v1.5

後期課題

「クォータービューACTIONゲームの作成」

3D表現を活かす為に、クォータービューのゲームを作成します。C++言語で中規模のゲームを開発する事で、言語理解と全体的なプログラミングの手法について学ぶ事ができます。ゲーム開発の一つの例として手法を理解していきましょう。

■参考ゲーム：メルヘンメイズ

1988年にナムコ（現バンダイナムコアミューズメント）からリリースされた、高難易度のジャンプアクションゲームとしてアーケードゲームで当時話題になった作品です。

ルール

視点はクォータービュー（斜め上方向からの視点）となり、右上の方向を正面として進んでいきます。

「8方向移動」と「攻撃」「ジャンプ」の2つのボタン操作となります。

床は自力で落ちる事はないですが、敵からの攻撃や浮遊する床を踏み外して下に落ちるとミスとなります。

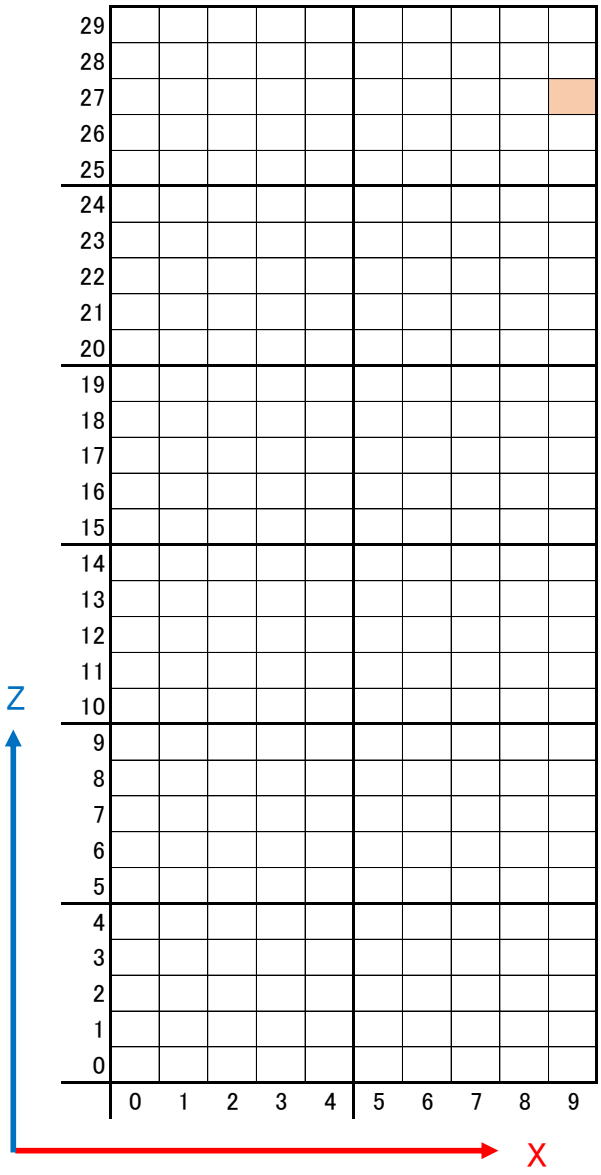
「ジャンプボタン」で浮遊する床や障害物を飛び越えながら進んでいく。

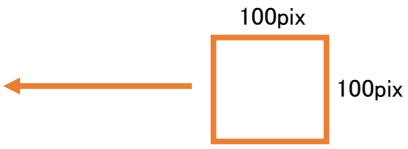
攻撃は「しゃぼん玉ボタン」を押すことにより吹き出されるしゃぼん玉で、これを敵に当てて倒したり、床下に落としたりする。

「しゃぼん玉ボタン」は押しっぱなしにして「巨大しゃぼん玉」を作り、敵をより遠くまで飛ばしたり倒したりすることができる。

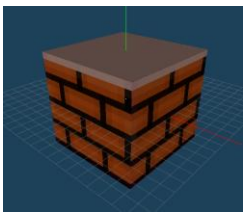
■フィールドの仕様

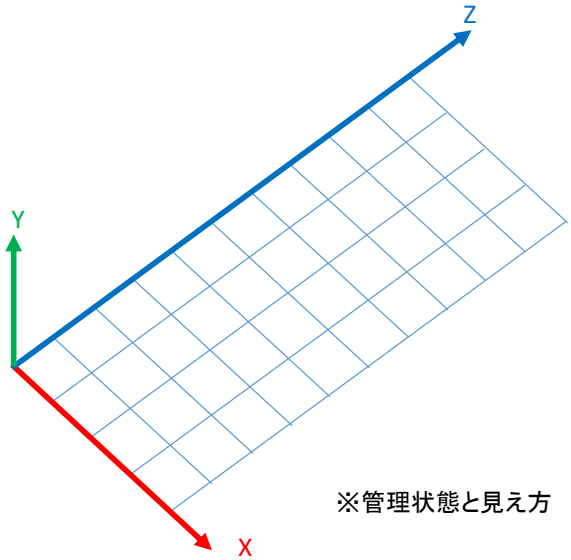
フィールドはとりあえず2次元配列で管理していきます。横10コマ×縦30コマぐらいにしておきます（サイズは自由です）
又、1コマの座標サイズは100×100ピクセルとしておきます。





※画面での見た目はナナメですが、実際の管理はX,Zの
真っ直ぐで行います。座標系の関係上、横をX（右にプラス）
縦をZ(奥にプラス）で管理していきます。
つまり、左下が原点（0,0）となります。

