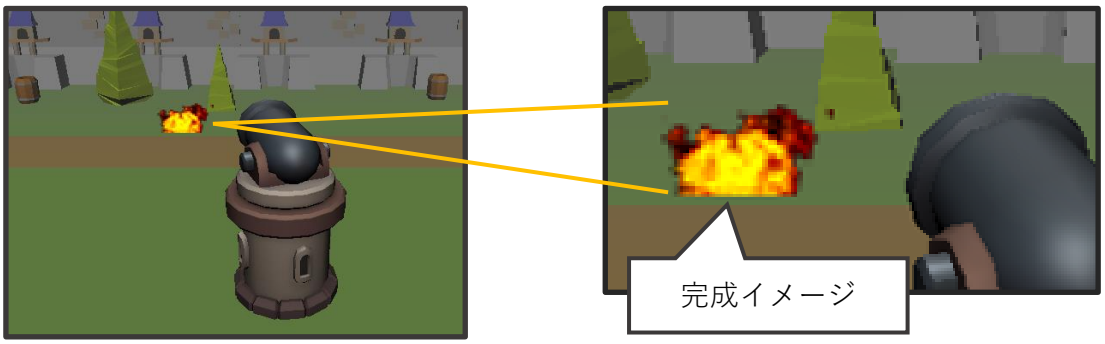
３Ｄワールドに２Ｄ描画（ビルボード）

弾がステージに衝突したら、爆発エフェクトを描画していきたいと思います。
かっこいい３Ｄエフェクトを描画したいところですが、次回プロジェクトの
お楽しみにとっておきます。

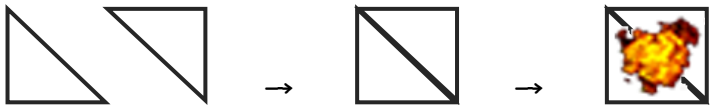
今回は、これまでと同じように、２Ｄ画像をアニメーションさせて、
爆発エフェクトを表現していきます。
３Ｄ空間で２Ｄ画像を描画する際に、ビルボードという便利な技術がありますので、
まずは、そちらを解説します。



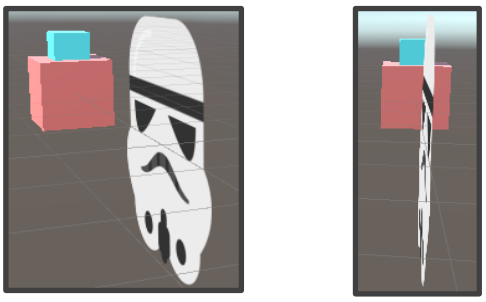
ビルボードとは？

ポリゴンを常にカメラに向ける技術のことです。

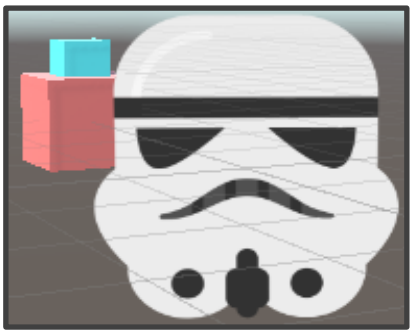
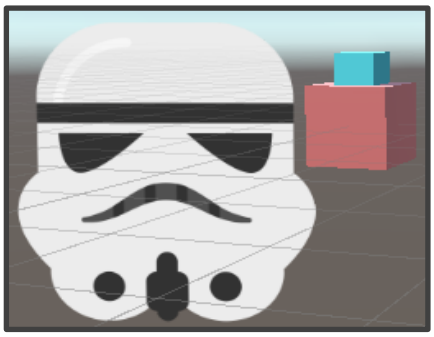
３Ｄ空間上に２Ｄ画像を描画したい場合、下図のように２つのポリゴンを
使って四角形を作り、この四角形に画像を貼り付けて描画します。



