内容

[**第一章　プログラムに慣れよう** 2](#_Toc509141895)

[**Lesson\_00\_0 tkEngne2** 2](#_Toc509141896)

[**Lesson\_00\_1 IGameObject** 3](#_Toc509141897)

[**Lesson\_00\_2 クラスを追加してみよう** 3](#_Toc509141898)

[**Lesson\_00\_3 キャラクターを表示しよう** 6](#_Toc509141899)

[**Lesson\_00\_4 まとめ** 7](#_Toc509141900)

[**Lesson\_00\_05 中間テスト** 8](#_Toc509141901)

[**Lesson\_01\_1 キャラクターを動かしてみよう。** 8](#_Toc509141902)

[**Lesson\_01\_2 パッドの入力でキャラを動かしてみよう。** 9](#_Toc509141903)

[**Lesson\_01\_3 キャラクターをジャンプさせてみよう。** 10](#_Toc509141904)

[**Lesson\_01\_4 キャラクターに重力を加えてみよう。** 10](#_Toc509141905)

[**Lesson\_01\_5 中間テスト** 10](#_Toc509141906)

[**Lesson\_02\_1 キャラクターを回転させてみよう。** 11](#_Toc509141907)

[**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。** 13](#_Toc509141908)

[**Lesson\_02\_3 中間テスト** 14](#_Toc509141909)

[**Lesson\_03\_01 走るアニメーションを再生してみよう。** 14](#_Toc509141910)

[**Lesson\_03\_02 ジャンプアニメーションを再生してみよう。** 15](#_Toc509141911)

[**Lesson\_03\_03 中間テスト** 16](#_Toc509141912)

[**Lesson\_04\_01 キャラクターに攻撃させてみよう** 16](#_Toc509141913)

[**Lesson\_04\_02 正しく攻撃を終了させよう** 17](#_Toc509141914)

[**Lesson\_05 カメラを動かしてみよう。** 18](#_Toc509141915)

[**Lesson\_05\_01　注視点を動かそう** 19](#_Toc509141916)

[**Lesson\_05\_02　視点を動かそう** 19](#_Toc509141917)

[**Lesson\_05\_03 近平面と遠平面** 20](#_Toc509141918)

[**Lesson\_05\_04　画角** 21](#_Toc509141919)

[**Lesson\_05\_05　まとめ** 22](#_Toc509141920)

[**Lesson\_05\_06　中間テスト** 23](#_Toc509141921)

# **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、技術力と資金力の高い会社で使われる。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。

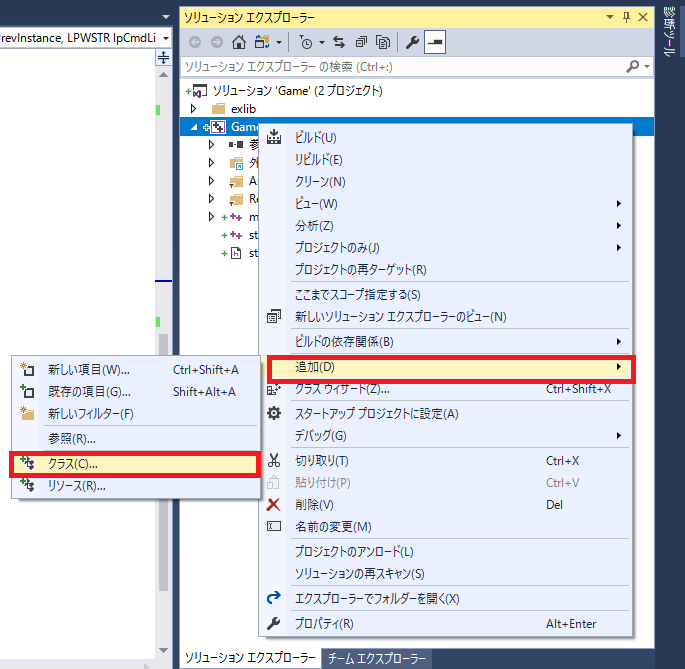


VisualStudioが立ちあがったら、Ctrl+F5を押してゲームを起動してみてください。すると下記のような灰色のウィンドウが起動したと思います。

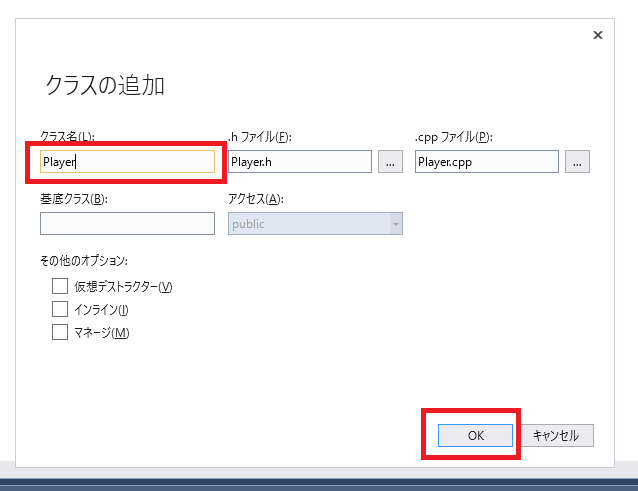


では、プログラムを追加して、キャラクターを表示してみましょう。まずはVisualStudioにPlayerクラスを追加しましょう。

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()  {  } |

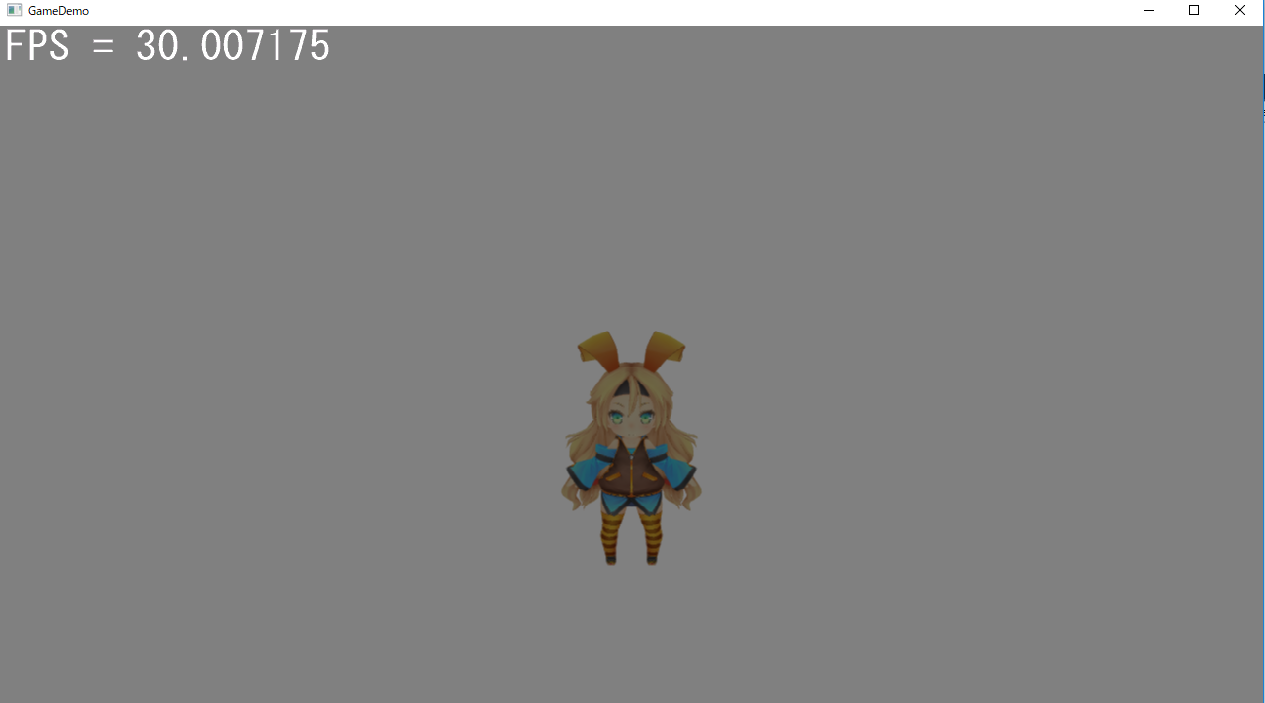
これでプレイヤークラスの作成は完了しました。しかし、クラスは設計図のようなもので、作成しただけでは何の意味もありません。自動車は設計図を用いて、車は製造されますよね？クラスも同じで設計図をもとにインスタンスを作成する必要があります。

