Unity3D を実用する

@Eurfi_Typhoon

はじめに

この記事は、Unity Technologies 社のゲーム開発用ミドルウェア、Unity の実用に関するものです。
スクリプトは UnityScript で記述しています。コンポーネントの先頭に this.をつけていることがありますが、どのオブジェクトのコンポーネントを指しているのか明確にするためであり、必須ではありません。

UnityScript How-To

紫色は Perl 正規表現であり、そのままでは動作しないことにご留意ください。

◆ GameObject を移動する

非物理的制御

```
1.  ObjectA.transform.Translate (x 移動量,y 移動量,z 移動量);
```

ObjectA の position に括弧内の値を加算します。相対的な移動です。呼び出されるたびに加算が繰り返されます。
上記記法は単純なので記載していますが、公式ドキュメントでは避けるよう書かれています[1]。Unity において、この記述は呼び出されるごとに ObjectA にアタッチされた transform コンポーネントを探す必要があり、コンポーネントを Awake 関数内でキャッシュした時に比べオーバーヘッドがかなり増大するからです。

1

```
private var MoveTarget : Transform;
function Awake(){ MoveTarget = ObjectA.transform; }
function Update(){ MoveTarget.Translate (x value,y value,z value);}
```

が推奨されています

```
2.  ObjectA.transform.position.[xyz] = [xyz]ワールド座標;
```

ObjectA の position に右項を代入します。指定した座標に ObjectA を配置するというのがイメージとして正しいと思います。s/=/+=/すると Translate と同じ意味になります。s/=/*=/とかすると ObjectA が吹っ飛んでいきます。たぶん推奨されないし扱いづらいであろう等加速度運動です。

```
3.  ObjectA.transform.localPosition.[xyz] = [xyz]ローカル座標;
```

ObjectA の localPosition[2]に右項を代入します。transform.position との違いは、親子関係のある GameObject の子にこのスクリプトを反映すると、親との距離が変化する点です。

```
4.  ObjectA.transform.position = ObjectB.transform.position;
```

ObjectA の position に ObjectB の position を代入します。xyz の成分ごとにごにょごにょ加算することで追従する感じになります。親子関係をつくると制御上よろしくないときに使えるのではないかと思います。

物理的制御

物理演算を行うので、処理負荷は大きくなります。また、Rigidbody コンポーネントが必要です。

```
5.  ObjectA.rigidbody.velocity.[xyz] = [xyz]移動量
```

これは非物理的制御と同様の動き方です。

```
6.  ObjectA.rigidbody.AddForce (“Force|Accelation|Impulse|VelocityChange”)
```

これは ObjectA に対して継続した力|継続した加速度|瞬間的な力|瞬間的な速度変化を与えるものです。

[2]A(1,0,0)が親で B(3,1,2)が子である時、B の localPosition は(2,1,2)

◆　GameObject を回転する

基本的に移動したい時と同じですが、回転においてはオイラー角とクォータニオンという 2 つのアプローチがあり、最初の難関となるであろう部分です。一般的にオイラー角、すなわち通常の“角度”で表現するのがスクリプトを見直した時にわかりやすいですが、Update 関数の中でオイラー角を指定して回転させると毎フレームごとにオイラー角→クォータニオン変換が行われるため、連続したアニメーションには適していないとされます。Unity におけるオイラー角での回転処理では、このクォータニオン変換で生じるオーバーヘッドと引き換えにジンバルロックを防いでいます。

```
1.  transform.Rotate (Vector3(x 回転量,y 回転量,z 回転量),座標系);
```

```
2.  transform.rotation =
    Quaternion.Slerp (transform.rotation,
      Quaternion.Euler(x value,y value,z value), float 型);
```

Slerp 以降の引数は(回転前の角度,回転後の角度,定数(下図))と定義されています。変更する割合を定数で定めているわけです。

これが意味を持つのは、Time.deltaTime の関数として最後の引数を定義した時です。ただ、Time.deltaTime * 0.5 としたとき、2 秒後に回転が終了してくれると嬉しい**のですが**、どんどん回転速度が遅くなって動作が遅れがちなので実際には 10 秒くらいかかります。扱いづらいです。ひょっとすると、対数のような動き方をするのかもしれません。すなわち、Time.deltaTime * 0.5 では 2 秒ごとに残りの回転量が 1/2 になるのかもしれません。動作させて検証したところ、そんな印象を受けます。