そのためカメラの視点を変えて、斜めや横から見てみると、
下図のようにペラペラに見えてしまいます。



このポリゴンの向きを、常にカメラの向きに合わせるよう、
角度を制御すると、２Ｄゲームの時のように、
常に２Ｄ画像を正面から見せることができますので、

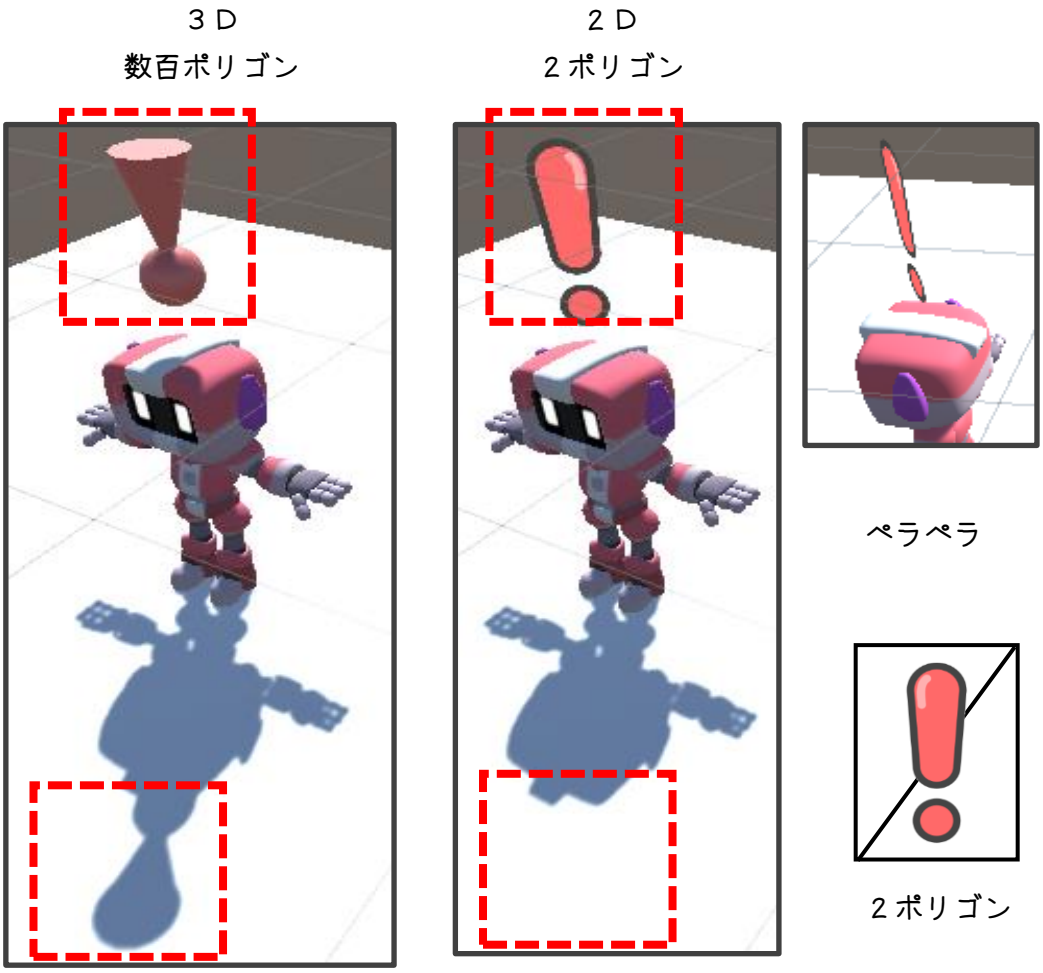
カメラの視点を変えても、常にコッチを見てくれるようになります。

このビルボード、３Ｄでも様々なシーンで使用されます。
例えば、３ＤのＲＰＧ系で、主人公がびっくりした時ですが、



こちら全て、２Ｄで描画されています。
ＵＩアイコン、感情アイコンなどと呼ばれたりしますね。

３Ｄよりも２Ｄの方が視認性が高かったりしますし、
処理負荷の面で見ても、３Ｄよりも２Ｄの方が軽いです。



軽量で見栄えが良いのあれば、それに越したことはないのですが、
影などにも影響が出ますし、演出したい表現によって、使い分けれると
良いと思います。

DxLibにおいて、ビルボード描画する関数は以下の通り。

```
int DrawBillboard3D(
    VECTOR Pos, float cx, float cy, float Size, float Angle,
    int GrHandle, int TransFlag ) ;
```

