Unity メモ

Unity とは,

ゲームの統合開発環境で，複数のプラットホームに対応するゲームエンジンでもある。Unity では Scene に GameObject を配置して scene graph を構築してゲームを作る。GameObject は，Transform, RigidBody, Collider，Script などの Component（構成要素）を持つ。

フレームとは描画処理（rendering）によって描画された 1 画面のことで，Unity エンジンは 1 秒間に 30 ～ 60 枚程度のフレームを構築し，フレームごとにゲームオブジェクトに設定されているスクリプトの Update() メソッドが呼び出され，Scene Graph を更新し rendering する。

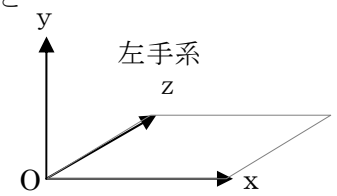
また，Unity の物理演算は描画に関係なく固定の interval（Fixed Timestep，デフォルトでは 0.02 秒，50fps）で計算され，FixedUpdate() が呼び出される。例えば Rigidbody に力を加える場合には，Update ではなく，毎フレームごとの FixedUpdate の中で力を適用する必要がある。なお，Update を最後に呼び出した時からの経過時間を取得するには Time.deltaTime を使用する。

```
Script サンプル：Update 毎秒 10m 移動
void Update() {
        float dz = Time.deltaTime * 10;
        transform.Translate(0, 0, dz);
}
```

対応済みプラットホーム

Web Browser, Android, iOS, Windows Phone 8, Windows, Mac OS X など

Unity の座標系：左手系



World 座標：Unity の 3 次元空間座標　X, Y, Z

Local 座標：親オブジェクトを原点とした座標

Screen 座標：ディスプレイの解像度と一致する。

マウスの座標はスクリーン座標で所得される。

Viewport 座標 : Screen 座標を 0～1 に規格化した座標で、マルチ解像度に対応するのみ便利である。

Camera Space :

Projection (Normalized Device) Space :

Scene Graph

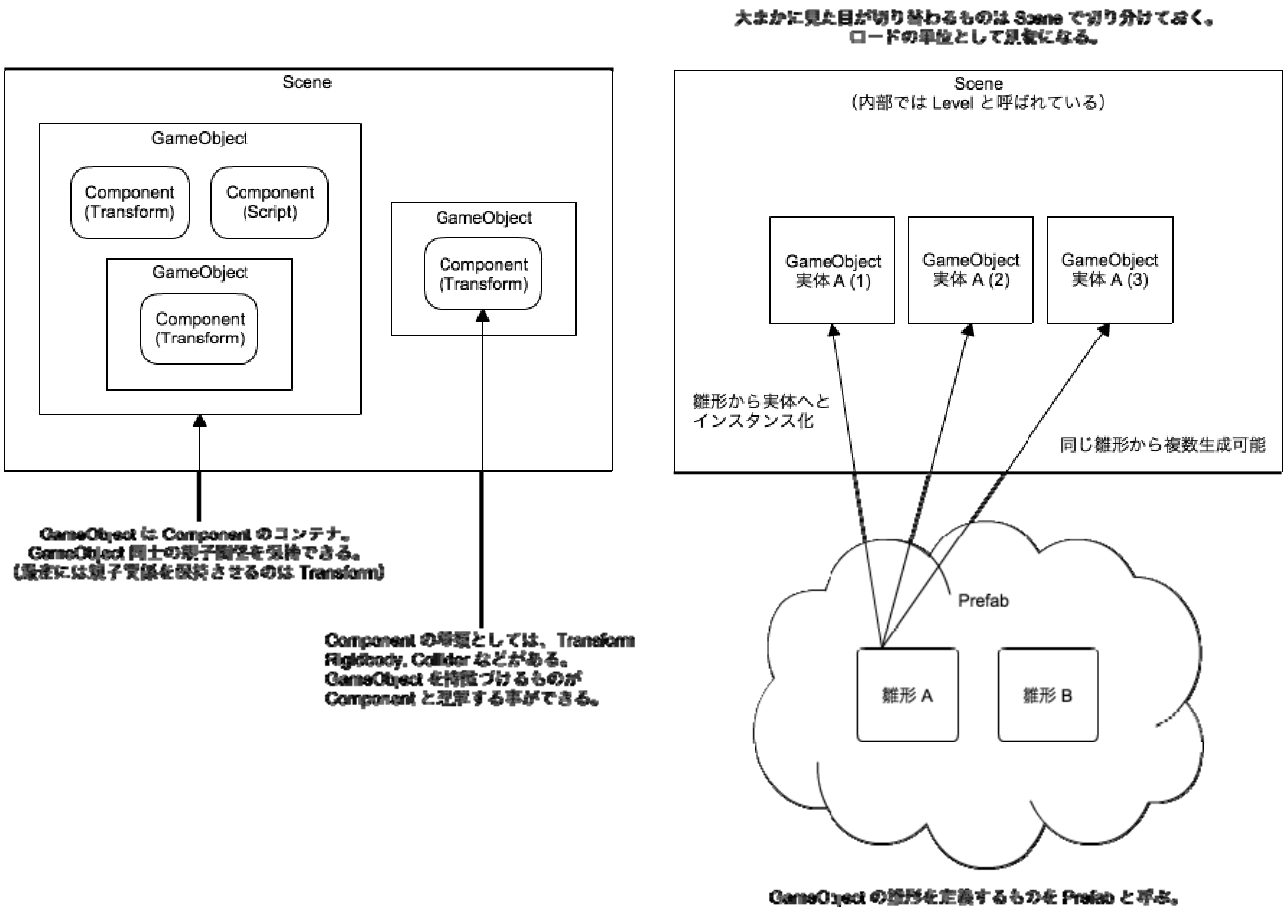
シーングラフはベクトルベースのグラフィックスでよく使われているデータ構造で，ゲームエンジンの多くはシーングラフを実装している。シーングラフは木構造(tree)で，scene を表現する。

