Unity講座2017

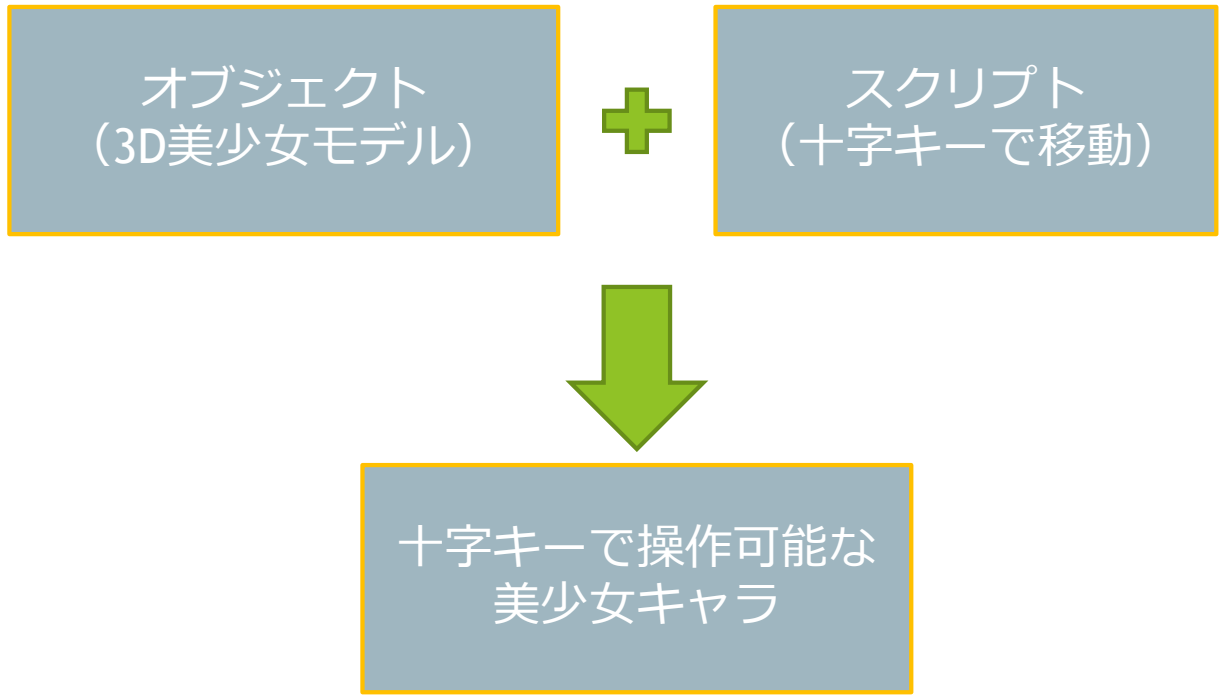
～第1回～

Unityの基本的な使い方

実際に動くものを作る

0. はじめに

- Unityでは（おおざっぱに言うと）物体であるオブジェクトと動きを管理するスクリプト（プログラム）を組み合わせてゲームを作ります



- というわけでUnityでのゲーム作りはプログラムを書くだけではなく
- オブジェクト生成とか色々やることあります

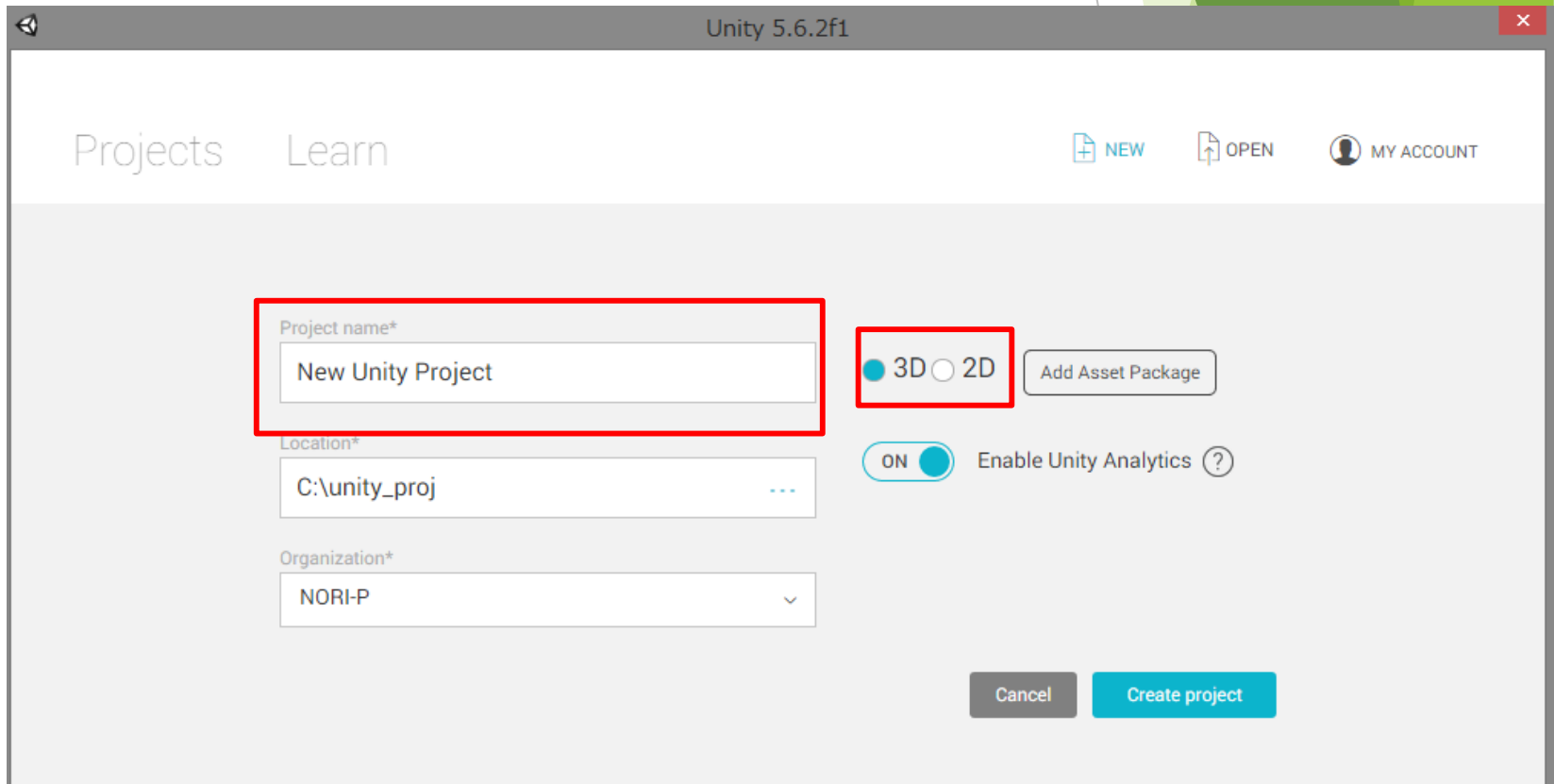
- 色々ありすぎてよくわかんないので
- この講座を手順通りやることでまずはゲームが1個出来た！
- という風にゲーム作りの楽しさを感じてもらいたいなと思います。

