレンダラーを削除。**  **DeleteGO(m\_spriteRender);**  }  **bool GameOver2D::Start()**  **{**  **//CSpriteRenderのインスタンスを生成**  **m\_spriteRender = NewGO<prefab::CSpriteRender>(0);**  **//２Ｄをロード。**  **m\_spriteRender->Init(**  **L"sprite/GAMEOVER.dds", //読み込むファイルのファイルパス。**  **1105.0f, //スプライトの幅。**  **115.0f //スプライトの高さ。**  **);**  **return true;**  **}** |

これでGameOver2Dクラスは完成です。

続いて、プレイヤーが死亡したときにGameOver2Dクラスのインスタンスを生成するようにしましょう。Player.cppに下記のコードを追加してください。

Player.cpp

|  |
| --- |
| #include "stdafx.h"  #include "Player.h"  #include "Bullet.h"  **#include "GameOver2D.h" //ヘッダのインクルードを忘れないように。** |

Player.cpp

|  |
| --- |
| //////////////////////////////////////////////////////////  // 一定間隔で呼ばれる更新処理。  //////////////////////////////////////////////////////////  void Player::Update()  {  　　　　　　・  　　　　　　・  　　　　　　・  　　　　　省略  　　　　　　・  　　　　　　・  　　　　　　・  //EnemyBulletという名前のゲームオブジェクトに対してクエリ(問い合わせ)を行う。  QueryGOs<Bullet>("EnemyBullet", [&](Bullet\* bullet)->bool {  //２点間の距離を計算する。  CVector3 diff = bullet->m\_position - m\_position;  if (diff.Length() < 500.0f) { //距離が500以下になったら。  //死亡。  prefab::CSoundSource\* ss = NewGO<prefab::CSoundSource>(0);  ss->Init("sound/explosion.wav"); //explosion.wavをロード。  ss->SetVolume(0.05f); //うるさいので音を小さくする。  ss->Play(false); //ワンショット再生。  //エフェクトのインスタンスの作成。  prefab::CEffect\* effect = NewGO<prefab::CEffect>(0);  effect->Play(L"effect/fire.efk");//fire.efkを再生。  effect->SetPosition(m\_position);//エフェクトの発生位置として敵機の座標を渡す。  **//GameOver2Dを作成する。**  **NewGO<GameOver2D>(0);**  DeleteGO(this);  //クエリ終了。  return false;  }  return true;  });  } |

これで自機が死んだときにゲームオーバーの２Ｄが表示されるようになります。実行して確認してみてください。

# **Lesson 14\_2 背景を表示してみよう。**

　続いて、背景の２Ｄを表示してみようと思います。Assets/sprite/background.jpgをVisualStudioに追加してください。Image Content Pipelineを設定するのを忘れないようにしてください。追加出来たら、Backgroundクラスを追加します。

　Background.hとBackground.cppに下記のコードを追加してください。

Background.h

|  |
| --- |
| #pragma once  class BackGround **: public IGameObject**  {  public:  BackGround();  ~BackGround();  b**ool Start();**  **prefab::CSpriteRender\* m\_spriteRender = nullptr;**  **};** |

Background.cpp

|  |
| --- |
| #include "stdafx.h"  #include "BackGround.h"  BackGround::BackGround()  {  }  BackGround::~BackGround()  {  **//スプライトレンダラーのインスタンスを削除。**  **DeleteGO(m\_spriteRender);**  }  **bool BackGround::Start()**  **{**  **m\_spriteRender = NewGO<prefab::CSpriteRender>(0);**  **m\_spriteRender->Init(**  **L"sprite/background.dds",**  **MainCamera().GetWidth(), //飛行機の後ろに背景を書くための処理。**  **MainCamera().GetHeight(), //飛行機の後ろに背景を書くための処理。**  **true**  **);**  **return true;**  **}** |

背景は飛行機(3Dモデル)の後ろに表示するので、少し特殊なことをしていますが、今は気にしないでください。

　これでBackgourndクラスの作成は完了です。あとは、インスタンスを生成して、背景が表示できるようにしましょう。main.cppを開いて、次のコードを追加してください。