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、網掛けんあ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。**

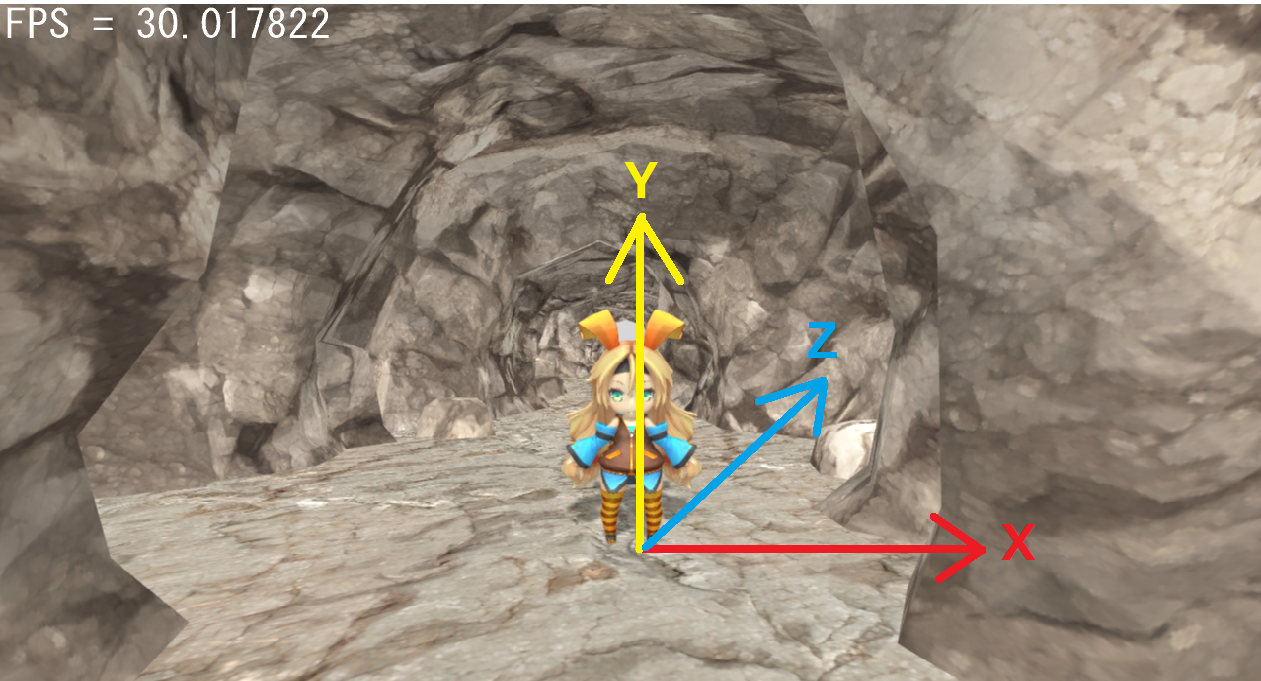
# **Lesson\_00\_05 中間テスト**

　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>

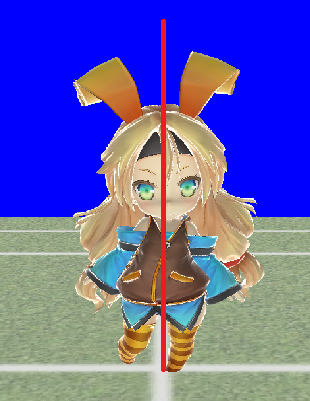
# **Lesson\_02\_1 キャラクターを回転させてみよう。**

　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るため、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。

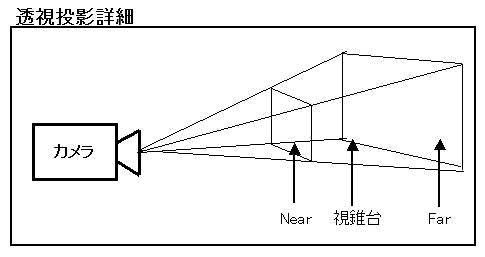
では「HandsOn 2 視点を動かす。」と記述されている箇所に次のプログラムを記述してください。

|  |
| --- |
| //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。

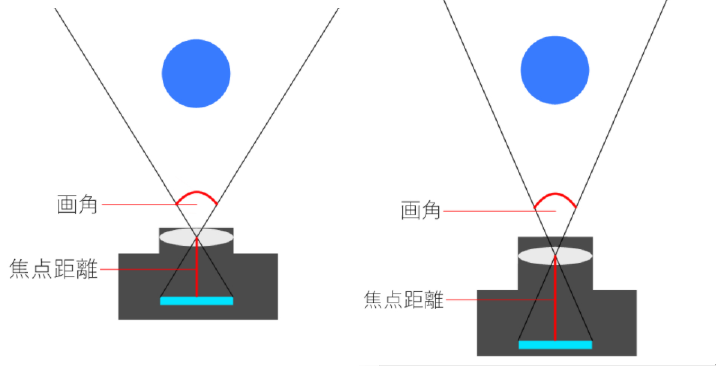
近平面(ニアクリップ)から遠平面(ファークリップ)の間にある、オブジェクトがスクリーンに表示されます。