Dr.JIRO Software

https://www.drjiro.com/game-engine/cocos2dx/cocos2d-x-scenegraph/ (2016.7.31)

Tips: Unity API Help ショートカット　Ctrl + ’

}



↑http://developers.mobage.jp/blog/true-unity-course

Start　初期化

Update　フレーム毎に呼び出される

Instantiate

GameObject

　　Inspector で Tag で命名すると Tag 名で UnityScript で参照できる。

Component：GameObject にアタッチして GameObject を構成し制御する。

　メソッド

　　Component GetComponent(Type type);

　　T GetComponent<T>();

　　Component GetComponent(string type);

　ジェネリック関数（http://docs.unity3d.com/jp/current/Manual/GenericFunctions.html）

　　スクリプトリファレンスのいくつかの関数（例えば，さまざまな GetComponent 関数） は T または型名を関数名の後に <> 記号で記されている変種があります:

```
//C#
void FuncName<T>();
//JS
function FuncName.<T>(): T;
```

これらはジェネリック関数として知られています。これらのスクリプティングにおける重要性は パラメーターの型かつ（または）関数の戻り値の型を指定できることです。JavaScript では， ダイナミックタイピングの制約を回避するために使用することができます:

```
// The type is correctly inferred since it is defined in the function call.
//In C#
var obj = GetComponent<Rigidbody>();
//In JS
var obj = GetComponent.<Rigidbody>();
```

C# では，多くの文字入力やキャストを節約できます:

```
Rigidbody rb = go.GetComponent<Rigidbody>();

// ...as compared with:

Rigidbody rb = (Rigidbody) go.GetComponent(typeof(Rigidbody));
```

スクリプトリファレンスに書かれている，ジェネリックの変種をもった関数は，すべてこの特別なコールの シンタックスを使用することができます。

Generics : http://www.buildinsider.net/language/tsgeneric/01
型引数 f<T>(x) f(x: T)

