３Ｄモデルの衝突判定

弾が思った方向に発射できるようになりましたので、
弾が３Ｄモデルのステージと衝突したら、爆発するように実装していきたいと思います。

しかし、砲身の角度が上向きだと、弾がステージと衝突しませんので、
弾に重力をかけて、ステージと衝突するようにしていきましょう。

１年生のロックマン課題の時にもやりましたが、
重力は、物体に及ぼす加速度です。
地球の場合は、物体の速度が毎秒9.8l m/sずつ増加するようですので、
ゲーム全体の定数として、SceneManagerに定義しておきます。

SceneManager.h

```
class SceneManager
{
public:

    static constexpr float DEFAULT_FPS = 60.0f;

    // 重力
    static constexpr float GRAVITY = 9.8lf;
```

弾に重力をかける(３Ｄワールドの下方向に力を加える)ためには、
ShotBaseの移動処理に手を加える必要があります。
ShotBaseに加速度をため込むメンバ変数を追加しましょう。

ShotBase.h

```
private:

    ～ 省略 ～

    // 重力
    float gravityPow_;
```

ShotBase.cpp

```
void ShotBase::CreateShot(VECTOR pos, VECTOR dir)
{

    ～ 省略 ～

    // 重力
    gravityPow_ = 0.0f;
}

void ShotBase::Update(void)
{

    ～ 省略 ～

    pos_ = VAdd(pos_, VScale(dir_, speed_));

    // 更に加速度的に重力を加える
    gravityPow_ += SceneManager::GRAVITY / SceneManager::DEFAULT_FPS;
    pos_.y -= gravityPow_;

    ～ 省略 ～

}
```

毎秒9.81 mとのことなので、60FPS(1秒間に60フレーム)に換算すると、
9.81 ÷ 60 になりますので、以前に作成したFPS定数で割り算しましょう。

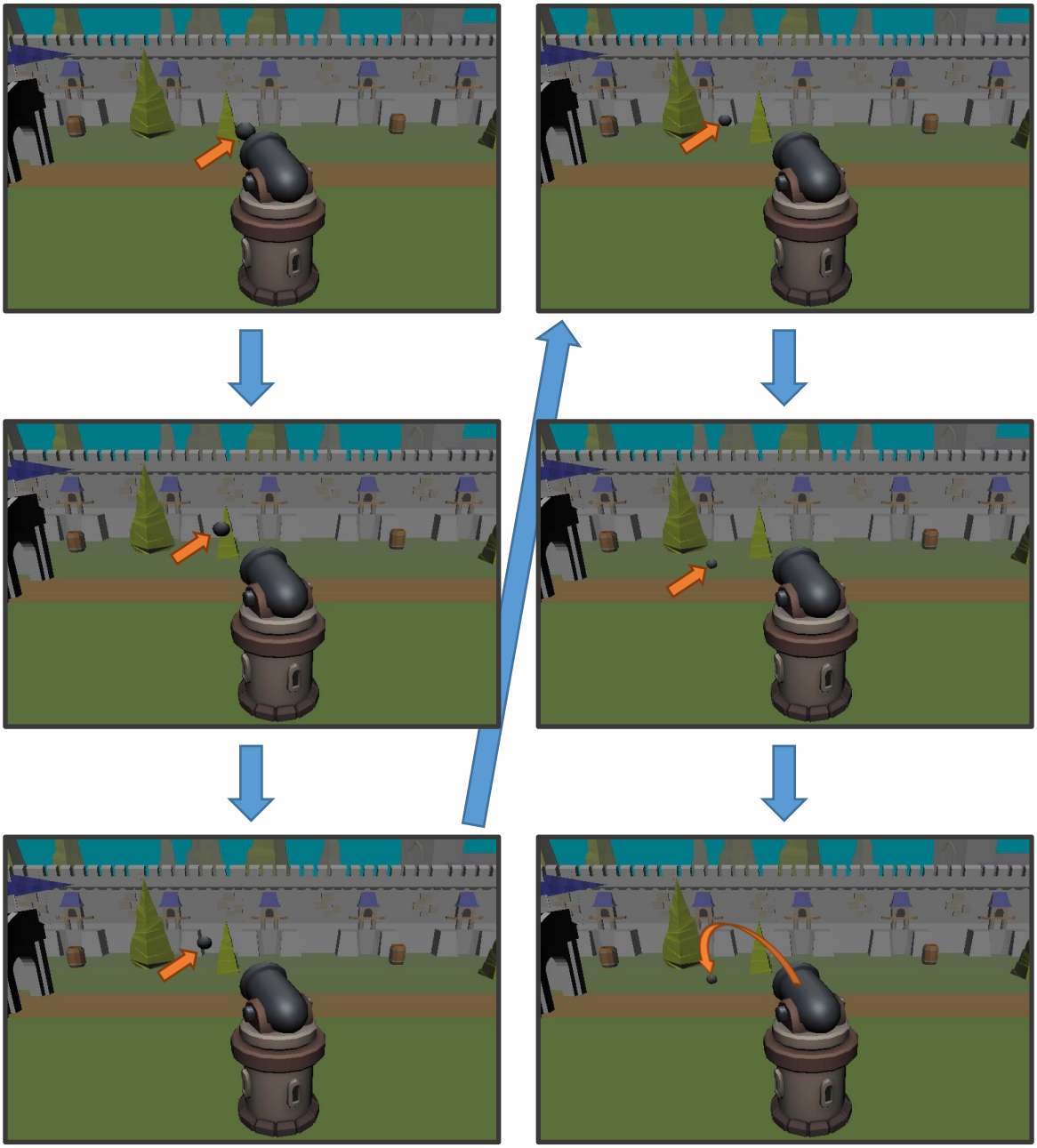
下方向は、Yの負の方向になりますので、Yの値に引き算を行っておりますが、
これも正確に式を書くと、移動量 = 方向 × スピード になりますので、

```
    pos_ = VAdd(pos_, VScale({ 0.0f, -1.0f, 0.0f }, gravityPow_));
```