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

**復習テスト\_01 (Lesson\_01～Lesson\_03**

MiniTest\_01を使用して下記の実装を行いなさい。

1. Player.cppの83行目「Question 1 パッドの入力でキャラクターを移動させてみよう。」と記述されている箇所にプログラムを追加して、キャラクターを移動できるようにしなさい。
2. Player.cppの「Question 2 パッドのAボタン(キーボードのJ)が押されたらキャラクターをジャンプさせてみよう。」と記述されている箇所にプログラムを追加して、キャラクターがジャンプできるようにしなさい。
3. Player.cppの「Question 3 パッドの入力でキャラクターの方向を変えられるようにしてみよう。」と記述されている箇所にプログラムを追加して、キャラクターの方向を変えられるようにしなさい。
4. Player.cppの「Question 4 パッドの入力で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プログラムを追加して、走りアニメーションを再生できるようにしなさい。
5. Player.cppの「Question 5 Aボタン(キーボードのJ)が押されたら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プログラムを追加して、ジャンプアニメーションを再生できるようにしなさい。
6. ゲームパッドのＢボタン(キーボードのＫ)が押されていると移動速度が2倍になるようにしなさい。(応用)

**Lesson 7 モンスターにプレイヤーを追いかけさせよう。**

　Lesson７ではプレイヤーとモンスターの距離が一定値以下になったら、プレイヤーを追いかけるという処理を追加してみましょう。

**Lesson 7\_01 有限状態機械(FSM)**  ここまではフラグというものを使って攻撃中かどうかを判定していました。しかし、ディジタルゲームのキャラクターは攻撃中だけではなく、逃走中、追跡中、休憩中などといった複数の状態を持つことがほとんどです。これらにフラグを使って管理をしていると、非常に煩雑になってしまい、プログラムが複雑になっていきます。そこで今回はenum(列挙型)を使用して有限状態機械(FSM)を使用して、プログラムを実装してみようと思います。

Lesson\_07/Question/Game/Enemy.h(25行目)

|  |
| --- |
| enum State {  State\_Idle, //待機状態  State\_Chase, //プレイヤーを追いかける。  State\_Attack, //攻撃状態。  }; |

Enemy.hにStateという名前の列挙型が用意されています。そして、32行目にState型のstateという変数が用意されています。この変数にモンスターの現在の状態を記録することでFSMを行えるようにしていきます。

**Lesson 7\_02 追跡中に一定の距離以下まで追いついたら攻撃を行うようにする**

　Questionのプログラムを実行すると、プレイヤーとモンスターの距離が一定値以下になると、モンスターが歩きモーションの再生を行います。Enemy.cppの63行目～66行目までのプログラムがモンスターに歩きモーションを再生させて、追跡状態に変更しているコードです。

Enemy.cpp(63行目～66行目)

|  |
| --- |
| else if (diff.Length() < 5.0f) {  animation.PlayAnimation(AnimationWalk, 0.2f);  state = State\_Chase;  } |

しかし、モンスターは歩きアニメーションを再生するだけで、プレイヤーを追いかけてきません。「Question1 プレイヤーの方に向かって進む。」と書かれている箇所に次のプログラムを記述してください。

|  |
| --- |
| //Question1 プレイヤーの方に向かって進む。  //(ヒント)  // プレイヤーの方に進むという処理を行うためには、プレイヤーの方向を向いている向きベクトルを求める必要がある。  // diffにはエネミーからプレイヤーまでのベクトルが入っている。  // このベクトルの大きさを１にすると(正規化)向きベクトルになる。  CVector3 toPlayerDirection = diff;  toPlayerDirection.Normalize();  moveSpeed = toPlayerDirection \* 2.0f; |

プレイヤーの方に向かって移動するようになったと思います。

**実習**

1. 「Question 2 プレイヤーとの距離が2m以下になったら攻撃を行う。」と記述されている箇所にプログラムを追加して、追跡中にプレイヤーに追いついたら攻撃を行うようにしなさい。
2. 「Question3　プレイヤーと敵の距離が10m以上になったら、追跡終了で待機状態に戻す。」と記述されている箇所にプログラムを追加して、モンスターが追跡をあきらめるようにしなさい。

**Lesson 8 ダメージ判定を実装しよう**

Lesson７までの実装でモンスターとプレイヤーが攻撃モーションを再生できるようになりました。Lesson8ではダメージ判定を実装して、攻撃をヒットさせてダメージモーションを再生できるようにしてみましょう。

**Lesson 8\_01 コリジョン**

　ゲームプログラミングでコリジョンというキーワードが出来たら、衝突データや衝突判定を意味しています。コリジョンデータには色々な形状があります。ボックス、カプセル、球etc。今回は最も簡単な球と球の衝突見ていきましょう。球と球の衝突判定は２点間の距離を計算することで行えます。

半径

半径

球と球の2点間の距離が二つの球の半径の合計以下になったら諸突していると判定できます。Enemy.cppの87行目～108行目を見てみてください。

|  |
| --- |
| //ダメージ判定。  void Enemy::CheckDamage()  {  if (state == State\_Damage) {  //ダメージ中ならリターン。  return;  }  //プレイヤーの攻撃コリジョンとのあたり判定を行う。  int numCollision = collisionWorld->m\_collisionList.size();  for (int i = 0; i < numCollision; i++) {  if (collisionWorld->m\_collisionList[i]->attr == enCollisionAttr\_PlayerAttack) {  //プレイヤーが発生させた攻撃コリジョンとのあたりを調べる。  //コリジョンとの距離を調べる。  CVector3 diff = collisionWorld->m\_collisionList[i]->pos - position;  if (diff.Length() < 3.0f) {  //ダメージを受ける。  animation.PlayAnimation(AnimationDamage, 0.2f);  state = State\_Damage;  }  }  }  } |

これはプレイヤーが発生させた攻撃コリジョンとモンスターのあたり判定を行っているコードです。プレイヤーが複数の攻撃コリジョンを発生させる可能性があるので、ループを使用していることに注意してください。

では、続いて攻撃コリジョンを発生させているコードを見てみましょう。Player.cppの188行目を見てみてください。

|  |
| --- |
| //攻撃コリジョンを発生させる。  EmitAttackCollision(position, 0.2f); |

これが攻撃コリジョンを発生させているコードです。第一引数は攻撃コリジョンの位置、第二引数は攻撃コリジョンが発生するまでの遅延時間(単位：秒)です。

**実習**

1. 「Question 1 攻撃コリジョンを発生させる。」と記述されている箇所に攻撃コリジョンを発生させるコードを追加して、中攻撃でもモンスターにダメージを与えられるようにしなさい。
2. 「Question ２ 攻撃コリジョンを発生させる。」と記述されている箇所に攻撃コリジョンを発生させるコードを追加して、大攻撃でもモンスターにダメージを与えられるようにしなさい。
3. 「 Question 3プレイヤーのダメージ判定。」と記述されている箇所にコードを追加して、プレイヤーもダメージを受けるようにしなさい。参考になるコードはEnemyのダメージ判定のプログラムです。

**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 9 自機と地面を表示してみよう。**

　Shooting\_00のプログラムを実行すると、空が一面に広がる画面が表示されたはずです。Lesson 9ではこの世界に自機と地面を表示するプログラムを実装しましょう。



