ゲームプログラミングⅡ

# はじめに

　この授業はある程度体系だったものとなりますが、tips集のような側面があります。Chpater1～Chapter5まではゲームを作るうえで、おそらく多くのゲームにとって、共通認識となるであろう内容となっています。

　Chapter6～Chapter9の内容は、いくつかのゲームの要素の実装事例を挙げています。これらは、どのゲームでも使えるというものではありません。これらのChapterで狙っているものは、みなさんに考え方のヒントを与えるためのものです。

　ゲームは０から１を作るものです。そのため、誰かに教えてもらったものだけを作って入ればいいわけではありません。しかし、先人が考えたことを学ぶのは重要です。まずは、先人の模倣を行って、新しいものを作成する力を身につけていきましょう。

# **Chapter 1 レベルデザイン**

## **1.1 レベルデザインとは**

レベルとはステージのことを指します。なので、レベルデザインとはステージを作成することを指しています。よく、難易度の調整などと勘違いされますが、これは間違いで、ステージのデザインがレベルデザインです。

　素晴らしいレベルデザインとは、ユーザーを道に迷わせることなく、適度な緊張と、達成感を与えることができます。

[【GDC 2013】発表！　「良いレベルデザインの10の原則」](https://game.watch.impress.co.jp/docs/news/593961.html)

## **1.2 レベルエディタ**

　ほとんどすべてのゲームの開発では、何かしらのレベルエディタが用意されています。Excelを使用しているところもあれば、3dsMaxやMayaといったDCCツールを利用しているところもあります。独自ツールを作っているところもあるでしょう。UnityやUnrealEngine4などのゲームエンジンは統合開発環境の中に、レベルエディタが組み込まれています。

## **1.3 レベルエディタに必要な機能**

　レベルエディタに最低限求められる機能は、ステージに配置されている、建物、ギミック、敵キャラクターなどの配置情報（位置情報、回転情報、拡大率）を出力できることです。

## **1.4 tkEngine2のレベルエディタ**

学内のゲームエンジンでは、maxScriptを使って3dsMaxの機能を拡張してレベルエディタとして活用しています。

### **1.4.1 【実習】3dsMaxを利用したレベルデザイン(時間 20分)**

　では、下記の動画を参考にして、*レベル*を作成してみてください。

<https://www.youtube.com/watch?v=bK8oyehhfYA&feature=youtu.be>

## **1.5 【実習】レベルデータ(tkl)を確認してみる。**

　1.4.1の実習でレベルデータを作成することができました。では、作成したレベルデータの中身を確認してみましょう。下記動画を参考にしてレベルデータの中身を確認してみてください。

<https://www.youtube.com/watch?v=KPVd13ulMQ8&feature=youtu.be>

## **1.6 レベルデータをゲームで使ってみる**

　tkEngineにはレベルをロードするために、下記の二つのクラスが用意されています。

①　CLevelクラス

　　 tklファイルをロードする機能を持つクラス。

②　CMapChipクラス

モデル表示、あたり判定登録の機能を持つクラス。

CMapChipクラスのインスタンスはCLevelクラスが保持しています。クライアント側ではCLevelクラスしか利用しません。

### **1.6.1 CLevelを利用したサンプルコード**

下記は、レベルをロードするサンプルコードです。

|  |
| --- |
| bool Game::Start()  {  　　　　　　…  　　　　　 省略  　　　　　　…  m\_level.Init(L"test.tkl", nullptr);  return true;  } |

### **1.6.2 【実習】Chapter\_01を改造して、レベルをロードしてみる。**

　では、Chapter\_01を改造して、[1.4.1 【実習】3dsMaxを利用したレベルデザイン(時間 20分)](#_1.4.1_【実習】3dsMaxを利用したレベルデザイン(時間_20分)で作成してレベルをロードしてみましょう。

まず、GameクラスにCLevelクラスのインスタンスを保持させます。

#### 【サンプルコード】1.6.2.1 CLevelクラスのインスタンスを保持させる。

|  |
| --- |
| class Game : public IGameObject  {  public:  Game();  ~Game();  void Update();  CLevel m\_level; //レベル  }; |

続いて、Gameクラスのコンストラクタでレベルを初期化します。

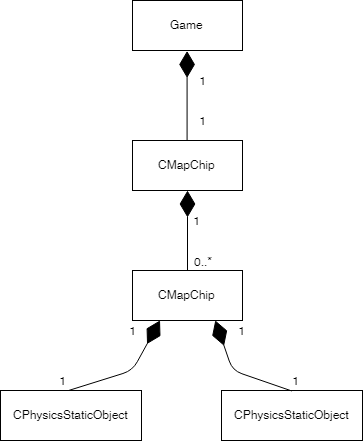
#### 【サンプルコード】1.6.2.2 レベルの初期化。Game.cpp（5行目）

|  |
| --- |
| Game::Game()  {  //カメラを設定。  …  　　　　省略  　　　　 …  m\_level.Init(L"test.tkl", nullptr);  } |