Main.cpp

|  |
| --- |
| /\*!  \*@brief main.cpp  \*/  #include "stdafx.h"  #include "Player.h"  #include "EnemyGenerator.h"  **#include "Background.h"** |

Main.cpp

|  |
| --- |
| int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nCmdShow)  {  ・  　　　　・  　　　　・  　　　　省略  　　　　・  　　　　・  　　　　・  //エンジンを初期化。  if (Engine().Init(initParam) == true) {  //カメラを初期化。  MainCamera().SetTarget({ 0.0f, 0.0f, 0.0f });  MainCamera().SetNear(0.1f);  MainCamera().SetFar(15000.0f);  MainCamera().SetPosition({ 0.0f, 0.0f, -10000.0f });  MainCamera().SetUpdateProjMatrixFunc(CCamera::enUpdateProjMatrixFunc\_Ortho);  MainCamera().SetWidth(20000.0f);  MainCamera().SetHeight(20000.0f \* 9.0f / 16.0f);  MainCamera().Update();  //敵機生成器のインスタンスを作成する。  NewGO<EnemyGenerator>(0);  **//背景のインスタンスを作成する。**  **NewGO<BackGround>(0);**  //プレイヤーのインスタンスを作成。  NewGO<Player>(0);    //ゲームループを実行。  Engine().RunGameLoop();  }  //エンジンの終了処理。  Engine().Final();  return 0;  } |

これで背景が表示されるようになったと思います。実行して確認してみて下さい。

# **Lesson 14\_3 中間テスト**

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSf9f5dkknqqxwGgM6WwSFZ8ezPcgxa_rGVVpfmMPNry-8lBiA/viewform?usp=sf_link>

*参考プロジェクト*

*Lesson\_14の内容がすべて実装されたサンプルプログラム。*

*Lesson\_14\_Answer*

# **Lesson 15\_1 ゲームで使用されるデザインパターン**

　Lesson15ではシューティングゲームの実装から少し離れて、デザインパターンというものを簡単に学んでいきましょう。

デザインパターンとは、先人たちが考えた優れた設計をたくさんの人が使いやすいようにカタログ化したものです。デザインパターンで最も有名なものはGOFデザインパターンと言われる23種類のデザインパターンです。

　デザインパターンとは、この23種類だけではなく、各種分野によってさまざまなデザインパターンが存在します。今回はゲームで古くから使われていて、今現在UnityやUnrealEngineなどでも使われているデザインパターンをいくつか紹介します。当然、今まで皆さんが見てきたプログラムでも使われています。

# **Lesson 15\_2 ゲームループ**

　ゲームループとはゲームプログラミングを行う上で欠かすことができないパターンになります。ほぼすべてのゲームで採用されていて、ゲーム以外のプログラムではあまり使われません。ここまで皆さん学んできたように、画面上に表示されるゲームキャラクタというのは、３Ｄゲームであれば３Ｄ空間上でどこにいるのかというベクトル型の座標のデータを保持しています。そして、３Ｄ空間上でキャラクターを動かすためには、この座標を移動させる必要がありました。例えば、コントローラーで右方向が入力されたらX方向に10づつ移動する場合、下記のようなコードを記述してきたはずです。

if(KeyInput().IsPressA() == true ){

position.x += 10;

}

では、上のプログラムをどこで、何回実行すればいいでしょうか？もしこのコードが一度しか実行されないのであれば、キャラクターはX方向に10移動した後はピクリともしなくなるでしょう。つまりこのコードはゲーム起動中ずっと実行される必要があるのです。そのため、ゲームプログラムにはゲーム起動中は終了することのないゲームループと言われるものが実装されています。

while(true){

if(KeyInput().IsPressA() == true ){

ゲームループ

position.x += 10;

}

}

さて、これでもまだ問題が残ります。このゲームのループする速度が実行される環境に依存してしまっています。例えば、core i7を積んでいるコンピューターであれば、このループは１秒間に100万回実行されるかもしれません。しかし core i3を積んでいるコンピューターであれば、このループは１秒回に10万回しか実行されないかもしれません。このように、コンピューターのスペックによってゲームの実行速度が大きく変わってしまうと、まともにゲームを遊ぶことはできません。そのため、多くのゲームでは１秒間にループできる回数の上限を設定しています。60fpsとか30fpsとか聞いたことある人もいると思います。