**Lesson 9\_01 wWinMain関数**

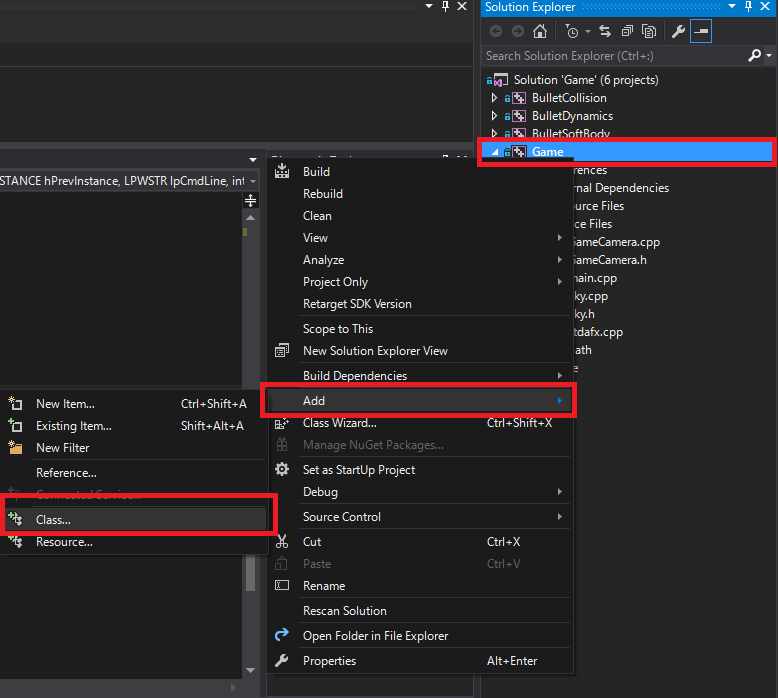
　main.cppの70行目のwWinMain関数を見てみてください。実はこの関数はWindowsプログラムのメイン関数と呼ばれるもので，C言語のmain関数と同じでようなものになります。プログラムが起動するとこの関数が実行され、Engine().RunGameLoop();というプログラムが実行されると、ゲームループと呼ばれる一定間隔でループする処理が動き出します。

|  |
| --- |
| int WINAPI wWinMain(  HINSTANCE hInst,  HINSTANCE hPrevInstance,  LPWSTR lpCmdLine,  int nCmdShow  )  {  //tkEngineの初期化。  InitTkEngine( hInst );  //空を追加。  NewGO<::Sky>(0, "Sky");  //ゲームカメラを追加。  GameCamera\* cam = NewGO<GameCamera>(0, "GameCamera");  ShadowMap().SetCamera(cam->m\_camera);  Engine().RunGameLoop(); //ゲームループを実行。  return 0;  } |
|  |

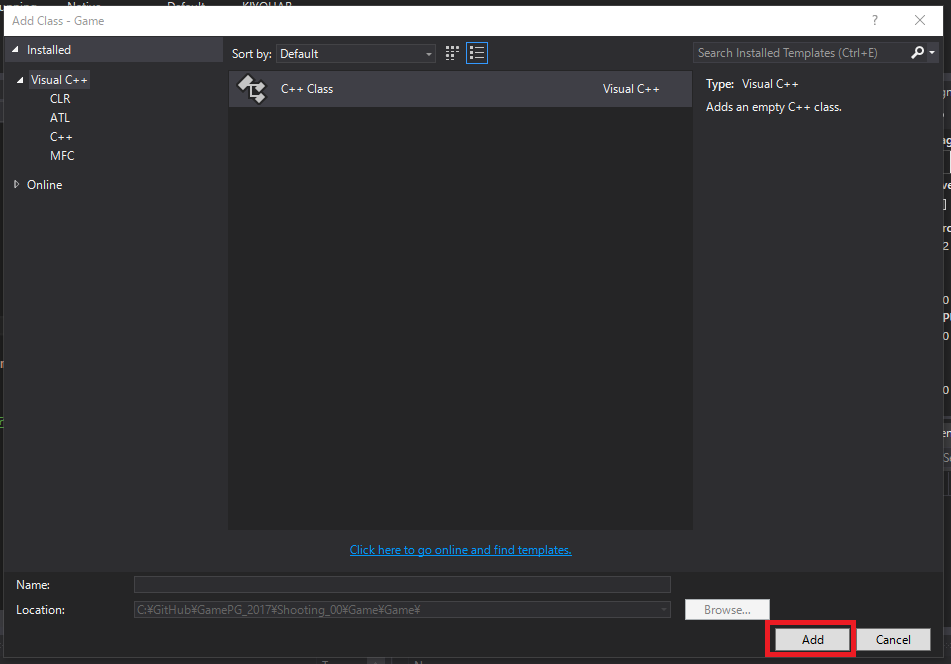
ゲームループが起動する前にNewGO<::Sky>(0, "Sky");というコードがあります。これが空を作成しているコードです。試しにこの行をコメントアウトしてみてください。空が表示されなくなります。コメントアウトすると空が消えることが確認できた人は、コメントアウトを元に戻して、空を復活させておいて下さい。

**Lesson 9\_02 プレイヤーのクラスの追加**

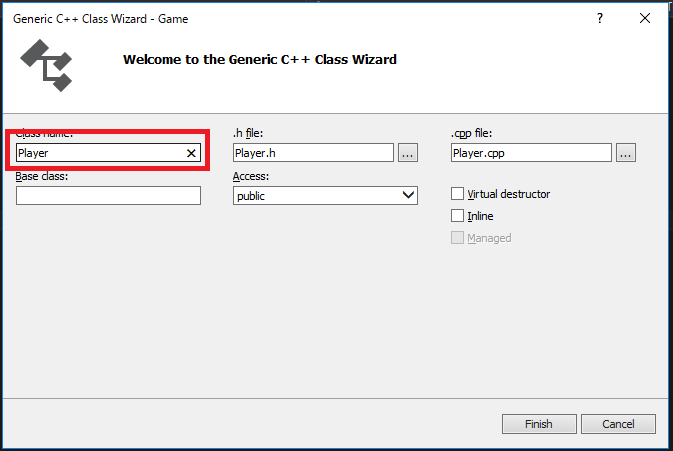
では、自機(プレイヤー)のプログラムを記述するためにプレイヤーのクラスを作りましょう。ソリューションエクスプローラーのGameプロジェクトを選択して右クリック/追加/クラスを選択してください。

