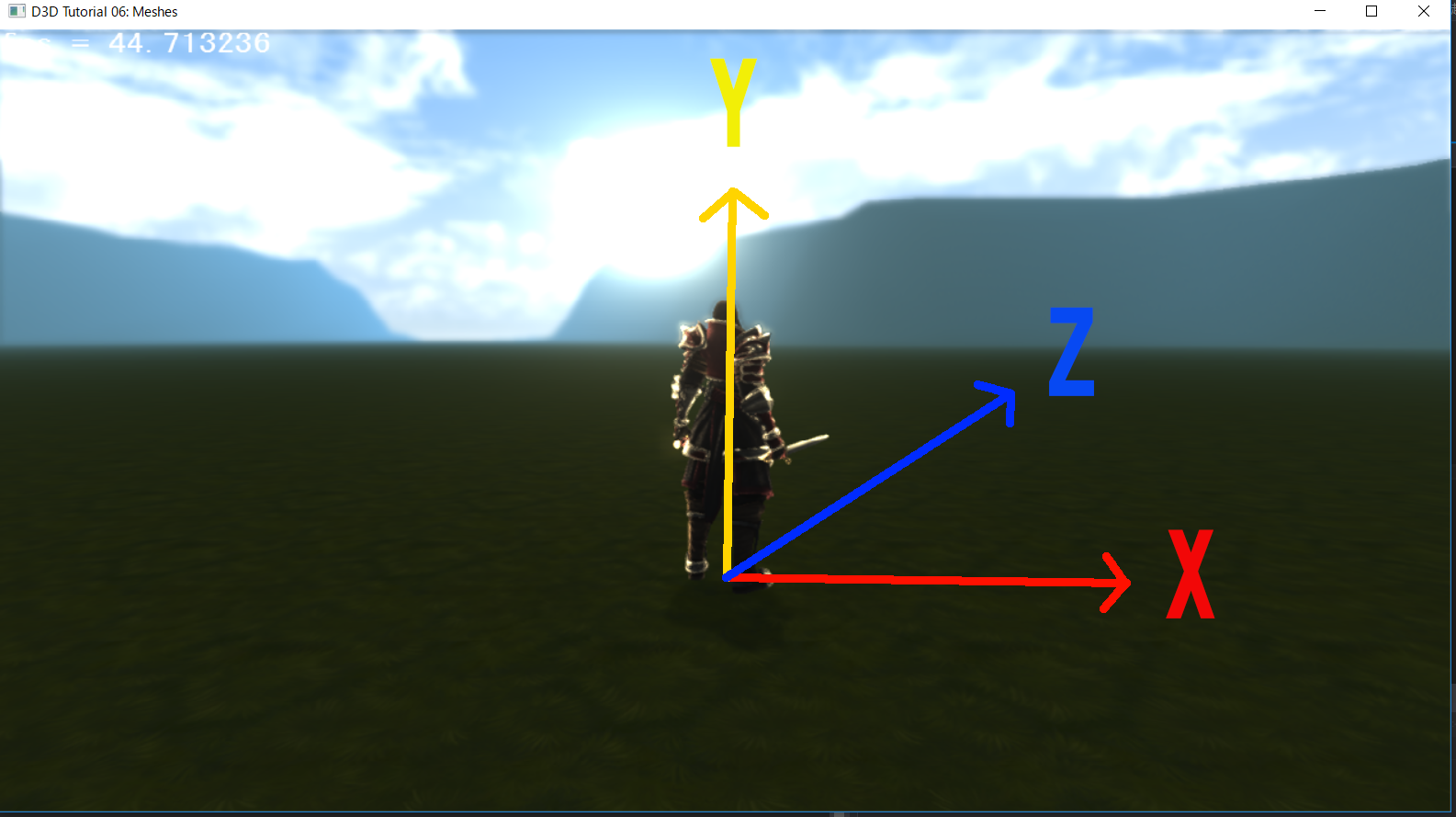
**Lesson\_01\_1 キャラクターを動かしてみよう。**

　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::Move()  {  //Question 1 キャラを左右に動かしてみよう。  moveSpeed.x **= 0.2f;**  //Question 2 キャラを上下に動かしてみよう。  //Question 3 キャラをジャンプさせてみよう。  //Question 4 重力を加えてみよう。  //移動。  characterController.Execute(moveSpeed, GameTime().GetFrameDeltaTime());  position = characterController.GetPosition();  } |

moveSpeedはキャラクターの移動速度を表す変数です。X軸方向に速度を設定すると、キャラクターはX軸上を移動します。

実習課題(１０分)

「Question 2 キャラを上下に動かしてみよう。」と記述されている次の行にキャラクターをZ軸上で動かすプログラムを改造しなさい。

**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。  moveSpeed.x = 1.0f; //移動速度を１にする。  }else{ //それ以外。  moveSpeed.x = 0.0f; //何も入力されていないので移動速度を０にする。  } |