となります。

これで、重力の実装はおしまいです。

下図のように、放物線を描く軌道で弾が移動するようになっていると思います。



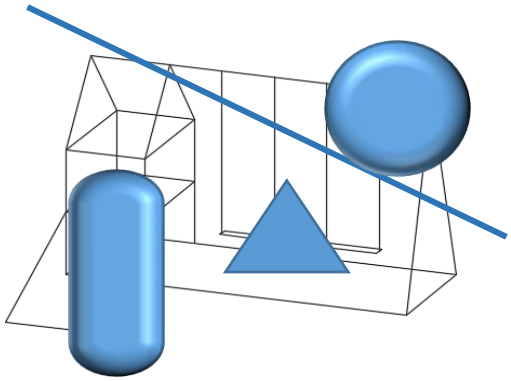
いよいよ３Ｄモデルとの衝突判定です。
DxLibには、代表的な３Ｄモデルとの当たり判定がいくつかありますので、
まずは、そちらを紹介します。

```
// 線とモデルの当たり判定
MV1CollCheck_Line

// 球とモデルの当たり判定
MV1CollCheck_Sphere

// カプセルとモデルの当たり判定
MV1CollCheck_Capsule

// 三角形とモデルの当たり判定
MV1CollCheck_Triangle
```



それではこれらの関数の使い方を解説します。

手順①　衝突専用の情報を構築する

衝突処理は、処理負荷の高い処理になります。
処理負荷を下げるために、ポリゴンの頂点情報を精査し、
衝突用に最適化したデータを構築します。

```
void Stage::Init(void)
{

    ～ 省略 ～

    // 衝突判定情報(コライダ)の作成
    MV1SetupCollInfo(modelId_);          引数は衝突情報を作りたい、
                                         モデルのハンドルＩＤ。
    // 背景画像読み込み
    imgBack_ = LoadGraph((Application::PATH_IMAGE + "Sky.jpg").c_str());

}
```

ここで作成された衝突情報は、モデル情報などと同じように解放しないと、
メモリに滞在し続けてしまいますので、

```
int MVITerminateCollInfo( int MHandle, int FrameIndex ) ;
```

上記の関数でメモリから解放する必要があるのですが、
既にRelease関数で実行されている、MVIDeleteModel関数内で、
衝突情報も一緒にメモリから解放されますので、省略します。

```
void Stage::Release(void)
{
    // 画像の解放
    DeleteGraph(imgBack_);

    // ロードされた３Ｄモデルをメモリから解放
    MVIDeleteModel(modelId_);
}
```

