**Chapter 1**

**キーボードのＡが押されたらプレイヤーの移動速度を倍にしてみよう。**

まず、キーボードのAが押されたということを記録できるようにする必要があります。Packmanのプロジェクトでキーボードの入力を記録しているプログラムは下記の二

のファイルです。

ソースファイル

.\Packman\tkEngine\Input\tkInput\tkKeyInput.cpp

ヘッダーファイル

.\Packman\tkEngine\Input\tkInput\tkKeyInput.h

ではヘッダーファイルを下記のように編集してください。太字になっている部分が変更点です。

|  |
| --- |
| /\*!  \* @brief キー入力。  \*/  #ifndef \_TKKEYINPUT\_H\_  #define \_TKKEYINPUT\_H\_  namespace tkEngine{  class CKeyInput{  enum EnKey{  enKeyUp,  enKeyDown,  enKeyRight,  enKeyLeft,  **enKeyA**,  enKeyNum,  };  public:  /\*!  \* @brief コンストラクタ。  \*/  CKeyInput();  /\*!  \* @brief デストラクタ。  \*/  ~CKeyInput();  /\*!  \* @brief キー情報の更新。  \*/  void Update();  /\*!  \* @brief 上キーが押されている。  \*/  bool IsUpPress() const  {  return m\_keyPressFlag[enKeyUp];  }  /\*!  \* @brief 右キーが押されている。  \*/  bool IsRightPress() const  {  return m\_keyPressFlag[enKeyRight];  }  /\*!  \* @brief 左キーが押されている。  \*/  bool IsLeftPress() const  {  return m\_keyPressFlag[enKeyLeft];  }  /\*!  \* @brief 下キーが押されている。  \*/  bool IsDownPress() const  {  return m\_keyPressFlag[enKeyDown];  }  **/\*!**  **\* @brief キーボードのAが押された。**  **\*/**  **bool IsAPress() const**  **{**  **return m\_keyPressFlag[enKeyA];**  **}**  private:  bool m\_keyPressFlag[enKeyNum];  };  }  #endif //\_TKKEYINPUT\_H\_ |

続いてソースファイルを下記のように変更します。

|  |
| --- |
| /\*!  \* @brief キー入力  \*/    #include "tkEngine/tkEnginePreCompile.h"  #include "tkEngine/Input/tkKeyInput.h"  namespace tkEngine{  /\*!  \* @brief コンストラクタ。  \*/  CKeyInput::CKeyInput()  {  memset(m\_keyPressFlag, 0, sizeof(m\_keyPressFlag));  }  /\*!  \* @brief デストラクタ。  \*/  CKeyInput::~CKeyInput()  {  }  /\*!  \* @brief キー情報の更新。  \*/  void CKeyInput::Update()  {  if (GetAsyncKeyState(VK\_UP) & 0x8000) {  m\_keyPressFlag[enKeyUp] = true;  }  else {  m\_keyPressFlag[enKeyUp] = false;  }  if (GetAsyncKeyState(VK\_DOWN) & 0x8000) {  m\_keyPressFlag[enKeyDown] = true;  }  else {  m\_keyPressFlag[enKeyDown] = false;  }  if (GetAsyncKeyState(VK\_RIGHT) & 0x8000) {  m\_keyPressFlag[enKeyRight] = true;  }  else {  m\_keyPressFlag[enKeyRight] = false;  }  if (GetAsyncKeyState(VK\_LEFT) & 0x8000) {  m\_keyPressFlag[enKeyLeft] = true;  }  else {  m\_keyPressFlag[enKeyLeft] = false;  }  **if ((GetAsyncKeyState('A') & 0x8000)**  **| (GetAsyncKeyState('a') & 0x8000) ) {**  **m\_keyPressFlag[enKeyA] = true;**  **}**  **else {**  **m\_keyPressFlag[enKeyA] = false;**  **}**  }  } |

これでキーボードのAが入力されると、m\_keyPressFlag[enKeyA]にtrueという値が記録されるようになりました。

では本当にキーボードのAが入力されたらtrueが設定されるか確認してみましょう。

先ほどの書き換えた部分にコードを下記の黒字のコードを追加してみて下さい。

|  |
| --- |
| if ((GetAsyncKeyState('A') & 0x8000)  | (GetAsyncKeyState('a') & 0x8000) ) {  m\_keyPressFlag[enKeyA] = true;  **MessageBox(NULL, "Aボタンが押されたよ！", "成功", MB\_OK);**  }else {  m\_keyPressFlag[enKeyA] = false;  } |

Aを押したらダイアログボックスがでましたか？

では、確認ができたらメッセージボックスを表示するコードは削除してください。

では、正しくキーボードから入力を受け取ることができることが確認できたのでプレイヤーの移動速度を倍にしてみましょう。プレイヤーの移動処理は下記のファイルに記述されています。

ソースファイル

　Packman\Packman\game\Player\ CPlayer.cpp

CPlayer.cppを下記のように書き換えてみて下さい。

|  |
| --- |
| /\*!  \* @brief プレイヤー  \*/  #include "stdafx.h"  #include "Packman/game/Player/CPlayer.h"  #include "Packman/game/CGameManager.h"  /\*!  \*@brief Updateが初めて呼ばれる直前に一度だけ呼ばれる処理。  \*/  void CPlayer::Start()  {  }  /\*!  \*@brief Update関数が実行される前に呼ばれる更新関数。  \*/  void CPlayer::PreUpdate()  {  Move();  }  /\*!  \*@brief 更新処理。60fpsなら16ミリ秒に一度。30fpsなら32ミリ秒に一度呼ばる。  \*/  void CPlayer::Update()  {  m\_sphere.SetPosition(m\_position);  m\_sphere.UpdateWorldMatrix();  CGameManager& gm = CGameManager::GetInstance();  CMatrix mMVP = gm.GetGameCamera().GetViewProjectionMatrix();  const CMatrix& mWorld = m\_sphere.GetWorldMatrix();  m\_wvpMatrix.Mul(mWorld, mMVP);  m\_idMapModel.SetWVPMatrix(m\_wvpMatrix);  IDMap().Entry(&m\_idMapModel); m\_shadowModel.SetWorldMatrix(mWorld);  ShadowMap().Entry(&m\_shadowModel);  }  /\*!  \*@brief 移動処理。  \*/  void CPlayer::Move()  {  **float moveSpeed = 0.02f; //移動速度。**  **if (KeyInput().IsAPress()) {**  **//キーボードのA押されていたら速度を倍にする。**  **moveSpeed \*= 2.0f;**  **}**  **if (KeyInput().IsUpPress()) {**  **m\_position.z += moveSpeed;**  **}**  **if (KeyInput().IsDownPress()) {**  **m\_position.z -= moveSpeed;**  **}**  **if (KeyInput().IsRightPress()) {**  **m\_position.x += moveSpeed;**  **}**  **if (KeyInput().IsLeftPress()) {**  **m\_position.x -= moveSpeed;**  **}**  }  /\*!  \*@brief 描画処理。60fpsなら16ミリ秒に一度。30fpsなら32ミリ秒に一度呼ばれる。  \*/  void CPlayer::Render(tkEngine::CRenderContext& renderContext)  {  CGameManager& gm = CGameManager::GetInstance();  m\_sphere.RenderLightWVP(  renderContext,  m\_wvpMatrix,  gm.GetFoodLight(),  false,  true  );  }  /\*!  \*@brief 構築。  \*必ず先にCreateShapeを一度コールしておく必要がある。  \*/  void CPlayer::Build( const CVector3& pos )  {  m\_sphere.Create(0.08f, 10, 0xffff0000, true );  m\_idMapModel.Create(m\_sphere.GetPrimitive());  m\_shadowModel.Create(m\_sphere.GetPrimitive());  m\_position = pos;  } |

**Chapter 2**

**ジャンプできるようにしてみよう。**

Chapter1のプログラムを改造して、キーボードのAが入力されたらプレイヤーがジャンプするようにしてみましょう。今回編集するプログラムは下記のファイルになります。