****

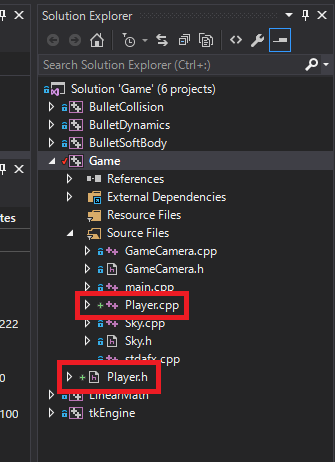
選択すると次のウィンドウが開くので追加を選択してください。



Addを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 9\_02 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:  ///////////////////////////////////////////////////////  //ここからメンバ関数。  ///////////////////////////////////////////////////////  Player();  ~Player();  bool Start() override;  void Update() override;  void Render(CRenderContext& rc) override;  ///////////////////////////////////////////////////////  //ここからメンバ変数。  ///////////////////////////////////////////////////////  CSkinModelDataHandle m\_skinModelData; 　　　//!<スキンモデルデータ。  CSkinModel m\_skinModel; 　　　　　//!<スキンモデル。  CAnimation m\_animation; 　　　　　//!<アニメーション。  CVector3 m\_position = CVector3::Zero; //!<座標。  CQuaternion m\_rotation = CQuaternion::Identity; //!<回転。  }; |

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"  #include "GameCamera.h"  Player::Player()  {  }  Player::~Player()  {  }  ///////////////////////////////////////////////  //インスタンスを生成して一度だけ呼ばれる処理。  ///////////////////////////////////////////////  bool Player::Start()  {  //3Dモデルデータをロードして初期化。  m\_skinModelData.LoadModelData("Assets/modelData/plane.X", &m\_animation);  m\_skinModel.Init(m\_skinModelData.GetBody());  m\_skinModel.SetLight(g\_defaultLight);  return true;  }  ///////////////////////////////////////////////  //毎フレーム呼ばれる更新処理。  //座標の移動とかはここに書くといいよ。  ///////////////////////////////////////////////  void Player::Update()  {  //モデルのワールド行列を更新。  m\_skinModel.Update(m\_position, m\_rotation, CVector3::One);  }  ///////////////////////////////////////////////  //毎フレーム呼ばれる描画処理。  //モデルの描画処理。  ///////////////////////////////////////////////  void Player::Render(CRenderContext& rc)  {  //GameCameraを検索。  GameCamera\* gcam = FindGO<GameCamera>("GameCamera");  m\_skinModel.Draw(  rc,  gcam->m\_camera.GetViewMatrix(),  gcam->m\_camera.GetProjectionMatrix());  } |

これでPlayerクラスが作成されました。Player::Start関数はPlayerのインスタンス(実態)が作成されたときに一度だけ呼ばれる関数です。Player::Update関数とPlayer::Render関数はゲームループから毎フレーム呼ばれる関数です。Update関数は座標の更新などを記述することが向いています。Render関数は3Dモデルなど、絵を描くための処理を記述するためのものです。

**Lesson 9\_03 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tkEngine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 hInst,  HINSTANCE hPrevInstance,  LPWSTR lpCmdLine,  int nCmdShow  )  {  //tkEngineの初期化。  InitTkEngine(hInst);  //空を追加。  NewGO<::Sky>(0, "Sky");  **NewGO<Player>(0, "Player");**  //ゲームカメラを追加。  GameCamera\* cam = NewGO<GameCamera>(0, "GameCamera");  ShadowMap().SetCamera(cam->m\_camera);  Engine().RunGameLoop(); //ゲームループを実行。  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。

|  |
| --- |
| #include "stdafx.h"  #include "GameCamera.h"  **#include "Player.h"**  #include "Sky.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。



**実習課題**　Groundクラスを作成して、地面を表示することができるようにしなさい。地面のモデルは次のパスにあります。Assets/modelData/ground.x

*参考プロジェクト*

*Lesson\_9の内容がすべて実装されたサンプルプログラム。*

*Shooting\_00\_Answer*

**Lesson 10 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 10\_1 Move関数を追加する。**　Game/Player.hを開いて、PlayerクラスにMove関数の宣言を追加してください。

|  |
| --- |
| class Player : public IGameObject  {  public:  ///////////////////////////////////////////////////////  //ここからメンバ関数。  ///////////////////////////////////////////////////////  Player();  ~Player();  bool Start() override;  void Update() override;  void Render(CRenderContext& rc) override;  **void Move();**  ///////////////////////////////////////////////////////  //ここからメンバ変数。  ///////////////////////////////////////////////////////  CSkinModelDataHandle m\_skinModelData; //!<スキンモデルデータ。  CSkinModel m\_skinModel; //!<スキンモデル。  CAnimation m\_animation; //!<アニメーション。  CVector3 m\_position = CVector3::Zero; //!<座標。  CQuaternion m\_rotation = CQuaternion::Identity; //!<回転。  }; |

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

|  |
| --- |
| ///////////////////////////////////////////////  //移動処理。  ///////////////////////////////////////////////  void Player::Move()  {  if (Pad(0).IsPress(enButtonLeft)) {  m\_position.x += 1.5;  }  if (Pad(0).IsPress(enButtonRight)) {  m\_position.x -= 1.5;  }  } |

Move関数は実装しただけでは何も起きません。関数はきちんと呼び出しを行う必要があります。Update関数からMove関数を呼び出すようにしてみます。

|  |
| --- |
| ///////////////////////////////////////////////  //毎フレーム呼ばれる更新処理。  //座標の移動とかはここに書くといいよ。  ///////////////////////////////////////////////  void Player::Update()  {  //移動処理を呼び出す。  **Move();**  //モデルのワールド行列を更新。  m\_skinModel.Update(m\_position, m\_rotation, CVector3::One);  } |

これで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。

**実習課題**　Move関数中身を改造して、上下のキーの入力で自機が上下の方向に移動するようにしなさい。

**Lesson 10\_2 Move関数に自機を回転させる処理を追加する。**

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

|  |
| --- |
| ///////////////////////////////////////////////  //移動処理。  ///////////////////////////////////////////////  void Player::Move()  {  if (Pad(0).IsPress(enButtonLeft)) {  m\_position.x += 1.5f;  //Z軸周りに-10.0°旋回する。  **m\_rotation.SetRotationDeg(CVector3::AxisZ, -10.0f);**  }  if (Pad(0).IsPress(enButtonRight)) {  m\_position.x -= 1.5f;  //Z軸周りに10.0°旋回する。  **m\_rotation.SetRotationDeg(CVector3::AxisZ, 10.0f);**  }  } |

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。

**実習課題**　Move関数中身を改造して、上に移動する時は自機が上を向く、下に移動するときは自機が下を向くようにしなさい。

**Lesson 10\_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。

|  |
| --- |
| void Player::Move()  {  if (Pad(0).IsPress(enButtonLeft)) {  m\_position.x += 1.5f;  //Z軸周りに-10.0°旋回する。  m\_rotation.SetRotationDeg(CVector3::AxisZ, -10.0f);  }  if (Pad(0).IsPress(enButtonRight)) {  m\_position.x -= 1.5f;  //Z軸周りに10.0°旋回する。  m\_rotation.SetRotationDeg(CVector3::AxisZ, 10.0f);  }  //もしもm\_position.xが150よりも大きくなったら。  if (m\_position.x > 150.0f) {  //150を代入する。  m\_position.x = 150.0f;  }  //もしもm\_position.xが150よりも小さくなったら。  if (m\_position.x < -150.0f) {  //-150を代入する。  m\_position.x = -150.0f;  }  } |