- 初めにUnityでゲーム作りを行う上で知っておいた方がよい知識を紹介します

0-0. 忘れてた

- 実際の画面を見ながらのがいいので
- Unityを起動
- 右上「NEW」から
  プロジェクトを作成

- 3Dを選択
- プロジェクト名は自由でどうぞ



1. 画面の見方

1-1. 画面の名称

- 人によって配置は
  異なります
  自分に合った配置を
  見つけましょう

1-2. Hierarchy

- ヒエラルキはSceneビューにあるゲームオブジェクトを表示する画面

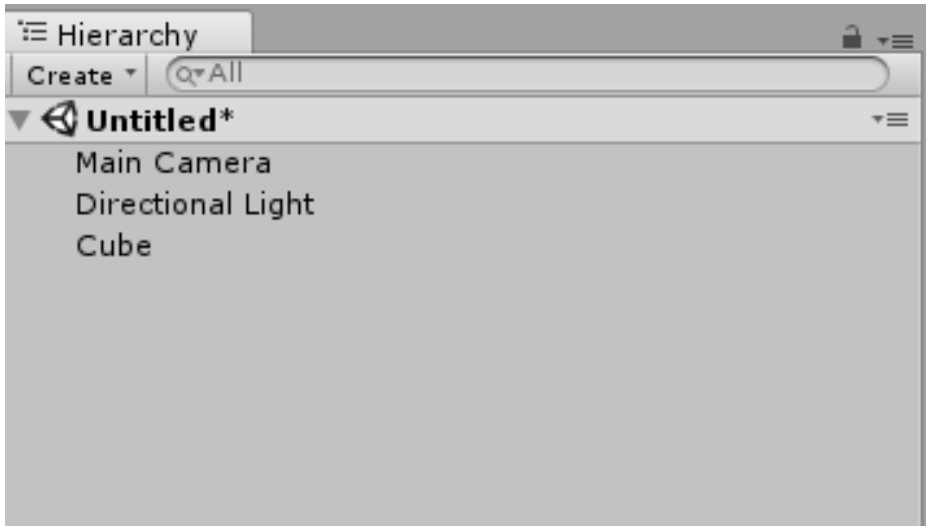
- →の状態だと、

  カメラオブジェクト

  ライトオブジェクト

  キューブオブジェクト

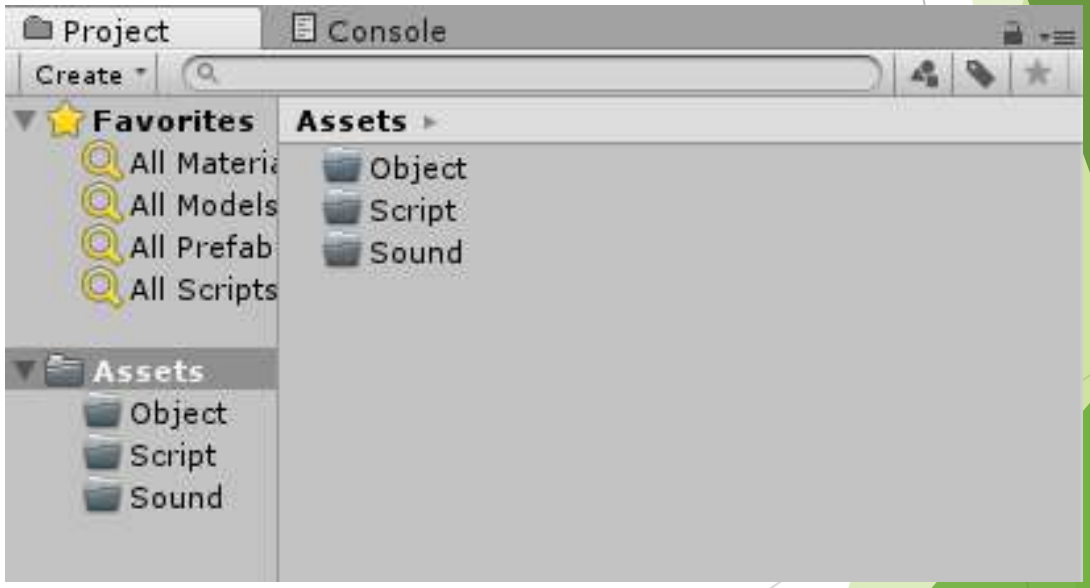
  が存在することが分かります



1-3. Project

- プロジェクトは保存先にある「Assets」フォルダの中身を表示する画面

- →の状態ではプロジェクト内に
  オブジェクト
  スクリプト（プログラムファイル）
  音楽ファイル
  をそれぞれ管理するフォルダがあることが分かります

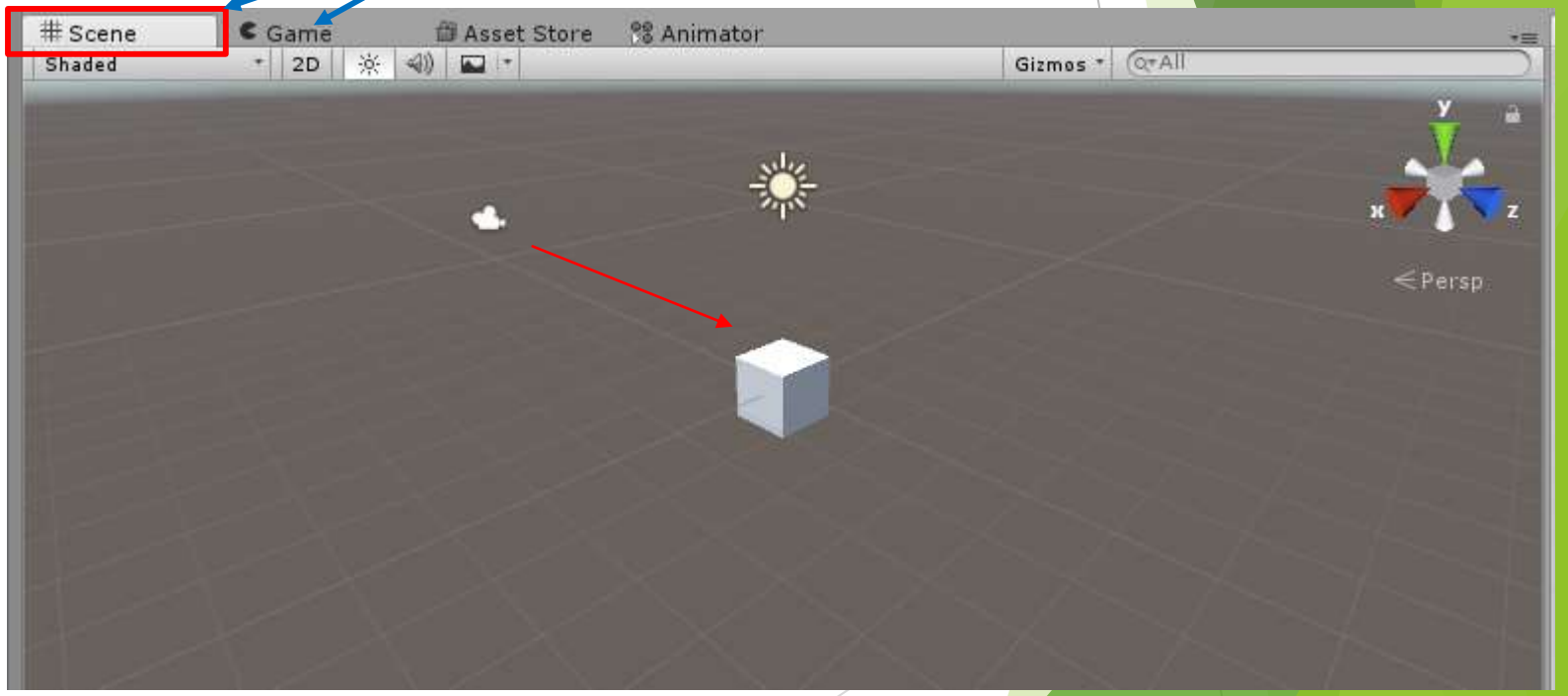


1-4. Scene

- シーンはゲームオブジェクトを配置する画面

- →の状態では
  カメラとライトと
  キューブオブジェクトが存在し
  カメラが矢印の方向を映している
  （どう見えるかは次ページ）

SceneとGameビューが重なってるときここで変更



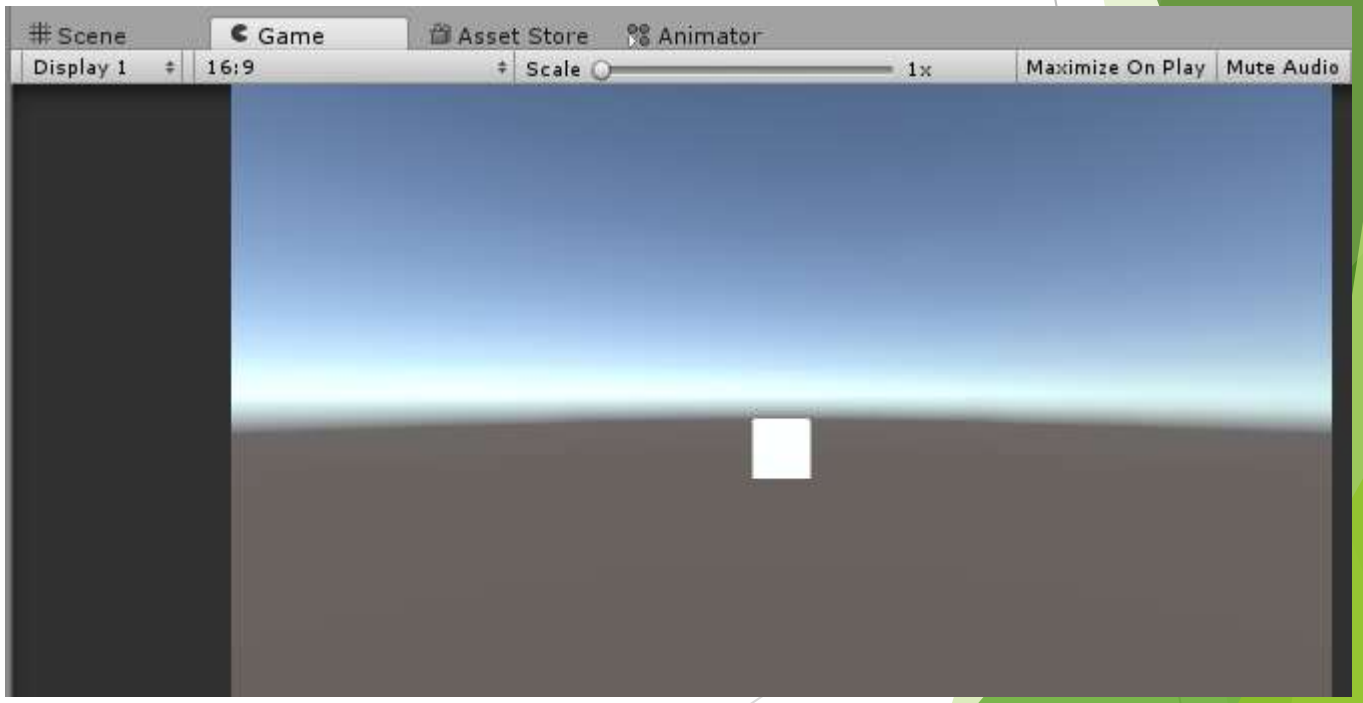
1-5. Game

- ゲームは実際にゲームを実行した時のカメラからみた画面

- →の状態だと

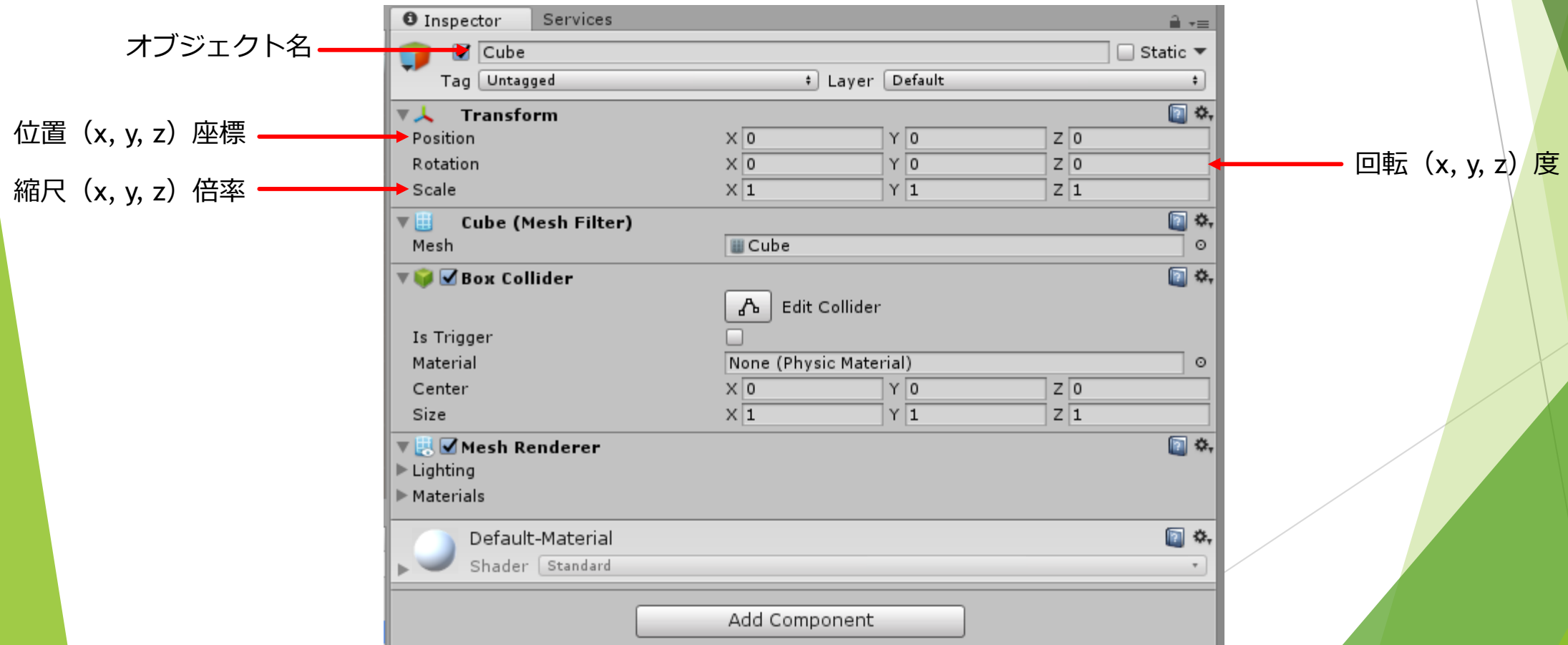
  1個のキューブオブジェクトを

  1面だけ映している



1-6. Inspector

- インスペクタはゲームオブジェクトのステータスを表示する画面

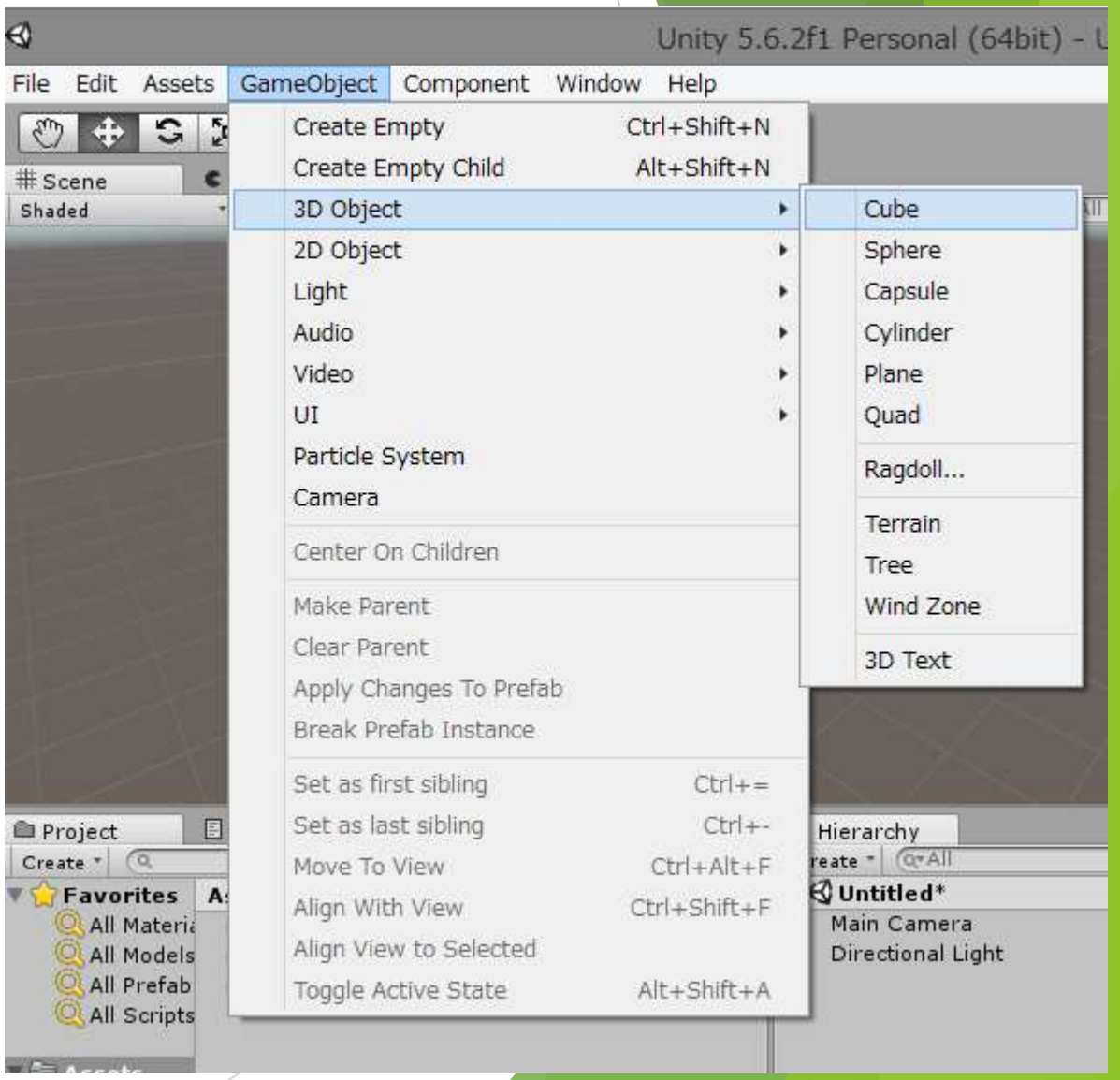


2. オブジェクト

2-1. オブジェクトの作り方

- 散々でてきたキューブオブジェクトを作る
- 「GameObject」→「3D Object」→「Cube」

- Sceneビューを確認
- あら不思議！キューブオブジェクトが！



2-2. オブジェクトの動かし方

- 画面左上の↓これ

左からSceneビュー内の移動、オブジェクトの移動、オブジェクトの回転、
オブジェクトの拡大・縮小、uGUIの操作です

- 基本的に左3つが使えれば大丈夫

- いちいちクリックで変更するのは大変なので

 → Qキー

 → Wキー

 → Eキー

- で操作できることを覚えておきましょう
- また  の状態で「ALT」を押している間だけ  に変化し
  Sceneビュー内の移動ではなく、視点変更ができるようになります。