ソースファイル

　Packman\Packman\game\Player\ CPlayer.cpp

　　ヘッダーファイル

　　　Packman\Packman\game\Player\ CPlayer.h

CPlayer.hを下記のように編集してください。

|  |
| --- |
| /\*!  \* @brief プレイヤー  \*/  #ifndef \_CPLAYER\_H\_  #define \_CPLAYER\_H\_  #include "tkEngine/shape/tkSphereShape.h"  class CPlayer : public tkEngine::IGameObject{  public:  CPlayer() :  m\_position(CVector3::Zero)  　{  　}  ~CPlayer(){}  /\*!  \*@brief Updateが初めて呼ばれる直前に一度だけ呼ばれる処理。  \*/  void Start() override final;  /\*!  \*@brief Update関数が実行される前に呼ばれる更新関数。  \*/  void PreUpdate() override final;  /\*!  \*@brief 更新処理。60fpsなら16ミリ秒に一度。30fpsなら32ミリ秒に一度呼ばれる。  \*/  void Update() override final;  /\*!  \*@brief 描画処理。60fpsなら16ミリ秒に一度。30fpsなら32ミリ秒に一度呼ばれる。  \*/  void Render(tkEngine::CRenderContext& renderContext) override final;  /\*!  \*@brief 構築。  \*必ず先にCreateShapeを一度コールしておく必要がある。  \*/  void Build( const CVector3& pos );  /\*!  \*@brief 移動処理。  \*/  void Move();  /\*!  \*@brief 座標を取得。  \*/  const CVector3& GetPosition() const  {  return m\_position;  　}  private:  tkEngine::CSphereShape m\_sphere;  CMatrix m\_wvpMatrix; //<ワールドビュープロジェクション行列。  tkEngine::CIDMapModel m\_idMapModel;  CVector3 m\_position;  tkEngine::CShadowModel m\_shadowModel; //!<シャドウマップへの書き込み用のモデル。  **CVector3 m\_moveSpeed; //!<移動速度。**  };  #endif |

プレイヤーにm\_moveSpeedという移動速度を覚えるためのメンバ変数が追加されました。

では、続いてtkPlayer.cppのMove関数を下記のように編集してください。

|  |
| --- |
| void CPlayer::Move()  {  　　//XZ平面での移動速度。  m\_moveSpeed.x = 0.02f;  m\_moveSpeed.z = 0.02f;  if (KeyInput().IsAPress()) {  //キーボードのAが押されていたら速度を倍にする。  m\_moveSpeed.x \*= 2.0f;  m\_moveSpeed.z \*= 2.0f;  m\_moveSpeed.y = 0.1f;  　　}  //Y方向への移動速度。  if (KeyInput().IsUpPress()) {  m\_position.z += m\_moveSpeed.z;  　　}  if (KeyInput().IsDownPress()) {  m\_position.z -= m\_moveSpeed.z;  　　}  if (KeyInput().IsRightPress()) {  m\_position.x += m\_moveSpeed.x;  　　}  if (KeyInput().IsLeftPress()) {  m\_position.x -= m\_moveSpeed.x;  　　}  m\_position.y += m\_moveSpeed.y;  //重力とかは考えない。  m\_moveSpeed.y -= 0.01f;  } |

これでプレイヤーはAを押すとジャンプするようになりました。ただし地面と衝突判定などは行っていないため、このプレイヤーはAを押さないと奈落の底に落下していきます。

次のChapterではプレイヤーが落下しないようにプログラムを変更してみます。

|  |
| --- |
| *Tips*  *ゲーム制作において、プレイヤーの挙動はそれっぽく見えればOKなのでまじめに物理計算を行う必要があるわけではありません。ですが先ほどのジャンプ処理を重力を考慮してプログラムを書いてみましたので、せっかくですのでご紹介します。変更点はCPlayerのMove関数のみです。*  void CPlayer::Move()  {  //Moveが呼ばれる感覚は16ミリ秒で固定で考える。  static const float deltaTime = 1.0f / 60.0f;  //速度の単位をm/sに変更する。  m\_moveSpeed.x = 1.f; //XZ平面での移動速度。  m\_moveSpeed.z = 1.f;  if (KeyInput().IsAPress()) {  //ジャンプ。  //初速度を2m/sで与える。  m\_moveSpeed.y = 2.0f;  }  CVector3 add(0.0f, 0.0f, 0.0f);  add = m\_moveSpeed;  add.Scale(deltaTime);  if (KeyInput().IsUpPress()) {  m\_position.z += add.z;  }  if (KeyInput().IsDownPress()) {  m\_position.z -= add.z;  }  if (KeyInput().IsRightPress()) {  m\_position.x += add.x;  }  if (KeyInput().IsLeftPress()) {  m\_position.x -= add.x;  }  m\_position.y += add.y;  //速度に重力加速度の影響を与える。  //重力加速度 9.8m/s^2  static const CVector3 gravity(0.0f, -9.8f, 0.0f);  CVector3 addVelocity = gravity;  addVelocity.Scale(deltaTime);  m\_moveSpeed.y += addVelocity.y;  }  *変更を加えたプログラムを下記のパスに上げました。Debugモードで実行すると処理が遅いのでもっさりした挙動になるので、Releaseモードで確認するといいでしょう。*  *(影も壁に落ちるように改良してます・・・)*  *\\mmnas01\student\GC2016\02\_授業\ゲームPG Ⅰ\Lesson02\Packman\_JumpGravity* |

**Chapter 3**

**地面に立てるようにしてみよう。**

Chapter2でパックマンがジャンプできるようになりましたが、Aボタンを押さないとパックマンは地面を突き抜けて自由落下していたはずです。ではこれを地面に立てるように改造してみましょう。地面の位置はY座標で0の位置にあるので、パックマンのY座標が0より小さくなれば落下を行わないようにすれば地面に立てるはずです。ではプログラムを見ていきましょう。

今回編集するソースは下記のファイルになります。

ソースファイル

Packman\Packman\game\Player\ CPlayer.cpp

Packmap\Packman\game\CCamera.cpp

|  |
| --- |
| /\*!  \*@brief 移動処理。  \*/  void CPlayer::Move()  {  //XZ平面での移動速度。  m\_moveSpeed.x = 0.02f;  m\_moveSpeed.z = 0.02f;  if (KeyInput().IsAPress()) {  //キーボードのAが押されていたら速度を倍にする。  m\_moveSpeed.x \*= 2.0f;  m\_moveSpeed.z \*= 2.0f;  m\_moveSpeed.y = 0.1f;  }  //Y方向への移動速度。  if (KeyInput().IsUpPress()) {  m\_position.z += m\_moveSpeed.z;  }  if (KeyInput().IsDownPress()) {  m\_position.z -= m\_moveSpeed.z;  }  if (KeyInput().IsRightPress()) {  m\_position.x += m\_moveSpeed.x;  }  if (KeyInput().IsLeftPress()) {  m\_position.x -= m\_moveSpeed.x;  }  m\_position.y += m\_moveSpeed.y;  //重力とかは考えない。  m\_moveSpeed.y -= 0.01f;  **if (m\_position.y < 0.0f) {**  **//座標が0以下になったので座標を補正。**  **m\_position.y = 0.0f;**  **}**  } |

黒字の部分が今回変更した部分になります。

さて、この変更でパックマンは地面をすり抜けて落下しなくなりましたが、実はまだ問題が残っています。カメラのプログラムを下記のように書き換えてみてください。

CCamera.cpp

|  |
| --- |
| void CGameCamera::Start()  {  CVector3 cameraTarget;  **m\_playerDist.Set(0.0f, 0.5f, -1.5f);**  cameraTarget = m\_playerDist;  cameraTarget.z = 0.0f;  m\_camera.SetPosition(m\_playerDist);  m\_camera.SetTarget(CVector3::Zero);  m\_camera.SetUp(CVector3::Up);  m\_camera.SetFar(100000.0f);  m\_camera.SetNear(0.1f);  m\_camera.SetViewAngle(CMath::DegToRad(45.0f));  m\_camera.Update();  } |

カメラが下記の図のようにプレイヤーの後方に移動したはずです。



実はパックマンの座標は球体の中心を指しています。そのためY座標0で判定を行うとパックマンが地面にめり込んでしまいます。つまり判定するY座標をパックマンの半径分押し上げてやればめり込まなくなるはずです。ではめり込まないようにプログラムを改良してみましょう。パックマンの半径は0.08とします。

CPlayer.cpp

|  |
| --- |
| /\*!  \*@brief 移動処理。  \*/  void CPlayer::Move()  {  //XZ平面での移動速度。  m\_moveSpeed.x = 0.02f;  m\_moveSpeed.z = 0.02f;  if (KeyInput().IsAPress()) {  //キーボードのAが押されていたら速度を倍にする。  m\_moveSpeed.x \*= 2.0f;  m\_moveSpeed.z \*= 2.0f;  m\_moveSpeed.y = 0.1f;  }  //Y方向への移動速度。  if (KeyInput().IsUpPress()) {  m\_position.z += m\_moveSpeed.z;  }  if (KeyInput().IsDownPress()) {  m\_position.z -= m\_moveSpeed.z;  }  if (KeyInput().IsRightPress()) {  m\_position.x += m\_moveSpeed.x;  }  if (KeyInput().IsLeftPress()) {  m\_position.x -= m\_moveSpeed.x;  }  m\_position.y += m\_moveSpeed.y;  //重力とかは考えない。  m\_moveSpeed.y -= 0.01f;  **if (m\_position.y < 0.08f) {**  **//座標が半径以下になったので座標を補正。**  **m\_position.y = 0.08f;**  **}**  } |