60fpsで動作するゲームの場合は1秒間に60回ループします。30fpsの場合は1秒間に30回ループします。

while(true){

if(KeyInput().IsPressA() == true ){

position.x += 10;

}

**//垂直同期待ち。ここでゲームループの処理が1/60秒経過するまで待機する。**

**WaitVSync();**

}

# **Lesson 15\_3 画面のリフレッシュレート**

　先ほどの１秒間にループする回数ですが、この回数は画面のリフレッシュレートに深くかかわっています。では画面のリフレッシュレートとは一体なんなのか見てみましょう。皆さんがいつも見ているパソコンのディスプレイや、テレビに表示されている画像は、リフレッシュレートが60のモニタであれば、１秒回に６０回ほど画面が切り替わっているパラパラ漫画のようなものになっています。この画面を切り替える回数がリフレッシュレート言われるもので、１秒間に30回画面を切り替えるもののりフレッシュレートは30になります。１秒間に120回切り替えるものであれば、リフレッシュレートは120になります。基本的なゲームループは一回の画面切り替えにかかる時間の倍数を垂直同期待ちの時間に設定していることが多いです。(このリフレッシュレートをわざと考慮してないゲームも存在します。その場合、ティアリングという現象が発生してしまうことになります。)

# **Lesson 15\_4 UpdateMethodパターン**

このデザインパターンは古くはナムコが開発したギャラクシアン、現在であればUnity、UnrealEngine4まで脈々と受け継がれてきているパターンです。そして、このパターンもtkEngine2でも使用されています。

このパターンはタスクシステムやゲームオブジェクトマネージャーなど色々な名前で呼ばれていますが、このパターンを短い文章で説明すると次のようになります。

**「インスタンスを登録すると、以降自動で毎フレーム更新関数が呼び出される。」**

tkEngine2を使用しているゲームであれば、下記のようなコードが該当します。

|  |
| --- |
| //プレイヤーのインスタンスを生成して、ゲームオブジェクトマネージャーに登録。  NewGO<Player>(0); |

このコードはPlayerクラスのインスタンスを作成して、そのインスタンスをゲームオブジェクトマネージャーに登録しています。登録されたインスタンスは以降、毎フレームUpdate関数やRender関数などが自動的にで呼び出されます。では、このパターンのサンプルコードを見てみましょう。非常にシンプルなサンプルですが、これはC++のポリモーフィズム(多態性)を活用しているコードです。

|  |
| --- |
| /////////////////////////////////////////////////////////////////////////////  //ゲームオブジェクトのインターフェースクラス。  /////////////////////////////////////////////////////////////////////////////  class IGameObject{  public:  virtual void Update() = 0; //純粋仮想関数。  };  /////////////////////////////////////////////////////////////////////////////  //IGameObjectのインターフェースを継承したEnemyクラス。  /////////////////////////////////////////////////////////////////////////////  dass Enemy : public IGameObject{  public:  void Update();  };  //EnemyのUpdate関数の実装。  void Enemy::Update()  {  　std::cout << “Enemyだよ！”;  }  /////////////////////////////////////////////////////////////////////////////  //IGameObjectのインターフェースを継承したPlayerクラス。  /////////////////////////////////////////////////////////////////////////////  class Player : public IGameObject{  public:  void Update();  }  //プレイヤーのUpdate関数の実装。;  void Player::Update()  {  std::cout << “プレイヤーだよ！”;  }  /////////////////////////////////////////////////////////////////////////////  // メイン処理  /////////////////////////////////////////////////////////////////////////////  const int MAX\_GAME\_OBJECT = 64;  IGameObject\* gameObjects[64];  int main()  {  　 int numEntryGameObjct = 2;  gameObject[0] = new Enemy; //Enemyのインスタンスを生成。  　 gameObject[1] = new Player; //Playerのインスタンスを生成。  while(true){ //ゲームループ。  for(int i = 0; i < numEntryGameObject; i++){  gameObject[i]->Update(); //これでEnemyとPlayerのUpdateが実行される。  　　　　　　　　　　　　　　　　//これがポリモーフィズム！！！  }  WaitVSync();  }  } |

