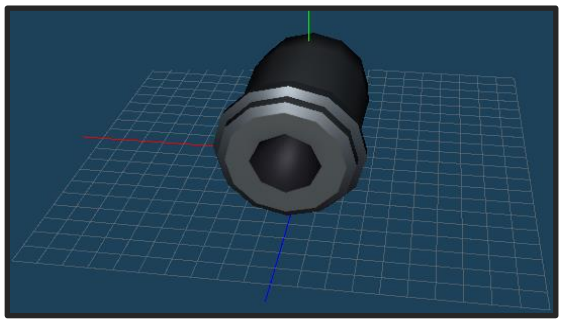
砲台と砲身の親子関係

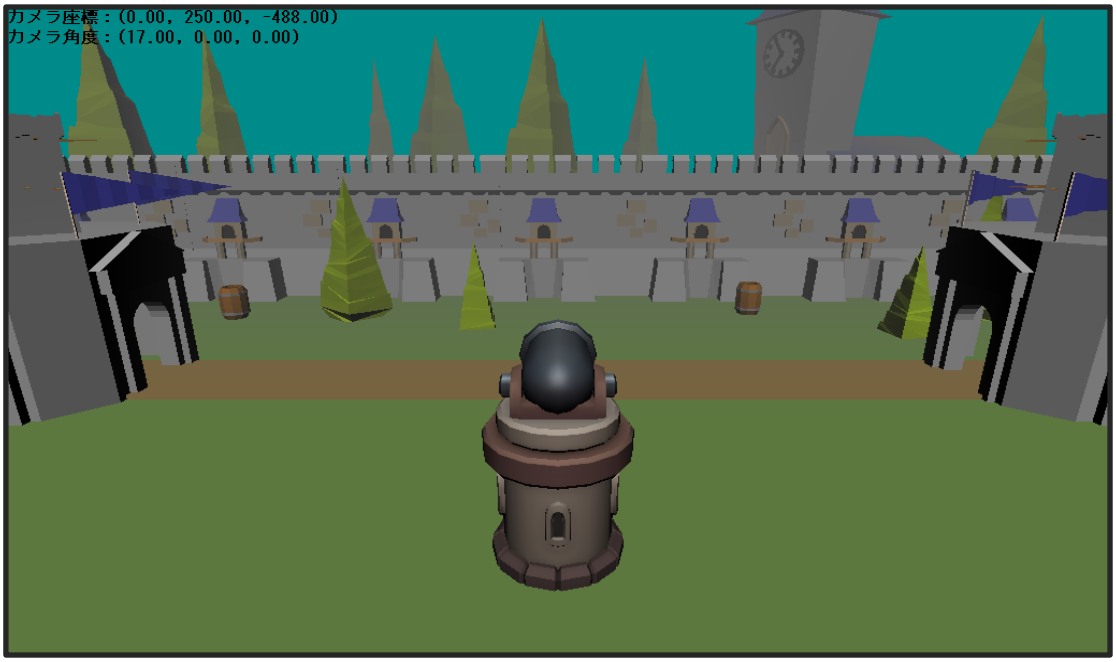
これまでのPlayerクラスにあたる、Cannonクラスを作成していきます。



モデルは土台と砲身に分かれており、
２つの外部ファイルを読み込む必要があります。

【演習】

Cannonクラスを作成して、２つモデルを下図のように描画してください。



条件

①それぞれのモデルの大きさ、角度、位置はメンバ変数で宣言すること

例　　VECTOR standScl_;
　　VECTOR standRot_;
　　VECTOR standPos_;

②下記の参考値の初期化はInit関数内で行うこと

③MVlSetScale等の３Ｄ制御関数はUpdate関数内で行うこと

参考値

土台	大きさ	：	{ 0.8f, 0.8f, 0.8f }
	角度	：	{ 0.0f, 0.0f, 0.0f }
	位置	：	{ 0.0f, 10.0f, -200.0f }
砲身	大きさ	：	{ 0.8f, 0.8f, 0.8f }
	角度	：	{ 0.0f, 0.0f, 0.0f }
	位置	：	{ 0.0f, 110.0f, -200.0f }