- Inspectorの座標や回転の値から直接オブジェクトを動かすことも可能です
- というよりInspectorから移動のが使うかも…

3. ついでに

3-1. ゲームの実行・一時停止

- 作成途中の段階で動くか確認したくなるよね？？？
- 上部真ん中にある↓のボタンで操作する

- 左から、再生、一時停止、1ステップ毎再生ボタン
- 左2つが使えれば大丈夫

※再生ボタンを押してから停止するまでに編集したものは
停止と共になかったことになります
オブジェクト追加やサイズ変更は再生前に行う！！！

- つまんない話はここまで

- ここからは実際にゲーム作りに入ります。
- 3までに最低限の知識を書きましたが、まだまだ足りません。
- その他必要なことは随所で説明しながら進みます！

- 何をつくるか？
- 初めてのゲーム作りの定番！ブロック崩し！

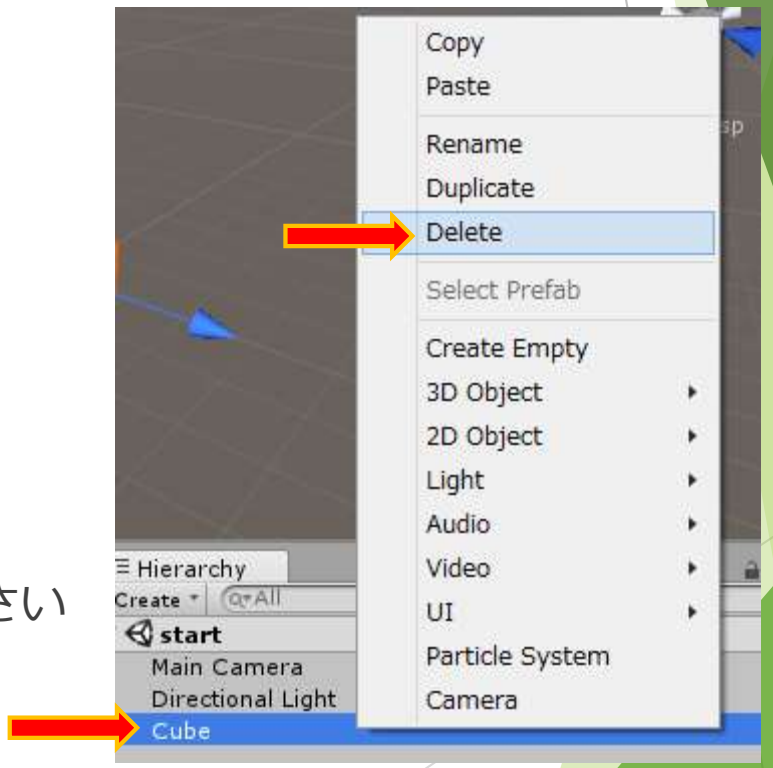
4. タイトル画面作成

4-1. タイトルを作る下準備

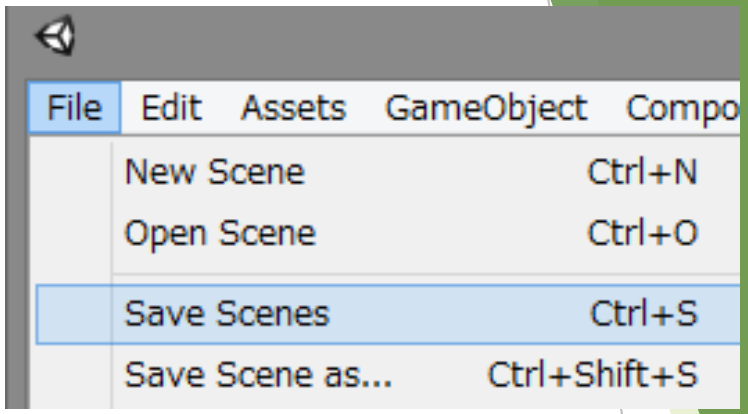
- どんなゲームでもタイトル・スタート画面はあります
- まずはここから作りましょう！

- 先ほど作成したキューブオブジェクトを削除します
- ヒエラルキにある「Cube」（名前は違うかも）
  を右クリック。「Delete」で削除

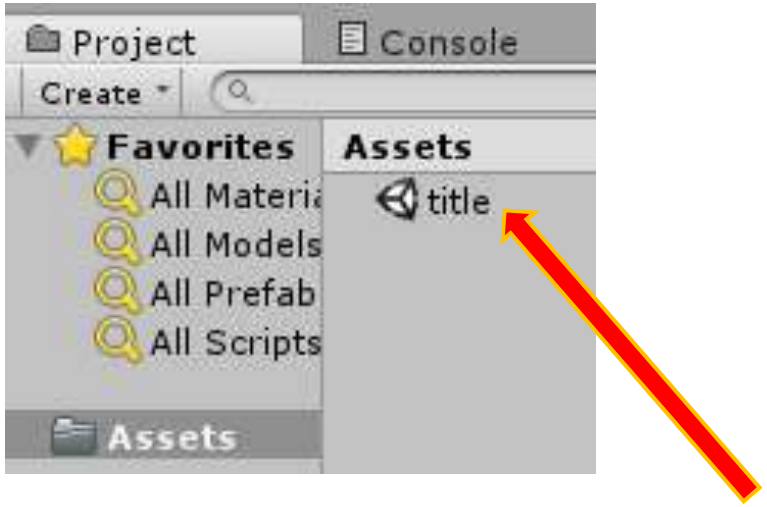
- 「Main Camera」と「Directional Light」だけにして下さい



- 元に戻したところでこのシーンを保存します
- 「File」→「Save Scenes」または「CTRL + S」で保存



- 名前は「title」で！



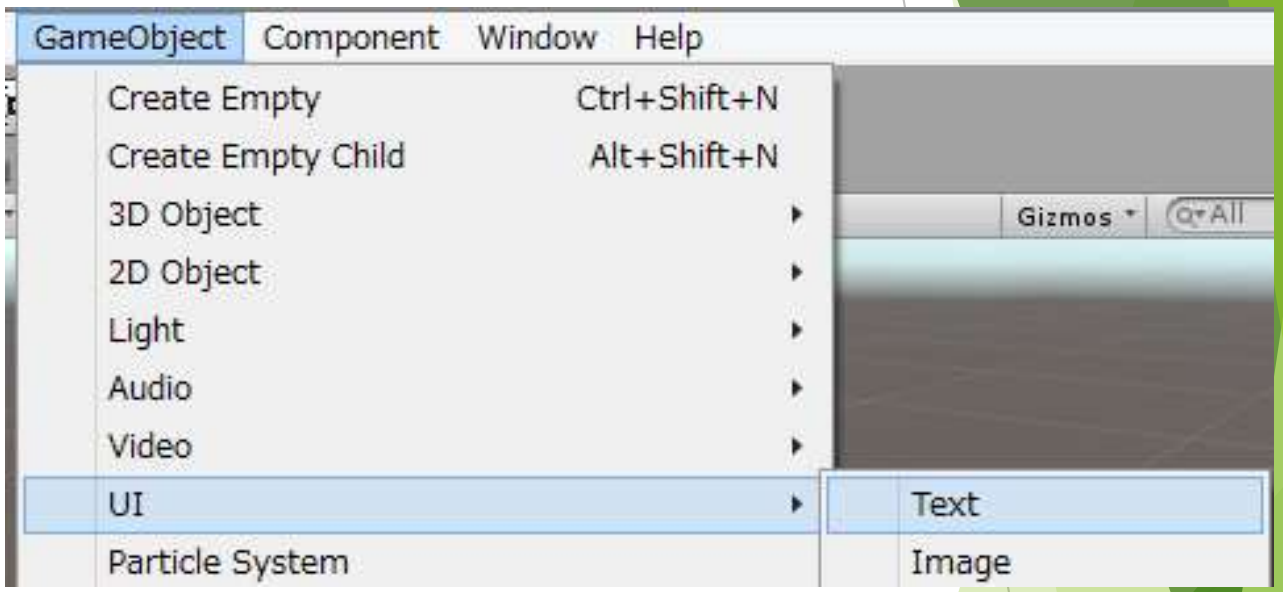
- するとプロジェクトビューの「Assets」にこんなのができてるはず
- シーンはヒエラルキではなくプロジェクトの「Assets」に保存されます

- このままタイトル画面を作っていきます

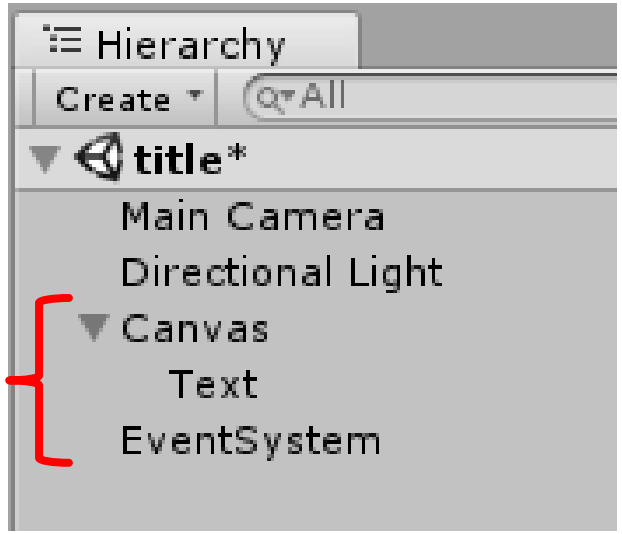
※こまめにCTRL + Sで上書き保存しましょう。急に止まった時死にます

4-2. テキストを作る

- タイトルっぽいテキストを書きましょう
- 「Game Object」→「UI」→「Text」



- ヒエラルキに色々追加されてる



- 「Canvas」・・・UI（テキストとかボタンとか）を置く場所
- 「Text」・・・テキスト（文字列とか）を管理する
- 「EventSystem」・・・UIへの入力を管理する