いかがでしょうか？パックマンが地面にめり込まなくなったはずです。

**Chapter 4**

**食べ物を食べられるようにしてみよう。**

このチャプターではパックマンと食べ物の距離が一定値以下になったら、食べ物を食べたと判定して、食べ物を削除する処理を実装してみましょう。

**4.1 ゲームループ**

まず、少し話がズレますが、どんなゲームのプログラムでもゲームが起動している間はループし続けるゲームループと言われるものがあります。このループは60fpsのゲームなら16ミリ秒に一度、30fpsのゲームなら33ミリ秒に一度の周期でループしています。そしてどのゲームでも必ず、このループの中にゲームの状態の更新処理や描画処理が記述されています。

|  |
| --- |
| int main()  {  //これがゲームループ。  while(true){  　　　//ゲームの状態を更新する。  //キャラの座標とか、ＨＰとか色々と。  　　　Update();  //画面に絵を描く。  　　　Render();  //画面のリフレッシュレートに同期させる。  　　　WaitVSync();  }  } |

非常に簡素なプログラムですが、どのゲームでも同様のコードが必ずあります。

皆さんが今まで見てきた、パックマンのプログラムであれば、CFood::UpdateやCPlayer::Updateが状態の更新処理。CFood::RenderやCPlayer::Renderが描画処理ということになります。

**4.2 ２点間の距離の計算**

今回のお題の食べ物を削除する処理はゲームの状態を更新する処理になります。ですので、どこかのUpdate関数にそのコードを記述すればいいことになります。

その手のコードをどこに記述するのかは、プログラマがやりやすいように決めていいのですが、今回はCFood::Update関数の中にそのコードを記述していくことにします。

では話を戻して、まずプレイヤーと食べ物の距離を計算することができないと削除を行うことはできません。プレイヤーと食べ物の距離は下記の計算で求まります。

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子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関数を用意しています。

|  |
| --- |
| //プレイヤーの座標を取得。  CVector3 p = Player().GetPosition();  Cvector3 e = m\_position;  CVector3 v;  v.x = p.x – e.x;  v.y = p.y – e.y;  v.z = p.z – e.z;  //これでLに長さが入る。  //関数の中で３平方の定理の計算をしている。  float L = v.Length(); |

いかがでしょうか？思っていたより簡単なコードではないでしょうか。あなたが記述した数式は引き算だけです。ゲーム会社ではベクトル計算の多くは関数として簡単に使用できるように用意されています。

慣れてくるまでは難しく感じるかもしれませんが、使い始めるとそこまで難しくありません。

**4.3 実際に消してみよう**

距離の計算の仕方も分かったので、実際に食べ物を消してみましょう。私の作ったエンジンでは食べ物を消すためには下記のような少々特殊なコードを記述する必要があります。

|  |
| --- |
| CGameObjectManager::Instance().DeleteGameObject(this); |

では、食べ物を削除するコードを記述します。

|  |
| --- |
| CVector3 p = Player().GetPosition();  CVector3 e = m\_position;  CVector3 v;  v.x = p.x - e.x;  v.y = p.y - e.y;  v.z = p.z - e.z;  float L = v.Length();  if (L < 0.08f) {  CGameObjectManager::Instance().DeleteGameObject(this);  } |

このコードをCFood::Update関数に追加すると食べ物を削除することができます。

**Chapter 5**

**パックマンと壁の当たり判定**

ゲームをプレイしていて、キャラクターが壁を簡単にすり抜けてしまうゲームは基本的にゲームとしては欠陥品になってしまいます。そのため、世に出回っているゲームのほとんどは何かしらの当たり判定を実装しています。このチャプターでは2Dゲームの頃から使われていた、簡単な当たり判定を実装してパックマンが壁にめり込まないようにしてみましょう。

**5.1 色々な当たり判定**

パックマンの当たり判定の説明を行う前に、3Dゲームで使われている、比較的簡単な当たり判定をいくつか紹介します。(簡単ではないかもしれませんが)

**5.1.1 球と球の当たり判定**

ゲームで最も良く使われる当たり判定かもしれません。非常に高速に動作して、実装も容易なためいろいろなゲームで使われています。

　実は、この当たり判定は皆さんすでに実装を行っています。なんのことか分かるでしょうか？皆さんにChapter4で、パックマンと食べ物の距離を調べてある一定距離以下なら食べ物を食べるというプログラムを組んでもらったと思います。実はあれが球と球の当たり判定になります。

半径

半径

球Aと球Bの中心座標を減算すると、球Aと球Bを結ぶベクトルが見つかります。そして、このベクトルの長さは３平方の定理を使用すれば求まります。

そして、このベクトルの長さが球Aの半径と球Bの半径を足した大きさよりも小さければ衝突していると判定できます。Chapter4で行ったことと全く同じです。

**5.1.2 AABBとAABBの当たり判定**

AABBというのは軸平行のバウンディングボックスと言われるものです。オブジェクトを内容する箱形状の当たり判定だと思ってください。AABBは一般的に下記のような構造体で表現されます。

|  |
| --- |
| struct Aabb{  Vector3 vMax; //箱の最大値。  Vector3 vMin;　 //箱の最小値。  }; |

そして、箱A(aabbA)と箱B(aabbB)の衝突判定は下記のような条件文で実装できます。

|  |
| --- |
| if(( aabbA.vMin.x <= aabbB.vMax.x )  && ( aabbA.vMax.x >= aabbB.vMin.x )  && ( aabbA.vMin.y <= aabbB.vMax.y )  && ( aabbA.vMax.y >= aabbB.vMin.y )  && ( aabbA.vMin.z <= aabbB.vMax.z )  && ( aabbA.vMax.z >= aabbB.vMin.z )  ){  //衝突している。  } |

**5.1.3 配列を使用した当たり判定**

　ここまで見てきた当たり判定は、みなさんが個人制作で3Dゲームの作成を行いだすと、恐らく一番お世話になる当たり判定になると思います。しかし、まだ少し難しいのではないかと思います。そこで今回は配列を使用した、古典的な、しかしゲームによっては今でも使える当たり判定を実装してもらいます。

パックマンのマップはCMapBuilderのsMap配列を使用して、構築されています。このSMap配列の値が１ならば壁なので移動しないという処理を記述してやれば、パックマンは壁にめり込まなくなるはずです。ではこのトリックをお教えしましょう。

パックマンの床と壁のサイズはGRID\_SIZEになります。

このGRID\_SIZEでパックマンの座標を除算してやると、パックマンがマップ上のどの場所にいるのかを判定することができます。

|  |
| --- |
| int x = (int)(m\_position.x / GRID\_SIZE);:  int z = (int)(m\_position.z / -GRID\_SIZE);  if(sMap[z][x] == 1){  //壁だよ！！！  } |

いかがでしょうか。非常に簡単なプログラムで壁の判定が行えました。では、ここまでの説明を参考にしてパックマンが壁をすり抜けないようにしてください。

今回はパックマンが壁に半分めり込むと思いますが、それは考慮しなくて構いません。

**5.2 壁にめり込まないようにしてみよう。**

　前節で行った当たり判定ですと、パックマンの体が半分めり込んでしまっていました。これはパックマンの中心座標を使って、壁との当たり判定を行っていたことが原因になります。

この座標で当たり判定を行っていた。

　　　では、バウンディングボックスを使ってもう少しだけマシにしてみましょう。中心座標で判定を行っているのがめり込みの原因ですので、めり込まないようにパックマンを内包するバウンディングボックスの４隅の座標を使って当たり判定を行ってみようと思います。

この４点を使う。

プログラムで記述すると下記のようになります。

|  |
| --- |
| //プレイヤーを内包する四角形の４隅の当たり判定を行う。  bool isHitWall = false;  float radius = 0.075f;  //左上  int x = (int)((pos.x - radius) / GRID\_SIZE);  int z = (int)((pos.z - radius) / -GRID\_SIZE);  if (sMap[z][x] == 1) {  //壁  isHitWall = true;  }  //右上  x = (int)((pos.x + radius) / GRID\_SIZE);  z = (int)((pos.z - radius) / -GRID\_SIZE);  if (sMap[z][x] == 1) {  //壁  isHitWall = true;  }  //左下  x = (int)((pos.x - radius) / GRID\_SIZE);  z = (int)((pos.z + radius) / -GRID\_SIZE);  if (sMap[z][x] == 1) {  //壁  isHitWall = true;  }  //右下  x = (int)((pos.x + radius) / GRID\_SIZE);  z = (int)((pos.z + radius) / -GRID\_SIZE);  if (sMap[z][x] == 1) {  //壁  isHitWall = true;  } |

