Unityはじめるよ

～初心者講座～

統合開発環境を内蔵したゲームエンジン
http://japan.unity3d.com/

※いろんな職業の方が見る資料なので説明を簡単にしてある部分があります。正確には本来の意味と違いますが上記理由のためです。ご了承ください。
この資料内の一部の画像、一部の文章はUnity公式サイトから引用しています。

本日のプレゼン内容

初心者向け講座
（Unityの基礎知識とスクリプト制御）

プログラムは触ったことがあるけど、
Unity自体あまり触ったことがない。

という方向けの内容となっています。

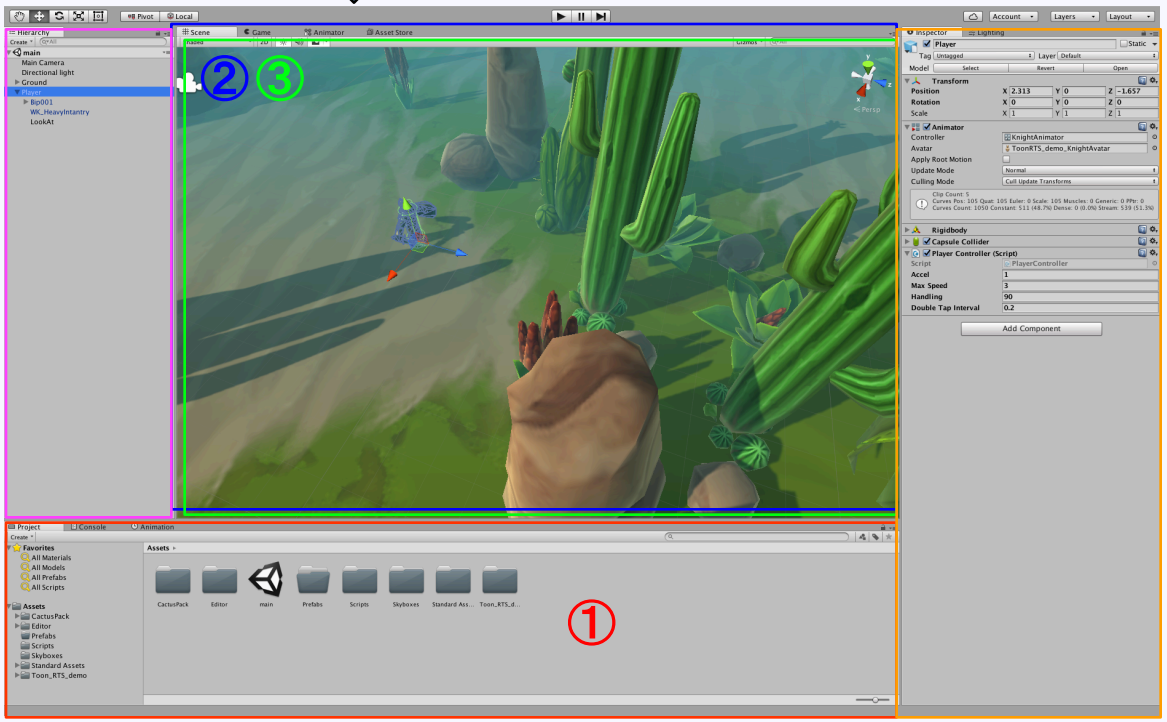
Unityの基礎知識

Unityの特徴

ゲーム開発に便利な機能が盛りだくさん
レンダリング、ライティング 、エフェクト、オーディオ、マテリアル、地形、物理演算 AI、アニメーション、ネットワークなどなど

