インタラクティブゲーム制作
ゲームプログラミング講座
第10回資料

竹内 亮太
(2011/7/6)

10　ゲームにおけるカメラワーク

10.1　映像とは異なる難しさ

今回は3DCGを使ったゲームにおいてのカメラワークについて述べます。映像作品では「何を見せるか」を念頭にカメラを動かしていきますが、ゲームの場合は一方的にカメラワークを押しつけるわけにはいきません。プレイヤーの操作に応じて、視点を的確に移動させてあげる必要があります。3DCGにおけるカメラは画面に映る物を決定する重要な要素ですので、カメラの操作が快適でないと非常にストレスがかかり、クソゲーの烙印を押されかねません。ゲームのカメラワークはユーザインタフェースの一部としてとらえ、慎重に調整するべきでしょう。

ここでは3Dを使ったゲームにおける典型的なカメラワークを整理し、その実現に必要な処理や考え方を紹介していくことにします。

10.2　1人称視点

1人称視点とは、**プレイヤーが操作するオブジェクトの視点**のことを指します。人間を操作するならば頭の位置、ロボットを操縦するゲームであればコックピットの位置から前方を向く視点のことになります。何と言っても臨場感ではトップクラスの迫力を誇りますので、3Dゲームと言えば1人称、という人も多いかと思います。

主に海外で大人気の、銃を持って殺したり殺されたりするゲームでよく用いられるため、**FPS**(First Person Shooting = 1人称視点シューティング)という1ジャンルを確立しています。しかし日本では「主人公が見えないのはヤダ」「3D酔いするからヤダ」などの理由により、あまり積極的には用いられないようです。

10.2.1　実装

1人称視点の実装は実に簡単です。以下のどちらかの手段で実現できます。

- 主人公のモデルをカメラとして登録する
- 主人公のモデルを親モデルとして、目線の位置にカメラを固定する

Carサンプルのドライバー視点が前者の方法で実現している例ですね。実戦では後者の方法を採ることが多いと思いますが、それほど難しくはないでしょう。

1人称視点の良いところは**キャラクターの操作がカメラ操作に直結していること**です。作る方も操作する方も悩まなくて済むわけですね。こういうシンプルさが海外でも受けるのかもしれません。

また、1人称視点はゲーム以外のコンテンツでも「ウォークスルー」と呼ばれて用いられることが多いです。空を飛ぶような視点の場合は「フライスルー」と呼ばれます。3DCGのプログラミングをするなら、一度は実装してみたいところですね。

10.2.2　マウスによるエイミング

FPSでよくある操作系で、**WASDキーで移動、マウス移動で視点回転**というパターンがあります。それを実現するための処理をサンプルとして公開しますので、詳細はそちらをご覧ください。

- 横方向の視点回転は glRotateWithVec(myPos, fk_Y, rad)
- 上下方向の視点回転は loRotateWithVec(0, 0, 0, fk_X, rad)

とするのが典型例ですが、このradの値をマウスカーソルの移動量から決定するのがポイントです。視点の方向にそのまま移動するとおかしなことになるので工夫が必要ですが、概ねこんな感じでOKです。

10.3　3人称視点

3人称視点とは、厳密な話をすれば「1人称ではない視点」の総称になります。操作キャラの後方からキャラクターとその周囲を俯瞰する視点や、2Dのゲームのように真上から見下ろしたり、真横から見たりするのも3人称視点に含まれます。

日本では操作キャラが見えているゲームの方が好まれる傾向にあるようで、FPS のようなゲーム性の物でも 3 人称視点になっているか、1 人称視点と切り替えが出来る物が多いです。**TPS**(Third Person Shooting) と呼んで区別することもありますが、単純に 3 人称にするだけなら FPS と実装方法も操作体系も大差はありません。カメラを操作キャラの子にして、後方に固定するだけですからね。プレイヤーは操作キャラのお尻を眺め続けることになるので「ケツゲー」と揶揄されるような呼び方をすることもあります。