**実習課題**　Move関数中身を改造して、上下に移動する時にも制限を設けなさい。

*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。*

*Shooting\_01\_Answer*

**Lesson 11 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。**

**Lesson 11\_1 Bulletクラスを作成する。**　Lesson 9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  #include "GameCamera.h"  class Bullet : public IGameObject  {  public:  ///////////////////////////////////////////////////////  //ここからメンバ関数。  ///////////////////////////////////////////////////////  Bullet();  ~Bullet();  bool Start() override;  void Update() override;  void Render(CRenderContext& rc) override;  ///////////////////////////////////////////////////////  //ここからメンバ変数。  ///////////////////////////////////////////////////////  CSkinModelDataHandle m\_skinModelData; //!<スキンモデルデータ。  CSkinModel m\_skinModel; //!<スキンモデル。  CVector3 m\_position = CVector3::Zero; //!<座標。  GameCamera\* m\_gameCamera = NULL; //!<ゲームカメラ。  }; |

Bullet.cpp

|  |
| --- |
| #include "stdafx.h"  #include "Bullet.h"  Bullet::Bullet()  {  }  Bullet::~Bullet()  {  }  ///////////////////////////////////////////////  //インスタンスを生成して一度だけ呼ばれる処理。  ///////////////////////////////////////////////  bool Bullet::Start()  {  //3Dモデルデータをロードして初期化。  m\_skinModelData.LoadModelData("Assets/modelData/bullet.x", NULL);  m\_skinModel.Init(m\_skinModelData.GetBody());  //ライトを設定。  m\_skinModel.SetLight(g\_defaultLight);  //毎フレームFindGOするのは重いので、初期化の時にゲームカメラを検索しておく。  m\_gameCamera = FindGO<GameCamera>("GameCamera");  return true;  }  ///////////////////////////////////////////////  //毎フレーム呼ばれる更新処理。  //座標の移動とかはここに書くといいよ。  ///////////////////////////////////////////////  void Bullet::Update()  {  //弾のモデルが小さすぎるので拡大率に30倍を設定する。  CVector3 scale;  scale.x = 30.0f;  scale.y = 30.0f;  scale.z = 30.0f;  m\_skinModel.Update(m\_position, CQuaternion::Identity, scale);  }  ///////////////////////////////////////////////  //毎フレーム呼ばれる描画処理。  //モデルの描画処理。  ///////////////////////////////////////////////  void Bullet::Render(CRenderContext& rc)  {  m\_skinModel.Draw(  rc,  m\_gameCamera->m\_camera.GetViewMatrix(),  m\_gameCamera->m\_camera.GetProjectionMatrix()  );  } |

これでBulletクラスの作成はひとまず完了です。弾のモデルが小さすぎるので、拡大率を設定している箇所に注意してください。

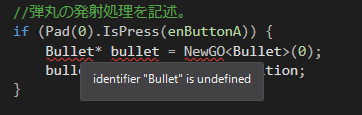
**Lesson 11\_1 Bulletクラスのインスタンス化**

　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

|  |
| --- |
| void Player::Update()  {  //移動処理を呼び出す。  Move();  //モデルのワールド行列を更新。  m\_skinModel.Update(m\_position, m\_rotation, CVector3::One);  //弾丸の発射処理を記述。  **if (Pad(0).IsPress(enButtonA)) {**  **Bullet\* bullet = NewGO<Bullet>(0);**  **bullet->m\_position = m\_position;**  **}**  } |

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 "GameCamera.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 11\_2 弾丸を動かそう。**　Lesson 11\_1の作業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

|  |
| --- |
| class Bullet : public IGameObject  {  public:  ///////////////////////////////////////////////////////  //ここからメンバ関数。  ///////////////////////////////////////////////////////  Bullet();  ~Bullet();  bool Start() override;  void Update() override;  void Render(CRenderContext& rc) override;  ///////////////////////////////////////////////////////  //ここからメンバ変数。  ///////////////////////////////////////////////////////  CSkinModelDataHandle m\_skinModelData; //!<スキンモデルデータ。  CSkinModel m\_skinModel; //!<スキンモデル。  CVector3 m\_position = CVector3::Zero; //!<座標。  **CVector3 m\_moveSpeed = CVector3::Zero; //!<移動速度。**  GameCamera\* m\_gameCamera = NULL; //!<ゲームカメラ。  }; |

Bullet.cpp

|  |
| --- |
| void Bullet::Update()  {  //弾を移動させる。  **m\_position += m\_moveSpeed;**  //弾のモデルが小さすぎるので拡大率に30倍を設定する。  CVector3 scale;  scale.x = 30.0f;  scale.y = 30.0f;  scale.z = 30.0f;  m\_skinModel.Update(m\_position, CQuaternion::Identity, scale);\  } |

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。

|  |
| --- |
| void Player::Update()  {  //移動処理を呼び出す。  Move();  //モデルのワールド行列を更新。  m\_skinModel.Update(m\_position, m\_rotation, CVector3::One);  //弾丸の発射処理を記述。  if (Pad(0).IsPress(enButtonA)) {  Bullet\* bullet = NewGO<Bullet>(0);  bullet->m\_position = m\_position;  **bullet->m\_moveSpeed.x = 0.0f;**  **bullet->m\_moveSpeed.y = 0.0f;**  **bullet->m\_moveSpeed.z = 10.0f;**  }  } |

**実習課題**　ここまでの実装では、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プログラムを改造しなさい。

**Lesson 11\_3 弾丸に寿命を設定しよう**　さて、Lesson 11\_2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 11\_3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に寿命を表すm\_lifeというメンバ変数を追加してください。

|  |
| --- |
| #pragma once  #include "GameCamera.h"  class Bullet : public IGameObject  {  public:  ///////////////////////////////////////////////////////  //ここからメンバ関数。  ///////////////////////////////////////////////////////  Bullet();  ~Bullet();  bool Start() override;  void Update() override;  void Render(CRenderContext& rc) override;  ///////////////////////////////////////////////////////  //ここからメンバ変数。  ///////////////////////////////////////////////////////  CSkinModelDataHandle m\_skinModelData; //!<スキンモデルデータ。  CSkinModel m\_skinModel; //!<スキンモデル。  CVector3 m\_position = CVector3::Zero; //!<座標。  CVector3 m\_moveSpeed = CVector3::Zero; //!<移動速度。  GameCamera\* m\_gameCamera = NULL; //!<ゲームカメラ。  **int m\_life = 60; //!<寿命。**  }; |

では続いて、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。

|  |
| --- |
| ///////////////////////////////////////////////  //毎フレーム呼ばれる更新処理。  //座標の移動とかはここに書くといいよ。  ///////////////////////////////////////////////  void Bullet::Update()  {  //弾を移動させる。  m\_position += m\_moveSpeed;  //弾のモデルが小さすぎるので拡大率に30倍を設定する。  CVector3 scale;  scale.x = 30.0f;  scale.y = 30.0f;  scale.z = 30.0f;  m\_skinModel.Update(m\_position, CQuaternion::Identity, scale);  **//寿命を減らす。**  **m\_life--;**  **if (m\_life == 0) {**  **//死亡。**  **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 12 小テスト

