**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。

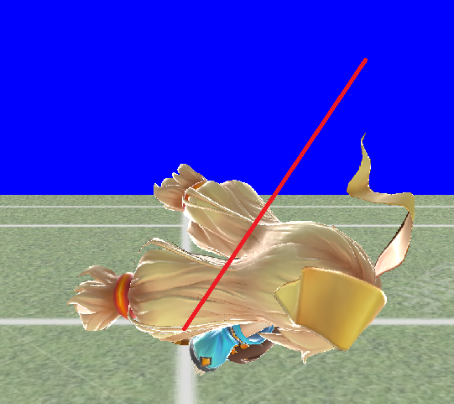
　　　　　　　　　　

Y軸周りに９０度回転

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)); |