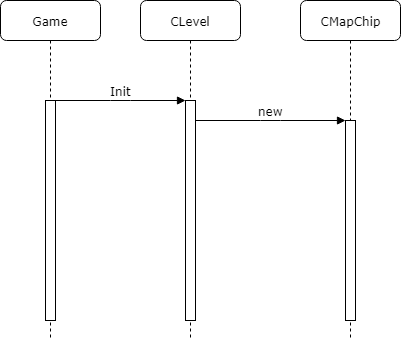
実装できた人は、実行してみてください。3dsMaxで配置したオブジェクトが表示されていると思います。

|  |
| --- |
| Tips  レベルに配置したオブジェクトをゲーム中に表示するためには、fbxファイルをVisualStudioに追加する必要があります。これを忘れないように注意してください。 |

### **1.6.3 【付録】CLevelを利用する場合のクラス図**



### **1.6.4 【付録】CLevelを利用する場合のシーケンス図**



### **1.6.5 オブジェクトのロードをフック(改造)する**

CLevelクラスを使って、レベルを初期化すれば、背景などの動かないオブジェクトは簡単に表示できます。では、ジャンプ台や敵キャラクターなどのオブジェクトを配置したい場合はどのようにすればいいのでしょうか？CLevelクラスを利用している場合は、オブジェクトのロード処理をフック(改造)することで実現できます。

　CLevel::Init関数の第二引数には**関数オブジェクト**と呼ばれるものを渡すことができます。この関数オブジェクトを使ってフックを行います。では、サンプルコードを見てみましょう。

#### 1.6.5.Sample1

【注意！】フックしたらtrueを返すのを忘れないように！

第一引数は読み込むレベルデータのファイルパス

|  |
| --- |
| m\_level.Init(  L"test.tkl",  [&](LevelObjectData& objData) {  if (objData.EqualObjectName(L"unityChan")) {  //配置しようとしているオブジェクトはユニティちゃん。  //フックして、プレイヤークラスを作る。  Player\* pl = NewGO<Player>(0);  //配置情報から座標と回転をプレイヤーに渡す。  pl->m\_position = objData.position;  pl->m\_rotation = objData.rotation;  return true;  }  return false;  　　}  ); |

第二引数は関数オブジェクト。この関数はオブジェクトをロードする直前で実行される。

### **1.6.6【実習】プレイヤーをフックする**

　では、Sample\_00を改造して、プレイヤーをフックしてみましょう。まず、ゲームコントローラ―の入力で動かすことができるプレイヤークラスを作成しましょう。

Playerクラスを追加して、下記のコードをPlayer.hとPlayer.cppを入力してください。

Player.h

|  |
| --- |
| #pragma once  class Player : public IGameObject  {  public:  Player();  ~Player();  bool Start();  void Update();  ////////////////////////////////  //ここからメンバ変数。  ////////////////////////////////  CVector3 m\_position; //キャラの座標。  CQuaternion m\_rotation; //キャラの回転。  CVector3 m\_moveSpeed; //キャラの移動速度。  CCharacterController m\_charaCon; //キャラクターコントローラ。  prefab::CSkinModelRender\* m\_modelRender; //スキンモデルレンダラー。  }; |

Player.cpp

|  |
| --- |
| #include "stdafx.h"  #include "Player.h"  Player::Player()  {  }  Player::~Player()  {  }  bool Player::Start()  {  m\_modelRender = NewGO<prefab::CSkinModelRender>(0);  m\_modelRender->Init(L"modelData/unityChan.cmo");  //キャラクターコントローラーの初期化。  m\_charaCon.Init(  20.0f, //カプセルコライダーの半径。  40.0f, //カプセルコライダーの高さ。  m\_position //キャラクターコントローラーの座標。  );  return true;  }  void Player::Update()  {  //ゲームパッドの入力から、移動させる量を決める。  m\_moveSpeed.x = Pad(0).GetLStickXF() \* -300.0f;  m\_moveSpeed.z = Pad(0).GetLStickYF() \* -300.0f;    if (Pad(0).IsTrigger(enButtonA)) {  //Aボタンが押されたらジャンプさせる。  //上方向に速度を設定する。  m\_moveSpeed.y = 600;  }  //重力により、下方向に力を加える。  m\_moveSpeed.y -= 20.0f;  //キャラコンを移動速度を渡して、キャラコンを動かす。  m\_position = m\_charaCon.Execute(m\_moveSpeed);  m\_modelRender->SetPosition(m\_position);  m\_modelRender->SetRotation(m\_rotation);  } |