このテキストでは、前述したような単純な TPS ではなく、操作キャラの動きに合わせてカメラを別途追従させるような処理が必要なカメラワークを中心に取り扱います。

10.3.1 単純な親子では片付かない

FPS や TPS では、常にキャラクターはカメラの向いている方向を向きます。しかし見下ろし型のカメラの場合は、カメラの向きとキャラの向きが一致しません。このため、カメラとキャラを単純に親子関係にするだけではうまく行きません。多少面倒かもしれませんが「キャラが動いた分だけカメラも動かす」というのがシンプルな対処方法です。

10.3.2 高さ方向への追従

3D のゲームなのですから、キャラクターがジャンプをすることも多いでしょう。ですがジャンプの上昇や落下に忠実な上下動をカメラにさせると、結構な違和感を覚えると思います。元々私が 3D 酔いする性質だからかもしれないですが。方向キーによる移動は「移動する操作をしている」という自覚がプレイヤーにあるので大丈夫なのですが、ジャンプの上昇や落下は物理法則による挙動なので、ある意味プレイヤーの想定を超えた動作が生じます。それにカメラを強引に合わせるので違和感が生じるのですね。

これを解消する方法として、私が個人的にあるゲームを解析した結果、次のような手法が有効であることが分かりました。

- 上昇中は追従せず、着地したときにひゅっと移動する。
- 落下中は、跳び始めの高さより下に落ち始めてから追従する。

もちろん、画面外に飛び出るような大ジャンプをさせるような場合は、これではフォローしきれない場合があるので、あくまで一例ということで紹介します。

10.3.3 目標地点への簡易移動アニメーション

ところで、カメラをひゅっと移動させるにはどうすればいいでしょうか?いきなり移動すると見た目がかっこ悪いので、じんわりと移動させてやりたいところです。真面目にやるのであれば、glTranslate() で移動量を少しずつ加えてやるか、パラメータ t を 0.0〜1.0 で変化させつつ $\mathbf{P} = (1-t)\mathbf{A} + t\mathbf{B}$ という線分の式を使って座標を求めるところですが、状態遷移の管理が面倒ですよね。

そこで「現在位置と目標位置」の座標だけで、ひゅっとかっちょよく移動させられるテクニックを紹介します。数式にすると $nowPos = 0.5 * nowPos + 0.5 * targetPos$ となりますが、いまいち分かりにくいですね。ではコードで示しましょう。

```
model.glMoveTo(0.5*model.getPosition()
                + 0.5*targetPos);
```

このコードを毎ループ呼ぶだけで、いい感じにひゅっとした動きになります。ひゅっと具合を変えたい場合は現在位置と目標位置のブレンド比率を変えてください。0.5:0.5 でも結構速いので、0.75:0.25 くらいでもいいかもしれません。

この処理でなぜいい感じになるかは、ちょっと考えれば分かると思います。現在位置と目標位置の中間、その位置と目標位置の中間、と位置を更新していくことにより、最終的にはほぼ目標地点と同一と言っていい座標に移動させられることになるわけです。微妙な数値誤差は出るでしょうが、ゲームでなら問題ないレベルでしょう。カメラワーク以外にも、UI のパーツが出たり引っ込んだりする時にも使えると思います。費用対効果に優れたコードなので、ぜひお試しください。

ちなみに姿勢 (角度) の場合も同じようなテクニックが使えますが、オイラー角や方向ベクトルは補間計算に適しません。クォータニオンを使う必要があります。詳細を語りだすと大学院レベルの数学理論の話になるので、コードで例示しておきます。

```
// 作業用にオイラー角とクォータニオンの変数を用
```

意

```
fk_Angle      nowAng, targetAng;
fk_Quaternion nowQ, targetQ, interQ;

// 今の姿勢と目標の姿勢をオイラー角で取得して、
// クォータニオンに変換
nowAng = model.getAngle();
targetAng = /* 目標姿勢を求めて代入 */;
nowQ.makeEuler(nowAng);
targetQ.makeEuler(targetAng);

// 補間姿勢を計算
// (0.5 は現在姿勢にかかる係数なのでお好みで)
interQ =
  fk_Math::quatInterSphere
            (nowQ, targetQ, 0.5);
// オイラー角に変換してモデルに反映
model.glAngle(interQ.getEuler());
```

はい、面倒ですね。ですが角度は非常にセンシティブなデータなので、このくらいの手間が必要なのです。これもカメラ以外に応用できると思います。ロックオンターゲットに追従する動きなどに使えるはずです。