マルチプラットフォーム対応
簡単な操作で各種プラットフォーム向けのビルドが可能。

画面説明



①プロジェクトブラウザー

作業中のプロジェクトにあるすべてのアセット（素材）を見ることが出来ます。検索やソートも簡単で、沢山のアセットを手軽に管理できる。

②シーンビュー

ゲームを作成するための作業を行うビューです。このビューに対して操作をしながら、ゲームで利用するシーンを作っていく。

③ゲームビュー

対象となるデバイスでどのようにゲームが表示されるかをいつでもエディターから確認できるゲーム表示ビュー。

④インスペクタービュー

シーン上の GameObject やプロジェクト内のアセットのプロパティーを確認したり編集することが出来る。

⑤ヒエラルキービュー

シーン内に含まれるすべての GameObject とその階層を表示する。もちろん、名前や型で素早く絞り込むこともできる。

プロジェクトとシーン

プロジェクト
一つのゲーム開発に必要な全てのデータのまとまり。
Unityの場合一つのフォルダに収まっている。

シーン
大雑把にいうと、タイトル画面、ヘルプ画面、
ステージ１、ステージ２、などの各画面のこと。
各画面には関わり深いオブジェクト同士がまとまっており、そのまとまりをシーンという単位で管理する。
※Unityでは、「.unity」という拡張子のファイルがシーン管理ファイル

一般的にシーンは、複数のオブジェクトで構成され、
カメラとライトも含んでいる。

シーンビュー上でゲームに必要なオブジェクトの
配置を行う。

ここで配置するオブジェクトのことをUnityでは

GameObject（ゲームオブジェクト）

という。

GameObject

Unityの世界では、
・キャラクター
・背景、
・ライト
・カメラ
・UI
などなど、すべてのオブジェクトは
GameObjectでできている。

コンポーネント

ゲームを作成するのに便利な機能を持った
部品(機能)のこと。

様々な種類のコンポーネントが存在し、
GameObjectの見た目や、動きなどの振る舞いを
決定づける。

コンポーネントはGameObjectにアタッチ(紐づけ)
することで、初めて効果を発揮する。

代表的なコンポーネント

・Transform　すべてのGameObjectに備わるコンポーネントで、
位置(Position)、回転(Rotation)、拡大縮小(Scale)の情報を持つ。

