弾の作成

次は、砲身から弾を発射できるようにしていきます。
弾といえば継承です。
今回は、プレイヤーの弾を作成していきますが、
今後、敵の弾を作るかもしれませんし、弾を複数種作るかもしれません。
機能的には似通った内容が多いですので、最初から継承元の基幹クラスを
作ってしまいましょう。

【弾の機能】

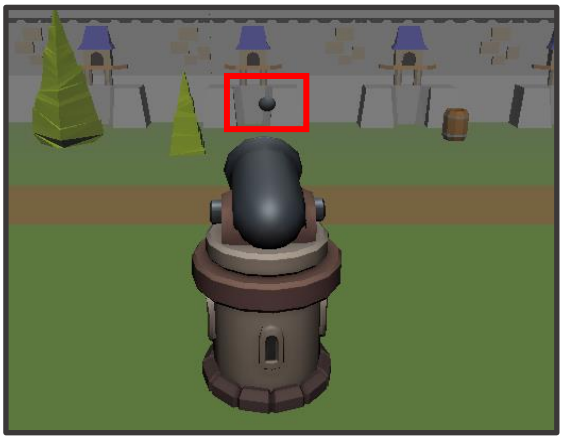
- Zキーで弾が発射されること
- 弾の描画には、３Ｄモデル("Cannon/Shot.mv1")を使用すること
- 複数発打てること
- 次に弾が打てる迄の硬直時間を設けること

⇒ ほとんどロックマンの弾機能と同じです。
作れそうな人は自分で作りましょう。※弾の大きさは0.8倍くらいで

【弾機能の設計】

いつも悩ましいですが、ロックマンをベースに設計します。

- 弾の実体は、Cannonクラスが持つ(撃つ者が持つ)
- 弾の更新や描画は、GameSceneで行う
  (後の衝突判定がやりやすいため)
- 使用しなくなった弾の実体は放置しておき、次の弾の実体に使いまわす
  (実体生成を少なるして、負荷を下げる)



【仮組みの機能】

- 弾の発射方向は、
  一旦、Zの正方向で良い
- 生存判定も後回し
- 衝突判定も後回し

黒い弾が真っ直ぐ出ればよい

ShotBase.h

```
#pragma once
#include <DxLib.h>

class ShotBase
{

public:

    // コンストラクタ(元となるモデルのハンドルID)
    ShotBase(int baseModelId);

    // デストラクタ
    virtual ~ShotBase(void);

    // 弾の生成(表示開始座標、弾の進行方向)
    void CreateShot(VECTOR pos, VECTOR dir);

    // 更新ステップ
    void Update(void);

    // 描画
    void Draw();

    // 解放処理
    void Release(void);

    // 生存判定
    bool IsAlive(void);

private:

    // 元となる弾のモデルID
    int baseModelId_;

    // 弾のモデルID
    int modelId_;

    // 方向
```

```
    VECTOR dir_;

    // 弾の大きさ
    VECTOR scl_;

    // 弾の角度
    VECTOR rot_;

    // 弾の座標
    VECTOR pos_;

    // 弾の移動速度
    float speed_;

    // 弾の生存判定
    bool isAlive_;

};
```

Cannon.h

```
#pragma once
#include <vector>
#include <DxLib.h>
class ShotBase;

class Cannon
{

public:

    ～ 省略 ～

    // 弾発射後の硬直時間
    static constexpr float SHOT_DELAY = 1.0f;

    ～ 省略 ～

    // 弾の取得
    std::vector<ShotBase*> GetShots(void);
```

```
private:

    ～ 省略 ～

    // ショット(ポインタ)
    std::vector<ShotBase*> shots_;

    // 弾のモデルID
    int shotModelId_;

    // 弾発射後の硬直時間計算用
    float stepShotDelay_;

    // 回転操作
    void ProcessRot(void);

    // 発射操作
    void ProcessShot(void);

    // 有効な弾を取得する
    ShotBase* GetValidShot(void);

};
```