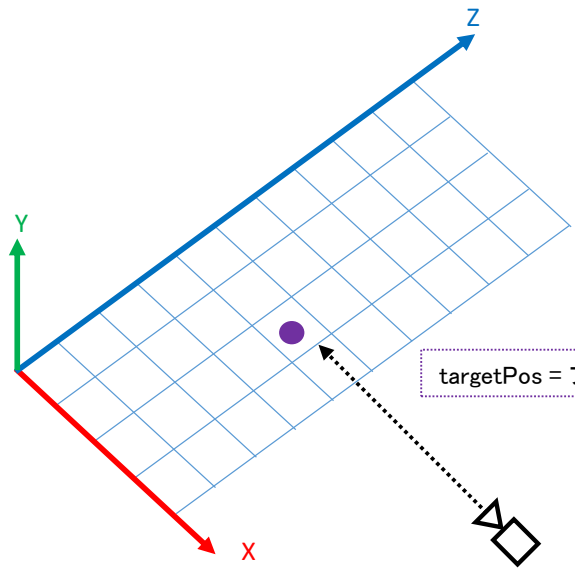




※管理状態と見え方

■カメラの仕様

クォータービューになるようにナナメ方向からの視点をセットします。

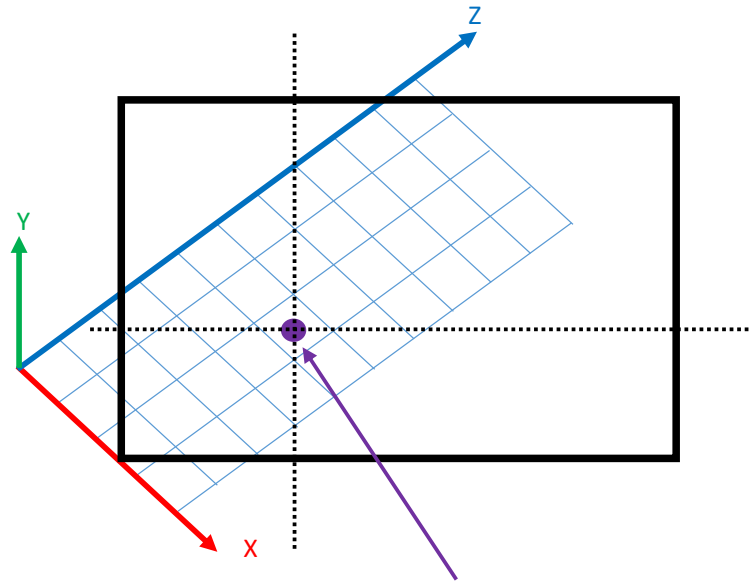



```
targetPos = プレイヤーの座標;
```

```
cameraPos.x = targetPos.x + 2000.0f;
cameraPos.y = targetPos.y + 2500.0f;
cameraPos.z = targetPos.z − 2000.0f;
```

1.カメラは常にプレイヤーを追従する様に設定します。つまり、カメラの注視点はプレイヤーという事になります。
2.カメラの座標は、ナナメ方向から見た位置に注視点からのオフセット量で設定します。

注視点をスクリーンのどの位置にするかの設定をしておきます。



正射影カメラをセットアップしておきます。
この設定はカメラから遠い物ものでも近いものでも同じ大きさに見えるようになり、今回の様なパースをかけないクォータービューの表現に向いています。

```
SetupCamera_Ortho(700);  // カメラを正射影に変更(700は下端から上端の間に表示される空間の範囲)
```

画面上でのカメラが見ている映像の中心座標を、画面左下方向の所に設定しておきます。
そうする事でプレイヤーを画面左下の方で固定して見る事が出来るようになります。

```
SetCameraScreenCenter(SCREEN_SIZE_X / 3, SCREEN_SIZE_Y / 3 * 2);// 消失点を画面左下にセット
```

■プレイヤーの仕様

プレイヤーキャラは、モーションデータ付きの3Dモデルを使用します。形式はfbx→mv1に変換したものが一般的となります。
「停止／歩き／攻撃／やられ」のモーションが必要になる為、素材として”東北ずん子”を使用します。
これらのモーションが準備できれば、各自で準備したモデルデータを使用しても問題はありません。

モデルデータを使用する場合の手順

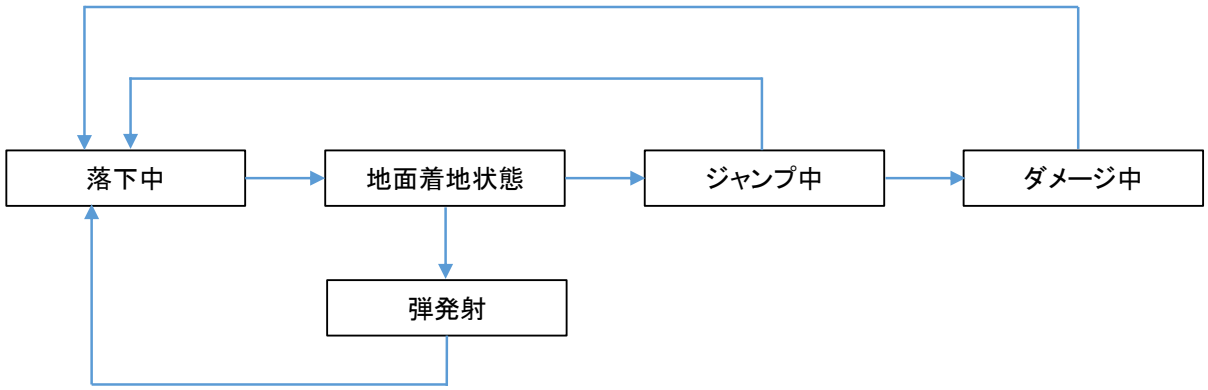
①3Dモデルを読み込む
②モデルの「拡大縮小」をセットする
③モデルの「回転」をセットする
④モデルの「移動（座標）」をセットする
⑤終了時に読み込んだモデルデータを削除する

モーション再生の手順

①今設定されているアニメーションを外す
②指定したanimNoのアニメを読み込む
③指定したアニメーションのトータルタイムを計る
④再生箇所を先頭にする
⑤指定したモデルデータの再生する所を指定する

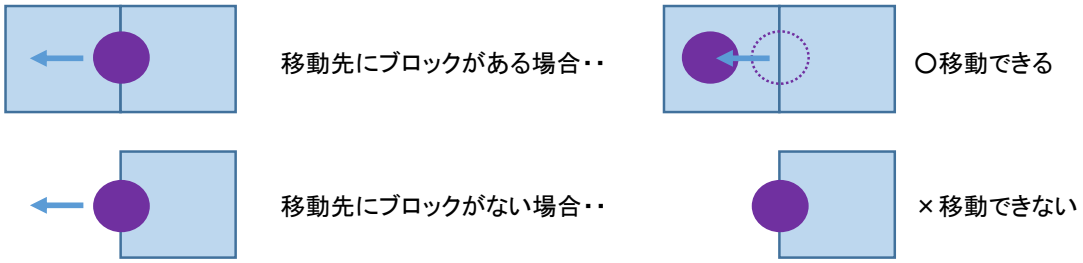
アクションについて

プレイヤーは、重力が働いており常に落下している状態で管理していきます。
ジャンプとショット発射は、地面に接地している時のみとし、それ以外では実行できない様にしておきます。