・Transform

変数

transform：GameObject にアタッチされた Transform を参照する変数
position：ワールド空間の位置
localPosition：親の Transform から見た相対位置
localScale：親の Transform から見た相対的スケール
rotation：ワールド空間での回転
Quaternion.AngleAxis を使うと，ある軸を中心にした回転を表現することができる。
例 1 ：Y 軸周りに秒間 120 度の角速度で回転

```
function Update () {
    transform.rotation = Quaternion.AngleAxis(Time.time * 120.0, Vector3(0, 1, 0));
}
```

例 2：「Y 軸周りに毎秒 120 度の回転」と「X 軸周りに毎秒 77 度の回転」を合成

```
function Update () {
    transform.rotation = Quaternion.AngleAxis(Time.time * 120.0, Vector3(0, 1, 0)) *
                         Quaternion.AngleAxis(Time.time * 77.0,  Vector3(1, 0, 0));
}
```

localRotation：親の Transform から見た相対的回転

関数

Translate(x, y, z)

public void Translate(Vector3 translation, Space relativeTo = Space.Self);

```
void Update() {
        transform.Translate(0, 0, Time.deltaTime);
        transform.Translate(0, Time.deltaTime, 0, Space.World);
    }
```

Rotate(x,y,z)　角度は度

public void Rotate(Vector3 eulerAngles, Space relativeTo = Space.Self);

```
void Update()
    {
        // Rotate the object around its local Y axis at 1 degree per second
        transform.Rotate(Time.deltaTime, 0, 0);

        // ...also rotate around the World's Y axis
        transform.Rotate(0, Time.deltaTime, 0, Space.World);
    }
```

transform.RotateAround(Vector3.zero, Vector3.up, 20 * Time.deltaTime);

・Rigidbody　　**var rb：Rigidbody = GetComponet<Rigidbody>();**

変数

mass, position, velocity, rotation, angularVelocity

関数

AddForce, AddForceAtPosition, GetPointVeleocity

・UnityScript　JavaScript を参考にした Unity 独自言語。Component の１つとして，GameObject にアタッチできる。

UnityScript

スクリプト名がクラス名となる。

クラス名：大文字で始まりラクダ記法

GameObject, Component, Rigidbody

関数名　大文字で始まりラクダ記法

OnStart，OnUpdate，　OnCollisionEnter，OnTriggerEnter,

変数名，プロパティ名：小文字で始まりラクダ記法

col.gameObject.name = "Cube"

変数宣言

アクセス修飾子 public, private, static

private 宣言しない限り public 扱いとなり，インスペクターに表示される。

static 宣言した変数は，グローバル（クラス）変数となり，

スクリプト名（クラス名）.グローバル変数名

でどこからでもアクセスできる。

：の後ろに型を明記

```
static var time : float;
public static var goal : boolean;
public var sceneName : String;
public var text : Text;
private var invisibleVar: int = 5
private var xfloat: float = 5.5f;
var playerGO : GameObject;
var playerGOTransform : Transform;
```

Input.GetKey(KeyCode.Space)　Space はなぜ大文字か？KeyCode enum parameter

数学関数

Mathf.　PI
Deg2Rad
Rad2Deg
Sin, Cos, Tan, Asin, Acos, Atan, Atan2
角度は rad

Time

Time.deltaTime 前フレームからの経過時間

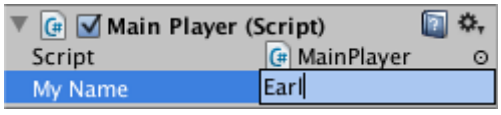
```
function UpDate(){          //Update は一定時間間隔で呼ばれない
    transform.Rotate( 0, 10*Time.deltaTime, 0 );          //角速度一定の回転
}
```

コンソールに出力　Debug.Log();

Input.GetKey(KeyCode.Space)

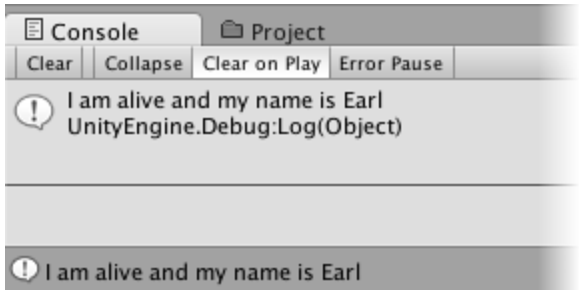
```
private var myObj:GameObject;
function Start(){
    myObj = GamaObject.Fing("Capsule");
}
```