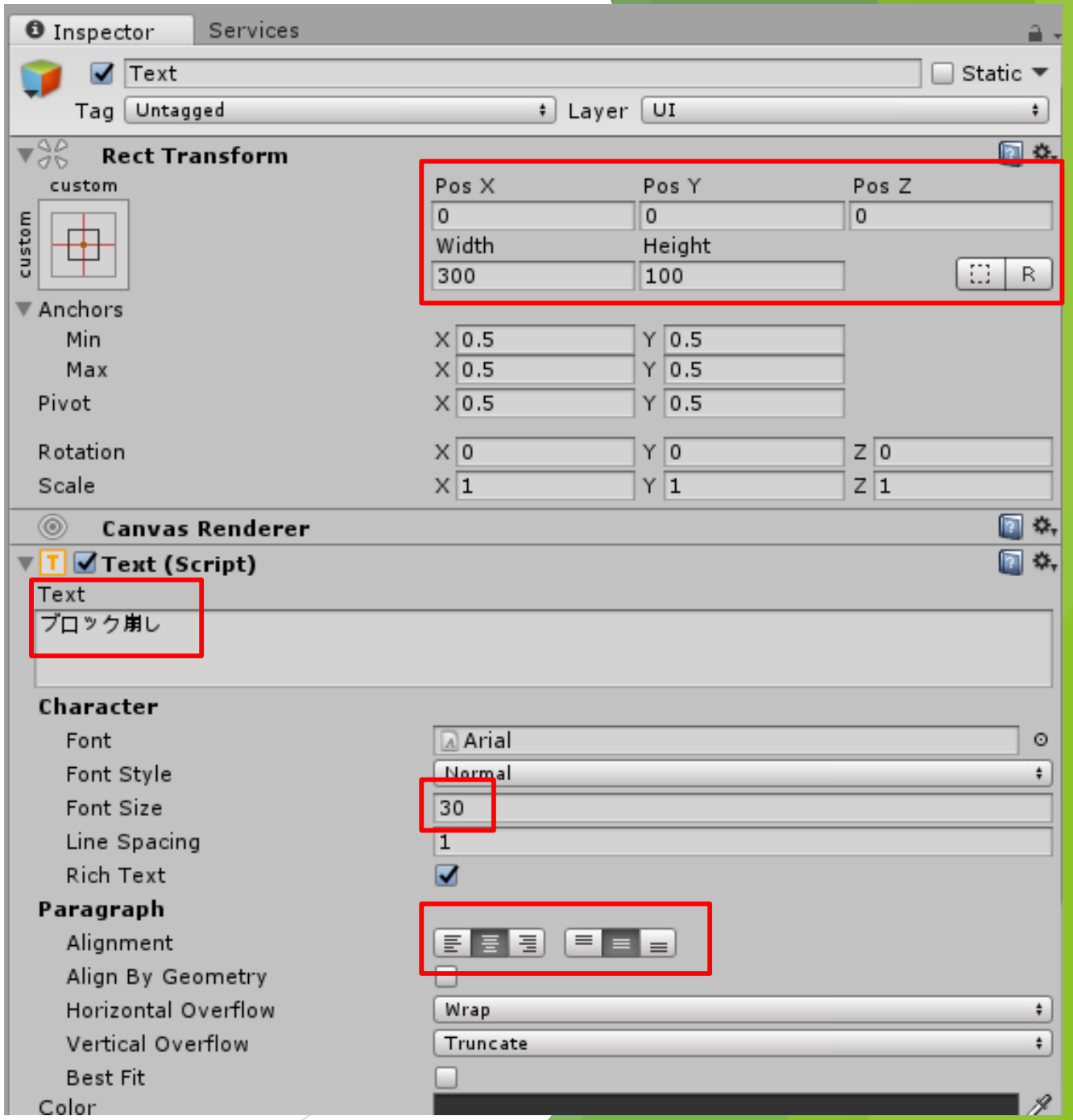
- EventSystem使ってなくね？とかいって削除しないように！！

- 実際にテキストを書きます
- 「Text」のインスペクタを見る

- インスペクタを→と同じ用に変更

- Pos X, Y, Zはテキストボックスの位置
- Width, Heightはテキストボックスの大きさ
- Textは表示する文字列
- Font Sizeは文字の大きさ
- Alignmentはテキストボックス内のどこに文字列を表示するか

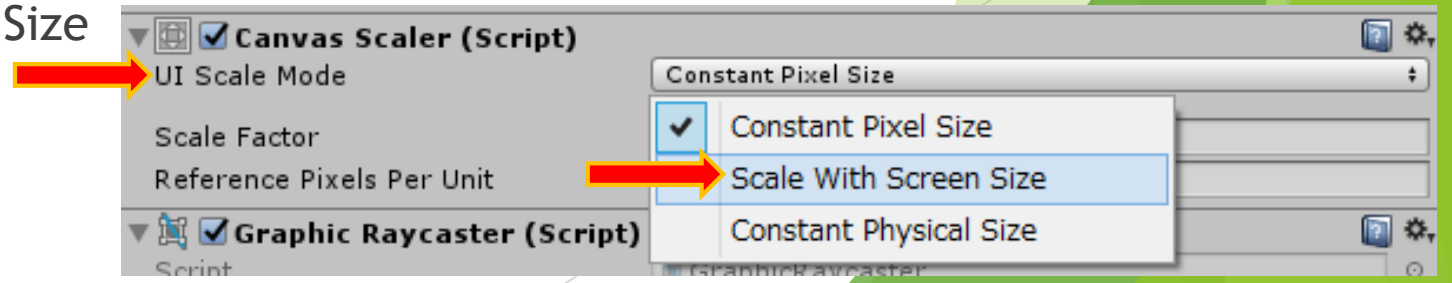
- となってます



- ゲームビューを確認

- こんな感じで文字が表示できていればおっけー

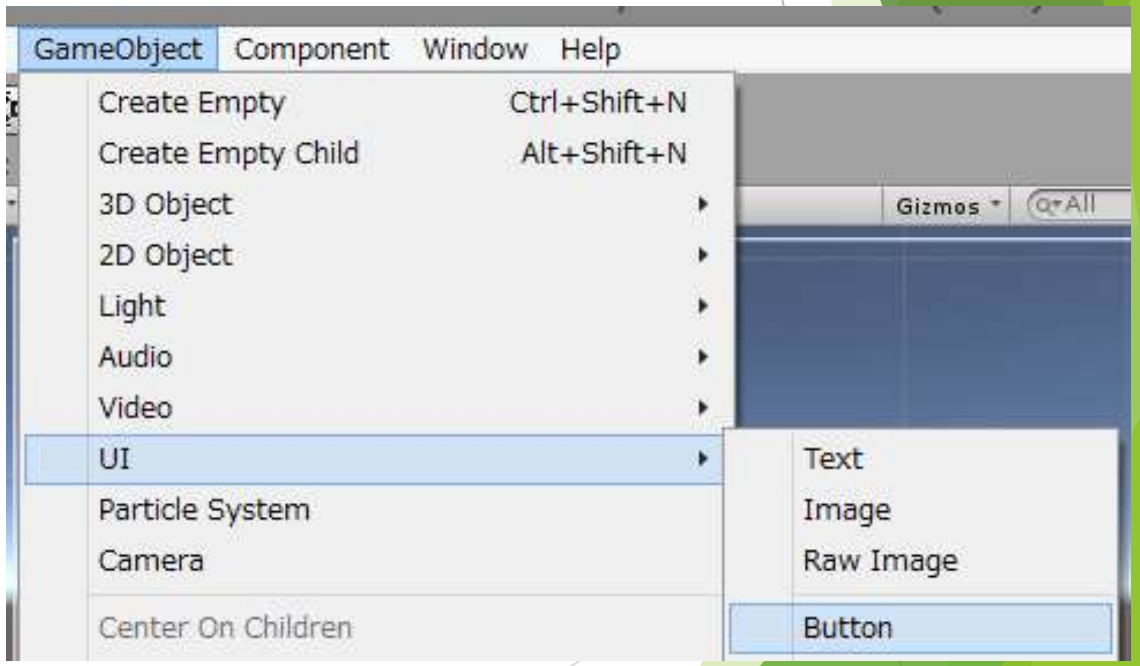
- 「Canvas」のインスペクタを見る
- 「UI Scale Mode」をScale With Screen Size に変更してください



4-3. スタートボタンの作成

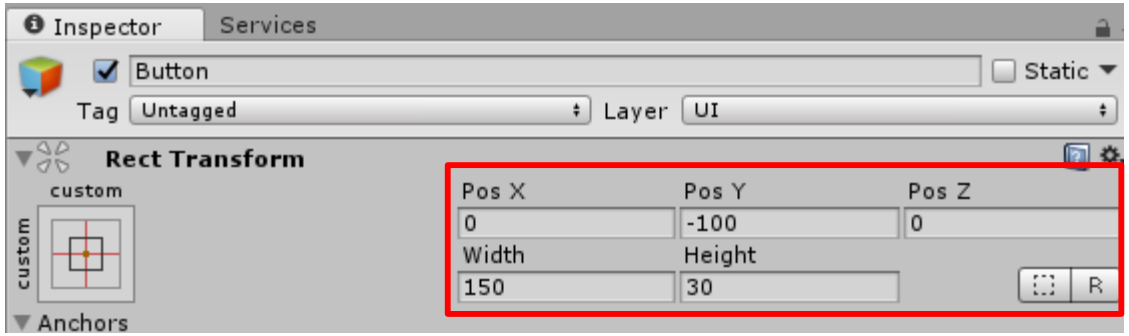
- タイトルからゲーム画面に遷移するためのボタンを作ります

- 「GameObject」→「UI」→「Button」
- ヒエラルキのCanvasにButtonが追加される

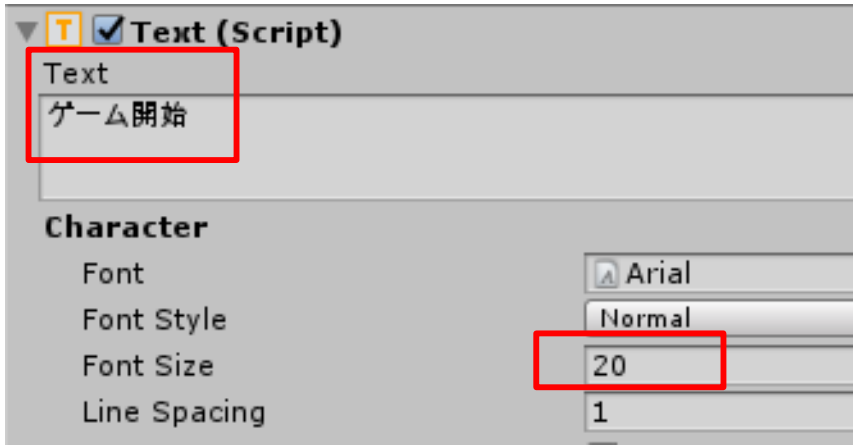


- Buttonの中にTextがあります
- 「Button」・・・ボタンの表示位置やクリックしたときの動作などを管理
- 「Text」・・・ボタンに表示する文章を管理