これで、プレイヤークラスが完成しました。続いて、レベルにunityChanというオブジェクトが配置されていたら、フックしてPlayerクラスのインスタンスを作成するようにしましょう。レベルをロードしている箇所を下記のように変更してください。

Game.cpp

|  |
| --- |
| Game::Game()  {  //カメラを設定。  MainCamera().SetTarget({ 0.0f, 70.0f, 0.0f });  MainCamera().SetNear(10.0f);  MainCamera().SetFar(10000.0f);  MainCamera().SetPosition({ 0.0f, 500.0f, 1000.0f });  MainCamera().Update();  m\_level.Init(L"test.tkl", [&](LevelObjectData& objData) {  if (objData.EqualObjectName(L"unityChan")) {  //ロードしようとしているオブジェクトはユニティちゃん。  //フックして、プレイヤークラスを作る。  Player\* pl = NewGO<Player>(0);  //配置情報から座標と回転をプレイヤーに渡す。  pl->m\_position = objData.position;  pl->m\_rotation = objData.rotation;  return true;  }  return false;  });  } |

入力出来たら実行して動作を確認してみてください。ゲームパッドでユニティちゃんを動かせるようになっているはずです。

### **1.6.7【実習課題】レベルに配置されている星(オブジェクト名star)をフックして、星を回転させる。**

レベルに配置されている星(オブジェクト名star)をフックして、下記の動画のように、星を回転させてください。

<https://www.youtube.com/watch?v=GI79NVCY46Y&feature=youtu.be>

# **Chapter 2 ステートフルクラス**

**ステートフルクラス**とは、状態を保持しているクラスの事です。これまで皆さんが作成してきたクラスは、ほとんどが**ステートレスクラス**、状態を保持していないクラスの事です。

　では、状態を保持しているとはどういうことでしょうか？下記の動画の星の処理を見てみてください。

ステートレス(星は回転し続けるだけ)

<https://youtu.be/91VYqRZn1Kg>

ステートフル(星は上に移動する状態と、下に移動する状態がある)

<https://youtu.be/wg9lHNS0oxg>

ステートレスの動画の星はずっと同じ方向に回転し続けるだけです。なので、この星のプログラムはステートレス(状態なし)になります。

一方ステートフルの動画の星は上に移動したり、下に移動したりします。つまり、この星は、**上に移動する状態**と**下に移動する状態**の二つの状態を切り替えて動作していることになります。なので、この星のプログラムはステートフル(状態保持)です。

　ステートフルクラスを作成するのは、とても簡単です。クラスに状態を表すメンバ変数を追加して、そのメンバ変数の値で処理を分岐してやればいいだけです。

　では、Chapter\_02のプログラムを見て、具体的にどのようにプログラミングされているのか見ていきましょう。

Star\_01.h

|  |
| --- |
| class Star\_01 : public IGameObject  {  public:  Star\_01();  ~Star\_01();  bool Start();  void Update();    //状態を表す列挙。  enum EnState {  enState\_Down, //下降中  enState\_Up, //上昇中  };  **EnState m\_state; //状態。**  int m\_timer; //タイマー。  CVector3 m\_position; //座標。  CVector3 m\_scale; //拡大率。  prefab::CSkinModelRender\* m\_modelRender; //スキンモデルレンダー。  }; |

Star\_01クラスにはm\_stateというメンバ変数があります。これが状態を表しているデータです。

　では、続いてm\_stateの値によって、処理を分岐させている箇所を見てみましょう。

Star\_01.cpp

|  |
| --- |
| void Star\_01::Update()  {  //タイマーをインクリメント。  m\_timer++;  if (m\_state == enState\_Down) {  //下降状態の時の処理。  m\_position.y -= 1.0f;  if (m\_timer == 120) {  //120フレーム経過したので、状態を切り替える。  m\_state = enState\_Up;  //タイマーをリセットする。  m\_timer = 0;  }  }  else if (m\_state == enState\_Up) {  //上昇状態の時の処理。  m\_position.y += 1.0f;  if (m\_timer == 120) {  //120フレーム経過したので、状態を切り替える。  m\_state = enState\_Down;  //タイマーをリセットする。  m\_timer = 0;  }  }  //座標を設定する。  m\_modelRender->SetPosition(m\_position);  } |