Cannon.cpp

```
void Cannon::Init(void)
{

    ～ 省略 ～

    // 弾のモデル
    shotModelId_ =
        MVlLoadModel((Application::PATH_MODEL + "Cannon/Shot.mvl").c_str());
    // 弾発射の硬直時間
    stepShotDelay_ = 0.0f;

    // 初期設定をモデルに反映(最初は実装しない)
    Update();

}
```

```
void Cannon::Release(void)
{

    MV1DeleteModel(standModelId_);
    MV1DeleteModel(barrelModelId_);
    MV1DeleteModel(shotModelId_);

    for (auto shot : shots_)
    {
        shot->Release();
        delete shot;
    }

}

void Cannon::ProcessShot(void)
{

    auto& ins = InputManager::GetInstance();

    // 攻撃キーを押すと、弾を生成
    if (ins.IsNew(KEY_INPUT_Z) && stepShotDelay_ <= 0.0f)
    {

        // 有効な弾を取得する
        ShotBase* shot = GetValidShot();

        // 弾を生成(方向は仮で正面方向)
        shot->CreateShot(barrelPos_, { 0.0f, 0.0f, 1.0f });

        // 弾発射後の硬直時間セット
        stepShotDelay_ = SHOT_DELAY;

    }

    // 弾発射後の硬直時間を減らしていく
    if (stepShotDelay_ > 0.0f) {
        stepShotDelay_ -= 1.0f / SceneManager::DEFAULT_FPS;
    }
```

```
}

ShotBase* Cannon::GetValidShot(void)
{

    size_t size = shots_.size();
    for (int i = 0; i < size; i++)
    {
        if (!shots_[i]->IsAlive())
        {
            return shots_[i];
        }
    }

    ShotBase* shot = new ShotBase(shotModelId_);
    shots_.push_back(shot);

    return shot;

}
```

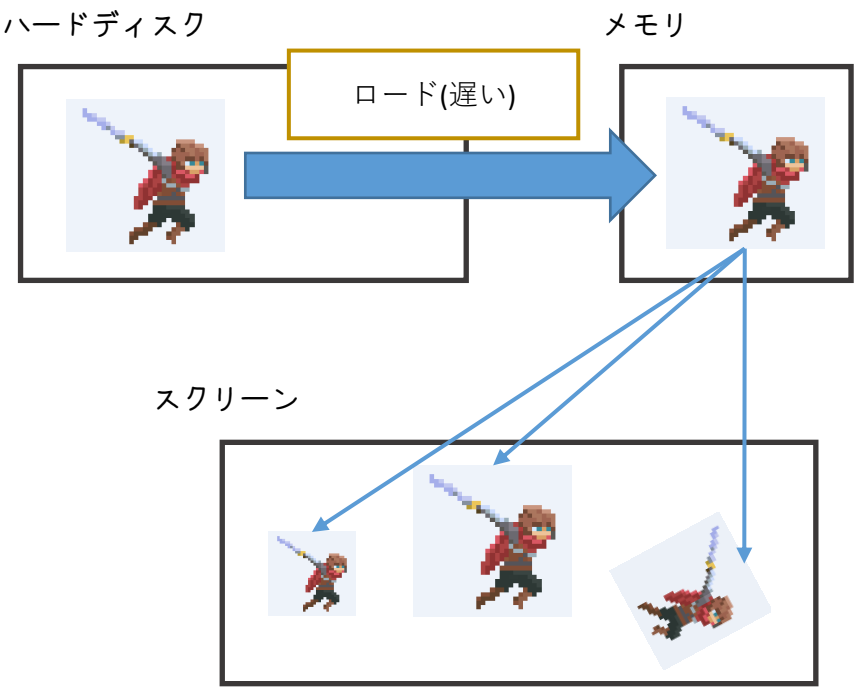
これで概ね、弾の処理が流れるための形はできましたので、
いよいよ弾の実装に入っていきます。

```
ShotBase::ShotBase(int baseModelId)
{
    baseModelId_ = baseModelId;
}
```