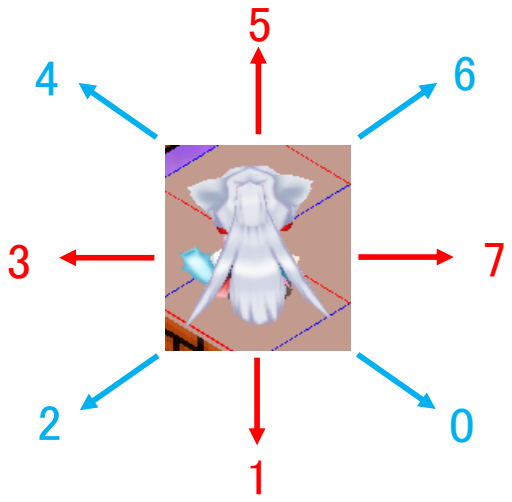
ブロックの判定について

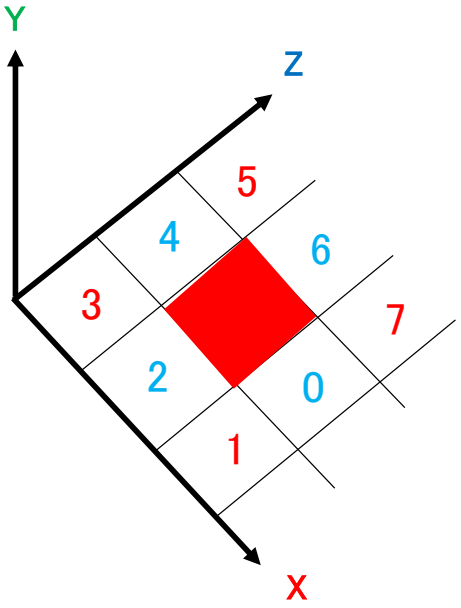
通常移動時で、ブロックの隅に来た時は、落下せずに踏みとどまる事ができます。



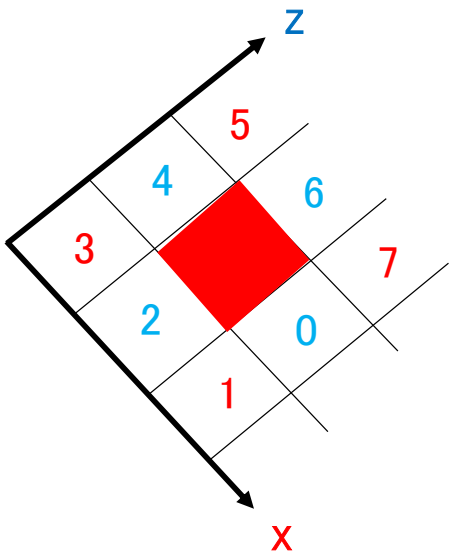
■キャラクターの向き

キャラクタは8方向に向いて移動ができます。弾の発射も8方向を基本としますので、どの方向かをint値で管理していきます。
方向としてはあくまでも画面に対して上下左右とナナメを設定しますので、0の方向は画面に向かって上方向になります。

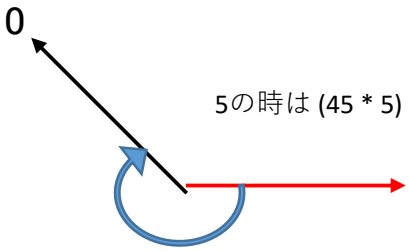




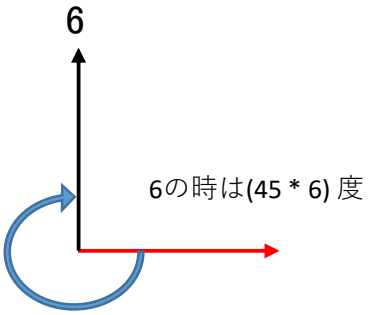
■向いた方向に弾を発射！



・5の時



・6の時



・方向(0～7)と移動量の式

```
mPos.x += cosf((DX_PI_F / 180) * (mDir * 45)) * 速度;
mPos.z -= sinf((DX_PI_F / 180) * (mDir * 45)) * 速度;
```