これでパッドの右ボタンが入力されるとキャラクターが右に移動するようになりました。では、続いて左ボタンが入力されたらキャラクターが左に移動するようにしてみましょう。

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  moveSpeed.x = 1.0f;　　　　　//移動速度を1にする。  }  else if (Pad(0).IsPress(enButtonLeft)) { //右のボタンが押されていなくて、左のボタンが押されているなら。  moveSpeed.x = -1.0f; //移動速度を-1にする  }  else {  moveSpeed.x = 0.0f; //何も入力されていないので移動速度を０にする。  } |

実習課題(１５分)

Question 2 キャラを上下に動かしてみよう。」と記述されている次の行からプログラムを追加して、パッドの上下のボタンを入力することによってキャラクターがZ軸上を動くようにしなさい。

**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。つまり、moveSpeed.yに速度を設定してやればいいことになります。次のコードをサンプルプログラムに記述してみてください。

Lesson01/Question/ Player.cpp

|  |
| --- |
| if (Pad(0).IsTrigger(enButtonA)) { //もしもAボタンが押されたら  moveSpeed.y = 2.0; //上方向に力を加える。  characterController.Jump();//おまじない。気にしないでOK。  } |

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大気圏を突破したのではないかと思います。

**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引力によって引っ張られているためです。この地球が引っ張る力のことを重力といいます。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 4 重力を加えてみよう。」と記述されている箇所に、次のプログラムを記述してみてください。

Lesson01/Question/ Player.cpp

|  |
| --- |
| //Question 4 重力を加えてみよう。  moveSpeed.y -= 0.1f; |

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\_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。

Lesson\_02/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。

Lesson\_02/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？

実習課題(２０分)

パッドの上下左右のボタンを押すことによって、キャラクターが移動方向を向くようにプログラムを改造しなさい。

**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。「Question 1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。

Lesson\_03/Player.cpp

|  |
| --- |
| void Player::AnimationControl()  {  //Question 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。  PlayAnimation(AnimationRun); //走るアニメーションを再生する。  }else {  //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。  } |

実習課題 1(１０分)

　ゲームパッドの下、左、右が入力された時も走るアニメーションを再生することができるように、if～else文を活用してプログラムを改造しなさい。

実習課題 2(１０分)

　実習課題 1をif～else文を使用せずに、||演算子を使用することで同様の結果を得られるようにプログラムを改造しなさい。

**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。

Lesson\_03/Player.cpp

|  |
| --- |
| //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);  } |

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。

|  |
| --- |
| void Player::AnimationControl()  {  //Question 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。  PlayAnimation(AnimationRun); //走るアニメーションを再生する。  }  **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！！！**  else {  //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)) {  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。  PlayAnimation(AnimationJump);  }  } |

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PlayAnimation(AnimationStand)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 (!characterController.IsJump()) { //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);  }  } |

characterController.IsJump()はジャンプ中かどうかを調べるためのものです。否定演算子！を使用していることに注意してください。否定演算子は意味を反転させるため、!characterController.IsJump()はジャンプ中じゃなければという意味になります。

**Lesson\_04\_01 キャラクターに攻撃させてみよう**

　Lesson\_04はLesson\_03の復習になります。コントローラーのボタンの入力で攻撃アニメーションを再生させてみましょう。

「Question 1 Xボタンが押されたら小攻撃アニメーションを再生する。」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。

|  |
| --- |
| //Question 1 Xボタンが押されたら小攻撃アニメーションを再生する。  if (Pad(0).IsPress(enButtonX)) {  //攻撃中じゃないときにXボタンが押されたら小攻撃・  //攻撃アニメーションを再生。  PlayAnimation(AnimationAttack\_00);  //攻撃中のフラグを立てる。  attackFlag = 1;  } |

全てここまでの授業の内容の復習となります。attackFlagという変数に1を代入して攻撃中のフラグを立てているのに注意してください。この処理がないと、キャラクターは正しく攻撃アニメーションを再生することはできません。

実習課題 1(１０分)

　ゲームパッドのYボタンが押されたら中攻撃(AnimationAttack\_01)、Bボタンが押されたら大攻撃(AnimationAttack\_02)を再生できるようにしなさい。

**Lesson\_04\_02 正しく攻撃を終了させよう**

　Lesson\_04\_01の実装でキャラクターが攻撃を行うことができるようになりました。しかし、攻撃アニメーションを最後まで再生するとキャラクターが固まってしまって動作しなくなってしまいました。これは攻撃中のフラグが立っているときは、キャラクターに移動やジャンプなどが行えないように制限をかけているせいです。そのため、攻撃アニメーションの再生が終了すると、攻撃中のフラグを下す必要があります。