手順②　衝突処理を行う

大砲の弾は、ほぼ球体ですので、MVICollCheck_Sphereを使用します。

```
void GameScene::Update(void)
{

    stage_->Update();
    cannon_->Update();

    // ステージモデルID
    int stageModelId = stage_->GetModelId();

    auto shots = cannon_->GetShots();
    for (auto shot : shots)
    {
        shot->Update();

        // ステージモデルとの衝突判定
        auto info = MVICollCheck_Sphere(
            stageModelId, -1, shot->GetPos(), ShotBase::COL_RADIUS);
```

モデルのハンドルＩＤのゲッター関数を作成する

```
        if (info.HitNum > 0)
        {
            shot->Blast();
        }

        // 当たり判定結果ポリゴン配列の後始末をする
        MVICollResultPolyDimTerminate(info);

    }

}
```

```
MVI_COLL_RESULT_POLY_DIM MVICollCheck_Sphere(
    int MHandle, int FrameIndex, VECTOR CenterPos, float r);
```

第１引数　　：　衝突判定を行いたいモデルのハンドルＩＤ
第２引数　　：　フレーム番号
　　フレームとは、モデル内で分けれらた
　　グループのようなもの。
第３引数　　：　衝突判定を行いたい球体の中心座標
第４引数　　：　衝突判定を行いたい球体の半径

この関数の返り値の型である、MVI_COLL_RESULT_POLY_DIM構造体は、
DxLibで以下のように定義されており、

```
struct MVI_COLL_RESULT_POLY_DIM
{
    // ヒットしたポリゴンの数
    int                             HitNum ;
    // ヒットしたポリゴンの配列( HitNum個分存在する )
    MVI_COLL_RESULT_POLY *          Dim ;
}
```

衝突したかどうかの判定だけであれば、以下の条件式のみで判別できます。

```
    if (info.HitNum > 0)
    {
        // モデルと衝突している
    }
```

衝突したポリゴン(複数の場合有り)の詳細情報が知りたければ、
構造体の中にある、１つ１つのポリゴン情報の構造体から
情報を取得します。

```
struct MVI_COLL_RESULT_POLY
{
    //（ MVICollCheck_Line でのみ有効 )ヒットフラグ
    //（ 1:ヒットした  0:ヒットしなかった ）
    int           HitFlag ;
    //（ MVICollCheck_Line でのみ有効 )ヒット座標
    VECTOR        HitPosition ;

    // 当たったポリゴンが含まれるフレームの番号
    int           FrameIndex ;
    // 当たったポリゴンが含まれるメッシュの番号
    //（ メッシュ単位で判定した場合のみ有効 ）
    int           MeshIndex ;
    // 当たったポリゴンの番号
    int           PolygonIndex ;
    // 当たったポリゴンが使用しているマテリアルの番号
    int           MaterialIndex ;
    // 当たったポリゴンを形成する三点の座標
    VECTOR        Position[ 3 ] ;
    // 当たったポリゴンの法線
    VECTOR        Normal ;
    // 当たった座標は、当たったポリゴンの三点の割合影響
    float         PositionWeight[ 3 ] ;
    // 当たったポリゴンの座標がそれぞれ最も影響を受けているフレームの番号
    int           PosMaxWeightFrameIndex[ 3 ] ;
}
```

ここで重要なのが、関数を使用して取得した衝突したポリゴン情報は、
不要になったら、メモリから削除する必要があり、

```
    MVICollResultPolyDimTerminate(info);
```

を忘れないようにしてください。簡単にメモリ不足になります。

なぜかというと、衝突したポリゴンが複数存在するため、
複数のポリゴン情報を構造体から得ることができるのですが、
何個のポリゴン情報を取得できるのかわからず、配列が固定にできない
ためです。動的に配列の数を拡張しているため、
明示的にメモリから解放するという手続きが必要になります。

ちなみに、線との当たり判定、MV1CollCheck_Lineは、
衝突したポリゴン情報を１つしか取得しないので、
このメモリの解放は不要となります。

DxLibには、ゲームエンジンのように何でもかんでも簡単に実装できるようには
なっておりませんが、ほどよく便利な機能が実装されております。

しかし、どうしても自分の力で衝突判定を実装したい、
数学的に理解しないと納得できない、という方がいらっしゃいましたら、
参考までに【衝突判定３Ｄ編】の資料を用意しておりますので、
そちらをご参照ください。

- 衝突判定３Ｄ_DxLibのポリゴン情報
- 衝突判定３Ｄ_三角形と球体
- 衝突判定３Ｄ_高速化ＡＡＢＢ

数学が得意な方は別ですが、チャレンジされるとしたら、
３年生以上になってからが良いかと思います。