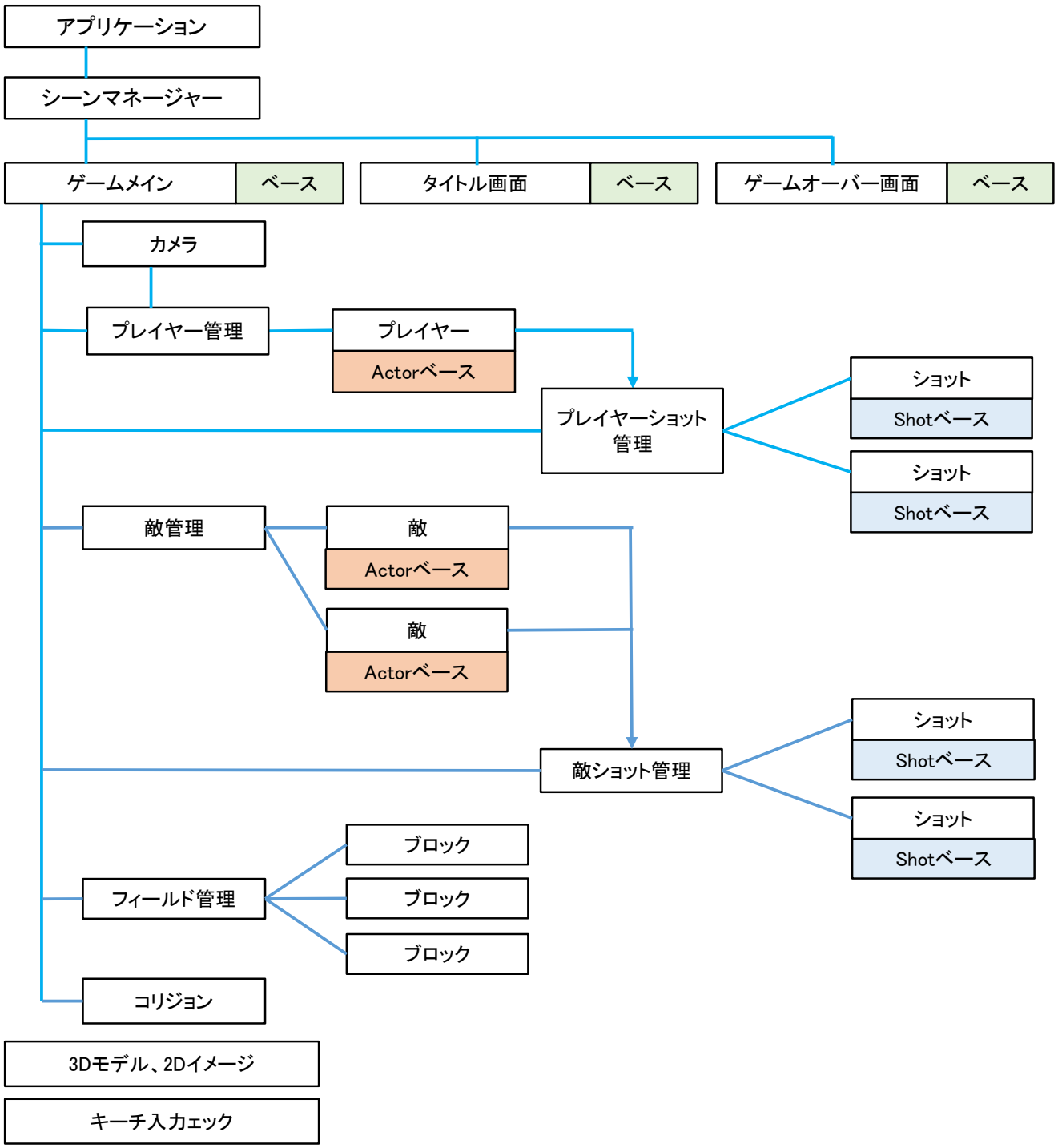
※Z軸は奥が+値なのでsin値をマイナスします。

■クラス構成

ゲームを構築する際に、最終的にどういう関係性になるのかの構成を考えておきます。
今回のゲームは、フィールド上にプレイヤーと敵がいて、それぞれが攻撃し合うというゲーム性と、3Dモデルの描画が主な内容となります。特にお互いのクラス情報のやり取りが発生するので、繋がりを意識しておく必要があります。

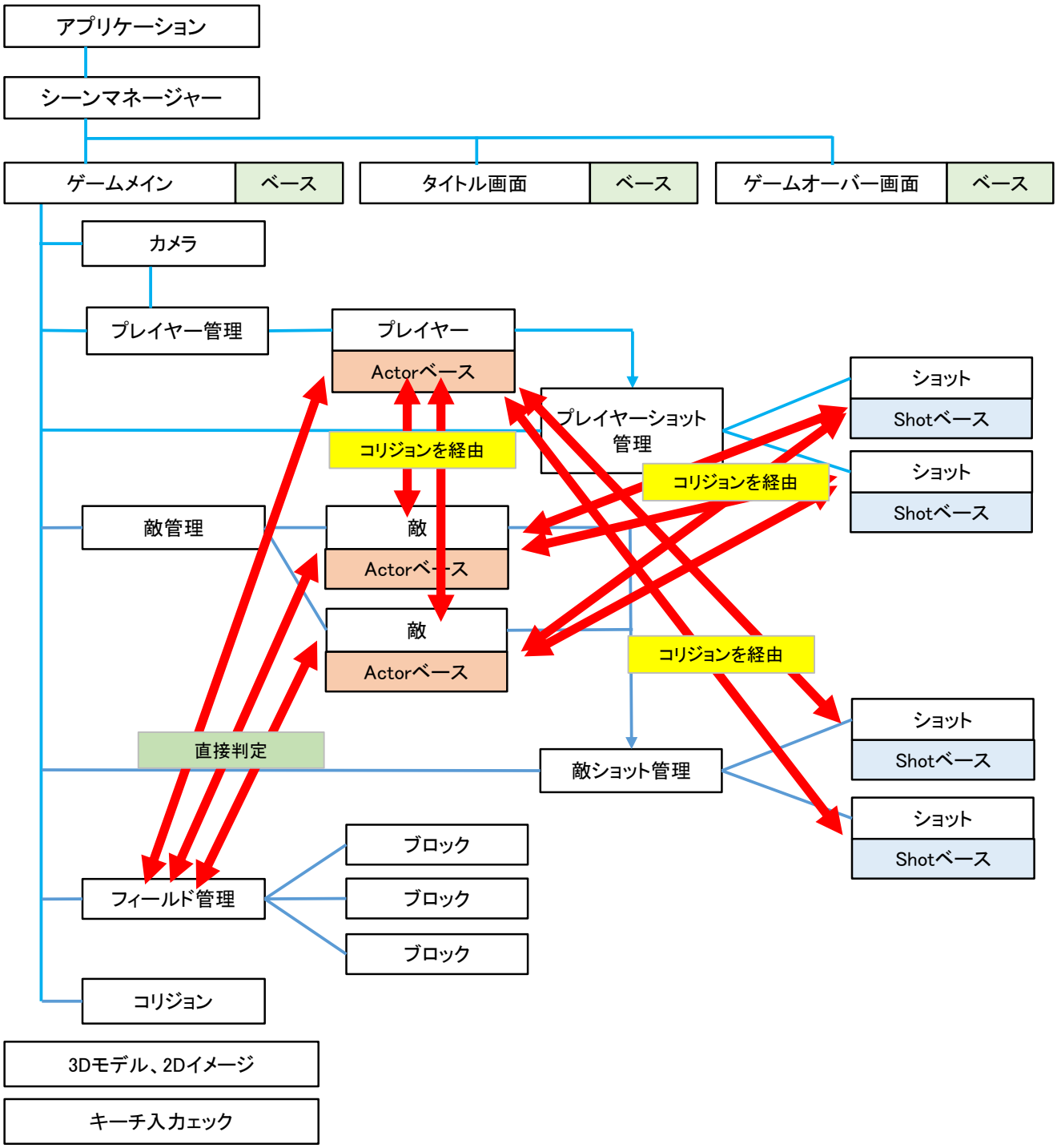
■基本のクラス構成

取りまとめを行うゲームシーンが、それぞれの管理クラスを生成して、役割りに生地て細部のクラスを管理していきます。
「プレイヤー」と「敵」、「プレイヤー弾」と「敵弾」は、共通の部分が多いのと、当たり判定の時の為に基底クラスを用いて共通の管理ができるようにしておきます（クラスの継承）。