設置して貰った砲身と砲台ですが、今の状態だと、
それぞれで、大きさ・角度、位置を管理していますので、
砲台の位置を移動させると、砲身も移動させないといけません。

２つで１セットのオブジェクトになりますので、
砲台を移動させたら、砲身も一緒に移動して欲しいところです。
こういった場合、２つのモデルに親子関係を持たせることで、問題は解消できます。

Unityなどのゲームエンジンを使用された方だとイメージしやすいかもしれませんが、

※Unityの画面

こんな感じで、入れ子構造にすると、ゲームエンジンの方で
勝手に親子関係が作られ、砲台が移動したら、砲身も移動して、砲台が回転したら、
砲身も回転するようになります。
これをゲームエンジンの力を借りず、プログラムで制御していきます。

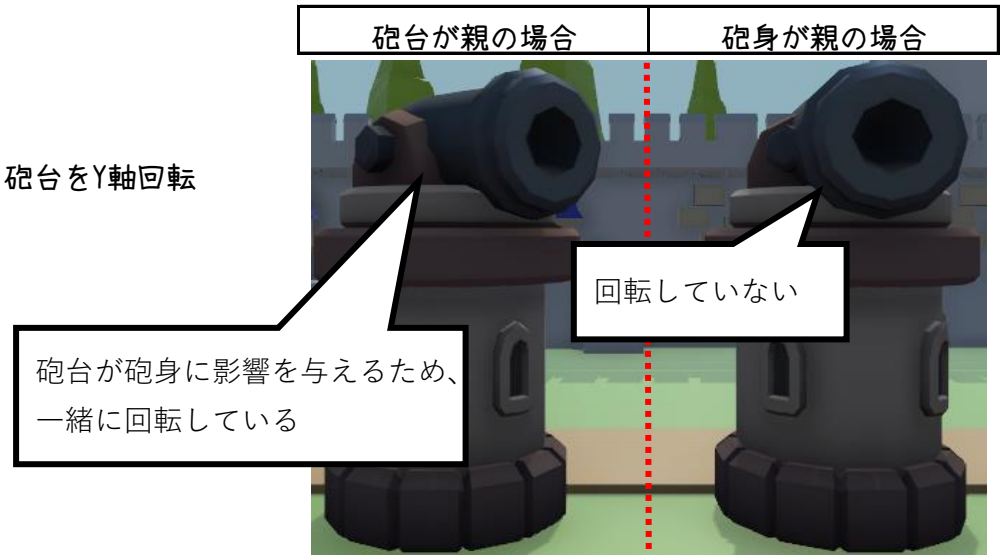
今回は、３Ｄ制御の３要素、大きさ、角度、位置のうち、
位置だけ親子関係を持たせていこうと思います。

どっちが親でどっちが子供？

今回は、砲台が親で、砲身が子供になります。
どうやって見分けるかというと感覚的なところもあるのですが、
イメージ的に土台側が親になります。
人間の場合、腰や背骨が親で、肩、肘、手首、指のような順番で子供に
なっていきます。

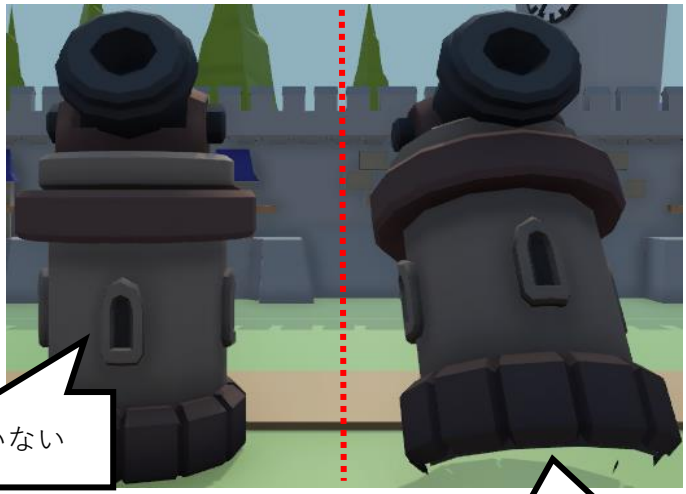
腰が回転すると上半身も回転しますし、肩が回転すると、
肘や手もそれに連れて、回転したり移動したりします。
逆に指を動かしても、肩や腰には影響がありません。
影響を与える側が親になります。