#pragma strict　//strict モードで実行(変数等の曖昧な記述法は禁止)




```
var rb : Rigidbody;          // : 後ろに型を明記
var myName : string;
public var player : GameObject;    //  アクセス修飾子 public を指定すると，inspector に表示される。
static 指定すると，グローバル変数となる。

//オブジェクトがシーンに表示される際に実行
function Start () {
    rb = GetComponet.<Rigidbody>();
    Debug.Log("I am alive and my name is " + myName);
```



```
        var rb = GetComponent.<Rigidbody>();
        rb.mass = 10f;
        transform.position = player.transform.position - Vector3.forward * 10f;
}

// フレームの切り替えの際に実行
function Update () {
        Debug.Log(rb.velocity);
}
```

を書き換える

```
//トリガーとなる当たり判定時に呼ばれる
//   Componet > Collider > Is Trigger を☑
function onTriggerEnter(col:Collider){
        if(col.gameObject.name == "" ){
                Debug.Log("衝突！");
        }
}
```

Unity では変数名に大文字がある場合，スペースを足してインスペクターのラベル名とする。この場合は，My Name がラベル名となるが，コード内では必ず元の変数名を使う必要がある。インスペクターに My Name とラベル表示され，編集可能なテキストフィールドに入力し値を設定できる。また，UnityScript では変数は private 宣言しない限り，デフォルトで public となる。


```
UI.Text
public var myText : UI.Text;        //inspector で表示されるプロパティ myText に Text を紐付け

duncton OnGUI(){
        myText = "Change";
}
```

Component の参照

コンポーネントはクラスのインスタンスであるので，アクセスするにはコンポーネントインスタンスの参照を取得します。これには GetComponent 関数を使います。

他の GameObject へのアクセス

ゲームオブジェクトを関連付けるもっとも分かりやすい方法は public であるゲームオブジェクト変数をスクリプトに追加することです。シーンまたは階層パネルからオブジェクトをこの変数の上にドラッグして割り当てる。もしスクリプトの中でコンポーネント型の public である変数を宣言すると，コンポーネントがアタッチされている任意のゲームオブジェクトをその上にドラッグできます。（var playerTransform : Transform ;）

名前またはタグでオブジェクトを見つける

名前を指定して個別オブジェクトを取得するには GameObject.Find 関数を使用します:

```
GameObject player;

void Start() {
        player = GameObject.Find("MainHeroCharacter");
}
```

オブジェクトまたはその集合をタグを使用して見つけるには GameObject.FindWithTag，GameObject.FindGameObjectsWithTag 関数を使用します:

```
GameObject player;
GameObject[] enemies;

void Start() {
    player = GameObject.FindWithTag("Player");
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
}
```

イベント関数

```
//物理エンジンが物理挙動の更新をする直前
void FixedUpdate() {
    Vector3 force = transform.forward * driveForce * Input.GetAxis("Vertical");
    rigidbody.AddForce(force);
}

//Update と FixedUpdate がシーン上の全てのオブジェクトに対して呼び出されて，全てのアニメーションが計算された後のタイミング
void LateUpdate() {
    Camera.main.transform.LookAt(target.transform);
}

//物理挙動イベント
void OnCollisionEnter(otherObj: Collision) {
    if (otherObj.tag == "Arrow") {
        ApplyDamage(10);
    }
}
```

オブジェクト

Transform

基本的にトランスフォームコンポーネントでスケール(Scale)を調整するべきではありません。実世界のサイズでモデルを作る最も良い方法は，トランスフォームのスケールを変更しないことです。２番目に良い方法は，インポート設定でインポートされたメッシュのスケールを調整することです。インポートサイズに応じて適切な最適化がなされます。スケール値が調整されたオブジェクトのインスタンス化はパフォーマンスを低下させます。スケールの最適化に関するさらに詳しい情報はリジッドボディ(Rigidbody)コンポーネントのセクションを参照してください。