**Chapter 6**

**経路探索**

ゲームAIを語る上で外せない要素の一つに経路探索と言われるものがあります。数年前まで、ゲームAIの技術といえば経路探索というくらいホットな話題でした。フォトリアルなFPSゲームで敵兵が壁をすり抜けてきたら興ざめすると思います。チャプター６ではLesson10のプログラムを使いながら経路探索について見ていきます。

**6.1 プログラムの説明**

　経路探索のプログラムは下記のパスにあります。

　　・tkEngine/AI/tkPathFinding.h

・tkEngine/AI/tkPathFinding.cpp

採用されているアルゴリズムはダイクストラ法。

　ネットワーク構造の経路探索データを受け取ることで経路探索が行えます。

　実際に経路探索をゲームで使用しているプログラムは下記のパスになります。

　　・PackMan/game/Enemy/CEnemy.h

・PackMan/game/Enemy/CEnemy.cpp

**6.2 アルゴリズム**

経路探索のアルゴリズムで代表的なものといえば下記の二つがあげられます。

**・ダイクストラ法**

**・A\*アルゴリズム**

A\*アルゴリズムはダイクストラ法の改良版となっており、多くのゲームにおいてはA\*アルゴリズムの方が高速なアルゴリズムになります。ただし、ダイクストラ法の方が高速な場合もあって、ゴール地点が決まっている場合はダイクストラ法の方が高速になります。FPSなどのような広大なマップでダイクストラ法を採用すると計算量が膨大になり、パフォーマンスが大きく低下するため、まずありえません。今回のサンプルプログラムではダイクストラ法を使用していますが、ダイクストラ法とA\*アルゴリズムとで実装難易度に差はありませんので、是非自分たちでチャレンジしてみて下さい。

**6.3 データ構造**

経路探索を行うためには、通れる道を表すデータが必要になります。現在のゲームで主流となって使用されているデータはナビゲーションメッシュと言われるものです。ナビメッシュを作るまでもないようなゲームになりますと、ウェイポイントと言われるものを使っているものもあります。

ナビゲーションメッシュ、ウェイポイントどちらを採用しても基本的なデータ構造に大きな違いはありません。両者とも隣接するノードとのリンクをもったネットワークのようなデータ構造になっています。

プログラムでは下記のような構造体が使われます。

|  |
| --- |
| struct SNode{  SNode\* linkNode[3]; //隣接ノード  CVector3 position; //ノードの座標。  }; |

実習課題

　Lesson10のプログラムは敵が最初から最後までプレイヤーを追いかけてくるだけの挙動になっています。この敵の挙動をもう少しマシにして面白いゲームにしてください。仕様は自由です。あなたの好きなように面白いゲームを作ってみてください。

**Chapter 7**

**ゲームで使用されるデザインパターン**

　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紹介します。今まで皆さんに見てもらってきたパックマンのプログラムでも使われています。

**7.1 ゲームループ**

まずは4.1でも紹介した、ゲームループについて見ていきましょう。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するということです。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();**

}

**7.1.2 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。)

**7.2 UpdateMethod**

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、皆さんが実習で使っていたパックマンのプログラムでも使用しています。

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。」**

パックマンのプログラムであれば、下記のようなコードが該当します。

**CGameObjectManager::Instance().NewGameObject<CEnemy>(0);**

このコードはCEnemy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関数とRender関数が自動で呼び出されます。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、C++を学んでいる最中の皆さんにとっては、UpdateMethodパターンよりもポリモーフィズム(多態性)を活用しているコードが興味深いかもしれません。

|  |
| --- |
| /////////////////////////////////////////////////////////////////////////////  //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。

**Chapter 8 テニスゲームを作ろう**

このチャプターでは未完成のテニスゲームのサンプルプログラムを使用して、ゲームを完成させてみましょう。

**8.1 プログラムの構成**

**Game.cpp,Game.h**

Gameクラス。ゲームのメイン関数のような処理になる。

Game::Update関数。ゲームループから毎フレーム呼ばれる更新関数。

　　 Game::Render関数。ゲームループから毎フレーム呼ばれる描画処理。

　　 Game::Start関数。ゲーム開始して一度だけ呼ばれる開始関数。

**Player.cpp,Player.h**

Playerクラス。ユーザーが操作するプレイヤークラス。

　　　Player::Update関数。Game::Updateからコールされている更新関数。

　　　Player::Render関数。Game::Renderからコールされている描画関数。

　　　Player::Init関数。Game::Start関数からコールされている初期化関数。

**ball.cpp,ball.h**

Ballクラス。テニスボールクラス。テニスコートの外に出ようとすると反射する。

プレイヤーに衝突しても反射します。

　　　Ball::Update関数。Game::Updateからコールされている更新関数。

　　　Ball::Render関数。Game::Renderからコールされている描画関数。

**Court.cpp,Court.h**

Courtクラス。テニスコートクラス。

**GameCamera.cpp,GameCamera.h**

GameCameraクラス。

**8.2 アニメーション付き3Dモデル表示。**

　今回のサンプルからアニメーション付き3Dモデルを表示する機能が追加されています。この節ではその機能について説明していきます。

**8.2.1 Xファイル**

Xファイルとはモデルフォーマットと言われるもので、DirectX9までサポートされていた形式になります。DirectX10以降はサポートされていないため、Xファイル自体は過去の遺物となっています。しかし、モデルフォーマットというのはどこに行っても通用する標準化されたものというものは存在しません。そして、3Dモデルを表示する基本的な理論というのは１０年以上前から変化がなく、枯れた技術となっています。そのためXファイルを使ったモデル表示を学ぶことは無駄にはなりません。

**8.2.2 CSkinModel、CSkinModelData**

この二つのクラスはXファイルを使用した、3Dモデルを表示するための機能を提供するクラスです。典型的な使用方法を下記に記述します。

**Xファイルのロード**

|  |
| --- |
| CSkinModelData modelData;  CSkinModel model;  void Init()  {  modelData.LoadModelData(“Assets/modelData/player.x”, NULL);  model.Init(&modelData);  } |

**ワールド行列の更新**

|  |
| --- |
| void Update()  {  model.UpdateWorldMatrix(position, CQuaternion::Identity, CVector3::One);  } |

**モデルの表示**

|  |
| --- |
| void Render(CRenderContext& renderContext)  {  model.Draw(renderContext, gameCamera->GetViewMatrix(), gameCamera->GetProjectionMatrix());  } |

実習課題

1. 対戦相手を表示できるようにする。
2. 対戦相手もボールを返せるようにする。
3. 対戦相手はAIで勝手に動作するようにする。

(AIというほど大したものである必要はありません。自動で動いていればＯＫです。)

**Chapter 9 回転**

このチャプターではLesson\_12のプログラムを使用して、3Dオブジェクトを回転させる方法を学んでいこうと思います。

**9.1 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。

**9.1.1 任意の軸周りの回転**

　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、下のようにユニティちゃんを回転させた場合はY軸周りに90度回転させることになります。

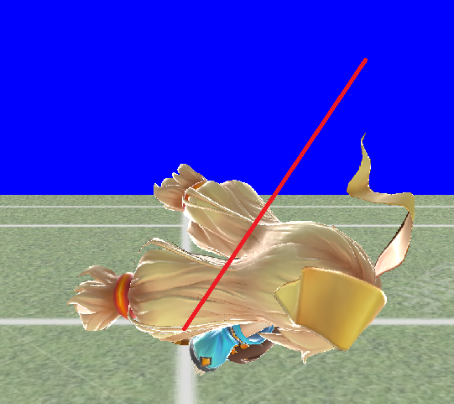
　　　　　　　　　　

Y軸周りに９０度回転

では次はX軸周りに回転させてみましょう。

では最後に下記のように斜めの軸で回してみましょう。

では、クォータニオンではこれらの回転をどのように表現するのかを見ていきましょう。

任意の軸をaxis、回転角度をθとして、回転を表すクォータニオンをqRotとすると

　　qRot.x = axis.x \* sin(0.5θ)

qRot.y = axis.y \* sin(0.5θ)

qRot.z = axis.z \* sin(0.5θ)

qRot.w = cos(0.5θ)

となる。

今回皆さんに提供しているサンプルプログラムには、この計算は関数化されているため、下記のようなプログラムを記述するだけで、回転クォータニオンを作成できます。

*Y軸周りに90度回転させるクォータニオンを作成するサンプルコード。*

|  |
| --- |
| CQuaternion qRot;  qRot.SetRotation(CVector3(0.0f, 1.0f, 0.0f), CMath::DegToRad(90.0f)); |

続いて、斜めの軸で回転させる場合のサンプルコードを紹介します。任意の軸というのは大きさ１のベクトルにする必要があるということに注意してください。

|  |
| --- |
| //斜めの軸を作成する。  CVector3 rotAxis(1.0f, 1.0f, 0.0f);  //大きさ１にするためにベクトルを正規化。  rotAxis.Normalize();  //Unityちゃんを回す。  CQuaternion qRot;  qRot.SetRotation(rotAxis, CMath::DegToRad(-90.0f)); |