とても短くてシンプルなプログラムですが、ポリモーフィズムの典型的な活用例になっています。

# **Lesson 16\_1 ゲームオブジェクトマネージャー**

　Lesson 16ではtkEngine2のゲームオブジェクトマネージャーについて学んでみましょう。これはtkEngine2の中で呼称している俗称なので、一般的に通じる用語ではありませんが、処理の自体はUnity、UnrealEngine4、各ゲーム会社の内製エンジンどこにでも似たようなものが存在しています。

# **Lesson 16\_2 Update Method パターン**

これは復習になりますが、Lesson 15\_4のUpdateMethodパターンを思い出してください。このパターンは「**インスタンスを登録すると、自動的に更新関数が呼ばれる**」というものでした。ゲームオブジェクトマネージャーはこのデザインパターンを活用しています。

# **Lesson 16\_3 IGameObject　インターフェースクラス**

ゲームオブジェクトマネージャーに登録ができるのは、IGameObjectを継承したクラスのインスタンスのみです。

|  |
| --- |
| class Player : public IGameObject  {  public:  Player();  ~Player();  bool Start(); 　//IGameObjectクラスのStart関数をオーバーライド。  void Update();　 //IGameObjectクラスのUpdate関数をオーバーライド。  } |

# **Lesson 16\_4 NewGO インスタンスの生成と登録**

IGameObjectを継承したクラスのインスタンスの生成と登録はNewGOというヘルパー関数で行うことができます。

|  |
| --- |
| //エネミーを２体生成する。  NewGO<Enemy>(0);  NewGO<Enemy>(0); |

作成したインスタンスには名前を付けることもできます。

|  |
| --- |
| //エネミーを２体生成する。  NewGO<Enemy>(0, "Enemy01");  NewGO<Enemy>(0, "Enemy02"); |

第一引数の数字は作成されたインスタンスの更新関数の実行優先度です。同じ優先度の場合、先に登録された方が優先的に実行されます。例えば、のコードの場合、１フレームにUpdate関数が呼ばれる順番は、Enemy03->Enemy01->Enemy02となります。

|  |
| --- |
| //エネミーを２体生成する。  NewGO<Enemy>(0, "Enemy01");  NewGO<Enemy>(0, "Enemy02");  NewGO<Enemy>(1, "Enemy03"); |

NewGOは作成したインスタンスを返してくれるので、メンバ変数などにキャッシュしておくことができます。

|  |
| --- |
| m\_player = NewGO<Player>(0); |

# **Lesson 16\_5 FindGO インスタンスの検索**

FindGOというヘルパー関数を使用すると、名前でインスタンスを検索することができます。

|  |
| --- |
| //Enemy01という名前が付けられている、インスタンスを検索する。  Enemy\* enemy = FindGO<Enemy>(, "Enemy01"); |

こいつしか検索できない！

インスタンスの名前は下記のように、同名をつけることができます。

|  |
| --- |
| //エネミーを４体生成する。  NewGO<Enemy>(0, "Enemy");  NewGO<Enemy>(0, "Enemy");  NewGO<Enemy>(0, "Enemy");  NewGO<Enemy>(0, "Enemy"); |

このように同名のインスタンスが複数登録されている場合に、下記のようなインスタンスの検索を行った場合、一番最初に見つかったものとなります。

|  |
| --- |
| //Enemyという名前が付けられている、インスタンスを検索する。  Enemy\* enemy = FindGO<Enemy>(, "Enemy"); |