ちなみに、Slerp とは Spherical linear interpolation(球面線形補間)の略で、クォータニオンを扱う際の補間方法です。

◆ GameObject 関連

```
1.  Destroy(ObjectA.gameObject);
```

ObjectA という GameObject を削除します。第二引数で消滅するまでの秒数を指定出来ます(省略可能)。

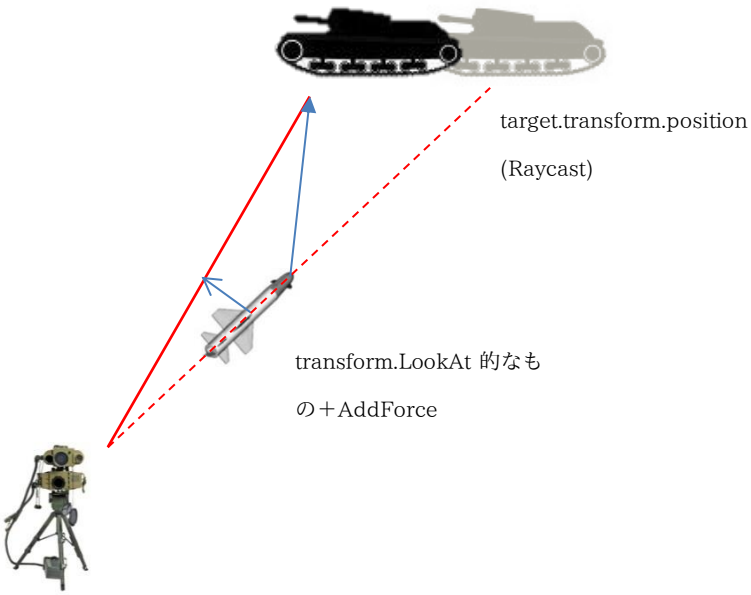
```
2.  var ObjectA : GameObject = Instantiate
      (Prefab name, position value, rotation value) as GameObject;
```

Prefab をインスタンスとして生成します。
生成された GameObject の名前は *Prefab 名(Clone)*となります。

Unity3D で作る FPS

パソコン部は本来自分でプログラムを組んだり絵を描いたりする部活ですので、ゲームを作る際もこのような“レベルエディタ”は使わないのが通例ですが、私は DirectX の初期化をやってのけてシェーダを一から書ける人間ではありませんし、新入部員の成果展示ではないのでどれだけ美しく面白いものを作れるかが重要だと考えています。それゆえ(レベルエディタとしては比較的大きい)自由度と無料であるという点から Unity を選択した次第です。
さて FPS では欠かせない弾の発射について、様々な手法での実装とそのメリット・デメリットについて検討していきたいと思います。
弾を発射するということになると、まず原始的な方法として考えられるのが当たり判定を持った GameObject が transform.translate で動いていくというものです。
この方法では、弾種ごとに Inspector から速度を調整する必要がありますが素朴でわかりやすいです。

次に考えられるのがレイキャスト（Ray を cast する)です。
レイキャストの利点はレイの延長上のオブジェクトの判定ができることです。例えば指令誘導式のミサイルを制御するには最適です。
レイキャストの難点は公式リファレンスでも指摘されている通り処理が遅いこと、そしてレイの起点によっては壁を貫通すること、低速の無誘導弾等には向いていない[3]ことです。また、反動を再現するのは厳しいです。というのもレイキャストは始点と終点だけで判断するので、ランダムに終点の位置を変えるにも銃口は一定の方向に固定されているわけではないからです。親子関係を持った transform を使えばできますが、どうも使い勝手が悪いです。
AddForce("Force")などを使うことによって、物理的に正しい銃弾の挙動が再現でき、また跳弾等も行えます。ですが弾丸それぞれの弾頭重量、バレルの長さなど（あくまでこだわるとすれば）細かい調査が必要となるという点で一番現実での準備が大変です。
紙面の都合上、ここで SACLOS(半自動指令照準線一致誘導方式)の実装案を載せることとします。



[3] Rigidbody を適用した弾頭の 3D モデルを AddForce("Force");が最適?