第１引数　　：　画像を描画する座標
第２引数　　：　描画する画像の中心座標(0.0f ～ 1.0f)
第３引数　　：　描画する画像の中心座標(0.0f ～ 1.0f)
第４引数　　：　描画する画像のサイズ
第５引数　　：　描画する画像の回転角度(ラジアン単位)
第６引数　　：　描画する画像のハンドルＩＤ
第７引数　　：　画像の透明度を有効にするかどうか

それでは爆発アニメーションの実装に移りたいと思います。
画像アニメーションを行うだけになりますので、Blastクラスは作成せず、
ShotBaseをSTATE分けして、処理していきたいと思います。

ShotBase.h

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

class ShotBase
{

public:

    // 衝突判定用の球体半径
    static constexpr float COL_RADIUS = 10.0f;

    // 弾の状態
    enum class STATE
    {
        NONE,
        SHOT,
        BLAST,
        END
    };
```

STATEデザインパターンを導入

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

    // 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);
    void UpdateShot(void);
    void UpdateBlast(void);
    void UpdateEnd(void);

    // 描画
    void Draw();
    void DrawShot();
    void DrawBlast();
    void DrawEnd();

    // 解放処理
    void Release(void);

    // 弾判定
    bool IsShot(void);

    ～ 省略 ～

private:

    // 弾の状態
    STATE state_;

    ～ 省略 ～

    // 重力
    float gravityPow_;

    // 爆発アニメーション画像配列のポインタ
```

弾ごとに画像をロードするのではなく、Cannonクラス側でロードして、画像のハンドルＩＤを引数で渡すようにする。

```
    int* blastImgs_;

    // 爆発アニメーション数
    int blastAnimNum_;

    // 爆発のアニメーション用カウンタ
    int blastCntAnim_;

    // 爆発のアニメーション速度
    float blastSpeedAnim_;

    // 爆発のアニメーション番号
    int blastIdxAnim_;

    // 状態遷移
    void ChangeState(STATE state);

};
```

ShotBase.cpp

```
ShotBase::ShotBase(int baseModelId, int* blastImgs, int blastAnimNum)
{
    baseModelId_ = baseModelId;
    blastImgs_ = blastImgs;
    blastAnimNum_ = blastAnimNum;
}

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

    ～ 省略 ～

    // 爆発のアニメーション用カウンタ
    blastCntAnim_ = 0;

    // 爆発のアニメーション速度
    blastSpeedAnim_ = 0.3f;

    // 状態遷移
    ChangeState(STATE::SHOT);
```

```
}


void ShotBase::Update(void)
{

    ※STATE別に処理を分けて、各STATEのUpdate関数を呼び出してください

}

void ShotBase::UpdateShot(void)
{

    ※今までの弾移動処理

}

void ShotBase::UpdateBlast(void)
{

    ※爆発のアニメーション処理
    ※アニメーションが終了したら、END状態へ遷移

    // 爆発アニメーションの終了判定
    if (blastIdxAnim_ + 1 >= blastAnimNum_)
    {
        ChangeState(STATE::END);
    }

}

void ShotBase::UpdateEnd(void)
{
    ※特に何もしなくてよい
}

void ShotBase::Draw()
{

    ※STATE別に処理を分けて、各STATEのDraw関数を呼び出してください
```

```
}

void ShotBase::DrawShot()
{

    ※今まで通り、弾モデルの描画

}

void ShotBase::DrawBlast()
{
    DrawBillboard3D(
       pos_, 0.5f, 0.5f, 80.0f, 0.0f, blastImgs_[blastIdxAnim_], true);
}

void ShotBase::DrawEnd()
{
    ※特に何もしなくてよい
}

bool ShotBase::IsShot(void)
{
    ※STATEがSHOT状態だったらtrue
}

bool ShotBase::IsAlive(void)
{
    return state_ != STATE::END;
}

void ShotBase::Blast(void)
{
    ※BLAST状態へ遷移させる
}


void ShotBase::ChangeState(STATE state)
{

    ※いつもの状態遷移処理を記述する
```