- まずはButtonから
- インスペクタを変更



- 続けてButton内のText
- Button-Textのインスペクタを変更



- シーンを確認すると↓の感じ

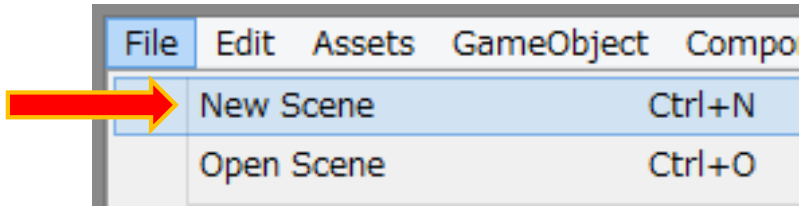
5. シーン遷移

5-1. シーン遷移の準備

- ゲームではタイトル→ゲーム→リザルト→タイトル. . .
  といったように各画面と矢印にあたる遷移が必要です

- 4でタイトル画面と遷移を起こすスタートボタンができたので
  次はボタンを押したときに遷移する先が必要です
- ので遷移先となるステージ画面（仮）を作ります

- 「File」→「New Scene」
- 現在のTitleシーンの上書きを求められたら上書きする



- カメラとライトだけの新しいシーンができたはず
- 早速CTRL + Sで保存
- 名前は「stage」

- プロジェクトのAssetsにシーンが2つあればおっけー

5-2. シーン遷移の作り方

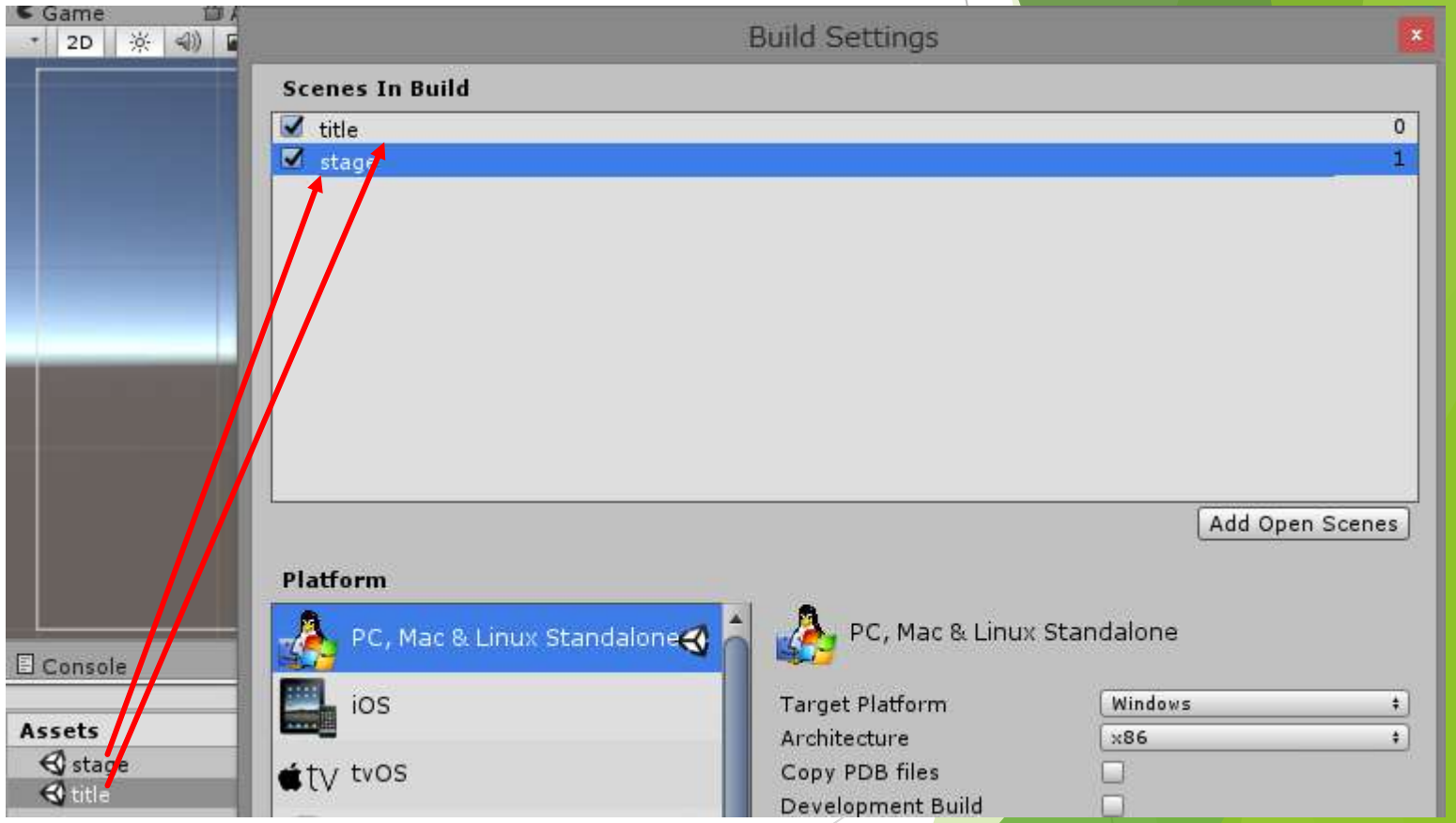
- Assetsからtitleシーンを開く（ダブルクリックで開きます）
- シーン遷移のためのやることがあと3つ

  ①ゲームビルドにシーンを入れる

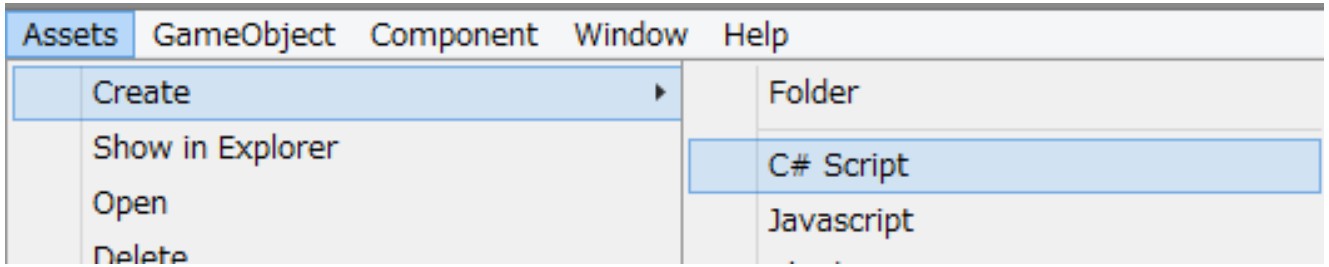
  ②遷移のためのプログラムを書く

  ③プログラムをボタンと結び付ける

- ①から
- 「File」→「Build & Settings」で↓の画面が出る
- Assetsにあるstageとtitileを
  両方ともScene In Buildに
  ドラッグ&ドロップする
- Build & Settingsを閉じて
  ①は終了



- 次は②
- ようやくプログラムを書きます



- 「Assets」→「Create」→「C# Script」
- AssetsにC#ファイルが追加されるので名前を「C_Start」に変更
- 「C_Start」をダブルクリックで開く

- このときVisual Studioが開く人とMono Developが開く人に分かれますがどちらでも問題ありません

- こだわりがあるならのちほど聞いてください

- 一旦プログラムについて．．．

- 必要なものをインポート

- ここからがメイン

- Start()関数

  →最初に1度だけ呼ばれる関数

- Update()関数

  →1フレーム毎に呼ばれる関数

C_Start.cs

C_Start

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

0 個の参照
public class C_Start : MonoBehaviour {

    // Use this for initialization
    0 個の参照
    void Start () {

    }

    // Update is called once per frame
    0 個の参照
    void Update () {

    }
}
```

- C_Startに赤枠を追加

- using ...
  →シーン遷移に必要なものをインポート

- public void TitleToStage()関数
  →この関数を呼んだ時に”stage”という
  シーンに遷移する
  publicをつけることでボタンから呼び出す
  ことができるようになります

- 書いたら保存
- ②終了

C_Start.cs

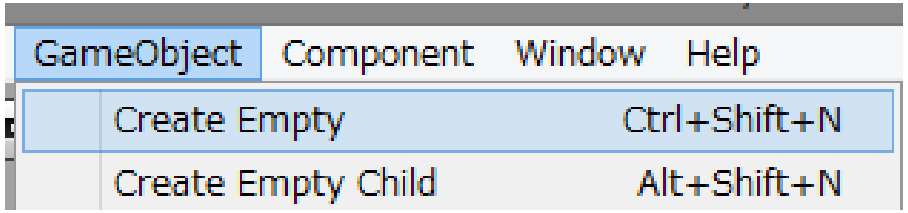
C_Start

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

0 個の参照
public class C_Start : MonoBehaviour {

    // Use this for initialization
    0 個の参照
    void Start () {

    }

    // Update is called once per frame
    0 個の参照
    void Update () {

    }

    //■■■タイトルからステージに遷移する関数■■■
    0 個の参照
    public void TitleToStage()
    {
        SceneManager.LoadScene("stage");
    }
}
```

- 最後に③

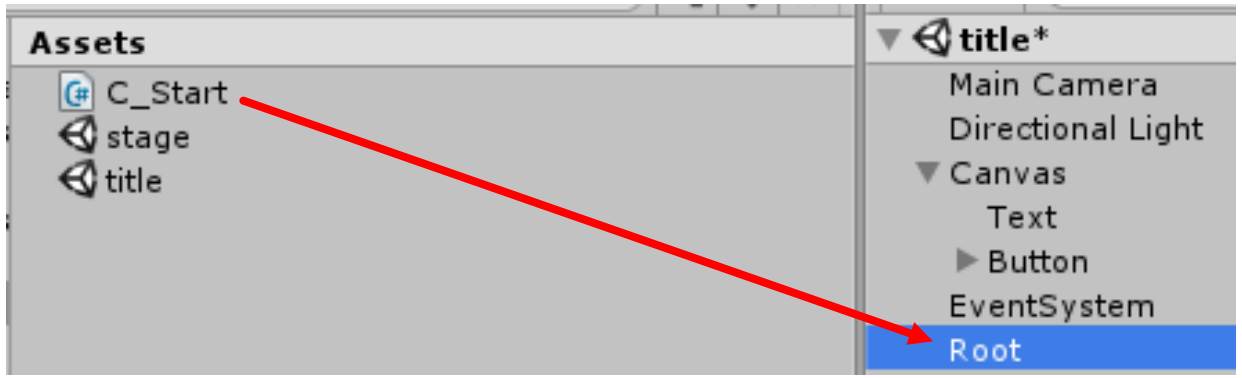
- ②で書いたプログラムをボタンから使えるようにします
- まずプログラムを保持するオブジェクトを適当に用意

