ソースコードPR書

作成者：杉山　寛於

〈目次〉

<ResourceManager.h/ ResourceManager.cpp>

『注力した点』

リソース全般を管理することができるResourceManagerをライブラリとして作成しました。基本的にリソースはゲームを制作する上でなくてはならないものだと考え、すべてのリソースを一括で管理できるライブラリがあると便利だと考え開発しました。このマネージャーの特徴として、CSVファイルで情報を管理しています。又、テクスチャとモデルの判別、必要なシーン、呼び出すためのキーなどを管理することができます。これにより、各シーンの初めにそのシーンで必要なリソースのみを読み込んで、シーンの終わりに破棄する仕様にしました。

『詳細』

ヘッダーファイルの45行目にある通り、このライブラリはシングルトンで管理しており、インスタンスは絶対に一つしか生成できない仕組みになっています。

```
44      //インスタンスの取得
45      static ResourceManager* GetInstance()
46      {
47          static ResourceManager instance;
48          return &instance;
49      }
```

ヘッダーファイルの82，83行目の各種ゲッターでは、設定されたキーを引数に入れることにより返り値でテクスチャ、もしくはCMOモデルを返す仕組みになっています。

```
81      //リソースの取得
82      ID3D11ShaderResourceView* GetTexture(const std::string& key);
83      DirectX::Model* GetCmoModel(const std::string& key);
84
```

実装ファイル188,198行目ではString型で汎用的なパスを事前に用意しており、リソースをあらかじめ決められたファイルに入れておけば、ファイル拡張子を含むファイル名のみでリソースを設定できるようになっています。

```
183             //ファイル名から相対パスを算出
184             else if (j == static_cast<int>(eFileReading::FILE_NAME)) {
185
186                 //識別子がテクスチャなら
187                 if (identifier == "T") {
188                     string omission = "Resources/Textures/";
189                     filename = omission+tileBuf;
190                     //マルチバイト文字をワイド文字に変換
191                     wcs = new wchar_t[filename.length() + 1];
192                     mbstowcs(wcs, filename.c_str(), filename.length() + 1);
193                     texturename = wcs;
194
195                 }
196                 //識別子がモデルなら
197                 else if (identifier == "M") {
198                     string omission = "Resources/Models/";
199                     filename = omission + tileBuf;
200                     //マルチバイト文字をワイド文字に変換
201                     wcs = new wchar_t[filename.length() + 1];
202                     mbstowcs(wcs, filename.c_str(), filename.length() + 1);
203                     cmoname = wcs;
204                 }
205             }
```

　又、これは後に説明するObjectModelクラスとも相性がいいです。

現状は、管理しているのがCSVファイルであるため直接手入力しなければならない点は変わらず、ヒューマンエラーが発生してしまうこともあるので今後はビルド時にキーをまとめたヘッダーファイルを自動作成し、キー入力も簡単にしていきたいと思っています。

追記：専用のツールをC#にて開発しました。これにより列挙型を引数に設定することで直接入力せずにリソースを取得することが可能になりました。

<ObjectModel.h/ ObjectModel.cpp>

『注力した点』

　このクラスを持たせてモデル情報をセットすると簡単にモデル描画できるライブラリを作成しました。

```
49      //リソースマネージャの準備
50      auto pRM = ResourceManager::GetInstance();  //リソースマネージャーのインスタンス取得
51      auto model = pRM->GetCmoModel("SunDome");   //キー指定
52
53      //天球モデルの設定
54      mpSphere->SetModel(model);                  //モデルのセット
55
56      //描画関数でこれを呼ぶ
57      mpSphere->Draw();                           //描画関数
58
59  }
```

※あらかじめコンストラクタで拡大率や座標なども標準のものを設定しているため上の処理だけでモデルを描画することができます。

『詳細』

ヘッダーファイルの77行目のSetModel関数では主にResourceManagerとの併用を前提に作成した関数で、モデル情報をそのまま渡すことができる関数です。

```
75
76      //モデルの指定
77      void SetModel(DirectX::Model* model);
```

59，62行目の変数を独立させた意図ですが、プレイヤーのモデルなどの『モデル描画はしないけど、当たり判定は欲しい』などに用いるために作成しました。これにより、Spriteなど一つ一つに当たり判定を持たせる必要がないモデルには当たり判定をつけずにObjectModelクラスで管理し、実際の当たり判定はBoxModelクラスなどの当たり判定を持つ継承先で管理することができるため、より処理を軽くできました。

```
62      //表示フラグ:true    表示     false:非表示
63      bool mDrawFlag;
64
65      //モデル使用フラグ
66      bool mModelUseFlag;
```