ビルボードで、
爆発アニメーション表示

```
}
```

Cannon.h

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

class Cannon
{
public:
    ～ 省略 ～

    // 爆発のサイズ
    static constexpr int BLAST_SIZE_X = 32;
    static constexpr int BLAST_SIZE_Y = 32;

    // 爆発のアニメーション数
    static constexpr int BLAST_ANIM_NUM = 16;

    ～ 省略 ～
private:
    ～ 省略 ～

    // 爆発の画像(本来は外部リソース用の管理クラスを作るべき。弾モデルも。)
    int blastImgs_[BLAST_ANIM_NUM];

    ～ 省略 ～
```

Cannon.cpp

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

    // 爆発エフェクト読み込み
    LoadDivGraph((Application::PATH_IMAGE + "Blast.png").c_str(),
        BLAST_ANIM_NUM, 4, 4, BLAST_SIZE_X, BLAST_SIZE_Y, blastImgs_, true);
```

```
    ～ 省略 ～
}

void Cannon::Release(void)
{
    ～ 省略 ～

    // 読み込んだ画像の解放
    for (int i = 0; i < BLAST_ANIM_NUM; i++)
    {
        DeleteGraph(blastImgs_[i]);
    }

}

ShotBase* Cannon::GetValidShot(void)
{
    ～ 省略 ～

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

    return shot;

}
```

Cannonで画像ロード

画像情報を引数で渡す

※C++では、固定長配列を引数に渡すことが
できません。可変長は可。
しかし、固定長配列のポインタであれば、
渡すことができます。(危険ですが)
せめてものリスクケアで、
固定長配列の要素数を一緒に渡して上げることで、
何要素目まで、配列操作してよいかの
ブロックを可能にしています。

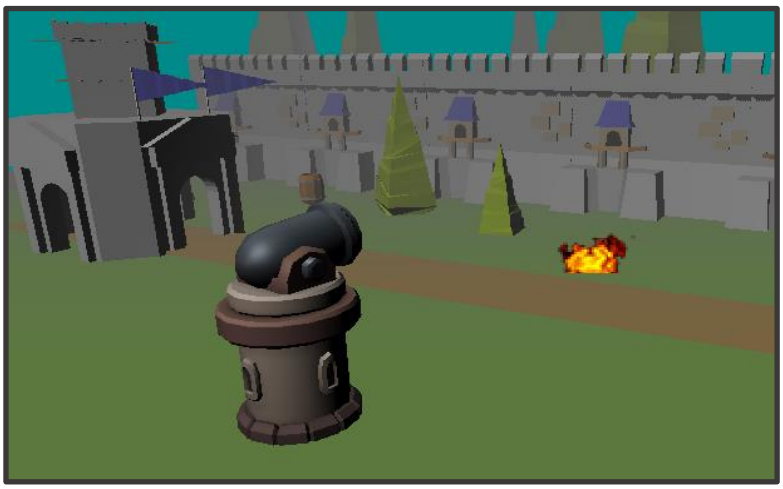
GameScene.cpp

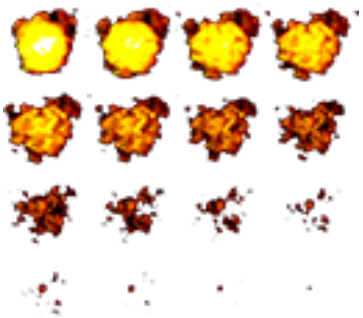
```
void GameScene::Update(void)
{
    ～ 省略 ～

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

```
        shot->Update();

        if (!shot->IsShot())
        {
            // 爆発中や処理終了後は、以降の処理は実行しない
            continue;
        }

        ～ 省略 ～

    }

}
```

上記のプログラムが上手く動作したら、
３Ｄ空間上にビルボードで２Ｄ描画できていると思います。

カメラがZ軸正面を向いていますので、
ビルボードによって、ポリゴンがちゃんとカメラの方向を向いてくれるか
心配かと思いますので、タイトル画面でカメラの位置や角度をある程度変えて、
ゲームシーンに遷移してみてください。





ゲームアングルは当然変わりますが、ビルボードで描画した
２Ｄ画像は正面を保ったままになっていることがわかるかと思います。