**9.2 クォータニオンの乗算**

ゲームで回転を扱いだすと、「Y軸に90度回した後で、X軸に45度回したいとか」、「毎フレームY軸周りに５度ずつ回したい」などと言ったことを実装したくなってきます。これらの仕様はクォータニオン同士の乗算を実装すると実現することができます。クォータニオン同士の乗算は下記のように行います。

X軸周りの回転を表しているクォータニオンをqRotX、Y軸周りの回転を表しているクォータニオンをqRotYとしたとき、乗算されたクォータニオンqRotは下記の計算で求まる。

qRot.w = qRotX.w \* qRotY.w - qRotX.x \* qRotY.x - qRotX.y \* qRotY.y - qRotX.z \* qRotY.z

qRot.x = qRotX.w \* qRotY.x + qRotX.x \* qRotY.w + qRotX.y \* qRotY.z - qRotX.z \* qRotY.y

qRot.y = qRotX.w \* qRotY.y - qRotX.x \* qRotY.z + qRotX.y \* qRotY.w + qRotX.z \* qRotY.x

qRot.z = qRotX.w \* qRotY.z + qRotX.x \* qRotY.y - qRotX.y \* qRotY.x + qRotX.z \* qRotY.w

皆さんに提供しているtkEngineにはクォータニオンの乗算を行う処理を用意していますので、上記の計算を行う必要はありません。tkEngineのCQuaternionを使用した場合のクォータニオンの乗算のサンプルコードを下記に示します。

|  |
| --- |
| CQuaternion qRotX, qRotY, qRot;  //X軸周りに45度回転するクォータニオンを作成する。  qRotX.SetRotation(CVector3(1.0f, 0.0f, 0.0f), CMath::DegToRad(45.0f));  //Y軸周りに90度回転するクォータニオンを作成する。  qRotY.SetRotation(CVector3(0.0f, 1.0f, 0.0f), CMath::DegToRad(90.0f));  //X軸周りの回転とY軸周りの回転を乗算する。  qRot.Multiply(qRotX, qRotY); |

クォータニオンの乗算は交換法則が成り立っていないとこに注意してください。例えば4×5は5×4にしても結果は20になります。これは交換法則が成り立っています。しかしクォータニオンはqRotX×qRotYとqRotY×qRotXで結果が異なります。

実習課題

Lesson\_13のプログラムを使用して下記の仕様を実装しなさい。解答例となる実行ファイルを下記のパスにアップしているので、挙動はそれを参考にしなさい。

Lesson\_13/解答のデモプログラム/ Answer.exe

1. キーボードの左右キーが押されるとユニティちゃんをY軸周りに回転させる。
2. キーボードの上下キーが押されるとユニティちゃんをX軸周りに回転させる。

**Chapter 10 拡大**

ここまで3Dゲームのキャラクターの位置を表すデータとして３要素のベクトル型、回転を表すデータとしてクォータニオンを紹介してきました。このチャプターではキャラクターの拡大を見ていこうと思います。

3Dゲームで拡大下記のように３要素のベクトル型として扱われます。

|  |
| --- |
| //プレイヤークラス。  class Player{  public:  CVector3 position; //座標  　CQuaternion rotation; //回転  **CVector3 scale; //拡大**  }; |

そして、プレイヤーを等倍で表示したい場合は拡大率を下記のように設定します。

|  |
| --- |
| Player player;  player.scale.x = 1.0f; //X軸方向の拡大率。  player.scale.y = 1.0f;　//Y軸方向の拡大率。  player.scale.z = 1.0f; //Z軸方向の拡大率。 |

もう勘のいい人は気づいているかと思いますが、プレイヤーをX軸方向に拡大したい場合はplayer.scale.xの値を変更してやることになります。Y軸であればplayer.scale.y、Z軸であればplayer.scale.zの値を変更してやることになります。

**10.1 ミラー**

拡大の考え方はそこまで難しいものではないと思いますので、この節では拡大のちょっとしたトリックのようなテクニックを使った、ミラーモデルと言われる、まるで鏡に映っているかのように左右反転しているモデルの表示の仕方を教えます。実はミラーモデルはX軸方向の拡大率を-1.0倍するだけで実現できます。

|  |
| --- |
| Player player;  player.scale.x = -1.0f; |

たったこれだけです。どうでしょうか非常に簡単でしょう？

モデルをX軸方向に-1.0倍するだけで、ミラーモデルの完成です。ただし、実はまだこれだけでは絵は正しく表示されません。実は3Dモデルを構成するポリゴンと言われるものには表面と裏面というものが存在します。そして、描画負荷を上げないために大抵の場合は裏面の描画は行われないようになっています。モデルをX軸方向に-1.0倍したということはモデルをひっくり返したことになりますので、ポリゴンの裏面が表面に来てしまうことになるので、このままではモデルはまともに表示されません。そのため、下記のようなコードをDraw関数の前で実行して、これから各モデルは裏面を描画しますよ～という風にGPUに教えてやる必要があります。

|  |
| --- |
| //これから描画するモデルが裏面描画であることをGPUに教える。  renderContext.SetRenderState(RS\_CULLMODE, CULL\_CW);  skinModel.Draw(renderContext,camera.GetViewMatrix(), camera.GetProjectionMatrix());  //モデルを書いたら表面描画に戻しておく。じゃないと、これ以降の描画がおかしくなってしますので。  renderContext.SetRenderState(RS\_CULLMODE, CULL\_CCW); |

実習課題

　Lesson\_14を使用して下記の仕様を実装しなさい。解答となるデモプログラムは下記のパスにアップしているので、それを参考にしなさい。

Lesson\_14/解答のデモプログラム/ Answer.exe

1. 上下のキーが押されるとユニティちゃんがY軸方向に拡大縮小する。
2. 左右のキーが押されるとユニティちゃんがX軸方向に拡大縮小する。
3. X軸方向のミラーモデルの処理を実装する。

**Chapter 10 行列**

ここまでで、3Dモデルをワールド空間で移動、回転、拡大する方法を見てきました。移動は3要素のベクトル(x,y,z)、回転はクォータニオン(x,y,z,w)、拡大は3要素のベクトル(x,y,z)を使用して表現していました。このチャプターまでの内容をしっかりと理解できていれば、3Dゲームのプレイヤーを作成しようと考えた場合、下記のようなクラスを作成することが考えられるはずです。

Player.h

|  |
| --- |
| //プレイヤークラスの定義。  class Player{  public:  //////////////////////////////////////////  　//メンバ変数  //////////////////////////////////////////  CVector3 position; //座標。  CQuaternion rotation; //回転。  　CVector3 scale; //拡大率。  ///////////////////////////////////////////  　//メンバ関数。  　//////////////////////////////////////////  //コンストラクタ。  Player();  //デストラクタ。  　~Player();  //更新処理。  　void Update();  }; |

Player.cpp

|  |
| --- |
| //コンストラクタ  Player::Player()  {  //コンストラクタでメンバ変数を初期化。  position.x = 0.0f;  　 position.y = 0.0f;  position.z = 0.0f;  rotation.x = 0.0f;  rotation.y = 0.0f;  rotation.z = 0.0f;  rotation.w = 1.0f;    scale.x = 1.0f;  scale.y = 1.0f;  scale.z = 1.0f;  }  void Player::Update()  {  if(KeyInput().IsLeftPress()){  //左のキーが押された。  　　　position.x -= 0.5f;  }  if(KeyInput().IsRightPress()){  //左のキーが押された。  　　　position.x += 0.5f;  }  if(KeyInput().IsUpPress()){  //左のキーが押された。  　　　position.z += 0.5f;  }  if(KeyInput().IsDownPress()){  //左のキーが押された。  　　　position.x -= 0.5f;  }  } |

しかし、実はposition、rotation、scaleの値をいくら変更しても3Dモデルはワールド空間上を移動、回転、拡大することはありません。3Dモデルをワールド空間で動かすためには最終的には行列というものを作成する必要があるからです。

**10.1 行列とは？**

　行列とはその名のとおり、行と列で成り立つデータです。3Dゲームでは主として、4×4行列か3×4行列が使用されます。このチャプターでは4×4行列を扱います。

4x4行列は下記のように表現されます。

プログラムで記述すると下記のようなコードになります。

|  |
| --- |
| float matrix[4][4]; |

3Dモデルをワールド空間で移動、回転、拡大するためには、それらの情報を使って**ワールド行列**と言われるものを作成する必要があります。

**10.2 平行移動行列**

まず、ワールド行列の前に平行移動行列について見ていきましょう。名前から推測できる人もいるかもしれませんが、これがワールド空間上の座標を表すことになります。平行移動行列は下記のような形になります。例えば、プレイヤーの座標が10、20、30になる場合は平行移動行列は下記のようになります。