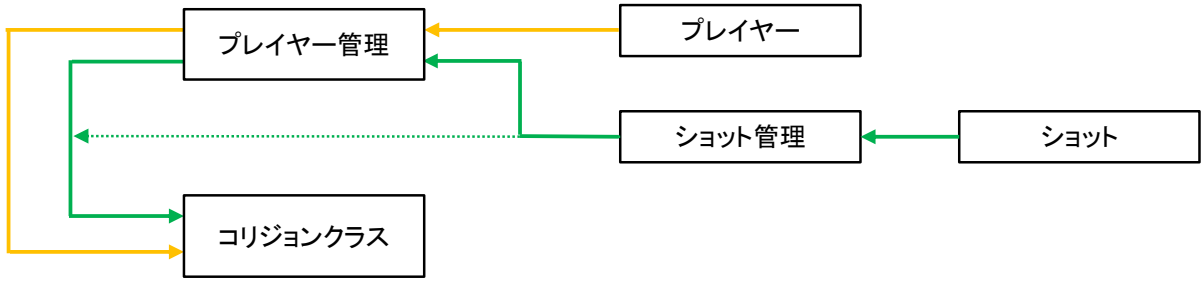
■コリジョンの検討

通常の管理状態では、素直にクラス設計していけば良いのですが、オブジェクト同士の当たり判定を考えた時に
管理上の前後関係が一気に破綻してしまいます。ゲーム全体が複雑になってくると、この当たり判定をどのように
していくかが設計の大きなカギとなります。面倒なので「全部グローバル変数で！」とはならないので注意が必要です。
又、当たり判定を一括で管理するコリジョンクラスを経由させる為に、情報を渡した後に判断結果を返す処理もあります。
かなり複雑な構成になりますが、判定処理そのものはコリジョンクラスに全部任せてしまいたい所です。

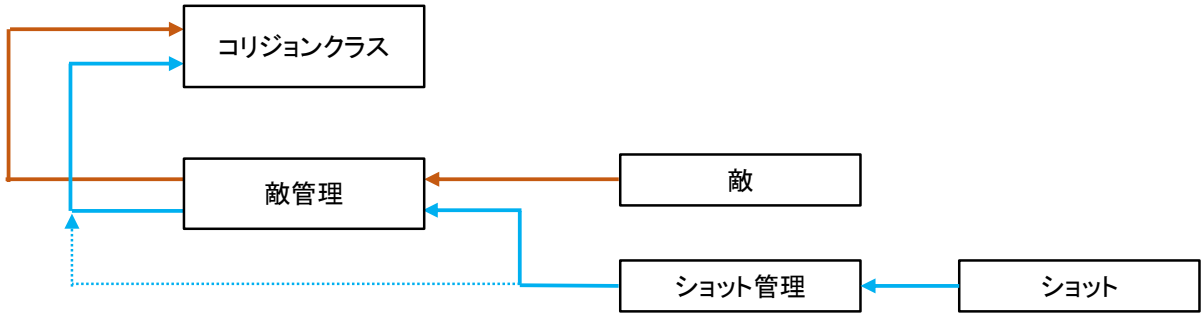


■「敵弾vsプレイヤー」「プレイヤー弾vs敵」の判定をコリジョンクラスで管理する為には、各オブジェクトの状態をコリジョンクラスが全部チェックする必要があります。
しかしながら、今回は敵や弾などは固定で数が確定していない為、その都度状況が変化しています。
当たり判定は「コリジョンクラス」からそれぞれのオブジェクトの判定をする必要がありますが、末端まで見に行くのは大変です。
その為、Updateのタイミングで生存しているオブジェクト自身が、コリジョンクラスに判定の意思表明をしていく様にします。
そうする事で、判定が必要なオブジェクトだけが抽出される事になり処理が容易になります。