10.3.4 回転操作で生じる問題

さて、見下ろし型のカメラで視点の回転をできるようにした場合、方向キーによって操作キャラが進む方向も変化することになります。1 人称視点ではローカル座標を基準にしていればだいたい良かったのですが、視点操作が分離するとそうもいきません。

ですが、先週までに習ってきたベクトルの演算をもってすれば、カメラの姿勢から操作キャラの進行方向を得ることなど造作もないことです。

- ↑キー方向：カメラの方向ベクトルを取得し、y 成分を 0 にした上で正規化
- ↓キー方向：↑キー方向の反対 (マイナスで反転)
- →キー方向：カメラの方向ベクトルとアップベクトルの外積
- ←キー方向：→キー方向の反対 (マイナスで反転)

この程度、いちいちコードで例示するまでもないですよね？

カメラの回転処理ですが、多くの場合は glRotateWithVec() を使うのが適します。映像のカメラワークでいうところの**オービット**に相当する動きになるわけです。この軸の取り方が案外厄介ですので、ここはコードを交えて解説します。

```
if(/* R_TURN */) {
    camera.glRotateWithVec
              (targetPos, fk_Y,  rad);
}
if(/* L_TURN */) {
    camera.glRotateWithVec
              (targetPos, fk_Y, -rad);
}
if(/* U_TURN */) {
    camera.glRotateWithVec
         (targetPos, targetPos+vecR,  rad);
}
if(/* D_TURN */) {
    camera.glRotateWithVec
         (targetPos, targetPos+vecR, -rad);
}
```

左右方向の回転は何ら問題ないはずです。注視点を通る、グローバルの y 軸方向を回転の中心とすれば、注視点を周回する軌道を描きます。

上下方向回転は、カメラのローカル座標における x 軸が回転軸方向となりますが、loRotateWithVec() を使うと注視点の指定が少々面倒です。そこで「任意の座標 2 点を通る回転軸」を指定することにし、注視点と、注視点から「カメラにとっての真横方向」にずれた位置を指定します。vecR は先ほど求めた、カメラの方向ベクトルとアップベクトルの外積を表します。回転の時でも外積は重要なんですね。

10.3.5 角度による回転のロック

上下方向回転を自由にさせてしまうと、地面にめりこんで床から注視点を見上げる羽目になったり、真上から見下ろす視点が猛烈なガクつきに襲われたりすることになります。この現象こそが**ジンバルロック**と呼ばれるもので、3DCG を始めとした姿勢制御における鬼門になっています。

このため、上下方向回転には制限をつけるべきです。地面に対して 10〜70 度程度の範囲をキープするようにしましょう。地面に対して今どのくらいの角度になっているかは、以下のコードで得ることができます。

```
fk_Vector vecF;
double    cosT, rad;
// Y 成分を 0 にした、床面と平行な進行方向を得る
vecF = camera.getVec();
vecF.y = 0.0;
```

```
    vecF.normalize();
    // 内積を求め、逆三角関数を使って角度（ラジアン）
を得る
    cosT = vecF*camera.getVec();
    rad = acos(cosT);
```

これで得た角度に応じて条件分岐でロックをかけてやればいいでしょう。何度オーバーしているかが分かるので、オーバーした分だけ戻してあげればいいのです。

10.4 今期の課題について

課題の提出についてですが、まとめてどーんと受け付けることにします。今期に出した (出す予定の) 課題は以下の通りです。

- (第 6 回) マテリアルを使った画面切り替えエフェクト
- (第 8 回) ベクトル演算による追跡 AI
- (第 9 回) モデル座標系と行列による変換・確認課題
- (第 11 回) 三次元幾何要素表現と交差交線計算・確認課題

2 年生にはこれらを解いて出してもらい、評価の基準とします。3 年生に関しては、コードレビューに参加してくれたチームには一定のボーナス点を積み増しした上で、前期終了時点での制作物 (α 版、ということになりますかね？) をもって評価とします。もちろんこれらの課題を提出したい場合は受け取って、評価対象に加えますので余裕のある人はどうぞ。

提出期限は 7/31 23:59 とします。早めに手をつけておくとよいでしょう。