　敵機を10体表示しなさい。敵機のモデルはAssets/modelData/enemy.xを使用しなさい。

テストができた人は下記のチェック項目にチェックを入れてください。

1. Enemyクラスを作成できた。(25点)
2. Enemyクラスのインスタンスを10個生成できた。(25点)
3. 敵機は重なることなく10体表示できた。(25点)
4. 敵機は自機と相対するように表示できている。(25点)

**Lesson 12-1 小テスト解説(Shooting\_03\_Answerを参照)**

1. Enemyクラスを作成できた。

Enemyクラスを作るにはenemy.hとenemy.cppを追加する必要がある。また3Dモデルを表示する機能が必要になるため、モデル表示に必要なメンバ変数、メンバ関数が必要になる。下記はenemyクラスの解答例です。

enemy.h

|  |
| --- |
| #pragma once  class Enemy : public IGameObject  {  public:  ///////////////////////////////////////////////////////  //ここからメンバ関数。  ///////////////////////////////////////////////////////  Enemy();  ~Enemy();  bool Start() override;  void Update() override;  void Render(CRenderContext& rc) override;  ///////////////////////////////////////////////////////  //ここからメンバ変数。  ///////////////////////////////////////////////////////  CSkinModelDataHandle m\_skinModelData; //!<スキンモデルデータ。  CSkinModel m\_skinModel; //!<スキンモデル。  CAnimation m\_animation; //!<アニメーション。  CVector3 m\_position = CVector3::Zero; //!<座標。  CQuaternion m\_rotation = CQuaternion::Identity; //!<回転。  }; |

enemy.cpp

|  |
| --- |
| #include "stdafx.h"  #include "Enemy.h"  #include "GameCamera.h"  Enemy::Enemy()  {  }  Enemy::~Enemy()  {  }  bool Enemy::Start()  {  //3Dモデルデータをロードして初期化。  m\_skinModelData.LoadModelData("Assets/modelData/enemy.X", &m\_animation);  m\_skinModel.Init(m\_skinModelData.GetBody());  m\_skinModel.SetLight(g\_defaultLight);  return true;  }  void Enemy::Update()  {  //モデルのワールド行列を更新。  m\_skinModel.Update(m\_position, m\_rotation, CVector3::One);  }  void Enemy::Render(CRenderContext& rc)  {  //GameCameraを検索。  GameCamera\* gcam = FindGO<GameCamera>("GameCamera");  m\_skinModel.Draw(  rc,  gcam->m\_camera.GetViewMatrix(),  gcam->m\_camera.GetProjectionMatrix());  } |

1. Enemyクラスのインスタンスを10個生成できた。(25点)
2. 敵機は重なることなく10体表示できた。(25点)
3. 敵機は自機と相対するように表示できている。(25点)

１でEnemyクラスは作成できたので、あとはインスタンス(実体)を作成することでUpdateやRenderが呼ばれるようになります。今回は10体Enemyを表示する必要があるので、Enemyのインスタンスを10個作成しましょう。重ならないように、座標を設定しているのと、相対するためにクォータニオンを使用して。180度回転させていることに注意してください。

main.cpp

|  |
| --- |
| int WINAPI wWinMain(  HINSTANCE hInst,  HINSTANCE hPrevInstance,  LPWSTR lpCmdLine,  int nCmdShow  )  {  //tkEngineの初期化。  InitTkEngine(hInst);    NewGO<::Sky>(0, "Sky"); //空を追加。  NewGO<Player>(0, "Player"); //プレイヤーを追加。  NewGO<Ground>(0); //地面を追加。  **//敵１体目。**  **Enemy\* enemy = NewGO<Enemy>(0, "Enemy");**  **enemy->m\_position.x = 0.0f;**  **enemy->m\_position.y = 0.0f;**  **enemy->m\_position.z = -100.0f;**  **enemy->m\_rotation.SetRotationDeg(CVector3::AxisY, 180.0f);**  **//敵２体目。**  **enemy = NewGO<Enemy>(0, "Enemy");**  **enemy->m\_position.x = 80.0f;**  **enemy->m\_position.y = 0.0f;**  **enemy->m\_position.z = -100.0f;**  **enemy->m\_rotation.SetRotationDeg(CVector3::AxisY, 180.0f);**  **//敵３体目。**  **enemy = NewGO<Enemy>(0, "Enemy");**  **enemy->m\_position.x = 160.0f;**  **enemy->m\_position.y = 0.0f;**  **enemy->m\_position.z = -100.0f;**  **enemy->m\_rotation.SetRotationDeg(CVector3::AxisY, 180.0f);**  **//敵４体目。**  **enemy = NewGO<Enemy>(0, "Enemy");**  **enemy->m\_position.x = -80.0f;**  **enemy->m\_position.y = 0.0f;**  **enemy->m\_position.z = -100.0f;**  **enemy->m\_rotation.SetRotationDeg(CVector3::AxisY, 180.0f);**  **//敵５体目。**  **enemy = NewGO<Enemy>(0, "Enemy");**  **enemy->m\_position.x = -160.0f;**  **enemy->m\_position.y = 0.0f;**  **enemy->m\_position.z = -100.0f;**  **enemy->m\_rotation.SetRotationDeg(CVector3::AxisY, 180.0f);**  **//敵６体目。**  **enemy = NewGO<Enemy>(0, "Enemy");**  **enemy->m\_position.x = 0.0f;**  **enemy->m\_position.y = 50.0f;**  **enemy->m\_position.z = -100.0f;**  **enemy->m\_rotation.SetRotationDeg(CVector3::AxisY, 180.0f);**  **//敵７体目。**  **enemy = NewGO<Enemy>(0, "Enemy");**  **enemy->m\_position.x = 80.0f;**  **enemy->m\_position.y = 50.0f;**  **enemy->m\_position.z = -100.0f;**  **enemy->m\_rotation.SetRotationDeg(CVector3::AxisY, 180.0f);**  **//敵８体目。**  **enemy = NewGO<Enemy>(0, "Enemy");**  **enemy->m\_position.x = 160.0f;**  **enemy->m\_position.y = 50.0f;**  **enemy->m\_position.z = -100.0f;**  **enemy->m\_rotation.SetRotationDeg(CVector3::AxisY, 180.0f);**  **//敵９体目。**  **enemy = NewGO<Enemy>(0, "Enemy");**  **enemy->m\_position.x = -80.0f;**  **enemy->m\_position.y = 50.0f;**  **enemy->m\_position.z = -100.0f;**  **enemy->m\_rotation.SetRotationDeg(CVector3::AxisY, 180.0f);**  **//敵１０体目。**  **enemy = NewGO<Enemy>(0, "Enemy");**  **enemy->m\_position.x = -160.0f;**  **enemy->m\_position.y = 50.0f;**  **enemy->m\_position.z = -100.0f;**  **enemy->m\_rotation.SetRotationDeg(CVector3::AxisY, 180.0f);**  //ゲームカメラを追加。  GameCamera\* cam = NewGO<GameCamera>(0, "GameCamera");  ShadowMap().SetCamera(cam->m\_camera);  Engine().RunGameLoop(); //ゲームループを実行。  return 0;  } |