「Question 4 攻撃アニメーションの再生が終わったら攻撃フラグを下す。」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。

|  |
| --- |
| if (!animation.IsPlay()) {  //アニメーションの再生が終わったので攻撃フラグを下す。  attackFlag = 0;  } |

これで、キャラクターは攻撃が終わると待機アニメーションに遷移して、移動やジャンプが行えるようになったと思います。

**Lesson\_05 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\_05\_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。Lesson\_05/Question/Game/GameCamera.cppを開いて「Question 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。

|  |
| --- |
| //Question 1 注視点を動かす。  CVector3 cameraTarget;  //注視点を計算する。  cameraTarget.x = playerPos.x;  cameraTarget.y = playerPos.y + 1.0f; //プレイヤーのちょっと上にする。  cameraTarget.z = playerPos.z;  //注視点をカメラに伝える。  camera.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。

実習

　注視点のY座標の補正の値を変更するとどのような変化が起きるか確認しなさい。

**Lesson\_05\_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「Question 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。

|  |
| --- |
| //Question 2 視点を動かす。  CVector3 cameraPos;  //視点を計算する。  cameraPos.x = playerPos.x;  cameraPos.y = playerPos.y + 1.5f; //プレイヤーのちょっと上にする。  cameraPos.z = playerPos.z - 4.0f; //プレイヤーの4.0分手前にする。  //視点をカメラに伝える。  camera.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\_05\_03　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\_06\_01 モンスターに攻撃をさせてみよう。**

　Lesson\_06ではモンスターの処理を実装していきます。Lesson06/Question/Game/Game.slnを起動して、Enemy.cppを開いてください。

今回はプレイヤーとモンスターの距離が2m以下になったら攻撃を行うという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モンスターの距離を計算する必要があります。プレイヤーやモンスターの座標は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、食べ物の座標をE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。

E

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

では、これを参考にしてenemy.cppの「Question 1 プレイヤーとの距離が2m以下になったら攻撃アニメーションを再生する。」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。

|  |
| --- |
| CVector3 playerPos = player->GetPosition();  //プレイヤーとの距離が一定値以下になったら攻撃する。  CVector3 diff = player->GetPosition();  diff = playerPos - position;  if (diff.Length() < 2.0f) {  //2m以下になったら  animation.PlayAnimation(AnimationAttack);  isAttackFlag = true; //攻撃フラグを立てる。  } |

実習

Lesson\_06\_01の実装をしたら、モンスターは一度だけ攻撃を行うようになったと思います。しかし、一度だけ攻撃を行ったら固まってしまい、二度と攻撃をしなくなってしまいました。これは攻撃フラグが立ちっぱなしになっていることが原因です。「Question 2 攻撃アニメーションの再生が終わったら待機アニメーションを再生する。」と記述されている箇所にこれを解決するためのプログラムを記述しなさい。参考になるプログラムはLesson\_04\_02の内容です。

**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**

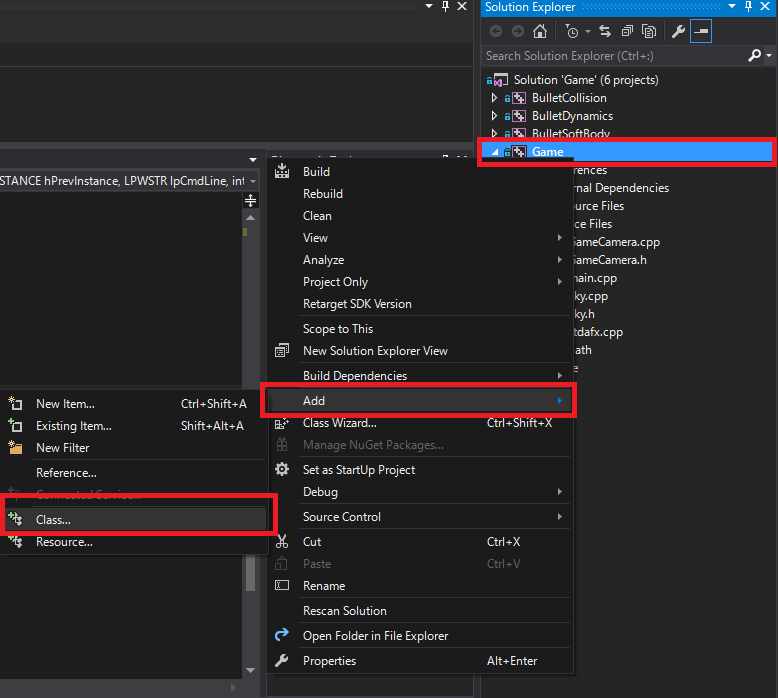
　main.cppの70行目のwWinMain関数を見てみてください。実はこの関数はWindowsプログラムのメイン関数と呼ばれるもので，C言語のmain関数と同じでようなものになります。プログラムが起動するとこの関数が実行され、Engine().RunGameLoop();というプログラムが実行されると、ゲームループと呼ばれる一定間隔でループする処理が動き出します。