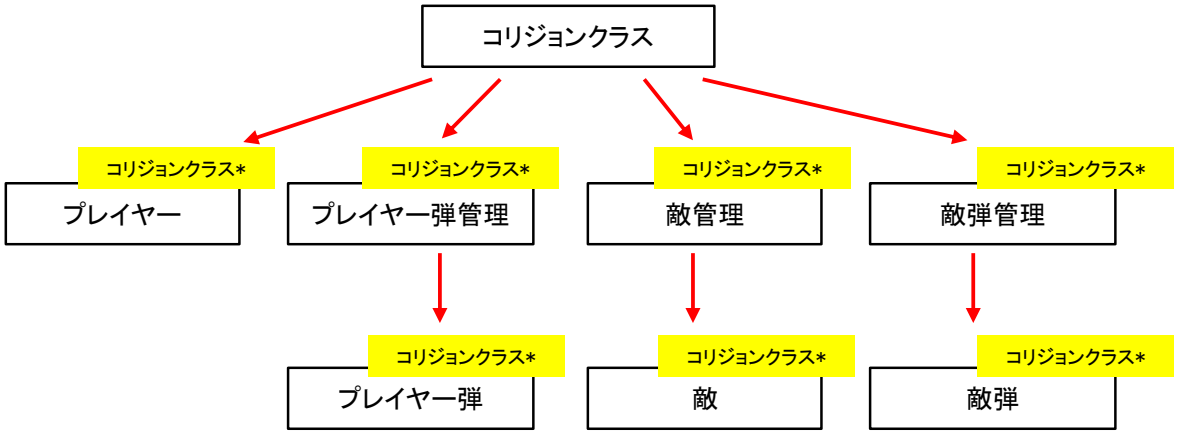
敵、敵弾との判定を依頼



プレイヤー、プレイヤー弾との判定を依頼



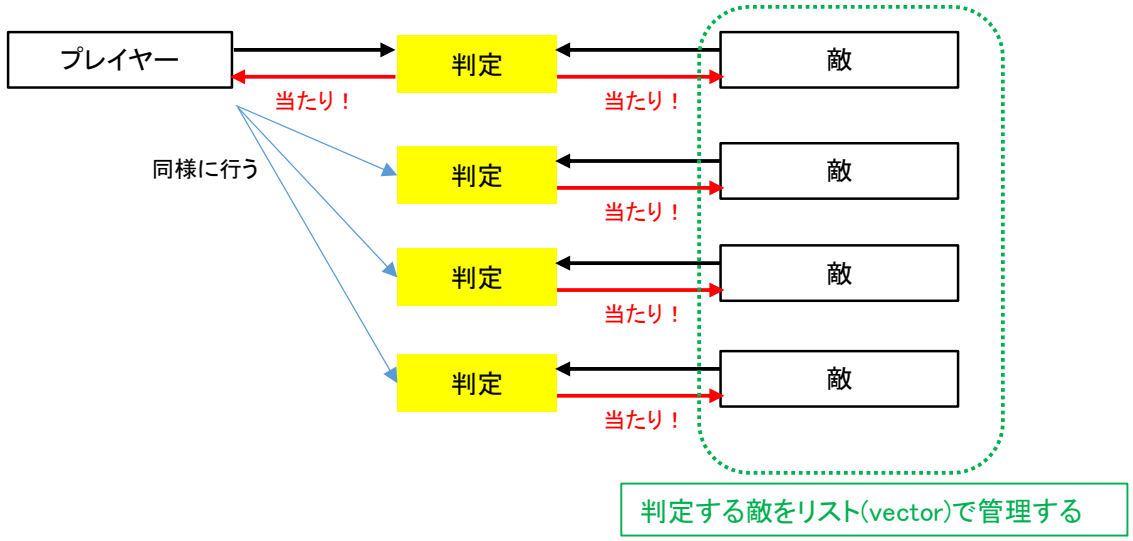
それぞれの末端のクラス「プレイヤー」「敵単体」「プレイヤー弾単体」「敵弾単体」からコリジョンクラスへのアクセスができるように、それぞれのオブジェクトがインスタンスされる時に、コリジョンクラスを渡して行く事にします。



■コリジョンクラスでの管理方法

コリジョンクラスでは、あくまでもオブジェクト同士の衝突判定を行い、衝突した事を返答するだけに特化していきます。
当たった後の処理については、それぞれのオブジェクトに任せてしまう訳です。
ただし、衝突時の必要な情報があれば、それらを渡す事までは行います（当たった場所、何に当たったか、当たった方向など）

コリジョンクラスでの判定方法



つまり、コリジョンクラスでそれぞれのオブジェクト毎のリスト(vector)を準備しておき、各オブジェクトがUpdateの度にリストにつなぐ事で、常に判定して欲しいオブジェクトの状態が出来上がる事になりますので、コリジョンクラスのUpdateでそれぞれのオブジェクト同士の衝突判定を行い結果を返す事ができる様になります。

```
#include <vector>

class SHOT_BASE;
class ACTOR_BASE;

class Collision
{
public:
    // 判定を行うオブジェクトの管理
    class ACTOR_BASE* mPlayer;

    std::vector<SHOT_BASE*> mHitPlayerShotList; // 判定用プレイヤーショットリスト

    std::vector<SHOT_BASE*> mHitEnemyShotList;  // 判定用敵ショットリスト

    std::vector<ACTOR_BASE*> mHitEnemyList; // 判定用敵リスト

}
```

オブジェクトをインスタンスする時に、ポインタを渡す方式では上手くいきません。特に、プレイヤーと敵は相関関係にあり、どちらを先に生成しても相互読み込みが発生してしまうからです。

例1)プレイヤーを先に生成した場合

```
player = new Player();
enemy = new Enemy(player);
```

※playerに敵は渡せない

例2)敵を先に生成した場合

```
enemy = new Enemy();
player = new Player(enemy);
```

※敵にプレイヤーは渡せない

じゃあ、親クラスを渡して全部見れるようにしよう！
playerとenemyのポインタを管理している親クラス「GameScene」そのものを全部のクラスで見れるようにすれば事実上、相互読み込みも解決できます。※全部公開する事になるので必要ないものも見えてしまう欠点もあります。

例1)GameSceneを全部に渡してそのポインタから全ての情報を得る

```
GameScene::GameScene()
{
	player = new Player(this);
	enemy = new Enemy(this);
}
```

```
Player::Player(GameScene* gScene)
{
	enemy.pos = gScene->enemy->pos;
}
```

```
Enemy::Enemy(GameScene* gScene)
{
	player.pos = gScene->player->pos;
}
```

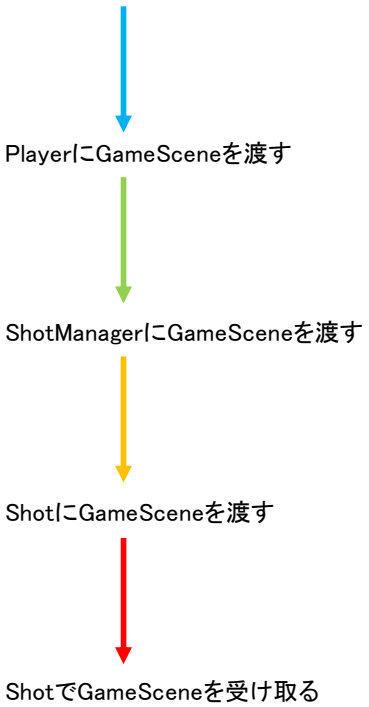
例2)その場合、末端までどんどん渡していく必要があります。

```
GameScene::GameScene()
{
	player = new Player(this);
}
```

```
Player::Player(GameScene* gScene)
{
	shotManager = new ShotManager(gScene);
}
```