親子関係のあるトランスフォームでは，子を設定する前に親の位置を 0,0,0 にします。あとで頭の痛い思いをせずに済むでしょう。

親子関係の設定(Parenting)

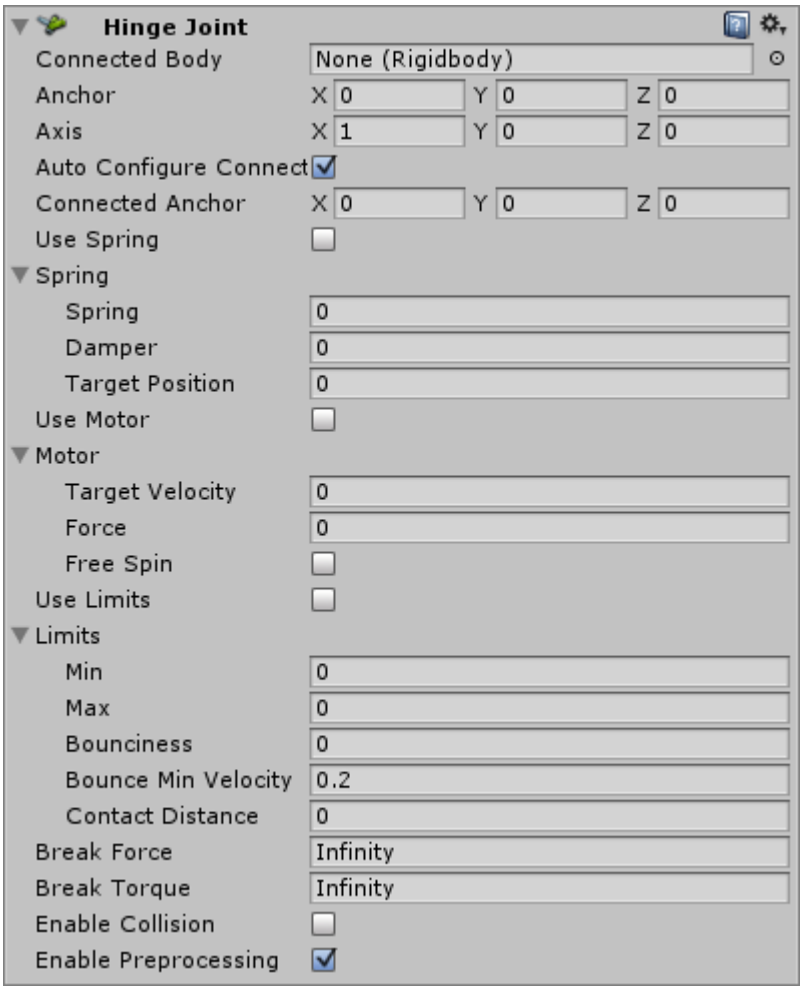
Unity では親子関係が設定できます。ゲームオブジェクトを他のオブジェクトの子供とするには，階層において，子オブジェクトを親にしたいオブジェクトの上にドラッグします。子供は親の移動や回転を受け継ぎます。親オブジェクトを開く(閉じる)ことで，ゲームに影響を与えずに親子関係を確認できます。

Physic Material

プロパティー:	*説明:*
Dynamic Friction	運動摩擦係数：通常は、0 から 1 の間の値を使用。0 の場合、氷のような感じになり，1 の場合、大きな力や重力がオブジェクトを押さない限り、素早く停止します。
Static Friction	静止摩擦係数：通常は、0 から 1 の間の値を使用。0 の場合、氷のような感じになります。1 の場合、強い力を加えないとオブジェクトは動きません。
Bounciness	反発係数：0 の場合、跳ね返りません。1 の場合はエネルギー損失なしで跳ね返ります。
Friction Combine	衝突するオブジェクト間の摩擦をどう処理するか。
- Average	2 つの摩擦力が平均化されます。
- Minimum	2 つの摩擦力のうち小さい方の値が使用されます。
- Maximum	2 つの摩擦力のうち大きい方の値が使用されます。
- Multiply	2 つの摩擦力が互いに乗算されます。
Bounce Combine	衝突するオブジェクト間の跳ね返し度合いをどう処理するか。Friction Combine と同じです。

