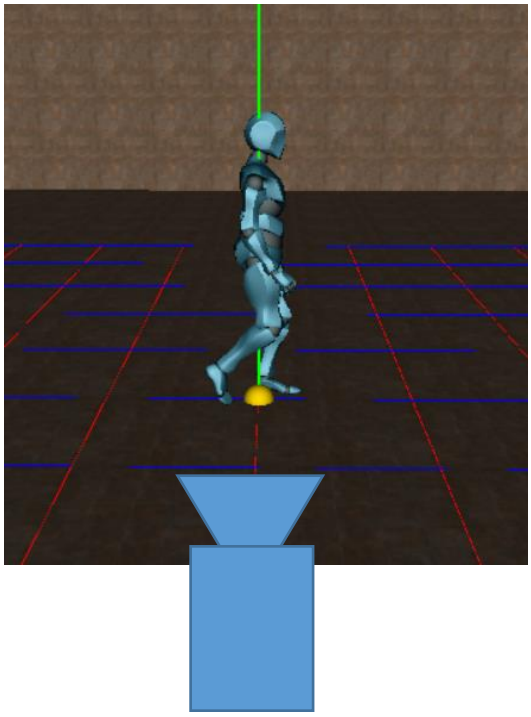
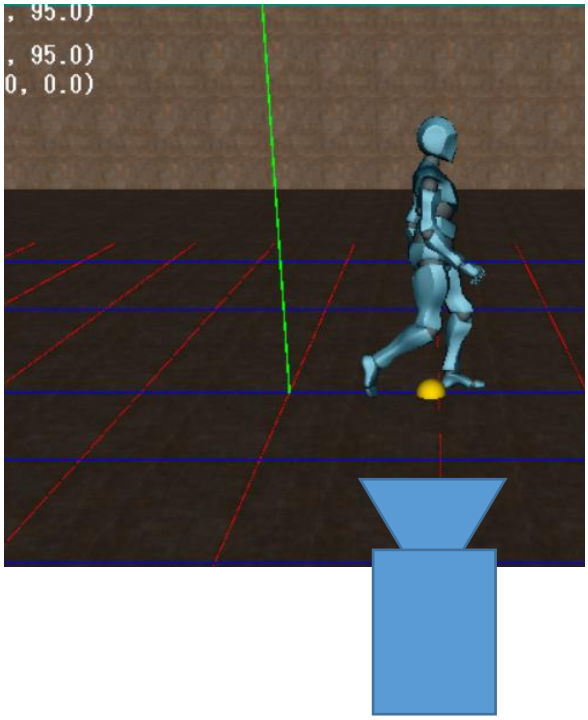
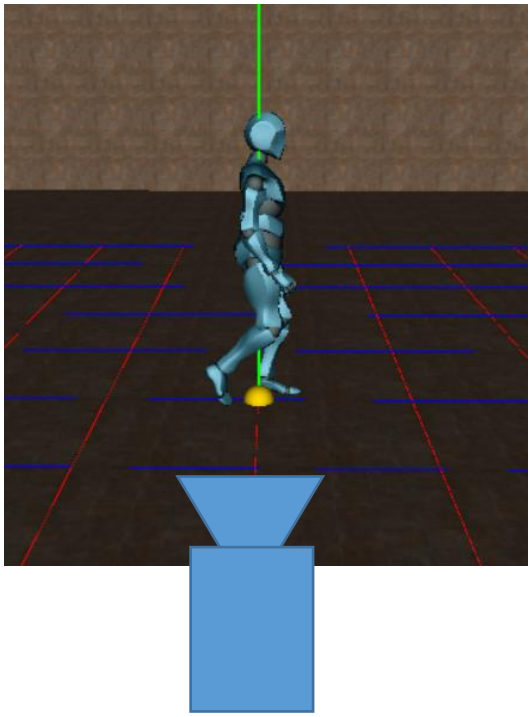
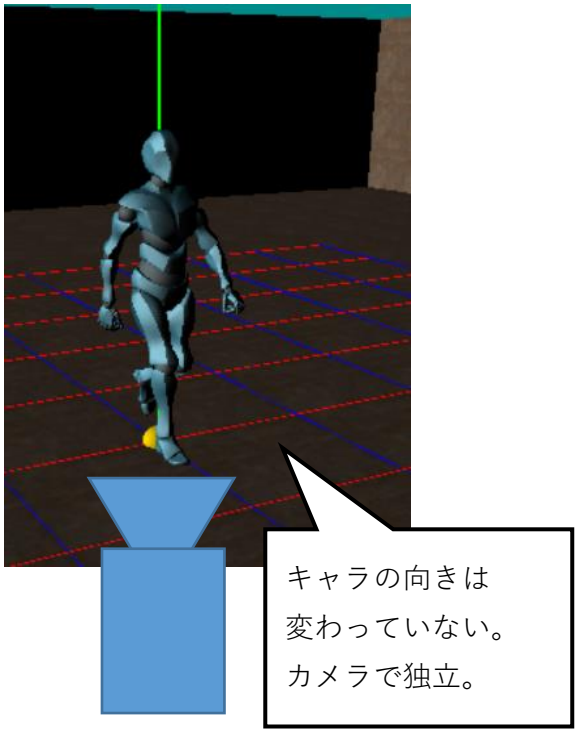
カメラの追従とバネの力