<GridLineManager.h/ GridLineManager.cpp>

『注力した点』

ブロックの配置を行う際、地面に敷いてあるグリッド線の交点にしかブロックを置けないような処理を行ったのですが、毎フレーム線分の当たり判定を取るにはとても処理が重く無駄な処理と考えたため、ブロック側に磁石を模した円の当たり判定をもたせ、グリッド線側では交点座標を可変長配列に保存し、比較的処理の軽い円と点の衝突判定を用いることで処理を軽くしました。

又、この際に作成した配列は後にブロックの座標設定の際にも使用しています。

```
/*==============================================
 渡された座標から最も近い交点を算出する
 引数：基準となる座標
 返り値：引数に最も近い座標
 ==============================================*/
DirectX::SimpleMath::Vector2 GridLineManager::GetIntersectionNearPosition(const DirectX::SimpleMath::Vector2& basePosition)
{
    //返り値用変数の宣言
    DirectX::SimpleMath::Vector2 returnPosition = mIntersectionNum[0];  //最初はエラー回避のため初番を入れておく
    //算出用変数の宣言
    float calculationPosition1, calculationPosition2;

    //基点からのベクトルの長さを算出
    calculationPosition1 =
        std::sqrtf
        (
            (std::abs(mIntersectionNum[0].x - basePosition.x)) * SQUARE +
            (std::abs(mIntersectionNum[0].y - basePosition.y)) * SQUARE
        );

    //要素数保存用変数の宣言
    int saveindex = 0;

    //簡略化用変数の宣言
    int size = static_cast<int>(mIntersectionNum.size());


    //交点群から対象に最も近い交点の座標を保存する
    for (int i = 1; i < size; i++)
    {
        //算出
        calculationPosition2 =
            std::sqrtf(
                (std::abs(mIntersectionNum[i].x - basePosition.x)) * SQUARE +
                (std::abs(mIntersectionNum[i].y - basePosition.y)) * SQUARE
            );

        //もし現段階で最も近い交点よりも距離が近い場合は
        if (calculationPosition2 < calculationPosition1)
        {
            //更新する
            calculationPosition1 = calculationPosition2;
            saveindex = i;//要素数を保存する
        }
    }

    //最も近い交点の情報を返す
    return mIntersectionNum[saveindex];
}
```

```
/*================================
 縦線と横線の交差判定:private
 引数：なし
 返り値：なし
 ================================*/
void GridLineManager::ChackHitVerticalLineHorizontalLine()
{
    //当たり判定関数群の取得
    Collider& pC = Collider::GetColliderInstance();

    //サイズの省略
    int hSize = static_cast<int>(mpHorizontalGridLine.size());
    int vSize = static_cast<int>(mpVerticalGridLine.size());
    //保存用変数の宣言
    DirectX::SimpleMath::Vector2 savePosition = DirectX::SimpleMath::Vector2::Zero;


    //敵の数だけ処理を回す
    for (int i = 0; i < vSize; i++)
    {
        //敵の数だけ処理を回す
        for (int j = 0; j < hSize; j++)
        {
            //当たっていたら処理をする
            if (pC.Intersect(
                *mpHorizontalGridLine[j]->GetLineCollider(),
                *mpVerticalGridLine[i]->GetLineCollider(), savePosition))
            {
                mIntersectionNum.push_back(savePosition);
            }
        }
    }
}
```