赤字になっている部分が平行移動成分となります。つまり、行列の４行目に平行移動成分が記録されることになります。

では、ベクトルから平行移動行列を作成するコードを見てみましょう。

|  |
| --- |
| CMatrix transMatrix;  //一行目を設定していく。  transMatrix.m[0][0] = 1.0f;  transMatrix.m[0][1] = 0.0f;  transMatrix.m[0][2] = 0.0f;  transMatrix.m[0][3] = 0.0f;  //２行目を設定していく。  transMatrix.m[1][0] = 0.0f;  transMatrix.m[1][1] = 1.0f;  transMatrix.m[1][2] = 0.0f;  transMatrix.m[1][3] = 0.0f;  //３行目を設定していく。  transMatrix.m[2][0] = 0.0f;  transMatrix.m[2][1] = 0.0f;  transMatrix.m[2][2] = 1.0f;  transMatrix.m[2][3] = 0.0f;  //４行目を設定していく。  transMatrix.m[3][0] = position.x;  transMatrix.m[3][1] = position.y;  transMatrix.m[3][2] = position.z;  transMatrix.m[3][3] = 1.0f; |

非常に長いコードになってしまいました。私が皆さんに提供しているtkEngineのCMatrixクラスには平行移動行列を生成するメンバ関数が用意されていて、それを使用すると下記のようなコードになります。

|  |
| --- |
| CMatrix transMatrix;  transMatrix.MakeTranslation(position); //平行移動行列を作成する。 |

**10.3 回転行列**

続いて回転を表す回転行列について見てみましょう。例えばY軸周りにθ回転する回転行列の場合は下記のようになります。

続いてX軸周りにθ回転する回転行列は下記のようになります。

最後にZ軸周りにθ回転する回転行列は下記のようになります。

X軸周りにθ、Y軸周りにθ回転する行列は行列の乗算を行うことで求めることができます。

では、クォータニオンから回転行列を作成するコードを見てみましょう。クォータニオンから回転行列を生成するプログラムは非常に複雑になるので、最初からtkEngineのCMatrixクラスのメンバ関数を使用します。

|  |
| --- |
| CMatrix rotationMatrix;  rotationMatrix.MakeRotationFromQuaternion(rotation); //クォータニオンから回転行列を生成。 |

**10.4 拡大行列**

最後に拡大を表す拡大行列について見ていきましょう。例えば、プレイヤーの拡大率がX軸に2.0倍、Y軸に1.5倍、Z軸に0.5倍の場合は下記のようになります。

このようになります。

では、拡大行列を生成するコードを見ていきましょう。このコードもCMatrixクラスのメンバ関数を使用します。

|  |
| --- |
| CMatrix scaleMatrix;  scaleMatrix.MakeScaling(scale); |

**10.5 ワールド行列**

　では、いよいよ3Dモデルをワールド空間で動かすためのワールド行列を見ていきましょう。ワールド行列は平行移動行列、回転行列、拡大行列を混ぜ合わせたものとなります。このまぜ合わせは行列の乗算によって実現できます(クォータニオンと似ています)。

CMatrixクラスには行列の乗算を行うメンバ関数が用意されています。その関数を利用して、ワールド行列を作成するコードを見てみましょう。

|  |
| --- |
| //平行移動行列を作成する。  CMatrix transMatrix;  transMatrix.MakeTranslation(position);  //回転行列を作成する。  CMatrix rotationMatrix;  rotationMatrix.MakeRotationFromQuaternion(rotation);  CMatrix scaleMatrix;  scaleMatrix.MakeScaling(scale);  //ワールド行列を作成する。  CMatrix worldMatrix;  worldMatrix.Mul(scaleMatrix, rotationMatrix);  worldMatrix.Mul(worldMatrix, transMatrix); |

行列の乗算はクォータニオンと同様に交換法則が成り立っていません。scaleMatrix×rotationMatrixとrotationMatrix×scaleMatrixの結果は異なります。

ワールド行列を作成するときの乗算の順番は基本的に下記のようになります。

**拡大行列×回転行列×平行移動行列**

この乗算順番を間違えると、意図しない結果になることがあります。

**10.6 モデルの頂点座標のワールド変換**

　3Dモデルがワールド空間で移動、回転、拡大するということはモデルの頂点が移動、回転、拡大していることになります。モデルの頂点座標は3次元のベクトルで表現されています。このモデルの頂点座標を先ほど作成したワールド行列で変換していくことでモデルはワールド空間を歩く、旋回、拡大することができるようになります。では3次元のベクトルを行列を使って変換する式を見てみましょう。変換前の頂点座標をV、変換後の頂点座標をV´とします。

=

行列とベクトルの乗算は下記のように計算されます。

この頂点座標の変換はGPUで実行されるシェーダープログラムで行われます。このシェーダープログラムもプログラマが記述する必要があります。今回はシェーダーの説明は行いませんが、シェーダーは非常に重要な技術でシェーダーがかけるプログラマというのは非常に重宝されることになります。

**Chapter 11 ゲームエンジン**

　ゲーム制作を強烈にサポートしてくれるものに、ゲームエンジンと呼ばれるものがあります。ゲームエンジンはUnity、UnrealEngine4といった商用のものと、各ゲーム会社が独自に作成を行っている内製ゲームエンジンがあります。下記に有名どころの内製ゲームエンジンを上げてみます。

**・ＭＴフレームワーク**

　　カプコンの内製エンジン。XBox360、PS3の世代で使用された。

**・クリスタルツールズ**

　　スクウェアエニックスのツールの集まりのような開発環境。ゲームエンジンと呼ぶほ

どのものでもなかった？FF13、FF14で使用されている。

**・Foxエンジン**

　　メタルギアソリッド5などで使用されたコナミ製のエンジン。

これらは社外に名前がでてきたものですが、名前が出てこないだけで各社独自のエンジンを保持しています。

**11.1　ゲームエンジン時代**

日本でスマートフォンアプリを中心にUnityが広く使われるようになり、PS4などのコンソール機でUnrealEngine4採用タイトルが多数発売されだしました。そのため、現在は各社内製エンジンの開発などを行わなくてもゲームを開発できるようになっています。しかし、今でも内製エンジンの開発を行っている会社があります。何故でしょうか？ 理由としては下記の点が考えられます。

**・UnrealEngine4、Unityのライセンス料**

　　　商用のゲームエンジンは当然ですがお金を支払わないと企業は使用することはでき

ません。Unityなどはライセンス料は比較的安価なのですが、UnrealEngine4だと高価になるため使用しない場合があります。

**・昔から開発していたタイトルだと、エンジンもあり、ノウハウもたまっているため外部エンジンを使うメリットが弱い。**

　　　続編物のタイトルなどでこの傾向が強い。

**・UnrealEngine4やUnityよりある一点に特化すれば強力なエンジンが作れる。**

　　　アンチャーテッド、CoD、FF15、ナルティメットストームなどがそれに該当します。

このような理由で今も各会社が作成した内製ゲームエンジンが使用されています。

**11.1 河原内製ゲームエンジン**

　ゲームＰＧの授業ではtkEngineという河原内製のDirectXを使用したゲームエンジンを使用して授業を進めてきました。ここから先は、このエンジンをさらに活用して多数のミニゲームを作成していきます。

**11.2 なぜUnityやUnrealEngine4を使わないの？**

この授業で内製ゲームエンジンを使用する理由は下記があげられます。

　　・C++を学ぶため

　　・コンソール系の会社への就職作品としてUnity、UnrealEngine4を使った作品では

厳しい。まず無理。

なぜか？

　C++を駆使してDirectX、OpenGLなどでゲームが作れる人は、Unity、UnrealEngine4は勉強すればほぼ使えるが逆は怪しい。

　Unreal専門会社のヒストリアでさえ、学生にはC++でゲームが作れるスキルを求めている。

　　・スマホ系の会社への就職作品としてもC++からやっておけばUnity、UnrealEngine4は割と簡単。

内製エンジンを使うというのは、難易度的には下のようになります。

**エンジンを作る**>>>>>>>>>>**内製エンジンを使う**>>>>**UnityやUE4などを使う**

2年生になるとエンジンを自分で作り、ゲーム大賞作品や就職活動作品を作ることになるの

ですが、1年生はtkEngineを使用してゲーム制作を進めていきます。

**Chapter 12 ゲームオブジェクトマネージャー**

このチャプターではtkEngine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tkEngineの中で呼称している俗称になるのですが、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。

**12.1 Update Method パターン**

復習になりますが、7.2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。

**12.2 インスタンスの登録**

では、実際にインスタンスを登録してみましょう。GamePG\_1/GameTmplateを適当な場所にコピーして、GameTemplate.slnを立ち上げてソリューションエクスプローラーからmain.cppを開いてください。