・MeshFilter　GameObjectの形状を管理するコンポーネント。

・MeshRenderer　GameObjectを画面に描画するためのコンポーネント。

・Rigidbody　物理演算を行うためのコンポーネント。

・〇〇Collider　あたり判定を制御するコンポーネント。

・スクリプト　自分で作ったスクリプトもコンポーネントの一種である。

MeshFilter

ドラゴン3Dモデル

GameObject

Transform
位置、回転、拡大縮小

スクリプト

火を吐かせる
スクリプト

スクリプト

空を飛ばす
スクリプト

Animator

アニメーション
データ

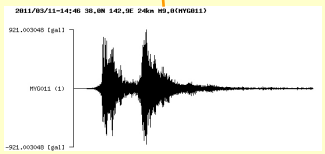
MeshRenderer

ドラゴンマテリアル
（テクスチャ＆質感）

RigidBody

物理演算

SoundManager



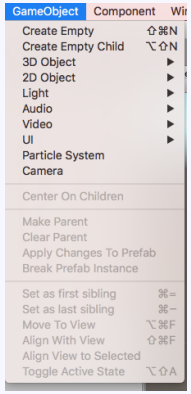
雄たけびサウンド

コンポーネントの振る舞いを Unityで確認してみよう！

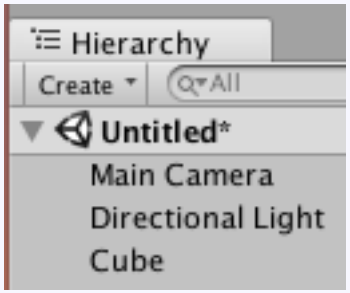
GameObjectの追加

GameObjectをシーンに追加する方法は何種類かある

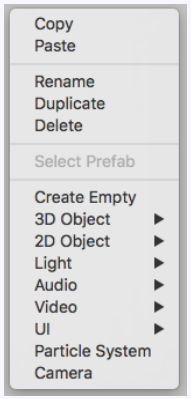
メニューから



ヒエラルキービューの Createボタン



ヒエラルキービューを 右クリック

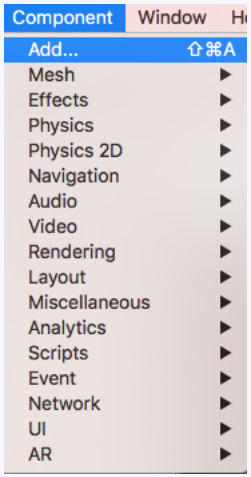


プロジェクトビューからシーンビューやヒエラルキービューにドラッグする方法もある

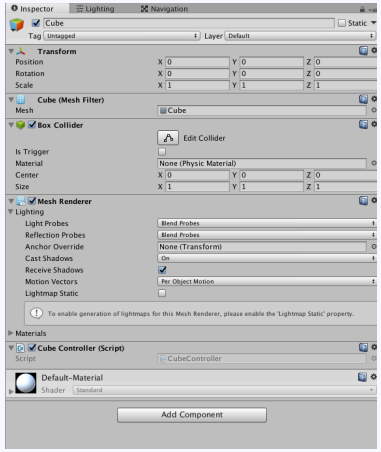
コンポーネントの追加

コンポーネントをGameObjectに追加する方法は何種類かある

メニューから



インスペクタビューの AddComponentボタン



プロジェクトビューからシーンビュー、ヒエラルキービュー、GameObjectにドラッグする方法、インスペクタビューにドラッグする方法がある。

シーンビューでの操作方法

マウスの左ボタン
　クリック　　：　選択（Shift + クリックで追加選択）

マウスホイール
　ドラッグ　　：　カメラの移動
　ホイール回転　：　カメラの拡大縮小

マウス右ボタン
　ドラッグ　　：　カメラの回転

オブジェクト選択状態での操作

画面の左上にある５つのボタン 

ショートカットキー(Q, W, E, R, T)

左から、
・カメラ移動
・選択中オブジェクトの移動
・選択中のオブジェクトの回転
・選択中オブジェクトの拡大縮小
・選択中オブジェクトの拡大縮小２

その横にある２つのボタン 

ショートカットキー(Z, X)

・回転や拡大縮小の基準点の切り替え
・ワールド座標／ローカル座標の切り替え

ショートカットキー

Command(Ctrl) + C　コピー
Command(Ctrl) + V　ペースト
Command(Ctrl) + D　複製
Command(Ctrl) + S　シーンを保存

の切り替え(Q, W, E, R, T)

の切り替え(Z, X)

スクリプト制御

スクリプトからGameObjectを操作する方法を解説します。

基本的な考え方は、スクリプトから、

「GameObject」にアタッチされている
コンポーネントを制御する

ということ。

スクリプト基礎知識(C#)

■スクリプトはコンポーネント
スクリプトはコンポーネントの一種であるため、GameObjectにアタッチ(紐づけ)しないと使えない。
※正確には「MonoBehaviourを継承したクラス」の場合は。

■ファイル名とクラス名は同一でなければならない
一度スクリプトファイルを作った後に、ファイル名を変える場合は、ソース内のクラス名をファイル名と同じにしておく。

■スクリプトファイルを作ったときに、初めから記述してある「Start()」「Update()」関数(メソッド)について
Start()　スクリプトがロードされたタイミングで一度だけ実行される関数
Update()　毎フレーム呼ばれる関数

■オーバーライド関数
決められた関数名をスクリプトファイル内に記述することで、決められた動作を行ってくれる。
例、OnCollisionEnter(あたり判定用)
FixedUpdate(物理演算用のUpdate関数)
OnMouseDown(マウス入力判定)などなど

■C#スクリプトの構成

```
using UnityEngine;
using System.Collections;

public class NewBehaviourScript : MonoBehaviour {

    public float TimeLimit = 30.0f;
    float m_timeLimitCount = 0f;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }

}
```