これでは困ることが多いと思います。このようなときはQueryGOsを使いましょう。

# **Lesson 16\_6 QueryGOs ゲームオブジェクトに対してクエリ(問い合わせ)を行う。**

QueryGOsというヘルパー関数は指定された名前のゲームオブジェクトのすべてに対して、クエリ(問い合わせ)を行うことができます。では、一番距離の近いEnemyという名前が付けられている、インスタンスの検索を行う問い合わせを見ていましょう。

|  |
| --- |
| float nearLen = 100000.0f; //一番近い距離を格納するための変数。  Enemy\* nearEnemy = NULL; //一番近いエネミーのインスタンスを格納するための変数。  //Enemyという名前のゲームオブジェクトに対してクエリ(問い合わせ)を行う。  QueryGOs<Enemy>("Enemy", [&](Enemy\* enemy)->bool {  //２点間の距離を計算する。  CVector3 diff = enemy->m\_position - m\_position;  if (diff.Length() < nearLen) { //こいつの方が近い。  nearEnemy = nearLen;　　　//最も近いエネミーのインスタンスを更新する。  nearLen = diff.Length(); //最も近い距離を更新する。  }  return true;  } |

Lesson 12\_3で弾丸とプレイヤーのあたり判定を行ったときに、QueryGOsを使っていました。

# **16\_7 DeleteGO インスタンスの破棄**

　インスタンスは適切なタイミングで破棄を行う必要があります。もし破棄することを忘れてしまうと、インゲーム中に遷移したのに、タイトル画面がまだ表示されているなんてことになってしまいます(Lesson 17でタイトルを実装するので、そこでさらに詳しく説明)。

IGameObjectを継承したクラスのインスタンスの破棄はDeleteGOを使う必要があります。

DeleteGOに破棄したいインスタンスのアドレスを渡すことで、破棄できます。

下記のコードはFindGOで削除したいインスタンスを検索して破棄しています。

|  |
| --- |
| Enemy\* enemy = FindGO<Enemy>("Enemey");  DeleteGO(enemy); |

メンバ変数などにキャッシュしていたアドレスを使うのもいいでしょう。

|  |
| --- |
| DeleteGO(m\_enemy); //メンバ変数にキャッシュしていたアドレスをDeleteGOに渡す。 |

# **Lesson 16\_8 中間テスト**

　下記のURLのテストを行いなさい。

# **Lesson 17\_1 シーン切り替えを行おう。**

デジタルゲームでは安全かつ完璧にシーンを切り替えることが必要になってくる場面があります。ではシーン切り替えについてみていきましょう。

# **Lesson 17\_2 スーパーマリオの場合**

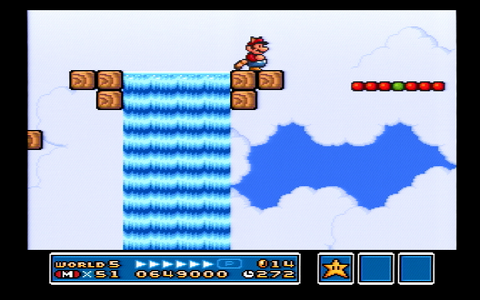
典型的なステージクリア型のアクションゲームのスーパーマリオを例に考えてみましょう。マリオは起動するとタイトル画面が表示されます。そしてタイトル画面でスタートボタンを押すとステージ選択画面に遷移します。ステージ選択画面でステージを選択するとインゲームに移動して、インゲームをクリアすると、ステージ選択画面に戻ってきます。この流れを図示化すると下記のようになります。



**タイトル画面**



**ステージ選択画面**



**インゲーム**

このように、スーパーマリオには少なくとも下記の3つのシーンがあることが想像できます。

　・タイトルシーン

　・ステージ選択シーン

　・ゲームシーン

# **Lesson 17\_3 安全かつ完璧にシーンを切り替える**

シーンの切り替えは安全かつ完璧に行う必要があります。「安全に」というのは「ゲームがクラッシュせずに」という意味で、「完璧に」というのは「不具合なく」という意味です。例えばステージ選択画面からインゲームに遷移することを考えてみましょう。その場合、プログラマは下記の点を注意する必要があります。

**・ステージ選択画面で描画されているものはすべて非表示にする必要がある**

**・ステージ選択画面で再生されているBGMはすべて停止する必要がある。**

もし、BGMを停止していなければインゲームでもステージ選択画面のBGMが再生されます。これはほとんどの場合不具合です。つまり、シーンが切り替わるときは正しく終了処理を記述する必要があるということです。