大砲の関係でいくと、回転に着目するとわかりやすいかと思います。

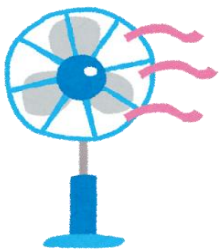


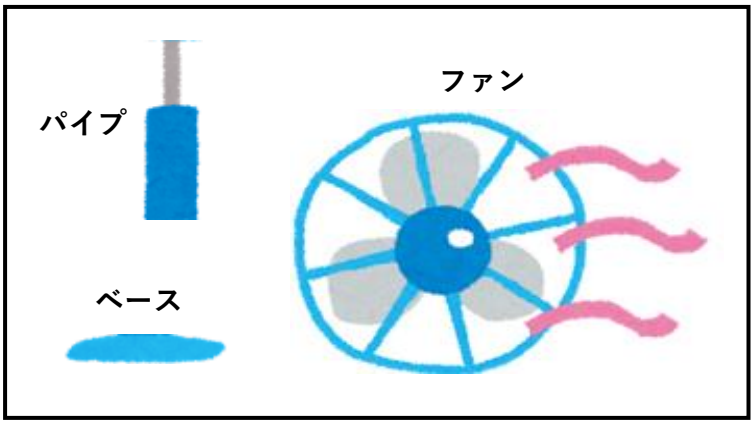
砲台が親の場合	砲身が親の場合

砲身をX軸回転



どちらが自然な親子関係か一目瞭然ですね。
砲台が親の方が、私達がやりたい制御だと思います。
それでは試しに、下図の扇風機を３つのパーツに分けます。





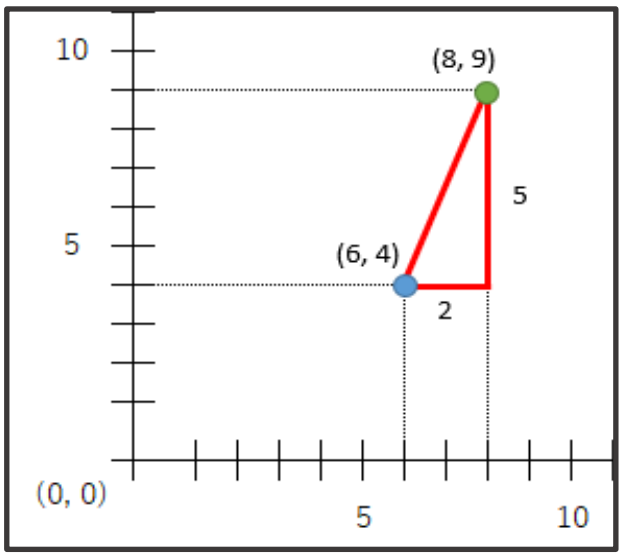
正しい親子関係はどれでしょうか？

	親	->	子	->	孫
ア	ファン	->	パイプ	->	ベース
イ	ファン	->	ベース	->	パイプ
ウ	パイプ	->	ベース	->	ファン
エ	ベース	->	パイプ	->	ファン