クラス

/// 制限時間
/// 制限時間カウント用

publicをつけた変数はエディタ上に表示される。
publicをつけない場合はprivateとなり、エディタ上に表示されない。

メンバ変数　クラス内のどこからでも触れる変数

関数　publicをつければ他のスクリプトからも呼び出すことができる

スクリプトの実行順について

他の開発環境からUnityに移ったときに戸惑うのは、

main関数

的なモノがないこと。

どこからプログラムが動き出して、どういう順番に制御されるのか。

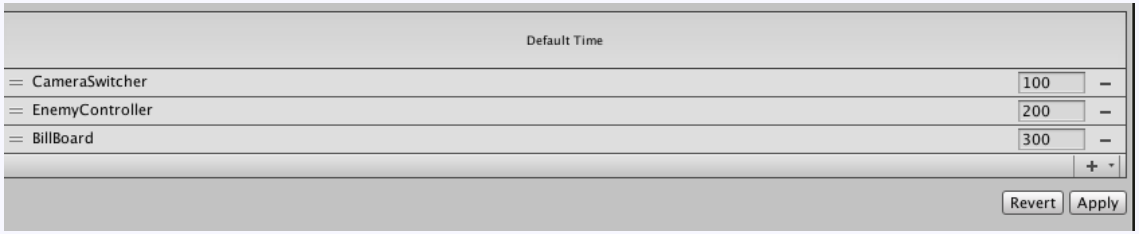
Unityではmain関数は意識しなくても良い作りになっている。

どのスクリプトから実行されるかは、明示的に指定しない限りは不定（Unityが決める）

明示的に指定する方法は、

Edit > Project Settings > Script Execution Order

から行う。



＋ボタンで
順番を管理したいスクリプトを追加する。
数字が少ないスクリプトから実行される。

スクリプトでの「Transformコンポーネント」の制御

Transformコンポーネントは位置、回転、拡大縮小値を、Vector3型で持つコンポーネント

位置の制御
ワールド座標での現在位置情報にアクセス
transform.position　　※transfrom変数はMonoVehaviourクラスで宣言されている

ローカル座標での現在位置情報にアクセス
transform.localPosition

例、
```
Vector3 pos = transform.position;   // 現在の位置を取得
// posを煮るなり焼くなり・・・
transform.position = pos;                    // 現在の位置をposにする
```

拡大縮小の制御
ワールド座標での拡大縮小情報にアクセス(情報を受け取ることしかできない)
transform. lossyScale

ローカル座標での拡大縮小情報にアクセス
transform. localScale

例、
```
Vector3 scale = transform.localScale;        // 現在の拡大縮小値を取得
// scaleを煮るなり焼くなり・・・
Transform. localScale = scale;               // 現在の拡大縮小値をscaleにする
```

可変フレームレートについて

FPSは実行環境によって変動する。
なので、オブジェクトを動かす際に「1フレームにどれだけ進むか」という計算の仕方だと、
実行環境でキャラの移動量が異なってしまい問題となる。
pos.x += 5f;　// ←こういう書き方はNG

実行環境によるFPS	1フレームにどれだけ進めるか	1秒に進む距離
10FPS	2m	20m
60FPS	2m	120m

1秒あたりに進む距離に差が！

なので「1秒でどれだけ進むか」という考え方で制御する。

pos.x += 5f * Time.deltaTime;　// ←1秒で5m進ませたいならこう書く

Time.deltaTimeには前フレームの処理にかかった時間が入っている。

拡大縮小の制御
ワールド座標での拡大縮小情報にアクセス(情報を受け取ることしかできない)
transform. lossyScale

ローカル座標での拡大縮小情報にアクセス
transform. localScale

例、

```
Vector3 scale = transform.localScale;          // 現在の拡大縮小値を取得
// scaleを煮るなり焼くなり・・・
Transform. localScale = scale;                 // 現在の拡大縮小値をscaleにする
```

回転の制御
Unity内部では回転情報をQuaternion型で扱っている。
※Vector3型だとジンバルロックと呼ばれる問題が発生するため
そのため、回転をスクリプトで扱うためには、Quaternion型からVector型に変換が必要。

ワールド座標での回転情報にアクセス
transform.rotation ← Quaternion型の値
transform.rotation.eulerAngles ← Vector3型に変換した値

ローカル座標での回転情報にアクセス
transform.localRotation
transform.localRotation.eulerAngles ← Vector3型に変換した値

例、

```
Vector3 rot = transform.rotation.eulerAngles;    // 現在の回転情報を取得
// rotを煮るなり焼くなり・・・
transform.rotation = Quaternion.Euler(rot);                    // 現在の回転情報をrotにする
```

スクリプトでの「Transformコンポーネント」の制御　向いている方向に進む

Transformコンポーネントのpositionを制御すれば各軸に対して移動はできますが、
向いている方向に進むとなると制御が大変になります。
安心してください。Unityでは向いている方向に進む処理も簡単に実装できます。

transform.forward
これが向いている方向を表すベクトルです。
なので、

```
transform.position += transform.forward * 10f * Time.deltaTime;
```