ヒンジジョイント

`Hinge Joint` は，2 つの Rigidbodies をグループ化し，互いにヒンジで連結されているかのように動くよう制約します。ドアに最適ですが，鎖や振り子などをモデル化するのにも使用できます。



プロパティー

プロパティー:	*説明:*
`Connected Body`	ジョイントが依存するリジッドボディへのオプションの参照。設定しないと，ジョイントはワールドに接続します。
`Anchor`	ボディが揺れる中心となる軸の位置。この位置はローカルなスペースで定義されます。
`Axis`	ボディが揺れる中心となる軸の方向。この方向はローカルなスペースで定義されます。
`Auto Configure`	有効の場合，接続されたアンカーの位置が，アンカーのプロパティにあるグローバ

プロパティー:	*説明:*
Connected Anchor	ルポジションと一致するように，自動的に計算されます。これは，デフォルトの挙動です。無効にした場合，接続されたアンカーの位置を，手動で調整する事ができます。
Connected Anchor	接続されたアンカーポジションの手動設定
Use Spring	スプリングは，リジッドボディをその連結されたボディと比較して，一定の角度に到達させます。
Spring	Use Spring を有効にした場合に使用されるスプリングのプロパティー。
Spring	オブジェクトが前述の位置に移動するのに出す力。
Damper	この値が高いほど，オブジェクトの速度は低下します。
Target Position	スプリングの対象角度。スプリングは度で測定されたこの角度に向けて引っ張られます。
Use Motor	モーターはオブジェクトを回転させます。
Motor	Use Motor を有効にした場合に使用されるモーターのプロパティー。
Target Velocity	オブジェクトが達成しようとする速度。
Force	オブジェクトが前述の速度を達成するのに適用される力。
Free Spin	有効にすると，モーターは回転にブレーキをかけるのに使用されず，加速にのみ使用されます。
Use Limits	有効にすると，Min と Max 値内にヒンジの角度が制限されます。
Limits	Use Limits を有効にした場合に使用される制限のプロパティー。
Min	回転が到達できる最低角度。

プロパティー:	*説明:*
`Max`	回転が到達できる最高角度。
`Bounciness`	最小または最大の停止に到達した際にオブジェクトが跳ね返る量。
`Contact Distance`	接触する距離内（ジョイントの位置から制限値まで）で，ジッター（Jitter）を回避するためリミットの効力が続く距離
`Break Force`	このジョイントが分解するのに適用される必要のある力。
`Break Torque`	`Break Torque` このジョイントが分解するのに適用される必要のあるトルク。
`Enable Collision`	このジョイントが分解するのに適用される必要のあるトルク。
`Enable Preprocessing`	無効にすることで安定して不可能なことを可能にするような設定を行うことができます。

詳細

1 つのヒンジを `GameObject` に適用する必要があります。このヒンジは，*Anchor* プロパティーで指定した点で回転し，指定した *Axis* プロパティー周辺で移動します。ジョイントの *Connected Body* プロパティーに GameObject を割り当てる必要は **ありません** 。ジョイントの `Transform` を追加したオブジェクトのトランスフォームに依存させたい場合にのみ，GameObject を *Connected Body* プロパティーに割り当てる必要があります。

ドアのヒンジがどのように機能するかを考えましょう。この場合の *Axis* は上で，Y 軸に沿って正になります。*Anchor* は，ドアと壁の間の交差部のどこかに置かれます。ジョイントは，デフォルトでワールドに連結されるので，壁を *Connected Body* に割当てる必要はありません。

次は，ドギードアのヒンジについて考えましょう。ドギードアの *Axis* は横で，相対的な X 軸に沿って正になります。メインドアを *Connected Body* に割当てる必要があるため，ドギードアのヒンジは，メインドアのリジッドボディに依存します。