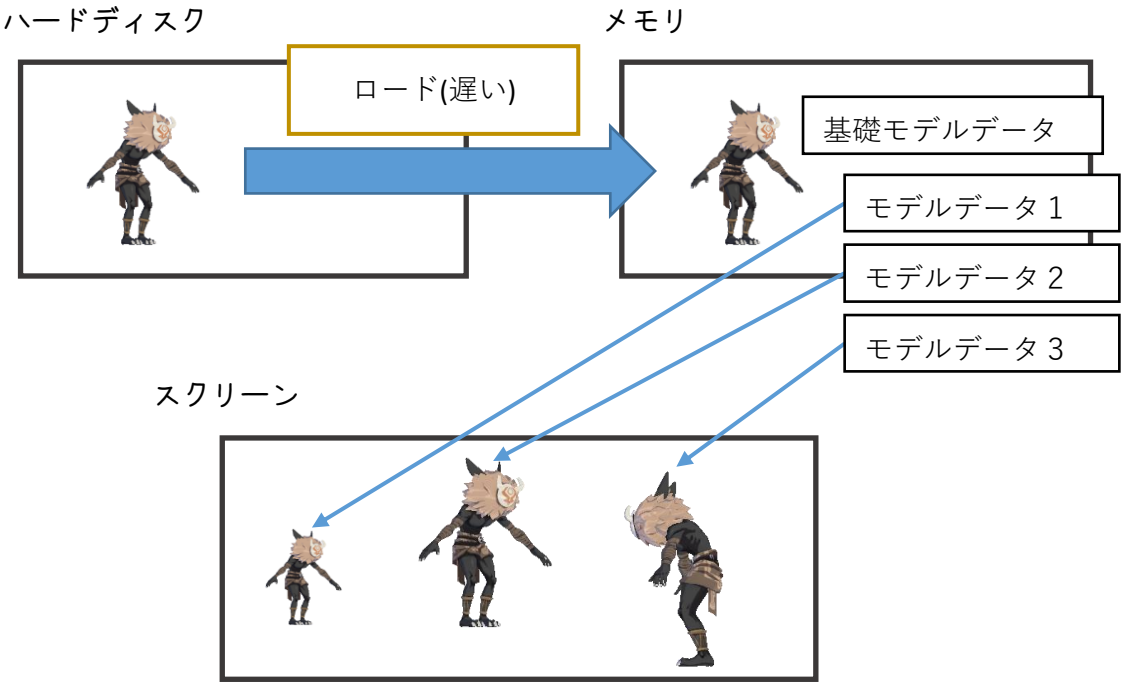
２Ｄ画像と同じように、ロード処理(HDDからメモリに移動する)を
省略するためにハンドルＩＤを引数から渡しているのですが、
３Ｄモデルの場合は、そのままでは使えません。
モデルには頂点情報があり、それぞれで、その大きさや角度、
位置が異なるため、固有のハンドルＩＤである必要があります。

とはいえ、ゼロからモデル情報を構築する必要もありません。
一度、ロードされたモデル情報から必要な情報を複製することができます。

【２Ｄ画像の場合】



【３Ｄモデルの場合】



この基礎モデルデータを複製して、新たに固有のモデルデータを作る機能が、
DxLibでは、MV1DuplicateModel関数として、用意されています。

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

    // 使用メモリ容量と読み込み時間の削減のため
    // モデルデータをいくつもメモリ上に存在させない
    modelId_ = MV1DuplicateModel(baseModelId_);

    // 弾の大きさを設定
    scl_ = { 0.8f, 0.8f, 0.8f };

    // 弾の角度を設定
    rot_ = { 0.0f, 0.0f, 0.0f };

    // 弾の発射位置を設定
    pos_ = pos;

    // 弾の発射方向の設定
    dir_ = dir;

    // 弾の速度
    speed_ = 8.0f;

    // 弾の生存判定
    isAlive_ = true;

}
```

大量にモデルを使う時には、
必ず複製するようにしましょう。
メモリ使用量と処理速度が、
かなり改善されます。

Update関数では、弾を移動させる必要がありますので、
弾の移動処理を実装していきます。
２Ｄであれ、３Ｄであれ、移動処理は変わりません。

移動処理とは、【座標 ＋ 移動量】で実装できます。
移動量は、【方向 × スピード】で求められます。

```
void ShotBase::Update(void)
{

    if (!IsAlive())
    {
        // 生存していなければ処理中断
        return;
    }

    // 弾を移動させる

    // 移動量の計算(方向×スピード)
    VECTOR movePow;

    // 移動処理(座標+移動量)
    ？？？

    // 大きさの設定
    MVISetScale(modelId_, scl_);

    // 角度の設定
    MVISetRotationXYZ(modelId_, rot_);

    // 位置の設定
    MVISetPosition(modelId_, pos_);

}
```

モデルの大きさや、角度、位置を変えないのであれば、Update関数で記載する必要はありませんが、弾は常に移動(位置の変更)を行いますので、Update内で、モデル制御を行います。これらの設定をしないと、移動されません。

モデルの描画、メモリ解放を忘れずに。

```
void ShotBase::Draw()
{

    if (!IsAlive())
    {
        // 生存していなければ処理中断
        return;
    }

    MVlDrawModel(modelId_);

}

void ShotBase::Release(void)
{
    MVlDeleteModel(modelId_);
}
```