- 今回は空のオブジェクト（キューブみたいに実体はないけど存在するもの）
  を「GameObject」→「Create Empty」でつくる

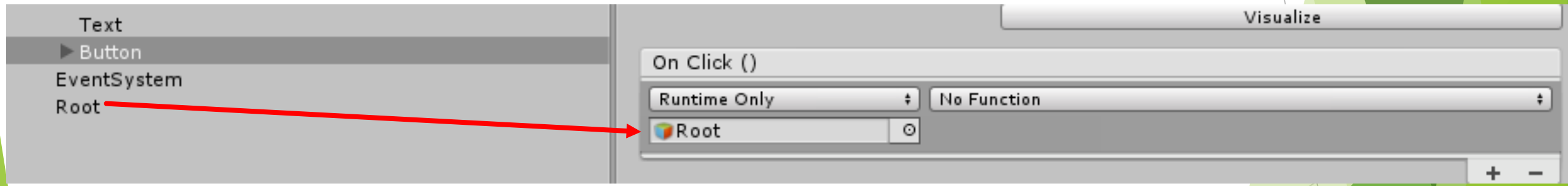


- 名前は「Root」

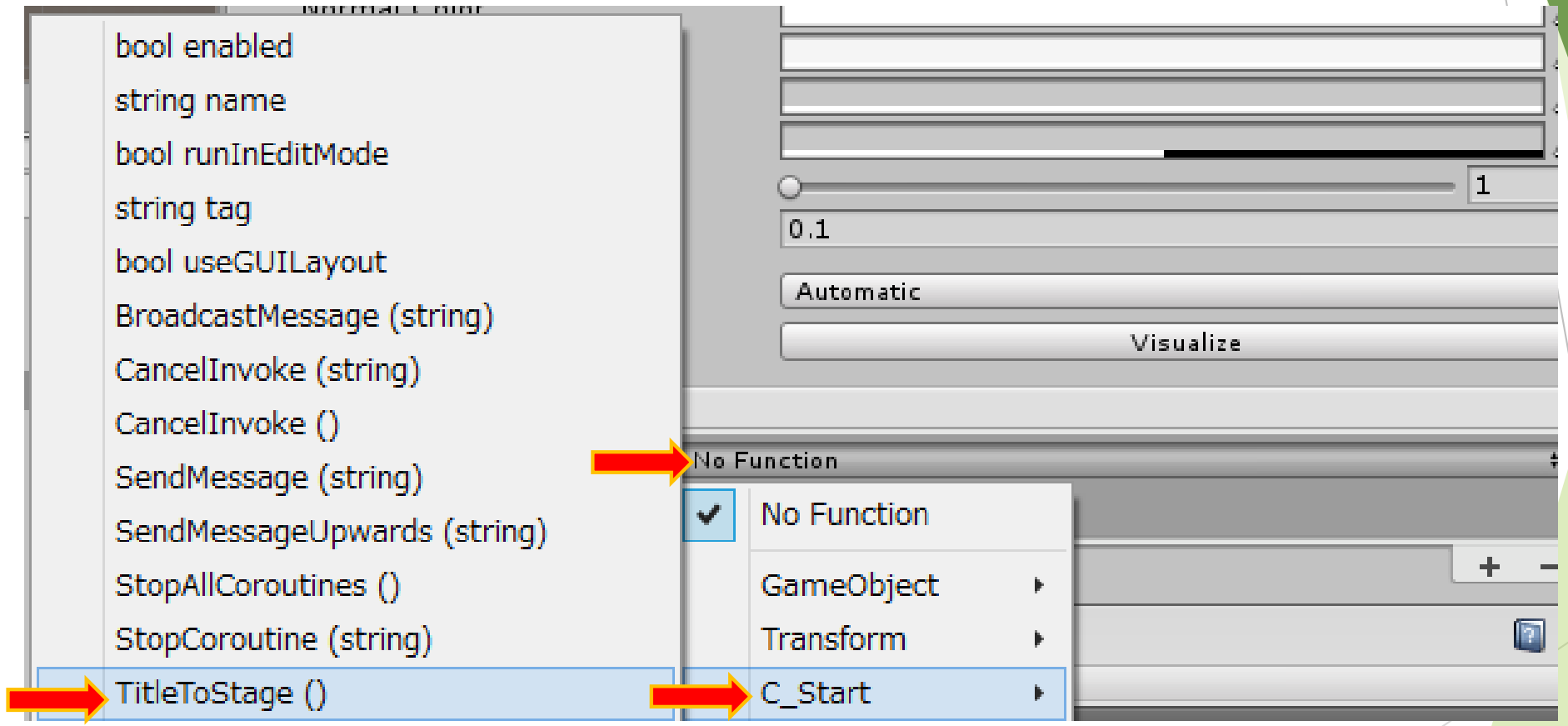
- C_StartをRootにドラッグ＆ドロップ



- 今度は一度Buttonのインスペクタを開き
- RootをOn Clickという項目の↓ここにドラッグ＆ドロップ

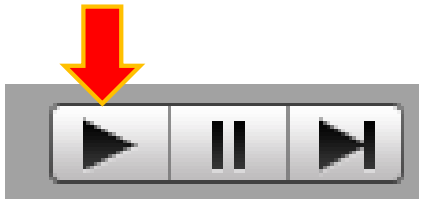


- No Functionという部分をクリックするとC_Startがあるはず
- その中のTitleToStageを選択

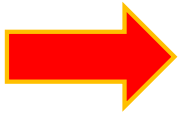


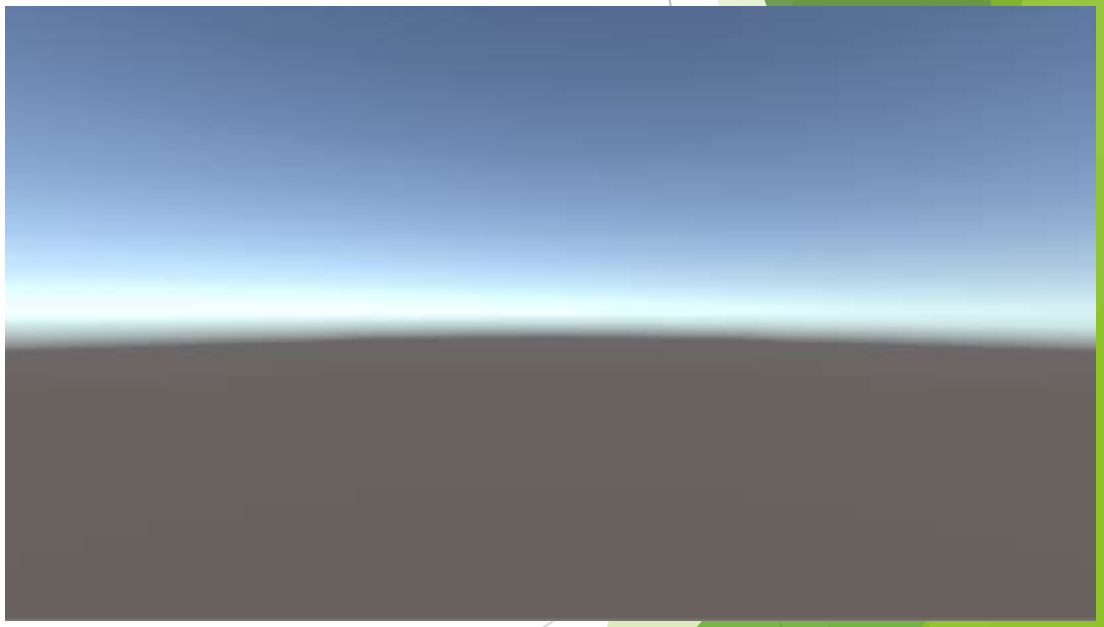
- ③終了

- 以上で遷移が可能になった！！
- →のやつで再生



- ゲーム開始ボタンを押すと画面が変わるはず！！！！





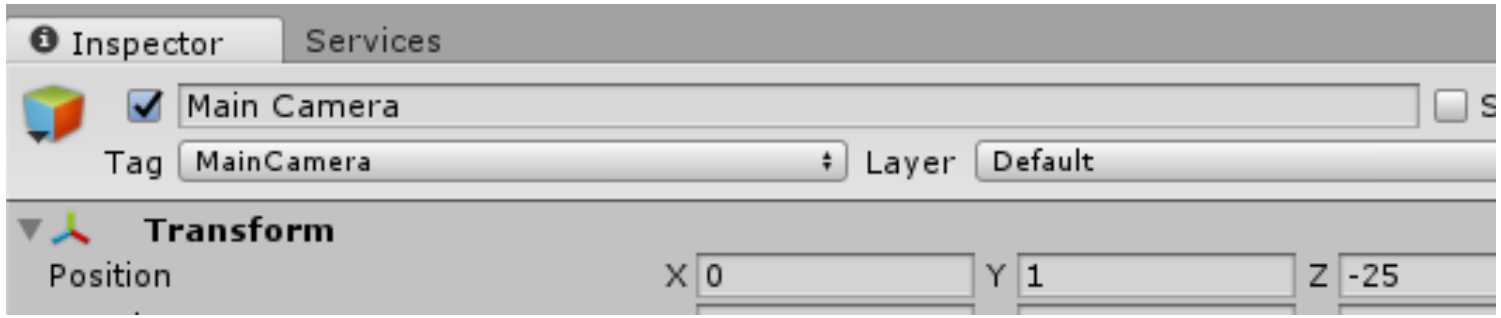
- シーン遷移はこの要領で行います

6. ステージ画面作成

6-1. 壁の作成

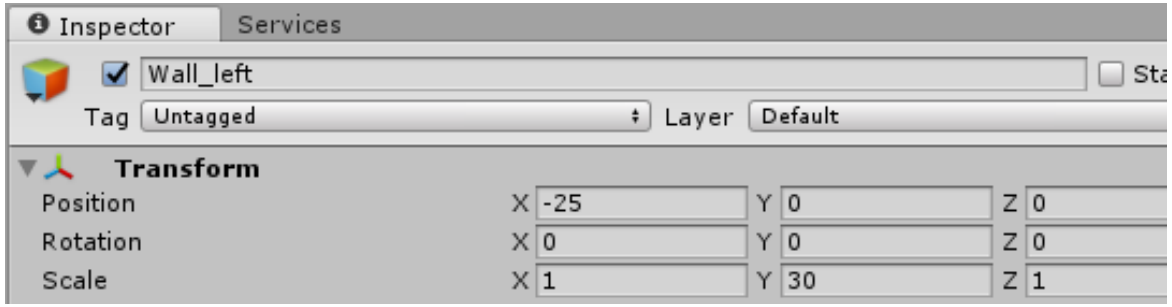
- ここからメインとなるステージ作成に入ります
- ブロック崩しに最低限必要なもの
  - ・ボール
  - ・バー
  - ・ブロック
  - ・壁
- 一番簡単な壁からいきます

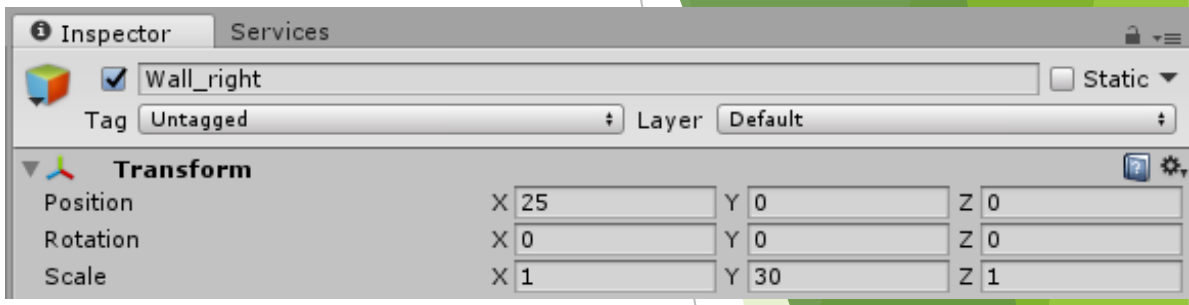
- Stageシーンを開いてまずはカメラ位置を変更
- Z座標だけ変更

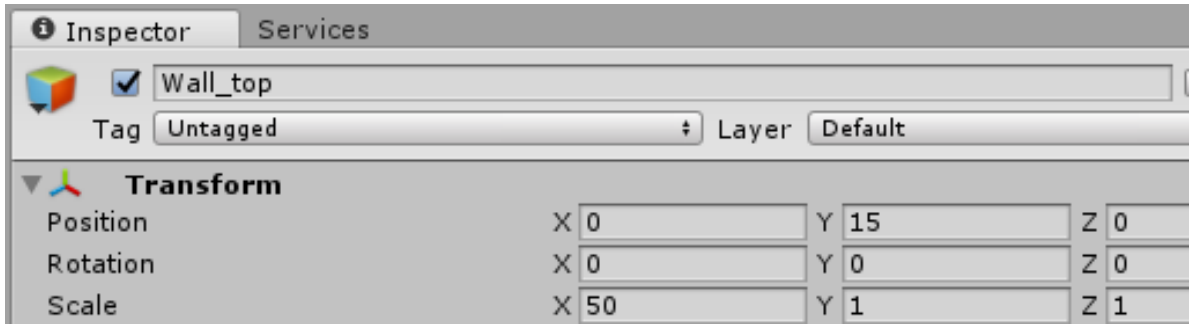


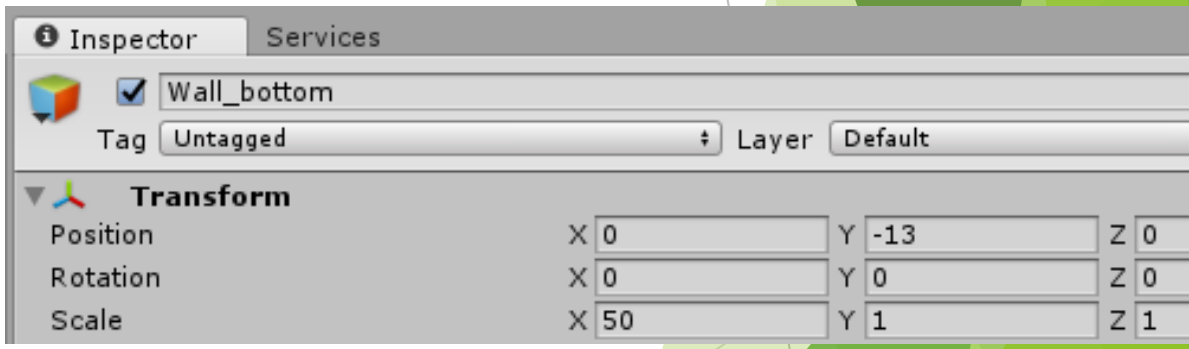
- 壁をつくる「GameObject」→「3D Object」→「Cube」
- 名前は「Wall_left」
- 同様に「Wall_right」「Wall_top」「Wall_bottom」
- と4つ壁を作ってください