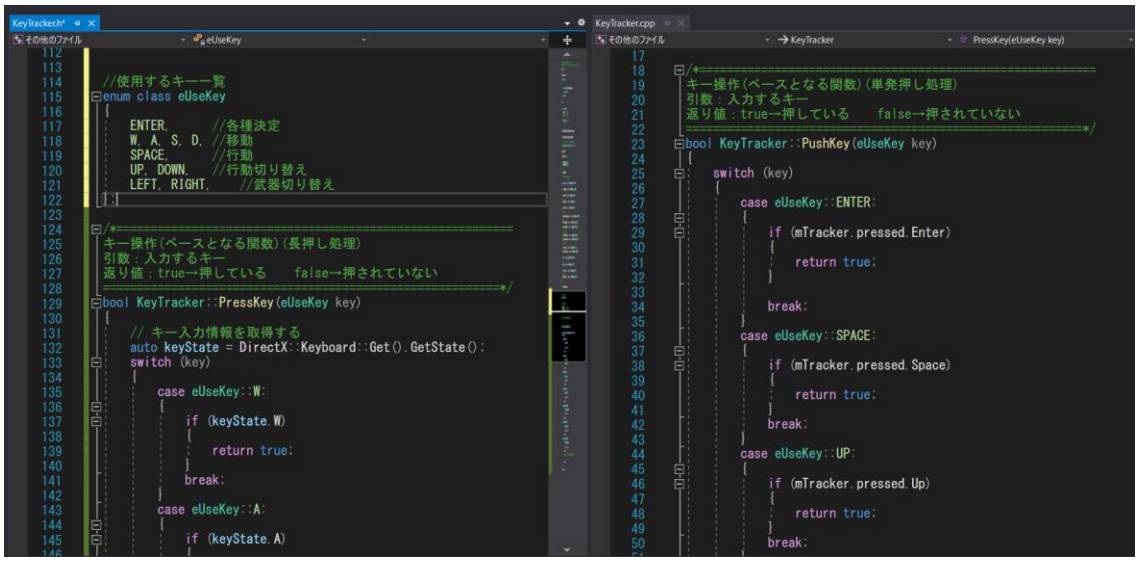
<KeyTracker.h/ KeyTracker.cpp>

『注力した点』

DirectXTK(DirectXToolKit)ではキーの単発押し処理とキーの長押し処理では呼び出し元が異なりそれぞれの関数を呼び出すと機能が複雑で扱いづらい面がありました。そこで新たに「ゲーム内のキー入力全てを管理するクラス」を作成しました。

『詳細』

デザインパターンの一種である「FacadePattern」を独学で習得し、用いました。始めに単発押し処理を管理する「KeyboardStateTracker」の変数と長押し処理を管理する「Keyboard」の変数をそれぞれ持たせ、”使用する機能のみを限定して”private な関数を作成します。



次に作成した関数をラップする形で”キーの用途別”で関数を作成します。

内容が同じで異なる名前の関数があるのはあえてそのような処理を行っています。

本来キー入力はゲーム内ではなるべく少ないほうがいいと思っています。よってキー入力情報そのものの追加は面倒な形にしました。

```
//更新処理
void KeyUpdate();

//使用用途の数だけ関数を作成する

//決定
bool Decision() { return this->PushKey(eUseKey::ENTER); }

//下へ
bool SelectUnder() { return this->PushKey(eUseKey::DOWN); }

//上へ
bool SelectOver() { return this->PushKey(eUseKey::UP); }

//歩行
bool WalkForward() { return this->PressKey(eUseKey::W); }
bool WalkBack() { return this->PressKey(eUseKey::S); }
bool WalkLeft() { return this->PressKey(eUseKey::A); }
bool WalkRight() { return this->PressKey(eUseKey::D); }

//強攻撃
bool PushActionStrongAttack() { return this->PushKey(eUseKey::SPACE); }
bool PressActionStrongAttack() { return this->PressKey(eUseKey::SPACE); }

//弱攻撃
bool ActionWeekAttack() { return this->PushKey(eUseKey::SPACE); }
//壁を建てる
bool ActionBlockCreate() { return this->PushKey(eUseKey::SPACE); }
```

さらにゲーム内部のプレイヤーなどはキーの細かい入力処理は不要だと考え、細かいキー設定はこの「KeyTrackerクラス」に統合してしまい、実際に使う処理毎に関数を作成することで修正を行いやすくしました。用途別に関数を作成したため、下の画像のような”今は同じ入力処理だけど後で変わるかもしれない”といった仕様変更にも柔軟に対応できます。

例として下の画像のように書いていた場合変更が大変になると思ったことから上記の仕様で作成しました。

```
void Update();
void Update2();


int main()
{
    //もしジャンプが押されたらジャンプする関数を呼び出す
    if (PushKey(eUsekey::ENTER))    Update();
}


void Update()
{
    /*ジャンプの処理を行う。*/

    //もしジャンプ中に再度押されたら2段ジャンプする
    if (PushKey(eUsekey::ENTER))
    {
        Update2();
    }
}

void Update2(){}
```