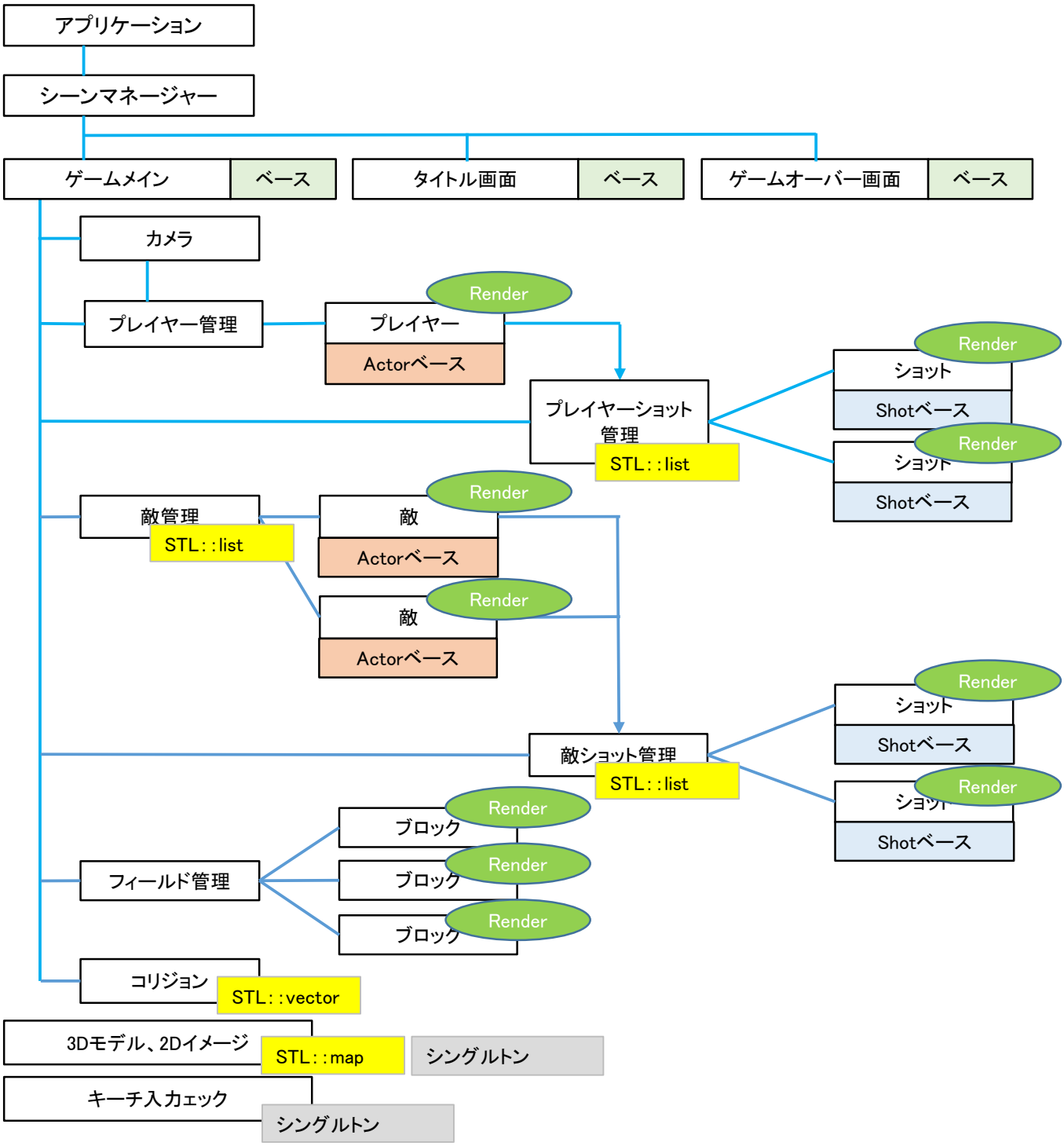
```
ShotManager::ShotManager(GameScene* gScene)
{
	shot = new Shot(gScene);
}
```

```
Shot::Shot(GameScene* gScene)
{
	this->gScene = gScene;
}
```



■もう一つの問題（画像関係）

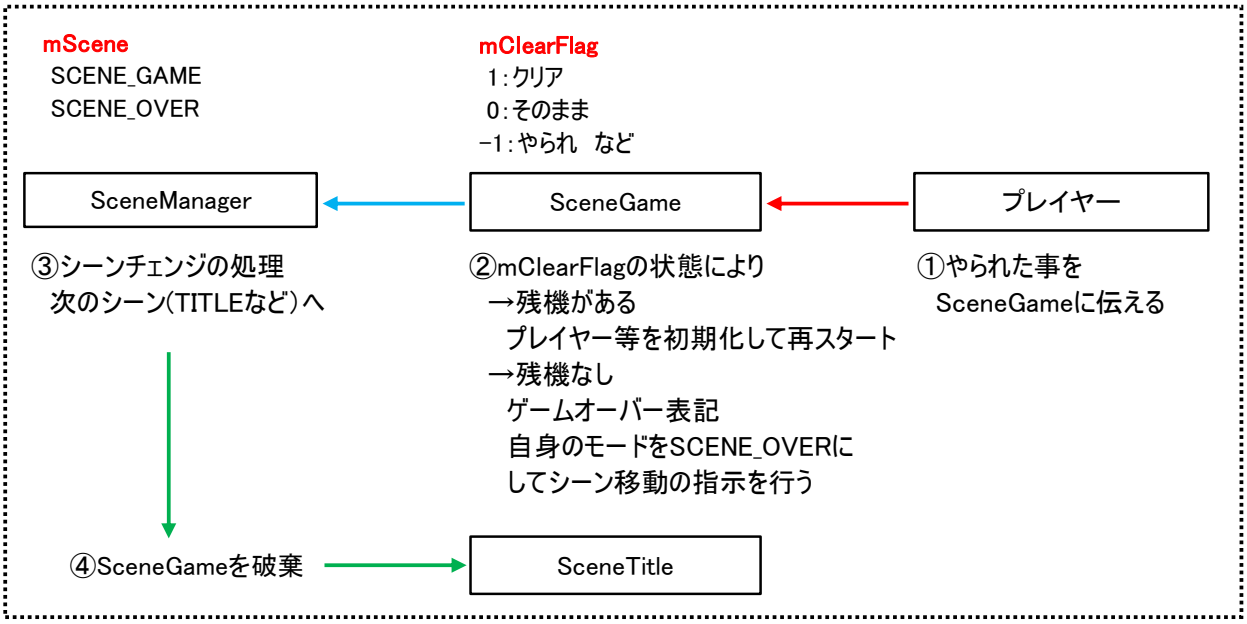
一つのクラスに一つの画像（3Dモデル）では気にしなくて良いのですが、複数の敵キャラクタやショットなどのモデルが出て来る場合は、その都度画像を読み込む事は無駄になりますので、どこかで一括管理をした方が適切です。
そうなると、それもGameSceneに置いてポインターを渡していく方法が有効になります。 ※とりあえずできます状態
しかしながら、このやり方だと画像の変数はpublicになるので将来的にはprivateにしたい所です。ゲット関数を使ってIDを渡す等の処理を加える事で何とか形になりそうです。GameSceneが画像の変数で溢れてしまうのも欠点です。
それならGameSceneに画像管理用のクラスを作って対応すれば良いのですが・・
この問題は、実際にどう設計すべきかを色々を検討していく部分になるので自分なりに考えていきましょう。