# **Lesson 17\_4 グローバル変数を利用したシーン切り替え**

　では、いくつかのシンプルなシーン切り替えの仕方を見ていきましょう。

最もシンプルな実装は下記のようにグローバル変数などを用いてシーンを切り替えることでしょう。

|  |
| --- |
| enum Scene{  scene\_Undef = -1, //シーン未定義。  scene\_Title, //タイトルシーン。  scene\_Game, //ゲームシーン。  };  Scene g\_scene; //現在のシーンを表すグローバル変数。  //シーンの切り替え  void ChangeScene(Scene nextSscene)  {  //各シーンの終了処理。  switch(g\_scene){  case scene\_Title:  //タイトルシーンの終了処理。  FinalTitleScene();  break;  case scene\_Game:  //ゲームシーンの終了処理。  FinalGameScene();  break;  }  //各シーンの初期化処理。  switch(nextScene){  case scene\_Title:  //タイトルシーンの初期化。  InitTitleScene();  break;  case scene\_Game:  //ゲームシーンの初期化。  InitGameScene();  break;  }  //現在のシーンを変更。  g\_scene = nextScene;  }  //更新処理。  void Update()  {  switch(g\_scene){  case scene\_Title:  //タイトルシーンの更新処理。  UpdateTitleScene();  break;  case scene\_Game:  //ゲームシーンの更新処理。  UpdateGameScene();  break;  }  } |

ChangeScene関数はシーンの切り替えを行う関数です。現在のシーンの終了処理を行った後で次のシーンの初期化処理を行っています。Update関数は現在のシーンに対応する更新処理を呼び出しています。

# **Lesson 17\_5ポリモーフィズムを活用したシーン切り替え**

前節のコードをもう少しマシなコードにするために、tkEngine2のゲームオブジェクトマネージャーの機能を使って、ポリモーフィズムを活用したシーン切り替えを実装してみましょう。

GameScene.h

|  |
| --- |
| class GameScene : public IGameObject{  public:  void Update();  bool Start();  }; |

GameScene.cpp

|  |
| --- |
| void GameScene::Update()  {  //ゲームシーンの更新処理を記述。  }  bool GameScene::Start()  {  //ゲームシーンの初期化処理を記述。  } |

TitleScene.h

|  |
| --- |
| class TitleScene : public IGameObject{  public:  void Update();  bool Start();  }; |

TitleScene.cpp

|  |
| --- |
| void TitleScene::Update()  {  //ゲームシーンの更新処理を記述。  if(Pad(0).IsPress(enButtonStart)){  //スタートボタンが押されたらゲームシーンに遷移する。  　　　NewGO<GameScene>(0); //ゲームシーンを生成。  DeleteGO(this);　　　　　　//自分自身を削除。  }  }  bool TitleScene::Start()  {  //ゲームシーンの初期化処理を記述。  } |

今回はIGameObjectを継承したGameSceneクラスとTitleSceneクラスを作成しました。シーンに応じて適切なシーンのインスタンスを生成してゲームオブジェクトマネージャーに登録を行っています。TitleSceneクラスのUpdate関数ではゲームシーンに遷移するサンプルコードが記述されています。GameSceneのインスタンスを生成してゲームオブジェクトマネージャーに登録を行い、TitleSceneのインスタンスを破棄しています。

# **Lesson 17\_6 Shootingゲームにタイトルシーンとゲームシーンを追加しよう。**

　まず、プロジェクトにTitleSceneクラスとGameSceneクラスを追加しましょう。

TitleSceneクラスとGameSceneクラスはIGameObjectを継承して、のように実装しておいてください。

TitleScene.h

|  |
| --- |
| #pragma once  class TitleScene : public IGameObject  {  public:  TitleScene();  ~TitleScene();  //IGameObjectのStart関数をオーバーライドする。  bool Start();  //IGameObjectのUpdate関数をオーバーライドする。  void Update();  }; |