- それぞれpositionとscaleを↓のように

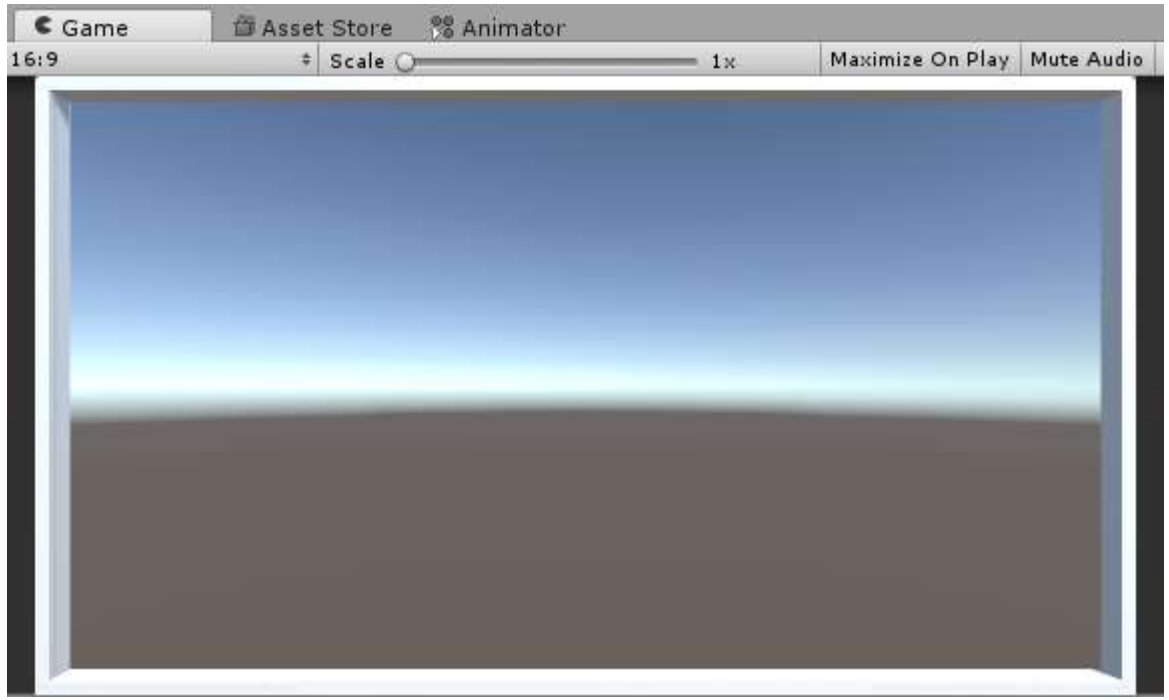








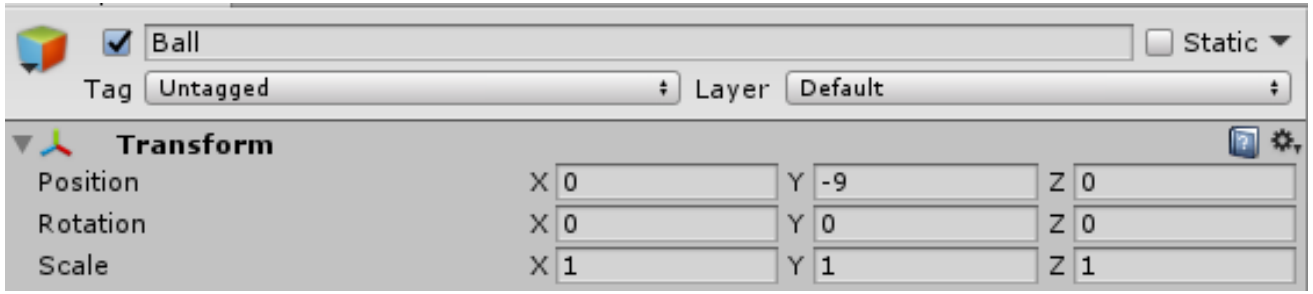
- ゲームビューがこんな↓感じになる



- 次はボール

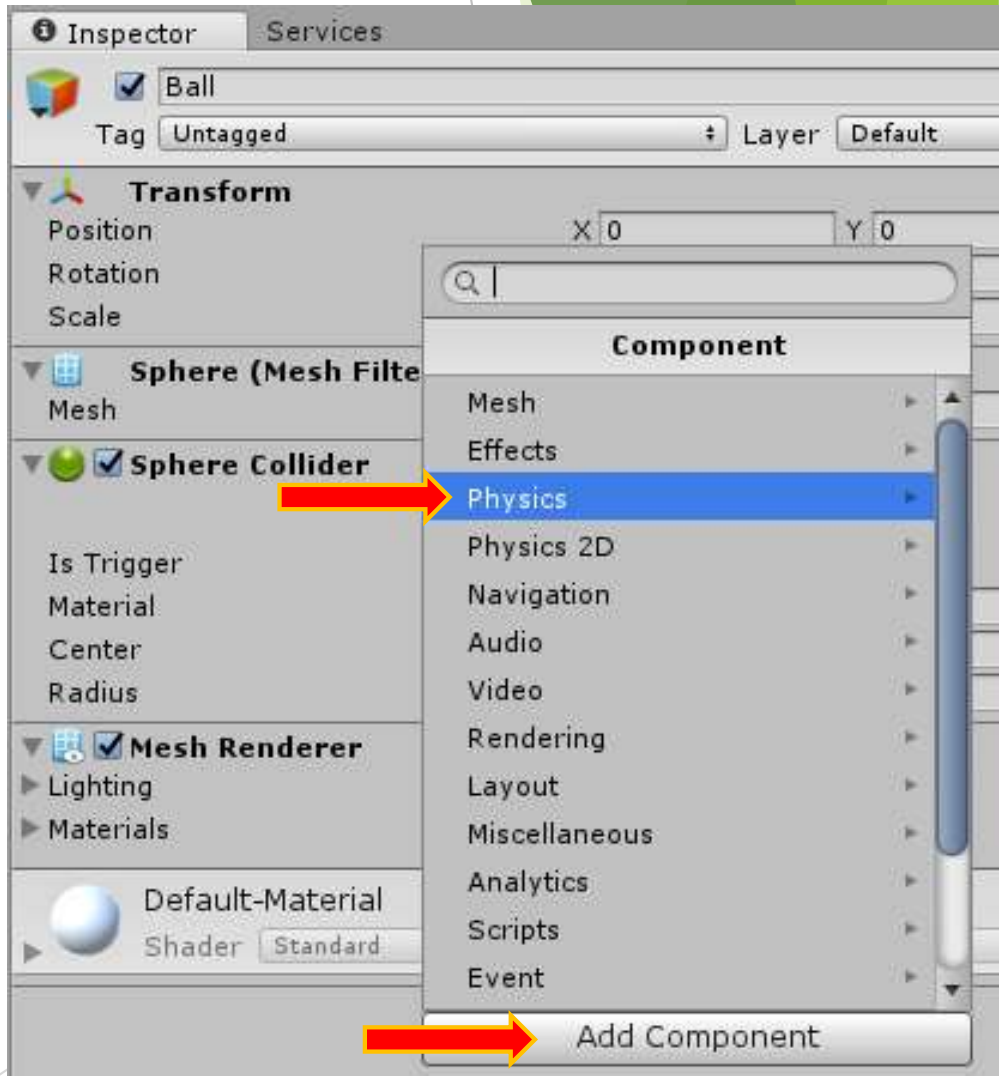
6-2. ボール作成

- 「GameObject」→「3D Object」→「Sphere」
- 名前は「Ball」
- インスペクタを↓と同じに

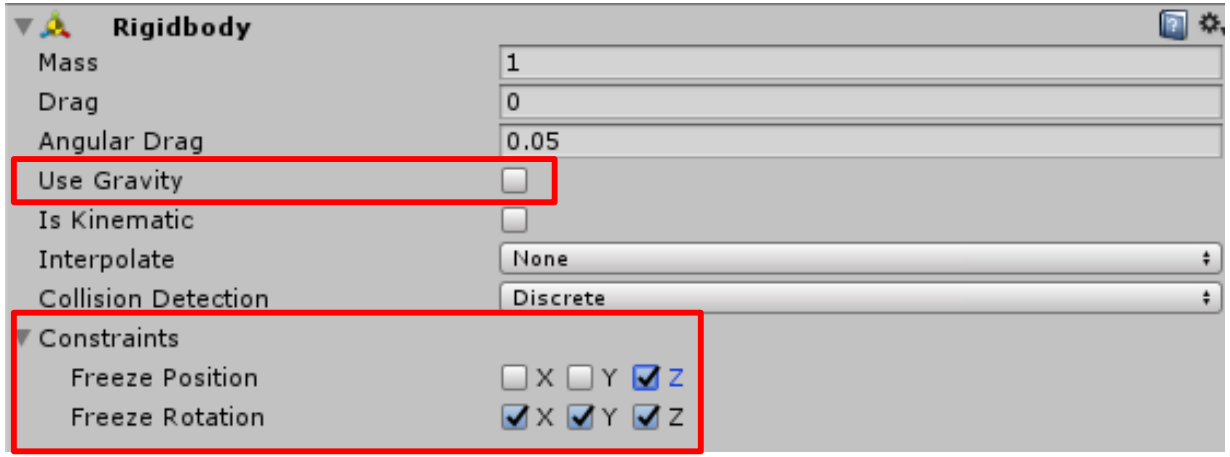


- この状態だと宙に浮くだけなので物理法則を付け加えます

- Ballのインスペクタの一番下にある「Add Component」から「Physics」→「Rigidbody」を選択



- Rigidbodyの中身を変更（する前にゲームを実行するとボールが重力落下します）



- Use Gravity

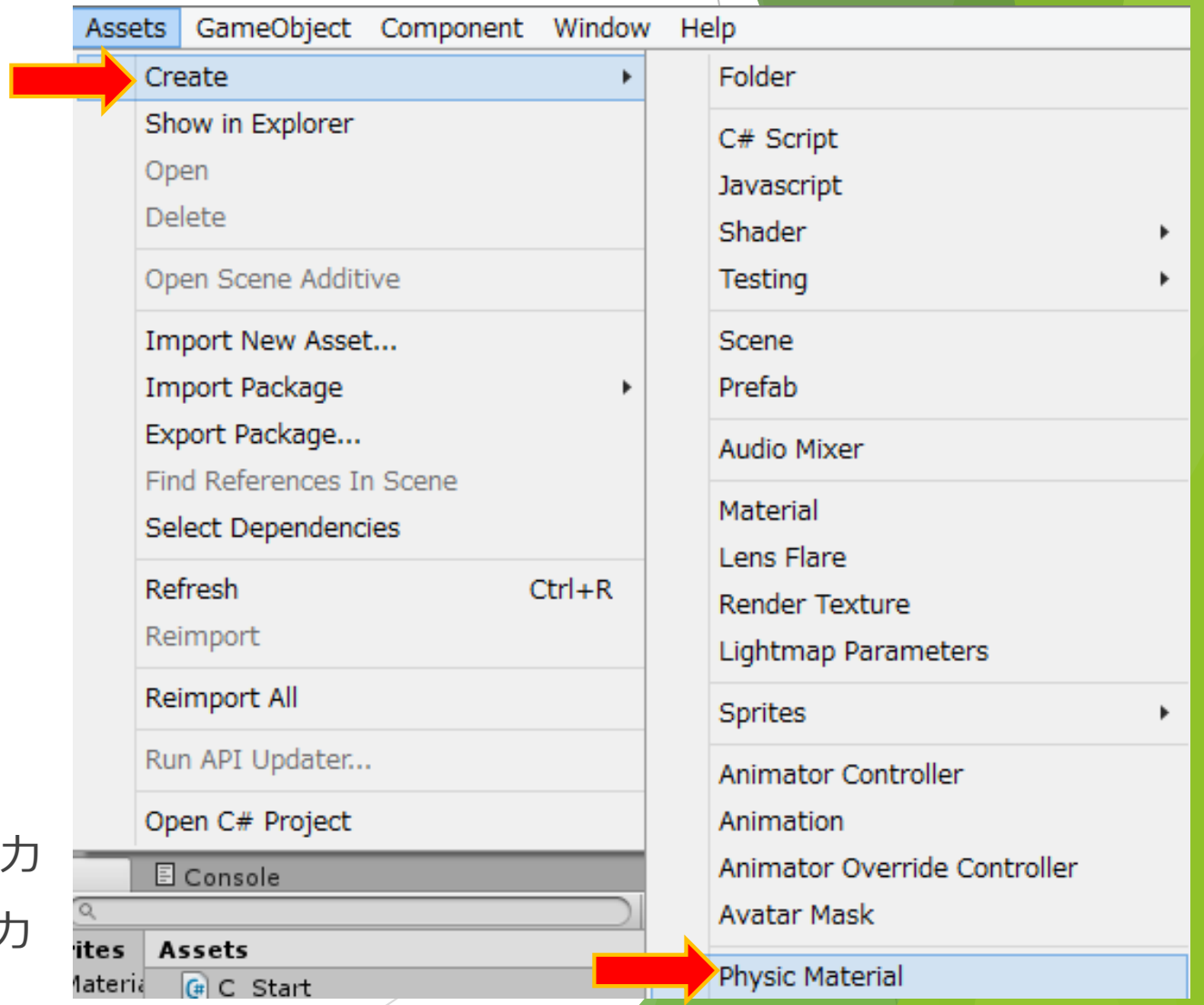
  →名前の通り重力を使用するか否か

- Constrains

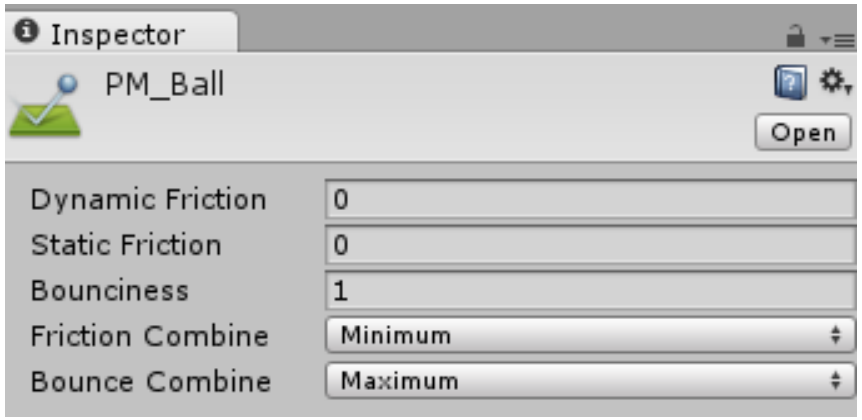
  →それぞれposition, rotation（座標、回転）を固定するか否か

  上図だとz軸には移動せずxyzどの軸にも回転はしない、という設定

- 今度はボールに特殊は物理法則を与えます
- 上メニューの「Assets」→「Create」→「Physic Material」

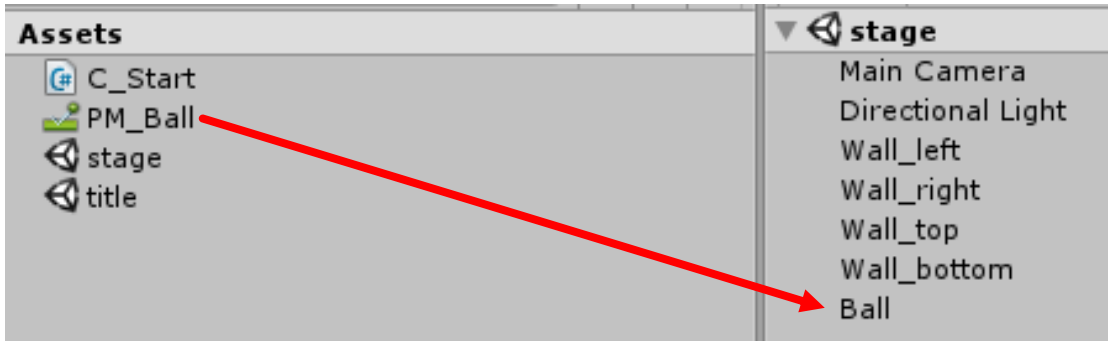


- 名前を「PM_Ball」に変更
- 「PM_Ball」のインスペクタ↓と同じに



- Dynamic Friction　→　動摩擦力
- Static Friction　→　静止摩擦力
- Bounciness　→　弾性力
- Friction Combine　→　衝突時の物体間の摩擦力
- Bounce Combine　→　衝突時の物体間の弾性力