まず、IGameObjectを継承してPlayerクラスを作成します。

|  |
| --- |
| class Player : public IGameObject{  public:  void Update(); //IGameObjectを継承したクラスは必ずUpdateを実装する必要がある。  }; |

ではPlayer::Updateを実装してみましょう。

|  |
| --- |
| void Player::Update()  {  if(GetKeyAsyncState(‘A’) != 0){ //Aボタンが押されたら  MessageBox(NULL, "Press A Button", "メッセージ", MB\_OK );  }  } |

キーボードのAが押されたらPress A Buttonというメッセージボックスが表示されるUpdate関数が実装できました。ではPlayerクラス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みましょう。wWinMain関数に下記のようなコードを追加してください。

|  |
| --- |
| int WINAPI wWinMain(  HINSTANCE hInst,  HINSTANCE hPrevInstance,  LPWSTR lpCmdLine,  int nCmdShow  )  {  //tkEngineの初期化。  InitTkEngine( hInst );  NewGO<Player>(0);　　　　　　　　　 //プレイヤーのインスタンスの生成。  Engine().RunGameLoop(); //ゲームループを実行。  return 0;  } |

NewGOはIGameObjectを継承したクラス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するための関数です。ちょっと難しいかもしれませんが、今はそういうものだと思っていてください。

では、これがどのようなケースで使われるのかFPSゲームを例に考えてみましょう。例えばFPSゲームでは銃のトリガーを引いたら弾を発射することができますよね？そのため下記のような実装が考えられます。

まずIGameObjectを継承したBulletクラスを作成します。

|  |
| --- |
| class Bullet : public IGameObject{  public:  void Update();  }; |

そして、BulletUpdateを実装します。

|  |
| --- |
| void Bullet::Update()  {  　MessageBox(NULL, “発砲されたよ”, “メッセージ”, MB\_OK);  } |

これで弾丸のクラスは作成できました。では先ほどのPlayer::Updateを改造して、キーボードのAボタンが押されたら弾丸を発射できるようにしてみましょう。

|  |
| --- |
| void Player::Update()  {  if(GetKeyAsyncState(‘A’) != 0){ //Aボタンが押されたら  NewGO<Bullet>(0); //弾丸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。  }  } |

**12.3 インスタンスの登録解除**

インスタンスの登録ができるのであれば、登録解除も行える必要があります。先ほどのFPSを例にすると、銃から発砲された弾丸は、キャラクターにヒットしたり、壁にヒットしたりしたらその寿命を終えます。ゲームの世界から消えたはずなのにいつまでもUpdate関数が呼ばれているのはおかしいですよね。そのため、役割を終えたインスタンスはゲームオブジェクトマネージャーから登録解除してやる必要があります。今回は２パターンの解除の方法を見てみましょう。

**12.3.1 NewGOの戻り値を使う**

NewGOは生成したインスタンスのアドレスを返してきます。それを使用して登録解除を行う方法を見てみましょう。

|  |
| --- |
| class Player : public IGameObject{  public:  　int numBullet = 0; //NewGOを使用して生成したインスタンスの数。  Bullet\* bulletArray[100]; //Bulletのポインタ型の配列。  　void Update();  }; |

Playerクラスのメンバ変数に生成したインスタンスの数を記録するメンバ変数と、Bulletのポインタ型の配列が追加されました。ではUpdate関数を見てみましょう。

|  |
| --- |
| void Player::Update()  {  if(GetAsyncKeyState(‘A’) != 0 && numBullet < 100){  //Bulletのインスタンスを生成。  bulletArray[numBullet] = NewGO<Bullet>(0);  　　//生成したので数をインクリメント。  numBullet++;  }  } |

キーボードのAが押されると最大100個まで弾丸を作れるようになりました。では、キーボードのBが押されると弾丸を一つ削除することができるようにプログラムを書き換えましょう。

|  |
| --- |
| void Player::Update()  {  if(GetAsyncKeyState(‘A’) != 0 && numBullet < 100){  //Bulletのインスタンスを生成。  bulletArray[numBullet] = NewGO<Bullet>(0);  　　//生成したので数をインクリメント。  numBullet++;  }  if(GetAsyncKeyState(‘B’) != 0 && numBullet > 0){  numBullet--;  **DeleteGO(bulletArray[numBullet]);**  }  } |

このように、tkEngineのゲームオブジェクトマネージャーはDeleteGOという関数に削除したいインスタンスのアドレスを渡してやることによって、削除を行うことができます。

**12.3.2 thisポインタを使う**

thisポインタを使用して、インスタンスが自分自身を削除することが可能です。例えば弾丸はインスタンスが生成されてから60フレーム経過すると削除されるという仕様だった場合、下記のようなコードを書く事ができます。

|  |
| --- |
| class Bullet : public IGameObject{  public:  int frameCount = 0; //フレーム数をカウントするためのメンバ変数。  void Update();  }; |

Bulletクラスにフレーム数をカウントするためのframeCountという変数が追加されました。ではUpdate関数の実装を見てみましょう。

|  |
| --- |
| void Bullet::Update()  {  frameCount++; //インスタンスが生成されてから経過したフレーム数をカウントアップ。  if(frameCount == 60){  //インスタンスが生成されてから60フレーム経過した  **DeleteGO(this); //自分自身を削除。**  }  } |

これでインスタンスが生成されてから60フレーム経過したら自動的に削除が行われます。

**Chapter 13 マップを作ろう**

　このチャプターではUnityのエディタ拡張を活用してマップの作成を行い、作成したマップをtkEngineを使用してゲーム中に表示する方法を勉強しましょう。

**13.1 tkToolsでモデルを配置**

下記のパスにUnityEditorを拡張したtkToolsというUnityのプロジェクトがあります。このツールを使用してオブジェクトを配置して、マップを作成していきましょう。

下記のパスにtkToolsの使い方の動画がありますので、そちらを参照してください。

***GamePG\_1/動画/tkToolsを使ったオブジェクト配置方法.mp4***

**13.2 配置情報を元にマップを表示する。**

　tkToolsから出力した配置情報を使用して、マップを表示しているサンプルは下記のパスにあります。

***GamePG\_1/MapChip***

ではサンプルプログラムの解説をしていきます。今回重要になるソースファイルは下記の４つになります。

**Map.cpp**

**Map.h**

**MapChip.cpp**

**MapChip.h**

**13.3 Mapクラス**

　MapクラスはtkToolsから出力された配置情報を使って、マップを構築しています。配置情報は下記のように#includeのトリックを使ってプログラムに組み込んでいます。

|  |
| --- |
| //マップの配置情報。  SMapInfo mapLocInfo[] = {  #include "locationInfo.h"  }; |

続いて、Map::Start関数でマップの配置情報を元にMapChipクラスのインスタンスを生成しています。MapChipクラスがマップを構成する各オブジェクトの描画を行うクラスになります。例えば、マップに100個のオブジェクトが配置されている場合は、100個のMapChipクラスのインスタンスが生成されます。

|  |
| --- |
| //マップにいくつのオブジェクトが配置されているか調べる。  int numObject = sizeof(mapLocInfo) / sizeof(mapLocInfo[0]);  //置かれているオブジェクトの数だけマップチップを生成する。  for (int i = 0; i < numObject; i++) {  MapChip\* mapChip = NewGO<MapChip>(0);  //モデル名、座標、回転を与えてマップチップを初期化する。  mapChip->Init(  mapLocInfo[i].modelName,  mapLocInfo[i].position,  mapLocInfo[i].rotation  ); |

**13.4 MapChipクラス**

　MapChipクラスはオブジェクトの一つ一つを表すクラスになります。下記のようなマップであればMapChipのインスタンスは二つ生成されることになります。



ではMapChipクラスのInit関数を見てみましょう。

|  |
| --- |
| void MapChip::Init(const char\* modelName, CVector3 position, CQuaternion rotation)  {  //ファイルパスを作成する。  char filePath[256];  sprintf(filePath, "Assets/modelData/%s.x", modelName );  //モデルデータをロード。  skinModelData.LoadModelData(filePath, NULL);  //CSkinModelを初期化。  skinModel.Init(&skinModelData);  //デフォルトライトを設定して。  skinModel.SetLight(&g\_defaultLight);  //ワールド行列を更新する。  //このオブジェクトは動かないので、初期化で一回だけワールド行列を作成すればおｋ。  skinModel.Update(position, rotation, CVector3::One);  } |

Init関数は引数にモデルの名前、配置座標、回転を受け取っています。この引数を元に、モデルをロードして、初期化を行っています。

あとは、描画を行うだけです。Draw関数を見ていましょう。

|  |
| --- |
| void MapChip::Render(CRenderContext& renderContext)  {  skinModel.Draw(  renderContext,  g\_gameCamera->GetViewMatrix(),  g\_gameCamera->GetProjectionMatrix()  );  } |