鎖

複数のヒンジジョイントを連結して，鎖を作成することもできます。鎖の各連結部分にジョイントを追加して，次の連結部を *Connected Body* として追加します。

ヒント

- *Connected Body* が機能するよう，ジョイントに割り当てる必要はありません。
- 動的なダメージシステムを作成するには，*Break Force* を使用します。プレイヤーが，ロケットランチャーで爆破する，あるいは車で突っ込んで，ドアとヒンジを切り離すことができるので非常に便利です。
- *Spring*，*Motor* および *Limits* プロパティーにより，ジョイントの動作を微調整できます。
- `Spring` と `Motor` は，意図して同時に使えないようになっています。両方を同時に使うと，予測できない結果になるためです。

オブジェクトを階層化(親子関係の設定)をすれば，簡単に任意のオブジェクトに追従させることができます。しかし，階層化すると親オブジェクトのスケールや回転も同時に影響を受けてしまいます。

目標とするターゲットと指定した距離を保って追従するスクリプトを作成しました。

Follow.cs

```
using UnityEngine;
using System.Collections;

public class Follow : MonoBehaviour {

	public GameObject objTarget;
	public Vector3 offset;

	void Start () {
		updatePostion();
	}

	void LateUpdate () {
		updatePostion();
	}

	void updatePostion()
	{
		Vector3 pos = objTarget.transform.localPosition;

		transform.localPosition = pos + offset;
	}
}
```

Inspector で目標とするオブジェクトと，保つ距離を指定します。
LateUpdate()を使っているのは，目標とするオブジェクトの更新が終わった後で位置を更新したい為です。
※このスクリプトは，目標とするオブジェクトの localPosition を参照して追従します。スクリプトを適用するオブジェクトと目標とするオブジェクトが同一の座標系(階層)にあり，目標とするオブジェクト自体が移動した場合にのみ動作します。

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour {
    void Update() {
        if (Input.GetKey("up"))
            print("up arrow key is held down");

        if (Input.GetKey("down"))
            print("down arrow key is held down");

    }
}
```

```
function  Update(){

        if(Input.GetKey(KeyCode.A)){
                transform.localScale.y += 0.1f;
        }

}
```

```
public var prefab : GameObject;
public var power : float;

function Update(){
        if(Input.GetMouseButtonDown(0)){
                var bullet = LoadBullet();
                var ray : Ray = Camera.main.ScreenPointToRay(Input.mousePosition);
                var dir : Vector3 = ray.direction.normalized;
                bullet.GetComponent.<Rigidbody>().velocity = dir*power;
        }
}

function LoadBullet(): GameObject{
        var bullet = GameObject.Instantiate(prefab)as GameObjrct;
        bullet.transform.parent = transform;
        bullet.transform.localPosition = Vector3.zero;
        retun bullet;
}
```

```
public    var power : float = 6.5;
private var target : GameObject;
function Start(){
        target = GameObject.FindGameObjectWithTag("DeathZone");
}
function FixedUpdate(){ //フレーム毎の呼び出し
        var direction : Vector3 = target.transform.position – target.position;
        GetComponent.<Rigidbody>().AddForce(direction.normalized*power);
        transform.LookAt(direction); //GameObject を direction の向きに変える
}
```

```
funcon Start(){
        Destroy(gameObject, 5.0); //5 秒後にスクリプトが設定された GameObject を消去

}
```

加速度センサー

```
var speed = 10.0;


function Update () {
	var dir : Vector3 = Vector3.zero;

	// we assume that the device is held parallel to the ground
	// and the Home button is in the right hand

	// remap the device acceleration axis to game coordinates:
	// 1) XY plane of the device is mapped onto XZ plane
	// 2) rotated 90 degrees around Y axis
	dir.x = -Input.acceleration.y;
	dir.z = Input.acceleration.x;

	// clamp acceleration vector to the unit sphere
	if (dir.sqrMagnitude > 1)
		dir.Normalize();

	// Make it move 10 meters per second instead of 10 meters per frame...
	dir *= Time.deltaTime;

	// Move object
	transform.Translate (dir * speed);
}
```