前置きが長くなりましたが、位置の親子関係を作っていきます。
位置を同期したり、合わせたりする時は、２Ｄの時と同じように
相対座標を使います。

相対座標とは？

ある特定の点(任意点)を基点として2点目以降の位置を指示する座標のこと



青色の点を基点とした場合、
緑色の相対座標は、
(2, 5)となる。

青色の点が動いても、
そこを基点に相対座標を
使って緑色の点の座標を
決めれば位置関係が
常に保たれる。

モデルのそれぞれの座標は、

砲身が、{ 0.0f, 110.0f, -200.0f }

砲台が、{ 0.0f, 10.0f, -200.0f }

相対座標を算出するには、子供から親の座標をマイナスすれば良いですので、

x = 0.0f - 0.0f = 0.0f
y = 110.0f - 10.0f = 100.0f { 0.0f, 100.0f, 0.0f }
z = -200.0f - (-200.0f) = 0.0f となる。

Cannonクラスのメンバ変数に、以下の相対座標を追加して、

```
// 砲台からの相対座標
VECTOR barrelLocalPos_;
```

Init関数で初期化する。

```
// 位置の設定
//barrelPos_ = { 0.0f, 110.0f, -200.0f };
// 土台からの相対座標とする
barrelLocalPos_ = { 0.0f, 100.0f, 0.0f };
```

モデルには相対座標ではなくて、ワールド座標を渡す必要がありますので、
相対座標からワールド座標に変換します。
先ほどは、親の座標をマイナスすることで相対座標にしましたので、
逆に親の座標をプラスすることでワールド座標に戻してあげます。

```
barrelPos_.x = standPos_.x + barrelLocalPos_.x;
barrelPos_.y = standPos_.y + barrelLocalPos_.y;
barrelPos_.z = standPos_.z + barrelLocalPos_.z;
```

３Ｄ授業で新しく登場した３つのfloat型の型を持つVECTORは
DxLibで定義されている構造体です。
DxLibではVECTOR同士の足し算は、
以下の関数で計算することが推奨されます。(xyz分、コードが長くなるので)

```
// VECTOR同士の加算 VAdd関数
barrelPos_ = VAdd(standPos_, barrelLocalPos_);
```

上級者の方へ

自作のクラスであれば、operator演算子をオーバーライドすれば、

```
barrelPos_= standPos_ + barrelLocalPos_;
```

という記述ができます。

例

```
Vector2 operator+(const Vector2& v) const {
    return Vector2(x + v.x, y + v.y);
}
```

DxLibの定義内容は以下の通り。

```
__inline VECTOR         VAdd( const VECTOR &In1, const VECTOR &In2 )
{
    VECTOR Result ;
    Result.x = In1.x + In2.x ;        ←元の式のように
    Result.y = In1.y + In2.y ;          足し算しているだけ
    Result.z = In1.z + In2.z ;
    return Result ;
}
```

ちなみに引き算は、

```
barrelLocalPos_ = VSub(barrelPos_, standPos_);

__inline VECTOR         VSub( const VECTOR &In1, const VECTOR &In2 )
{
    VECTOR Result ;
    Result.x = In1.x - In2.x ;
    Result.y = In1.y - In2.y ;
    Result.z = In1.z - In2.z ;
    return Result ;
}
```

VECTORの３つの数字に一律、同じfloat型の掛け算を行う場合は、

```
movePow_ = VScale(dir_, 3.0f);        方向×スピード = 移動量

__inline VECTOR         VScale( const VECTOR &In, float Scale )
{
    VECTOR Result ;
    Result.x = In.x * Scale ;
    Result.y = In.y * Scale ;
    Result.z = In.z * Scale ;
    return Result ;
}
```

VECTOR同士の掛け算は、内積として扱われることが多いですので、

```
// ベクトルの内積
__inline float          VDot( const VECTOR &In1, const VECTOR &In2 )
{
    return In1.x * In2.x + In1.y * In2.y + In1.z * In2.z ;
}
```

ついでに外積。

```
// ベクトルの外積
__inline VECTOR         VCross( const VECTOR &In1, const VECTOR &In2 )
{
    VECTOR Result ;
    Result.x = In1.y * In2.z - In1.z * In2.y ;
    Result.y = In1.z * In2.x - In1.x * In2.z ;
    Result.z = In1.x * In2.y - In1.y * In2.x ;
    return Result ;
}
```

今後、バシバシ使っていきます！