のように、現在位置に、向いている方向 * 進む距離をプラスしてあげれば、前進する処理が叶います。

他にも、

```
transform.position += transform.TransformDirection(Vector3.back) * 10f * Time.deltaTime;
```

とすれば、前後左右上下の好きな方向に進ませることができます。

自分(スクリプト)がアタッチされているGameObjectの他のコンポーネントにアクセスする方法

自分(スクリプト)は、コンポーネントに過ぎないので、周りのコンポーネントの存在は基本的には知らない。
他のコンポーネントをスクリプトから制御したい場合は、GetComponent関数を使う。

他のコンポーネントの取得
GetComponent<コンポーネント名> ();

例、(Rigidbodyの場合)

```
Rigidbody _rigidBody;  // コンポーネント受け取るためにRigidbody型の「_rigidBody」という名の変数を宣言
_rigidBody = GetComponent<Rigidbody> ();                    // Rigidbodyを取得
```

マウス操作、キーボード操作の情報の受け取り方(Input系)

マウス操作の取得
Input.GetMouseButtonDown(マウスのボタン番号) ← ボタンを押した瞬間だけtrue、それ以外はfalse
Input.GetMouseButtonUp(マウスのボタン番号) ← ボタンを離した瞬間だけtrue、それ以外はfalse
Input.GetMouseButton(マウスのボタン番号) ← ボタンを押している間はtrue、それ以外はfalse

※マウスのボタン番号　0:左ボタン 1:右ボタン 2:ホイール

マウスの位置
Input.mousePosition　Vector2型でマウスの座標を取得

利用例、

```
If (Input.GetMouseButtonDown(0) == true) {
    // 左ボタンが押されたときに行う処理を書く
}
else if (Input.GetMouseButtonUp(0) == true) {
    // 左ボタンが離されたときに行う処理を書く
}
Debug.Log(Input. mousePosition); // マウスカーソルの座標をコンソールに表示
```

キーボード操作の取得
Input.GetKey (キーコード) ← キーが押されている時はtrue、押されていない時はfalse

キーコードとは、キーの種類を表す列挙体
スペースキーの場合 KeyCode.Space
Aキーの場合 KeyCode.A

利用例、

```
if (Input.GetKey (KeyCode.Space) ) {
    // スペースキーが押された場合の処理をここに書く
}
else {
    // スペースキーが押されていない場合の処理をここに書く
}
```

スクリプトからのインスタンスの作成

インスタンスとは
利用可能な状態のGameObjectのこと。
つまりヒエラルキービューに表示されているGameObjectのこと

インスタンスの作成方法
Instantiate (GameObject obj, Vector3 position, Quaternion rot);

わかりやすく書くと
Instantiate (元となるGameObject, 初期位置, 初期向き);

生成したインスタンスを受け取る場合は、
GameObject obj = (GameObject)Instantiate (prefab, position, rotation);

利用例、

```
// プレハブを取得 (Resources/Prefabs/Bullet)
GameObject prefab = (GameObject)Resources.Load ("Prefabs/Bullet");
// プレハブからインスタンスを生成
GameObject obj = (GameObject)Instantiate (prefab, position, Quaternion.identity);

// プレハブからインスタンスを作成（自分より少し前の位置に作成する例）
GameObject obj = (GameObject)Instantiate (prefab,
                                          transform.position + (transform.TransformDirection (Vector3.forward) * 2.0f),
                                          transform.rotation);
```

スクリプトからのインスタンスの破棄

インスタンスの破棄方法
GameObject.Destroy (GameObject obj);

わかりやすく書くと
GameObject.Destroy(破棄したいGameObject);

利用例、
```
// インスタンスを破棄
GameObject.Destory(obj);
```

指定時間後にインスタンスを破棄する方法
GameObject.Destroy (GameObject obj, float time);

わかりやすく書くと
GameObject.Destroy(破棄したいGameObject, 何秒後に破棄するか);