|  |
| --- |
| 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;  } |
|  |

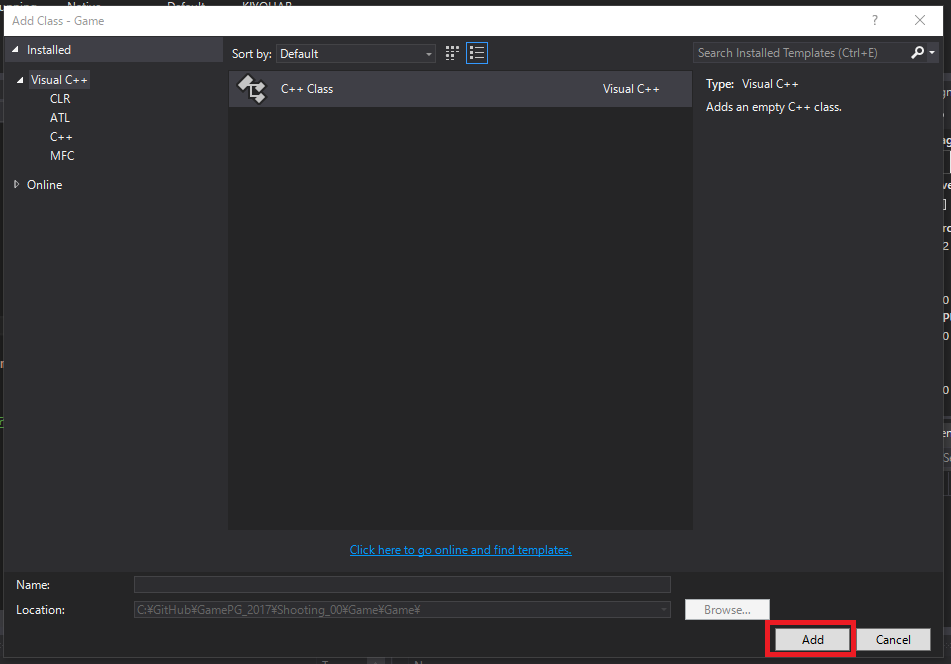
ゲームループが起動する前にNewGO<::Sky>(0, "Sky");というコードがあります。これが空を作成しているコードです。試しにこの行をコメントアウトしてみてください。空が表示されなくなります。コメントアウトすると空が消えることが確認できた人は、コメントアウトを元に戻して、空を復活させておいて下さい。

**Lesson 9\_02 プレイヤーのクラスの追加**

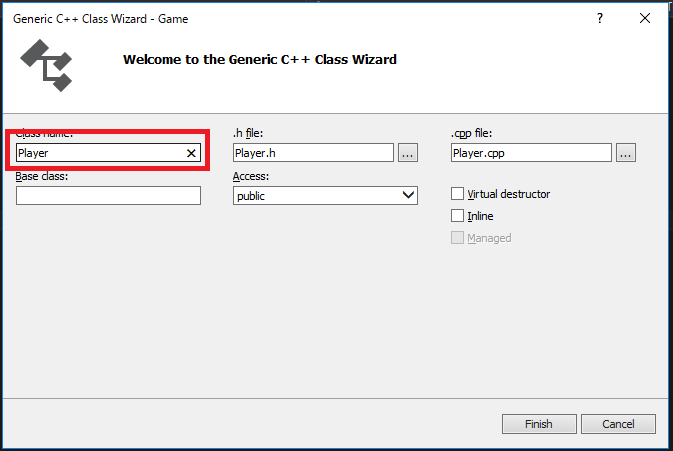
では、自機(プレイヤー)のプログラムを記述するためにプレイヤーのクラスを作りましょう。ソリューションエクスプローラーのGameプロジェクトを選択して右クリック/追加/クラスを選択してください。

****

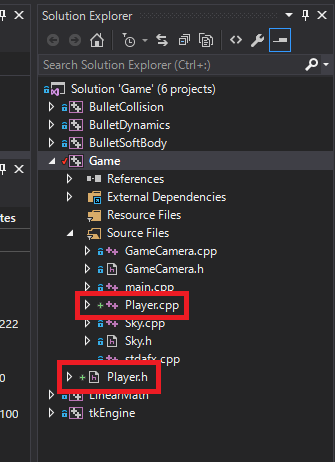
選択すると次のウィンドウが開くので追加を選択してください。



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