３Ｄビューワで作成した追従カメラは、
カメラの位置と注視点は、操作キャラクターと連動し、
カメラの角度(向き)は、カメラ独自のものとしていました。

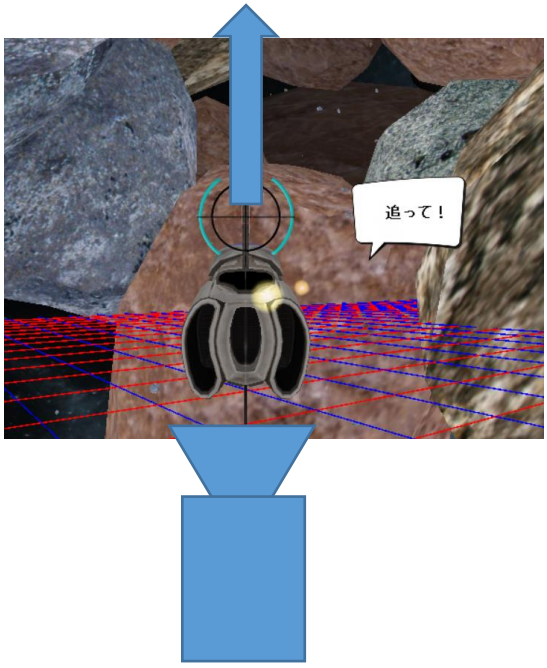


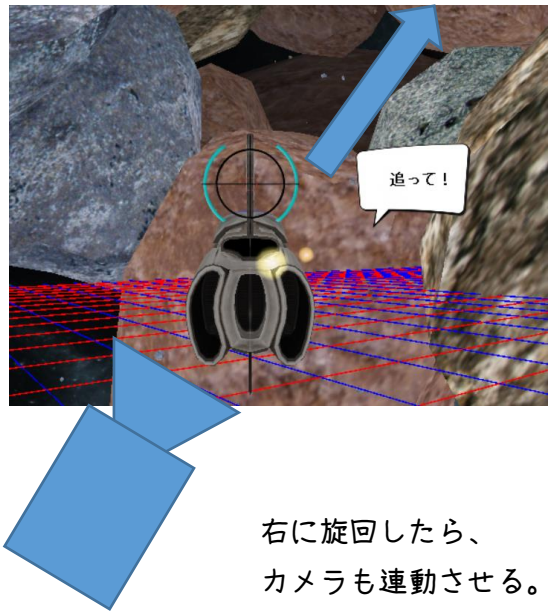






今回の３Ｄシューティングにおいては、
自機は、自立移動(勝手に前に進む)を行いますので、
プレイヤー操作としては、移動はなく、旋回のみとなります。
その場合、カメラも合わせて、回転させた方が操作性が良いですので、
カメラに独立した角度を持たせない形で実装していきます。