◆　指定したGameObjectの方向を瞬時に向く

```
#pragma strict
var target : Transform;

function Start(){
  target = GameObject.Find("ObjectA").transform

function Update(){
  this.transform.LookAt(target.position);
}
```

◆　指定したGameObjectの方向をゆっくり向く

```
#pragma strict
var target : Transform;
var TraceSpeed : float;

function Start () {
  target = GameObject.Find("ObjectA").transform;
}

function Update () {
   this.transform.rotation = Quaternion.Slerp(transform.rotation,
    Quaternion.LookRotation(target.position - this.transform.position),
      TraceSpeed );
}
```

このクォータニオン関連では、納得行く回転を実現するには回転速度を決定する値と実際の動作の対応を確かめるほかありません。「仕様です」。

◆　警備 AI

ここでは、視界内に敵が入ってくると気づき、特定の処理を行うという単純な AI を考えたいと思います。

私がこれを実装するために簡単に考えたのは

- 視界といってもレイキャストを端から端まで空間的にはできない
- 視界を Sphere Collider にすることによってそもそも視程内にいるかを判定し、コライダ接触+レイキャスト貫通で初めて感知(レイキャストの範囲を制限することによる仮想的な視界)
- 視界の外周なら瞬時に反応するとは限らない

```
#pragma strict
var NoticeRate : float;
private var PlayerDirection : Vector3;
private var r : float;

function OnCollisionStay(detection : Collision){
  if (detection.gameObject.tag == "Player"){
    PlayerDirection =
     Vector3.Normalize(detection.gameObject.transform.position
     - this.transform.position);
    if (PlayerDirection.z >= 0.0f){
      ViewJudge();
    }
  }
}
```

```
function ViewJudge(){
  if(Physics.Raycast(this.transform.position,detection.gameObject.tra
  nsform.position,ViewTry))
  {
    ViewTryCheck();
  }
}

function ViewTryCheck(){
  if(ViewTry.collider.gameObject.tag == "Player"){
    r = Random.value;
    if (r < NoticeRate || 0.5f < PlayerDirection.z){
      //気づいた！処理(略)
    }
  }
}
```

内容としては、最大視野を Collider で設定し、その中に Player タグのオブジェクトが入っている間(OnCollisionStay)、方向ベクトル(0,0,1)と、AI の位置を始点としたプレイヤーの方向ベクトルを正規化したものの内積を求め(この場合正規化後のプレイヤーへの方向ベクトルの z 成分がそのまま内積となる)、

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos\theta$$

において z 成分 $= 1 \cdot 1 \cdot \cos\theta$ となり $z = \cos\theta$ であるため、$z \geqq 0$ として半球状の視野を実装し、その中にプレイヤーがいれば視線が貫通するかチェック(ViewJudge)するためレイキャストを行う。その結果視線が貫通すれば(ViewTryCheck)、その発見時角度が上下左右各 60°($\cos\theta = 1/2$)以内の時は必ず発見する。視野の外周部では 0〜1 の小数乱数を生成し発見されるかの判定を行う。というものです。

Unity の Web 連携

Unity で JSON を用いて通信を行う方法について解説します。

```
var ValidationForm : WWWForm = new WWWForm();
function Validation_toka_tekito(){
  ValidationForm.AddField(“serialcode”, “ABCD1234abcd”);
  var ServerSend : WWW = new WWW(”http://example.com" , ValidationForm);
  yield ServerSend;
  CheckResult;
}

function CheckResult(){
  var Result : JSONObject = new JSONObject(ServerSend.text);
  var Confirmation : JSONObject = Result.GetField("ValidResult”);
    if(Confirmation == "OK”){
      SuccessProcess();
    }
    else{
      failProcess();
    }
}
```

WWWForm とは、送るデータファイルです。

WWW という命令は、第 1 引数に URL、第 2 引数に送るデータの変数名を入れて送信します。yield でレスポンスを待機します。ServerSend の結果をテキストとして読み込み、ValidResult というキーの値を取得し判定するというものです。

最後に

Unityは3Dゲーム用ミドルウェアとしてはとても汎用性が高いものです。ゲームでなくとも、2D スプライトを用いてアプリケーションを作ることもできるところにその一端が表れています。
単純なゲームを作るだけであれば既に用意されたスクリプトを適用するだけで出来るでしょうし、複雑なゲームにしても実現したいことを達成するための機能・属性が準備されています。
とはいうものの、ゲームを構成するシステムがよくできていても、肝心の3D 世界のビジュアルが残念では興ざめであります。今回は 3D モデルには触れませんでしたが、模型のように各パーツを精巧に作っていくという手法で機械類は美しく再現できるのではないかと思います。
人間を作るのは私には無理でした。多分これが作れる人は絵として人間を描くこともできると思います。2D で描けないものは 3D で作れないのです…

ともかくも稚拙なコードと文章をお読みいただき、ありがとうございました。

(暗視装置風のエフェクトを適用した Unity シーンのキャプチャ)