**Lesson 13 弾丸と敵機の衝突判定を行う**

　弾丸と敵機の衝突判定を行うためには、EnemyクラスでBulletのインスタンスにアクセスする必要があります。今回はtkEngineのFindGO関数を使用してBulletのインスタンスにアクセスする方法を見ていきましょう。FindGOでアクセスするために、まずBulletのインスタンスに名前を付ける必要があります。Bulletのインスタンスを生成する箇所で名前を指定するようにしましょう。

Player.cpp

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

続いてEnemyクラスで弾丸とプレイヤーとの衝突判定を行いましょう。今回はラムダ式という関数型プログラミングを行うコードを書いています。

Enemy.cpp

|  |
| --- |
| //指定した名前のゲームオブジェクトを検索して、見つかったらラムダ式の処理を実行する。  FindGO<Bullet>(  "Bullet",  //見つかったときの処理。  [&](Bullet\* bullet) {  CVector3 diff = m\_position - bullet->m\_position;  if (　　　①　　　< 20.0f) {  //死亡。  DeleteGO(this);  　　　}  　}  ); |

実習

下線部の１に当てはまるコードを記述しなさい。

**Lesson 14 敵も弾丸を撃てるようにする**　Lesson12で敵を10体表示できるようになりました。Lesson14では敵も弾丸を撃てるようにしてみましょう。

**Lesson 14\_1 Bulletクラスの再利用**

　敵が発射する弾丸はLesson11で作成したBulletクラスを使って実装してみようと思います。EnemyのUpdate関数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

|  |
| --- |
| void Enemy::Update()  {  //モデルのワールド行列を更新。  m\_skinModel.Update(m\_position, m\_rotation, CVector3::One);    **//弾丸を生成。**  **Bullet\* bullet = NewGO<Bullet>(0, "EnemyBullet");**  **bullet->m\_moveSpeed.x = 0.0f;**  **bullet->m\_moveSpeed.y = 0.0f;**  **bullet->m\_moveSpeed.z = -5.0f;**  **bullet->m\_position = m\_position;**  //指定した名前のゲームオブジェクトを検索して、見つかったら匿名メソッドの処理を実行する。  FindGO<Bullet>(  "Bullet",  //見つかったときの処理。  [&](Bullet\* bullet) {  CVector3 diff = m\_position - bullet->m\_position;  if ( diff.Length() < 20.0f) {  //死亡。  CSoundSource\* s = NewGO<CSoundSource>(0);  s->Init("Assets/sound/explosion.wav");  s->Play(false);  DeleteGO(this);  }  }  );  } |

次の動画のようになったら成功です。

[動画\Lesson14\_1.mp4](動画/Lesson14_1.mp4)

**実習課題**

　Lesson 14\_1の実装では、Bulletのインスタンスが毎フレーム作成されているため、弾丸が連続して射出されてしまっています。Lesson 11\_2の実習課題を参考にして敵機の弾丸の射出に50フレームのインターバルを設定しなさい。実装できると下記の動画のようになります。

[動画\Lesson14\_1\_実習.mp4](動画/Lesson14_1_実習.mp4)

**Lesson 14\_2 弾丸に当たったら自機が消えるようにする**　では続いて敵機が射出した弾丸に自機が衝突したら、自機が消えるようにしてみましょう。Lesson 13で行った内容とほとんど同じになります。下記の網掛けになっているコードをPlayerのUpdate関数に追加してください。

|  |
| --- |
| void Player::Update()  {  //移動処理を呼び出す。  Move();  //モデルのワールド行列を更新。  m\_skinModel.Update(m\_position, m\_rotation, CVector3::One);  //弾丸の発射処理を記述。  if (m\_interval == 0) {  //発射間隔が経過した。  if (Pad(0).IsPress(enButtonA)) {  Bullet\* bullet = NewGO<Bullet>(0, "Bullet");  bullet->m\_position = m\_position;  bullet->m\_moveSpeed.x = 0.0f;  bullet->m\_moveSpeed.y = 0.0f;  bullet->m\_moveSpeed.z = 10.0f;  m\_interval = 5; //5フレーム間隔をあける。  }  }  else {  m\_interval--;  }  **// 指定した名前のゲームオブジェクトを検索して、見つかったらラムダ式の処理を実行する。**  **FindGO<Bullet>(**  **"EnemyBullet",**  **//見つかったときの処理。**  **[&](Bullet\* bullet) {**  **CVector3 diff = m\_position - bullet->m\_position;**  **if (diff.Length() < 20.0f) {**  **//死亡。**  **CSoundSource\* s = NewGO<CSoundSource>(0);**  **s->Init("Assets/sound/explosion.wav");**  **s->Play(false);**  **DeleteGO(this);**  **}**  **}**  **);**  } |

次の動画のようになったら成功です。

[動画\Lesson14\_2 .mp4](動画/Lesson14_2%20.mp4)

**Lesson 15 サウンドを再生しよう**

　最後にサウンドの再生について見ていきましょう。サウンドはCSoundSource(音源という意味です)というクラスのインスタンスを生成することで再生することができます。

**Lesson 15\_1 BGMを再生しよう**　では、まずBGMを再生するプログラムを書いてみましょう。main.cppのwWinMain関数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(wWinMain関数)

|  |
| --- |
| //ゲームカメラを追加。  GameCamera\* cam = NewGO<GameCamera>(0, "GameCamera");  ShadowMap().SetCamera(cam->m\_camera);  **//音源クラスを作成。**  **CSoundSource\* soundSource = NewGO<CSoundSource>(0);**  **//BGMをロード。**  **soundSource->Init("Assets/sound/bgm.wav");**  **//ループフラグを立てて再生。**  **soundSource->Play(true);**  Engine().RunGameLoop(); //ゲームループを実行。  return 0; |

これだけだとコンパイルエラーが発生していると思います。main.cppにCSoundSourceというクラスが、どういったクラスであるかを教えてやる必要があります。main.cppの先頭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

|  |
| --- |
| #include "stdafx.h"  #include "GameCamera.h"  #include "Player.h"  #include "Ground.h"  #include "Sky.h"  #include "Enemy.h"  **#include "tkEngine/sound/tkSoundSource.h"** |

これでBGMが再生できるようになりました。

**実習課題**

* 1. プレイヤーが敵機の弾丸と衝突して墜落する時に爆発のSEを再生するようにしなさい。爆発のSEはexplosion.wavというファイルです。
  2. 敵機がプレイヤーの弾丸と衝突して墜落する時に爆発のSEを再生するようにしなさい。爆発のSEはexplosion.wavというファイルです。