利用例、
```
// 3秒後にインスタンスを破棄
GameObject.Destory(obj, 3.0f);
```

物理演算エンジンを使う方法

Rigidbody（衝突判定の形状を示すコンポーネント）
GameObjectにRigidbodyコンポーネント追加する
Materialを設定することで、反発定数などの衝突時の反発の仕方を指定することができます。

物理演算をスクリプトで制御する

```
Rigidbody rb = GetComponent<Rigidbody> (); // Rigidbodyコンポーネントを取得
// 自分の向いてる方向に100fの力を加える（力のかけ方はImpulse）
rb.AddForce (transform.TransformDirection (Vector3.forward) * 100.0f, ForceMode.Impulse);
```

ForceModeの種類（ Rigidbody.AddForce を使用して力を適用する方法のためのオプション。）

Force	質量を使用して、リジッドボディへ継続的な力を加えます。
Acceleration	その質量を無視して、リジッドボディへ継続的な加速を追加します。
Impulse	その質量を使用し、リジッドボディにインスタントフォースインパルスを追加します。
VelocityChange	質量を無視して、リジッドボディにインスタント速度変化を追加します。

物理演算の力を無効にする方法

```
Rigidbody rb = GetComponent<Rigidbody> (); // Rigidbodyコンポーネントを取得して
rb.velocity = Vector3.zero;  // かかっている力を0にする
```

スクリプトであたり判定を検出する方法

コライダーがアタッチされているGameObjectはあたり判定を検出が可能
コライダーコンポーネントのIsTriggerフラグのON/OFFで呼ばれる関数が変わる

IsTriggerフラグについて（デフォルトはオフ）
ONだと衝突時に発生する反発効果がなくなる。
つまりぶつかっても物理演算は行われず、すり抜けることとなる。
用途はエリア判定（ゴール地点に到着など）に使われることが多い。

IsTriggerフラグがオフだと、OnCollision系の関数が呼ばれ、フラグがONだとOnTrigger系の関数が呼ばれる。

OnCollision系なら引数のCollisionから衝突相手の情報を受け取ることができ、
OnTrigger系なら引数のColliderから衝突相手の情報を受け取れる。

スクリプト例（衝突検出を行いぶつかった相手の名前をコンソールに表示する）

IsTriggerがオフの場合

```
// 衝突した瞬間の1フレームだけ呼ばれる関数
void OnCollisionEnter(Collision collision) {
    Debug.Log ("OnCollisionEnter " + collision.gameObject.name);
}

// 衝突状態から離れた瞬間の1フレームだけ呼ばれる関数
void OnCollisionExit(Collision collision) {
    Debug.Log ("OnCollisionExit " + collision.gameObject.name);
}

// 衝突している間、何度も呼ばれる関数
void OnCollisionStay(Collision collision) {
    Debug.Log ("OnCollisionStay " + collision.gameObject.name);
}
```

IsTrigerがオンの場合

```
// 衝突した瞬間の1フレームだけ呼ばれる関数
void OnTriggerEnter(Collider collider){
    Debug.Log ("OnTriggerEnter " + collider.gameObject.name);
}

// 衝突状態から離れた瞬間の1フレームだけ呼ばれる関数
void OnTriggerExit(Collider collider){
    Debug.Log ("OnTriggerExit " + collider.gameObject.name);
}

// 衝突している間、何度も呼ばれる関数
void OnTriggerStay(Collider collider){
    Debug.Log ("OnTriggerStay " + collider.gameObject.name);
}
```

スクリプトでUIにアクセスする方法

uGUIの各パーツにアクセスする方法

スクリプトからuGUIを扱う場合は
using UnityEngine.UI;
が必須

簡単な例

エディタからインスタンスを受け取れるようにpublicで宣言しておく
public Text TextTimer;
～

enabledプロパティは表示のON/OFFを表す
TextTimer.enabled = false;　　//　Textに限らずImageやButtonなどUI全般が持つプロパティ。

まとめ

後半はスクリプトばっかりで、覚えることが多いと感じたかもしれませんが、どの機能もかなり少ない手順で利用可能です。

Unityの素晴らしいところは、

ユーザーが多いこと

これにより、膨大な情報がインターネット上に公開されていて、わからないことも、ネットを調べればすぐ解決できます！

ご清聴ありがとうございました