＜GroundObjectManager.h / GroundObjectManager.cpp＞

『注力した点』

当たり判定の作成に注力しました。

『詳細』

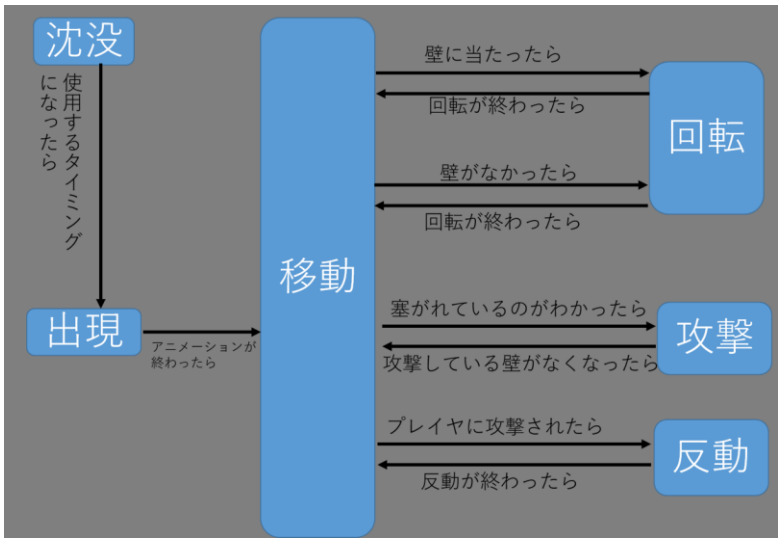
```
/*=============================================================
オブジェクトのサイズを測定する再帰関数:private
=============================================================*/
void GroundObjectManager::Search(int& chipX, int& chipY, int objectNum)
{
    //横隣が同じオブジェクトなら
    if (mGroundObjectMapChip[chipX + 1][chipY] == objectNum)
    {
        chipX++;
        this->Search(chipX, chipY, objectNum);

    }
    //縦を見る
    if(mGroundObjectMapChip[chipX][chipY + 1] == objectNum)
    {
        chipY++;
        this->Search(chipX, chipY, objectNum);
    }
}
```

```
/*=============================================================
CSVファイルからある程度のまとまりのオブジェクトを探し出す:private
引数：ファイルのパス
=============================================================*/
void GroundObjectManager::SeekLoadMapChipCohesive()
{
    mSaveInfo.clear();

    //一時保存用構造体を実体化
    SaveColliderInfo colliderInfo(eGroundObject::NONE, DirectX::SimpleMath::Vector2::Zero, DirectX::SimpleMath::Vector2::Zero);

    //Y軸の処理
    for (int y = 0; y < GOMANAGER_CSV_MAX_Y; y++)
    {
        //変換用変数
        int chipX = 0, chipY = 0;

        //X軸の処理
        for (int x = 0; x < GOMANAGER_CSV_MAX_X; x++)
        {
            if (mGroundObjectMapChip[x][y] == static_cast<int>(eGroundObject::NONE))continue;
            if (mGroundObjectMapChip[x][y] == static_cast<int>(eGroundObject::CONIFER_GROUP))continue;
            if (this->OverlapCheck(x, y))continue;  //重複チェック

            //オブジェクトがあったら再帰開始
            chipX = x;
            chipY = y;
            this->Search(chipX, chipY, mGroundObjectMapChip[x][y]);

            eGroundObject object(static_cast<eGroundObject>(mGroundObjectMapChip[x][y]));
            colliderInfo.mObjectType = object;


            colliderInfo.mMinPos = DirectX::SimpleMath::Vector2(static_cast<float>(x), static_cast<float>(y));
            colliderInfo.mMaxPos = DirectX::SimpleMath::Vector2(static_cast<float>(chipX), static_cast<float>(chipY));
            mSaveInfo.push_back(colliderInfo);
        }
    }
}
```

画像の関数を作成し、オブジェクト一つ一つに当たり判定を持たせるのではなく、ある程度まとまったオブジェクトを囲う形で当たり判定を作成しました。

＜敵のAI＞

Stateベースの敵AIを作成しました。



上記の図は仕組みを簡略化した図となります。青色の四角が各種Stateとなっています。

しかし、Managerに全て書いてしまい、EnemyManagerクラスの役割が現在多すぎる状態かつ、かなり限定した状態でのAIとなっており

(GroundObjectに上から当たった場合の想定をしていないなど)

課題も多いため、今後は一度設計を見直し、CommandPatternやビヘイビアをベースとしたAIに組みなおそうと考えています。

追記：改めて敵AIを組みなおし、行動を階層ごとに分けました。これによりManagerクラスの処理を削減することができました。

詳しくは[敵仕様.pptx]をご覧ください。