m\_stateの値によって、処理が分岐しているだけで、特に難しいところはないと思います。

## **2.1 実習課題**

・左右に動く星を作りなさい。

・Y軸周りに45°回転とY軸周りに－45°回転を交互に繰り返す星を作成しなさい。

# Chapter 3 FSM(有限状態機械)

　初めにFSM(有限状態機械)はChapter2のステートフルクラスと全く同じものと考えてＯＫです。それを踏まえて、FSMを見ていきましょう。

## **3.1 FSMとは**

　FSMとは有限の状態があり、一定のルールで状態を遷移していくものです。言葉が難しいので、ゲームで例えてみましょう。例えば次のような、ゲームの雑魚敵のよくある仕様を考えてみましょう。

「**普段はダンジョンをウロウロしているが、プレイヤーを見つけると追いかけてきて、追いつくと攻撃してくる。そして、プレイヤーとの距離が離れたら、追いかけるのをあきらめて、またダンジョンをウロウロする。**」

　この雑魚敵には3つの状態があることが考えられます。

　①　徘徊状態(ダンジョンをウロウロ)

②　追跡状態

　③　攻撃状態

プレイヤーを見つけたら

追跡

プレイヤーが離れたら

攻撃

徘徊

プレイヤーに追いついたら

プレイヤーとの距離が開いたら

## **3.2　FSMを実装してみる**

### **3.2.1 追跡状態を作ってみる(HandsOn)**

　では、サンプルプログラムのChapter\_03を使って、敵のFSMを実装してみましょう。まずはプレイヤーとの距離が一定以内に入ったら、追跡状態に遷移するようにしてみましょう。まず追跡状態を表すenumを追加しましょう。Enemy.hを開いて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.h(52行目)

|  |
| --- |
| //状態。  enum EnState {  enState\_Idle, //待機状態  **enState\_Tracking, //追跡状態。**  }; |

続いて、プレイヤーとの距離が400以下になったら追跡状態に遷移するコードを追加しましょう。Enemy.cppを開いて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(51行目)

|  |
| --- |
| if (m\_state == enState\_Idle) {  **//待機状態の時の処理。**  **//まずプレイヤーのインスタンスを検索する。**  **CPlayer\* player = FindGO<CPlayer>("Player");**  **//敵からプレイヤーに向かって伸びるベクトルdiffを計算する。**  **CVector3 diff = player->GetPosition() - m\_position;**  **//diffの長さを求めてdistanceに代入する。**  **float distance = diff.Length();**  **//プレイヤーとの距離が400以下になったら追跡状態に遷移する。**  **if (distance < 400) {**  **//歩くアニメーションを再生する。**  **m\_skinModelRender->PlayAnimation(enAnimation\_Walk);**  **//状態を追跡にする。**  **m\_state = enState\_Tracking;**  **}**  } |

最後に追跡状態の時にプレイヤーを追いかけるコードを追加しましょう。Enemy.cppを開いて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(51行目)

|  |
| --- |
| if (m\_state == enState\_Idle) {  　　　　…  　　　　省略  　　　　…  }  **else if (m\_state == enState\_Tracking) {**  **//追跡状態の時の処理。**  **//まずプレイヤーのインスタンスを検索する。**  **CPlayer\* player = FindGO<CPlayer>("Player");**  **//敵からプレイヤーに向かって伸びるベクトルvを計算する。**  **CVector3 v = player->GetPosition() - m\_position;**  **//vを正規化して大きさ１にする(方向のみのベクトルにする)**  **v.Normalize();**  **//vの大きさを３倍にして、敵の座標に足し算する。**  **m\_position += v \* 3.0f;**  **}** |

### **3.2.2 攻撃状態を作ってみる(実習課題)**

では、Enemyのプログラムを改造して、**追跡中にプレイヤーとの距離が50以下になったら、攻撃する**という仕様を実装してください。

### **3.2.3 攻撃状態中に攻撃アニメーションが終了したら、待機状態に遷移できるようにする(実習課題)**

最後に攻撃状態から待機状態に遷移できるようにしてみましょう。

またアニメーションの再生の終了判定は下記のようなコードで行えます。

|  |
| --- |
| if (m\_skinModelRender->IsPlayingAnimation() == false) {  //アニメーションの再生が終了した  } |

# Chapter 4 あたり判定

# Chapter 5 アニメーションイベント

# Chapter 6 ギミック　動く床

# Chapter 7 ギミック　ジャンプ台

# Chapter 8 経路を巡回するNPC

# Chapter 9 視野角があるNPC