TitleScene.cpp

|  |
| --- |
| #include "stdafx.h"  #include "TitleScene.h"  TitleScene::TitleScene()  {  }  TitleScene::~TitleScene()  {  }  bool TitleScene::Start()  {  return true;  }  void TitleScene::Update()  {  } |

GameScene.h

|  |
| --- |
| #pragma once  class GameScene : public IGameObject  {  public:  GameScene();  ~GameScene();  //IGameObjectのStart関数をオーバーライドする。  bool Start();  //IGameObjectのUpdate関数をオーバーライドする。  void Update();  }; |

GameScene.cpp

|  |
| --- |
| #include "stdafx.h"  #include "GameScene.h"  GameScene::GameScene()  {  }  GameScene::~GameScene()  {  }  bool GameScene::Start()  {  return true;  }  void GameScene::Update()  {  } |

では、続いてmain.cppを改造していきます。

これまでは、main.cppでエンジンの初期化が終わって、ゲームループに入る前に自機、敵機、背景、プレイヤーの生成を行っていましたが、エンジンの初期化が終わった後で、最初に遷移するのがタイトルシーンになったので、これらをすべてコメントアウトして、タイトルシーンのインスタンスを生成するようにコードを変更しましょう。

main.cpp

|  |
| --- |
| /\*!  \*@brief main.cpp  \*/  #include "stdafx.h"  **#include "TitleScene.h"**  **//これらは不要になる。**  **//#include "Player.h"**  **//#include "EnemyGenerator.h"**  **//#include "Background.h"**  int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nCmdShow)  {  　　　　　・  　　　　　・  　　　　省略  　　　　　・  　　　　　・  //エンジンを初期化。  if (Engine().Init(initParam) == true) {  /\*//これらはすべて不要になるのでコメントアウト。後でGame.cppに移動させるので消さないでね。  //カメラを初期化。  MainCamera().SetTarget({ 0.0f, 0.0f, 0.0f });  MainCamera().SetNear(0.1f);  MainCamera().SetFar(15000.0f);  MainCamera().SetPosition({ 0.0f, 0.0f, -10000.0f });  MainCamera().SetUpdateProjMatrixFunc(CCamera::enUpdateProjMatrixFunc\_Ortho);  MainCamera().SetWidth(20000.0f);  MainCamera().SetHeight(20000.0f \* 9.0f / 16.0f);  MainCamera().Update();  //敵機生成器のインスタンスを作成する。  NewGO<EnemyGenerator>(0);  //背景のインスタンスを作成する。  NewGO<BackGround>(0);  //プレイヤーのインスタンスを作成。  NewGO<Player>(0);  //BGMを再生。  prefab::CSoundSource\* ss = NewGO<prefab::CSoundSource>(0);  ss->Init("sound/bgm.wav");  ss->SetVolume(0.03f);  ss->Play(true);\*/ //タイトルシーンを作成。  **NewGO<TitleScene>(0);**  //ゲームループを実行。  Engine().RunGameLoop();  }  //エンジンの終了処理。  Engine().Final();  return 0;  } |

これで実行すると、なにも表示されない灰色の画面になったと思います。では、タイトルシーンでタイトル画面を表示してみましょう。

Assets/sprite/Title.pngをVisualStudioに追加して、項目の種類をImageContentPiplineに変更してゲームで使えるようにしてください。

追加出来たら、TitleScene.hとcppを下記のように変更しましょう。

TitleScene.h

|  |
| --- |
| #pragma once  class TitleScene : public IGameObject  {  public:  TitleScene();  ~TitleScene();  //IGameObjectのStart関数をオーバーライドする。  bool Start();  //IGameObjectのUpdate関数をオーバーライドする。  void Update();  //スプライトレンダラー。  prefab::CSpriteRender\* m\_spriteRender = nullptr;  }; |

TitileScene.cpp

|  |
| --- |
| #include "stdafx.h"  #include "TitleScene.h"  TitleScene::TitleScene()  {  }  TitleScene::~TitleScene()  {  **DeleteGO(m\_spriteRender);**  }  bool TitleScene::Start()  {  **//CSpriteRenderのインスタンスを作成。**  **m\_spriteRender = NewGO<prefab::CSpriteRender>(0);**  **//タイトル画像をロード。**  **m\_spriteRender->Init(L"sprite/Title.dds", 1280.0f, 720.0f);**  return true;  }  void TitleScene::Update()  {  } |