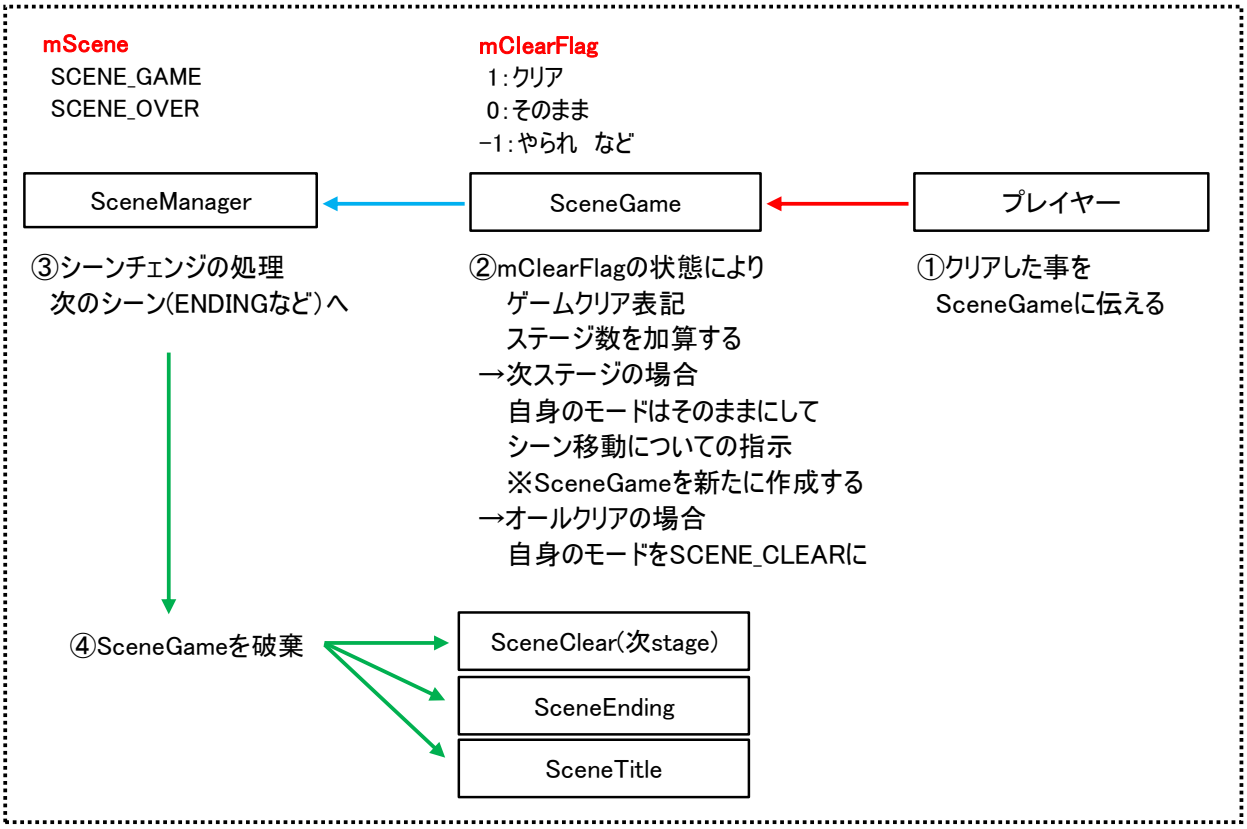
■ゲームクリアとゲームエンド

ゲームエンド・ゲームクリアの判定は、各オブジェクトがシーン管理に状況を伝達する形で行います。
各ゲームシーン(SceneGameなど)は、その情報を受け取ってそれぞれの処理を行いシーンチェンジの準備をします。

やられた時の遷移図



クリアした時の遷移図



■コーディング規約

プログラミングをする際にしっかり「決まり事」を設定して取り組む様にしてください。会社毎で決まっているやり方もありますが、それまでも自分なりの統一したルールーでコーディングしましょう。
今回の課題では、下記の様に統一していきます。

□命名規約

【クラス名】 先頭大文字、それ以降の単語区切りも大文字 （PascalCase）
【ベースクラス名】 すべて大文字，単語区切りをアンダースコア _ でつなぐ （SNAKE_CASE）
【(ローカル)変数名】 先頭小文字，それ以降の単語区切りは大文字 （camelCase）
【メンバ関数名】 先頭大文字、それ以降の単語区切りも大文字 （PascalCase）
【メンバ変数名】 先頭にmを付けて、それ以降は変数名と同様 （mCamelCase） ※先頭に"_"を付ける場合もあります
【定数名】 すべて大文字，単語区切りをアンダースコア _ でつなぐ（SNAKE_CASE）
【マクロ、列挙体】すべて大文字，単語区切りをアンダースコア _ でつなぐ （SNAKE_CASE）
【その他】 宣言した直後にその変数を使用し、以降使用しない場合の変数名は1文字でも良い。（Player* p;）

スネークケース（単語を_でつなぐ） 例：Player_Shot
キャメルケース（単語を大文字でつなぐ）例：PlayerShot

□if文

{}の省略は禁止。{の記述は行末にするか改行するかは決めておく。 ※for文も基本的に{}が必要です。

○
```
if(condition){

}else{


}
```

○
```
if(condition)
{
}else{


}
```

×
```
if(cpndition)
    return 0;
else
    return 1;
```

if 分の内容が簡潔であれば、1行if文の{}は省略可

○
```
if(condition) return0;
```

□インデント

インデントは半角スペース4つ分のTABを入れておく

□ヘッダーファイル（クラス定義）

ヘッダーファイル内でヘッダーファイルを極力インクルードしない。 ※基底クラスなどで必要な場合もあります。

□クラスの前方宣言

クラスが定義される前にそのクラス名を利用するためのもので、ヘッダーでの別のヘッダのインクルードを減らす目的があります。例2の様な記述も可能です。どちらかに統一して使用しましょう。

```
#include "B.h"
class A
{
    B* b;
};
```

例1
```
class B;
class A
{
    B* b;
};
```

例2
```
class A
{
    class B* b;
}
```