特別特殊な処理はありませんね？MapChipクラスは単にモデルをロードして、描画しているだけのクラスです。

**Chapter 14 コリジョン処理**

　このチャプターではゲームを作る上で避けて通ることができない、コリジョン処理について見ていきましょう。このチャプターではサンプルプログラムとしてGamePG\_1/CollisionDemoを使用します。

**13.1 衝突処理**

　ゲームにおいて物体同士の衝突処理というのはとても難易度の高いプログラムです。特に3Dゲームにおける衝突処理の難易度は2Dゲームの比ではありません。衝突処理には大きく分けて二つのフェーズがあります。　一つ目は衝突検出、そして二つ目は衝突解決です。衝突検出は物体がぶつかっているかどうかを調べる処理、衝突解決は物体が衝突している場合にそのめり込みを解決する処理です。

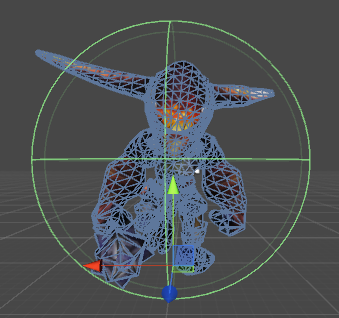
**13.2 物理エンジン**

　衝突処理はコリジョン処理とも呼ばれ、以降はコリジョンというキーワードを使用します。前節でコリジョン処理は非常に難易度が高いという話をしましたが、物理エンジンの登場によりその難易度は大幅に下がりました。河原内製エンジンのtkEngineはbulletPhysicsというオープンソースの物理エンジンを使用しています。そのため、今回皆さんがコリジョン処理の難易度の高い部分をプログラミングするということはありません。ありがたく物理エンジンを活用しましょう。

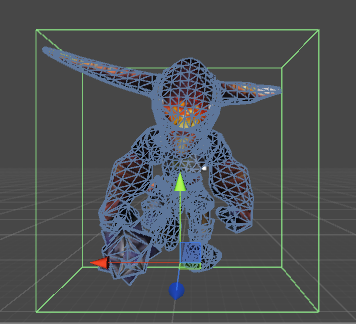
**13.3 Collider**

コリジョン処理を行うためには物体の形状を表すデータを作成する必要があります。このデータのことをUnityではColliderと呼称しています。この教材ではUnityに合わせて当たりデータのことをColliderと呼称します。Colliderにはいくつか種類があります。代表的なColliderを下記に示します。

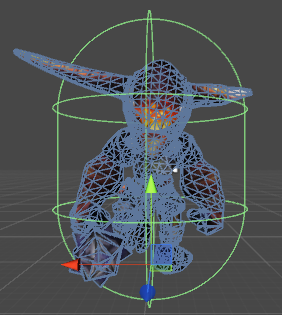
**・SphereCollider　(**球体形状のCollider)



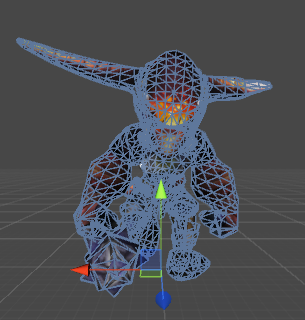
**・BoxCollider (**箱形状のCollider)



**・CapsuleCollider** (カプセル形状のCollider)



**・MeshCollider(**メッシュ形状のCollider)



多くのゲームではモデルの形状をそのままColliderにするMeshColliderを使うことはありません。コリジョン処理は複雑な形状で処理を行うほど、処理が重くなり、コリジョン抜けが発生する可能性が上がります。そのため、BoxCollider、SphereCollider、CapsuleColliderを組み合わせてColliiderを作成することが一般的です。

**13.4 MeshCollider**

前節で多くのゲームではMeshColliderを使うことはないといいましたが、MeshColliderであれば、モデルデータをそのまま流用してColliderを作成することが出来ます。そのため、ここではMeshColliderを使用してオブジェクトに当たりデータを作成してみようと思います。GamePG\_1/CollisionDemo/MapChip.hを開いて下さい。

|  |
| --- |
| class MapChip : public IGameObject  {  ・  　　　　　　　・  　　　　　　　・  **CMeshCollider meshCollider; //メッシュコライダー。**  **・**  **・**  **・**  }; |

MapChipクラスにmeshColliderというメンバ変数が追加されています。

続いて、MapChip.cppを開いてください。

|  |
| --- |
| void MapChip::Init(const char\* modelName, CVector3 position, CQuaternion rotation)  {  　　　　　　　・  　　　　　　　・  　　　　　　　・  **//メッシュコライダーの作成。**  **meshCollider.CreateFromSkinModel(&skinModel, &skinModel.GetWorldMatrix());**  **・**  **・**  **・**  } |

MapChip::Init関数にMeshColliderを作成するコードが追加されています。

第一引数にMeshColliderの元になるskinModelのインスタンスを渡しています。第二引数にはskinModelのワールド行列を渡しています。これでMeshColliderの作成は完了です。

**13.4 RigidBody**

　前節のプログラムでMeshColliderを作成することが出来ました。しかしまだコリジョン処理を行うことが出来ません。物理エンジンでコリジョン処理を行うためには剛体を作成して、その剛体を物理ワールドに登録する必要があります。剛体とは変形しない物体ということです。では剛体を作成するプログラムを見ていきましょう。MapChip.hを開いてください。

|  |
| --- |
| class MapChip : public IGameObject  {  　　　・  　　　・  　　　・  **CRigidBody rigidBody; //剛体。**  **}**; |

MapChipクラスにCRigidBody型のrigidBodyというメンバ変数が追加されました。

では剛体の作成のコードを見てみましょう。MapChip.cppを開いてください。

|  |
| --- |
| void MapChip::Init(const char\* modelName, CVector3 position, CQuaternion rotation)  {  　　　　・  　　　　・  　　　　・  //剛体の作成。  RigidBodyInfo rbInfo;  //剛体のコライダーを渡す。  rbInfo.collider = &meshCollider;  //剛体の質量。0.0だと動かないオブジェクト。背景などは0.0にしよう。  rbInfo.mass = 0.0f;  rbInfo.pos = position;  rbInfo.rot = rotation;  rigidBody.Create(rbInfo);  //作成した剛体を物理ワールドに追加する。  PhysicsWorld().AddRigidBody(&rigidBody);  } |

MapChip::Init関数に剛体を作成して、作成した剛体を物理ワールドに登録しています。

まず、剛体を作成するための情報となるRigidBodyInfoを作成しています。剛体の形状を表すcolliderの設定や質量の設定などを行っています。質量が0.0になっている点に注意してください。質量を0.0にすると動かないオブジェクトになります。建物や地面などの動くと困るオブジェクトは質量を0.0にしましょう。

**13.5 CharacterController**

　これで背景のコライダーの作成もできて、剛体も物理ワールドに登録できました。あとはキャラクターを動かしてコリジョン処理を行うだけです。tkEngineにはキャラクターの制御を行ってくれるCharacterControllerというクラスがあります。では、GamePG\_1/CollisionDemo/Player.hを開いてください。

|  |
| --- |
| #include "tkEngine/character/tkCharacterController.h"  class Player : public IGameObject  {  　　　・  　　　・  　　　・  CCharacterController characterController //キャラクタ―コントローラー。  }; |

PlayerクラスにCCharacterController型のcharacterControllerというメンバ変数が追加されています。ではGamePG\_1/CollisionDemo/Player.cppを開いて下さい。

|  |
| --- |
| void Player::Start()  {  　　　・  　　　・  　　　・  characterController.Init(0.5f, 1.0f, position);  } |

PlayerのStart関数でcharacterControllerの初期化を行っています。第一引数はキャラクターの半径。第二引数はキャラクターの高さです。第三引数はキャラクターの初期位置です。

|  |
| --- |
| void Player::Update()  {  //キャラクターの移動速度を決定。  CVector3 move = characterController.GetMoveSpeed();  move.x = -Pad(0).GetLStickXF() \* 5.0f;  move.z = -Pad(0).GetLStickYF() \* 5.0f;  //決定した移動速度をキャラクタコントローラーに設定。  characterController.SetMoveSpeed(move);  //キャラクターコントローラーを実行。  characterController.Execute();  //実行結果を受け取る。  position = characterController.GetPosition();  //ワールド行列の更新。  skinModel.Update(position, CQuaternion::Identity, CVector3::One);  } |

CharacterControllerに移動速度を設定して、Execute関数を実行するとCharacterControllerの座標が移動速度に応じて移動します。CharacterController::Execute関数は中でコリジョン検出と解決を行っています。そのため、本関数を読んだ後でCharacterControllerのGetPositionを使用すると移動した結果の座標を返してきます。これでキャラクターのコリジョン処理が実装できました。