実装出来たら、実行してみてください。タイトル画面が表示されたと思います。

では、続いて、タイトル画面でAボタンが押されたらゲームシーン遷移できるようにしてみましょう。

Title.cpp

|  |
| --- |
| #include "stdafx.h"  #include "TitleScene.h"  **#include "GameScene.h"**  ・  　　・  　　省略  　　・  　　・  void TitleScene::Update()  {  **//もしもAボタンが押されたら。**  **if (Pad(0).IsPress(enButtonA)) {**  **//GameSceneのインスタンスを生成。**  **NewGO<GameScene>(0);**  **//シーンが切り替わるのでTitleSceneのインスタンスを削除。**  **DeleteGO(this);**  **}**  } |

実装出来たら、タイトル画面でAボタンを押してみてください。灰色のウィンドウが表示されたと思います。Aボタンが押されると、GameSceneクラスのインスタンスが生成されて、TitleSceneのインスタンスは削除されています。TitleSceneのデストラクタで、m\_spriteRenderが指しているインスタンスも削除されているので、タイトル画面の表示がなくなっています。これを忘れてしまうと、インゲームでもタイトルの2Dが表示されてしまいます。

　では、続いてGameクラスのStart関数で自機、敵機、背景、プレイヤーの生成を行いましょう。main.cppのコメントアウトしたコードをコピー＆ペーストすればOKです。

Game.cpp

|  |
| --- |
| #include "stdafx.h"  #include "GameScene.h"  **#include "Player.h"**  **#include "EnemyGenerator.h"**  **#include "Background.h"**  GameScene::GameScene()  {  }  GameScene::~GameScene()  {  }  bool GameScene::Start()  {  **//カメラを初期化。**  **MainCamera().SetTarget({ 0.0f, 0.0f, 0.0f });**  **MainCamera().SetNear(0.1f);**  **MainCamera().SetFar(15000.0f);**  **MainCamera().SetPosition({ 0.0f, 0.0f, -10000.0f });**  **MainCamera().SetUpdateProjMatrixFunc(CCamera::enUpdateProjMatrixFunc\_Ortho);**  **MainCamera().SetWidth(20000.0f);**  **MainCamera().SetHeight(20000.0f \* 9.0f / 16.0f);**  **MainCamera().Update();**  **//敵機生成器のインスタンスを作成する。**  **NewGO<EnemyGenerator>(0);**  **//背景のインスタンスを作成する。**  **NewGO<BackGround>(0);**  **//プレイヤーのインスタンスを作成。**  **NewGO<Player>(0);**  **//BGMを再生。**  **prefab::CSoundSource\* ss = NewGO<prefab::CSoundSource>(0);**  **ss->Init("sound/bgm.wav");**  **ss->SetVolume(0.03f);**  **ss->Play(true);**  return true;  }  void GameScene::Update()  {  } |

実装出来たら実行してみてください。タイトル画面からゲーム画面に遷移できるようになっていると思います。

# **Lesson 17\_7 まとめ**

　シーンを切り替えるということ自体は別段難しいことではありません。グローバル変数を活用してもいいですし、ポリモーフィズムを活用してもいいでしょう。シーン切り替えで重要なことは、確保したリソース(メモリやファイルハンドラなどなど)を正しく開放することです。そしてこれは注意してコードを書かないと難しいことです。ゲームを作る場合、早期の段階で簡単なシーン切り替えを用意しておき随時テストを行えるようにするべきでしょう。

# **Lesson 17\_8 中間テスト**

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSdTKr0Nb6dYasG6MPxtApv4p2CcTt049lTLvjg3SZMj9nhK7Q/viewform?usp=sf_link>

*参考プロジェクト*

*Lesson\_17の内容がすべて実装されたサンプルプログラム。*

*Lesson\_17\_Answer*