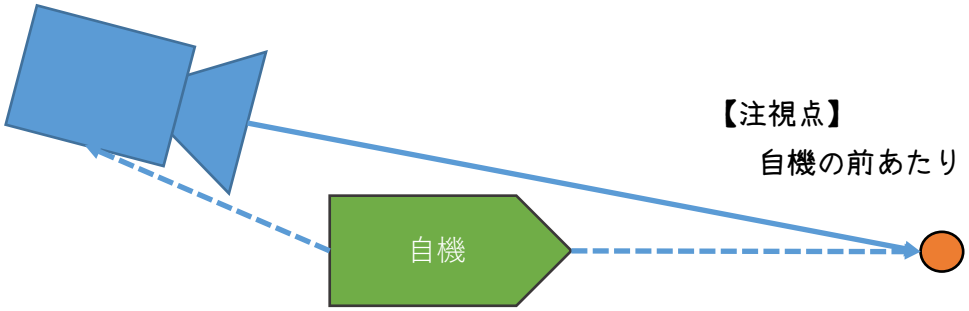




右に旋回したら、
カメラも連動させる。

自機の座標を中心に、カメラ位置と注視点を設定します。

【カメラ位置】自機の後ろのちょっと上



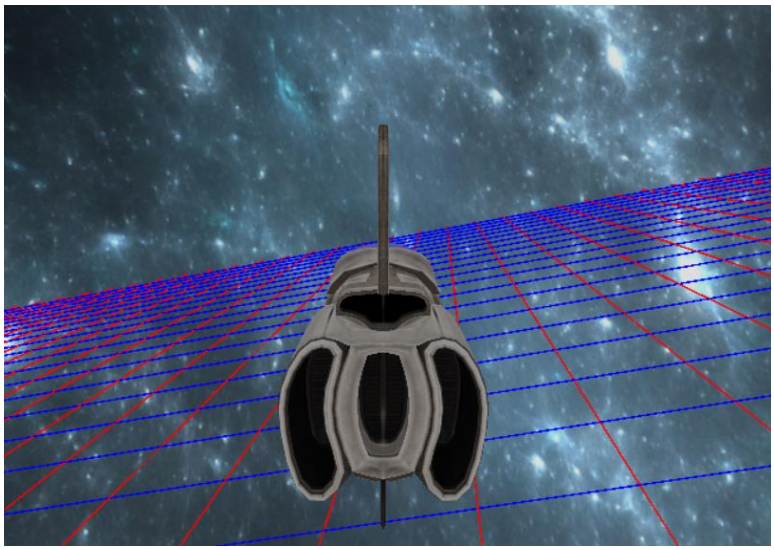
追従の実装が終わったら、上手くいっているか確認するため、
自機の旋回機能も実装していきましょう。

今回は、自由な旋回が可能な操作にしていきますので、
クォータニオンで計算していきましょう。
移動は、前方に移動するという処理を既に作っているかと思いますので、
自機の向きだけ、変えて上げれば大丈夫です。

・・・

実装ができ、宇宙空間を自由に飛び回れるようになりましたでしょうか？

移動や旋回処理自体は問題がないのですが、



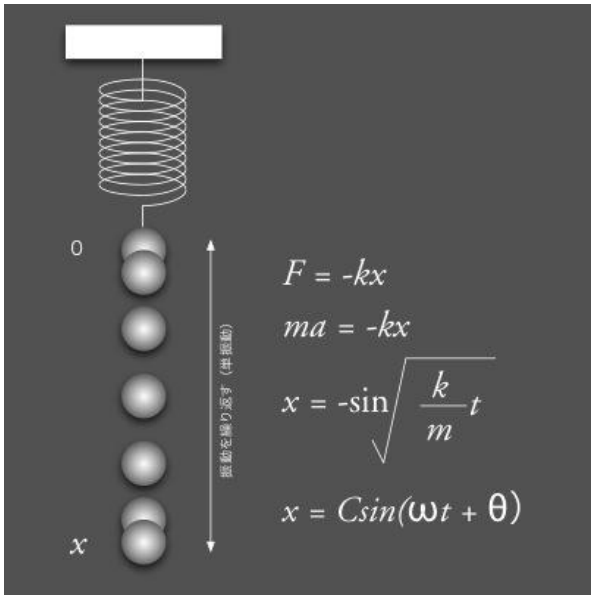
自機とカメラの動きが完全に連動しているので、
自機が常に前方姿勢を保つことによって、
旋回していないように見えます。

レーシングゲームやシューティングなど、
カメラの旋回が独立していない場合に、このような現象が起きてしまいます。

【常に】前方姿勢を保ってしまうのが問題になっていますので、
改善策としては、
完全に連動させるのではなく、ラグ(時間差)を作って上げるのが
良さそうです。

バネの力

数学的に定義すると、以下のようになるようです。



バネの力 = -バネの強さ × バネの伸び - 抵抗 × カメラの速度

1つずつ計算していきましょう。

バネの強さ(硬さ)　→　上図のコイル(バネ)の硬さになりますので、
抵抗の力として、定数などで定義しましょう。

バネの伸び　→　本来(理想)のカメラ位置と、
現在のカメラ位置との差になります。

理想の位置は、前述で実装した、
自機からの相対位置となります。

```
mDiffPos = VSub(mPos, backPos);
```

抵抗　→　減衰。教本ゲームプログラミングC++より抜粋。

```
float dampening = 2.0f * sqrt(POW_SPRING);
```
※POW_SPRING　=　バネの強さ(硬さ)

カメラの速度　　→　　現在のカメラ速度。メンバ変数で定義。
バネの力に合わせて、早くなったり、
遅くなったりする。
バネの力を生成後、更新を忘れずに。

一連の処理をまとめると、

```
// 理想位置
VECTOR idealPos = VAdd(shipPos, relative);

// 実際と理想の差
VECTOR diff = VSub(mPos, idealPos);

// 力 = -バネの強さ × バネの伸び - 抵抗 × カメラの速度
VECTOR force = VScale(diff, -POW_SPRING);
force = VSub(force, VScale(mVelocity, dampening));

// 速度の更新
mVelocity = VAdd(mVelocity, VScale(force, delta));
```

上記のような感じになります。

カメラ位置や注視点の更新や、定数の定義は省いていますので、
参考にしながら、カメラの新しいモード、
FOLLOW_SPRINGを追加して、自分で実装してみましょう。