- 「PM_Ball」を「Ball」にドラッグ＆ドロップ



- 特殊な物理法則を持ったボールが出来たので動きを与えるプログラムを書きます

- 「Assets」→「Create」→「C# Script」
- 名前は「C_Ball」

- 「C_Ball」も「Ball」にドラッグ＆ドロップ

- 「C_Ball」を開いて下のように書く（Start内に1行加えるだけ）

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

0 個の参照
public class C_Ball : MonoBehaviour {

    // Use this for initialization
    0 個の参照
    void Start () {
        GetComponent<Rigidbody>().velocity = new Vector3(10.0f, 10.0f, 0.0f);
    }

    // Update is called once per frame
    0 個の参照
    void Update () {

    }
}
```

- GetComponent<Rigidbody>().velocity

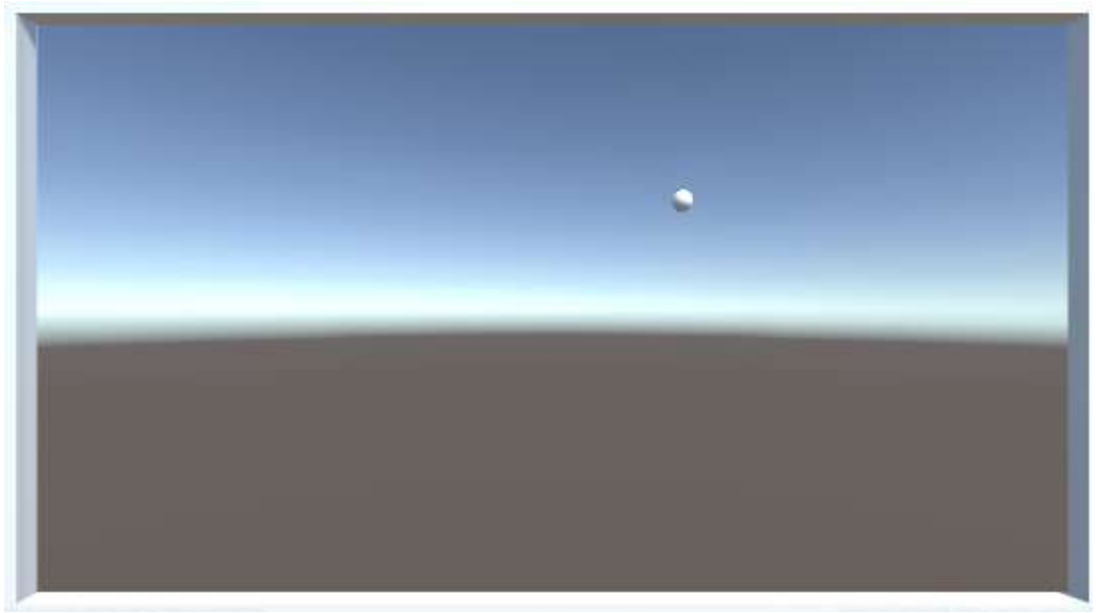
  →このプログラムを持つオブジェクトの「Rigidbody」の

  velocity（x,y,zベクトルの速度情報）を指定

- = new Vector3(10.0f, 10.0f, 0.0f)

  →3軸ベクトルを生成し上記velocityに代入

- ゲーム実行
- ボールが右上に動き出して壁に当たると跳ね返るはず

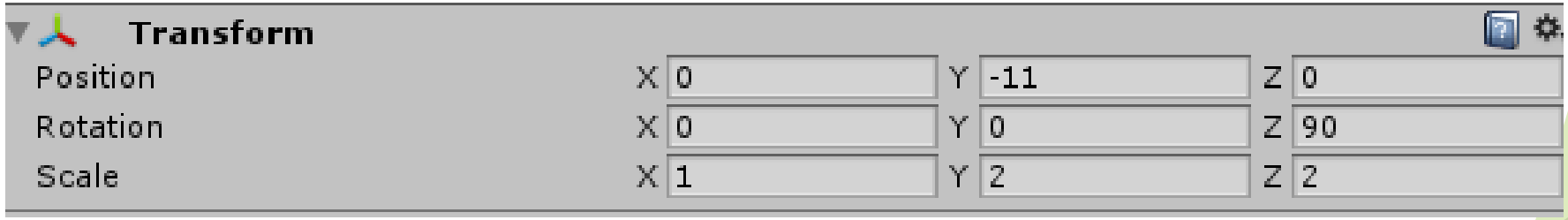


- 与えたのは初速だけ、PM_Ballにより跳ね返る物理法則を付与したため後は勝手に動いてくれる

- 次はバー（プレイヤーが操作するもの）を作る

6.3 バー作成

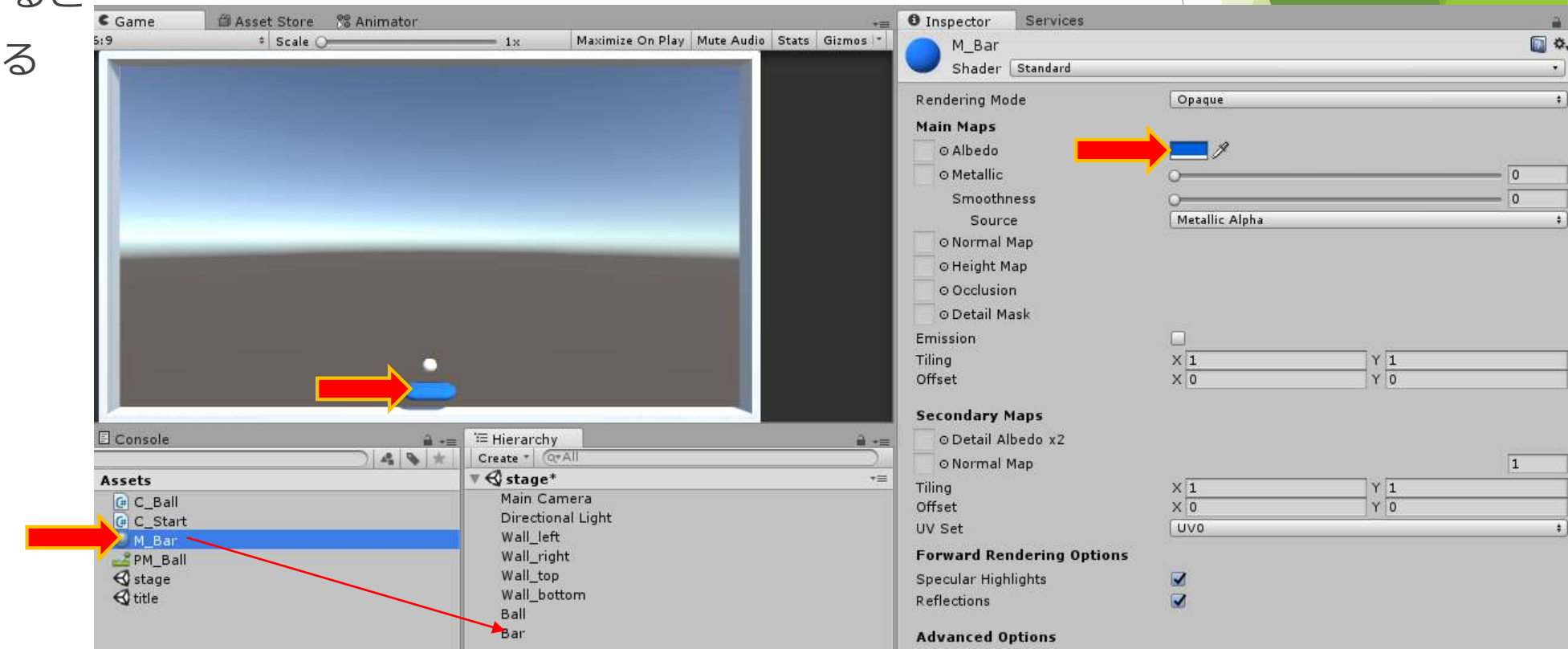
- 左右カーソルキーで動くバーを作る
- 「GameObject」→「3D Object」→「Capsule」
- 名前は「Bar」
- インスペクタを↓と同じに



- ゲームビューでボールの下の方にバーがある状態

- 壁が白、ボールは白、バーも白だと見づらいのでバーに色を付けます
- 「Assets」→「Create」→「Material」
- 名前は「M_Bar」

- 「M_Bar」を「Bar」にドラッグ＆ドロップ
- M_Barのインスペクタを開く
- 好きな色を選択すると
  バーの色が変化する



- 次はカーソルキーでバーを動かすプログラムを書きます
- 「Assets」→「Create」→「C# Script」
- 名前は「C_Bar」
- C_BarをBarにドラッグ＆ドロップ
- C_Barを開く

- 次ページの通りに書く

```
public class C_Bar : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {
        if(Input.GetKey(KeyCode.LeftArrow))
        {
            transform.position += new Vector3(-10.0f,0.0f,0.0f) * Time.deltaTime;
        }
        else if(Input.GetKey(KeyCode.RightArrow))
        {
            transform.position += new Vector3(10.0f, 0.0f, 0.0f) * Time.deltaTime;
        }
    }
}
```

- Input.GetKey(KeyCode.～)は～というキーが押されている間trueを返す
- transform.positionはこのプログラムを持つ(今はBar)のpositionを示す
- += new Vector3(...)はx方向に-10or+10だけ加算する
- *Time.deltaTimeは上記-10or+10を秒速換算する

- 書けたら保存してゲーム実行

- 左右カーソルキーでバーが動けばおっけー
- 動かないorエラーがすごいという人は高確率で大文字小文字を間違えています
- Unityでコード書くときは大小文字に気を付けていきましょう

- 今はどういう意味で書いてるか詳しく分からないと思いますが
  慣れてくると分かってきます。慣れましょう

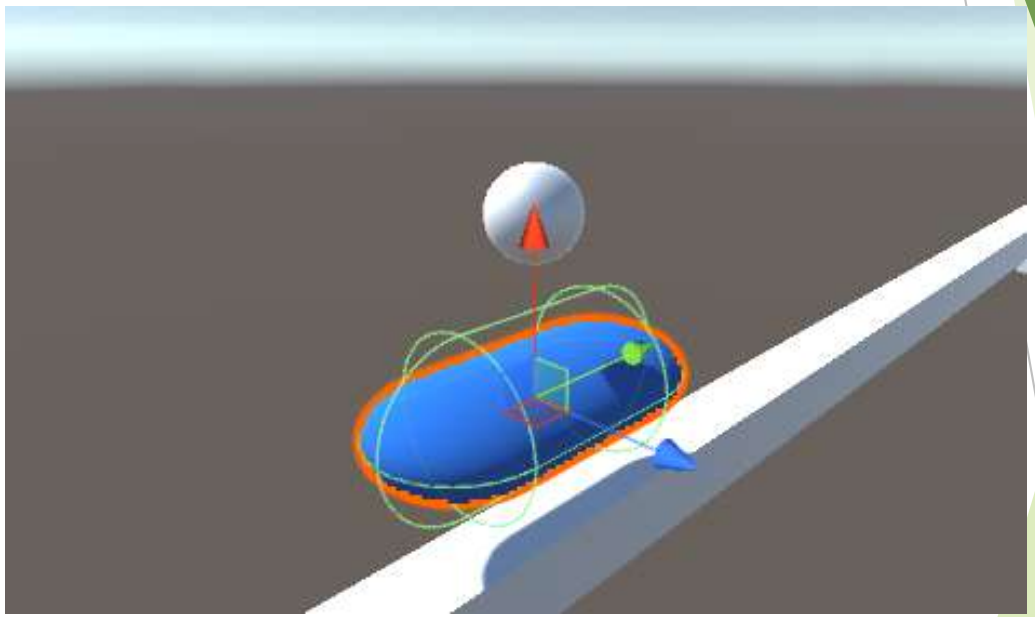
- で、実はこのバーはおかしくて
  ①壁をすり抜ける
  ②よく見るとボールがバーのちょっと上で跳ね返る
  となっています
- ①から修正します

- C_Barを開く
- Update内に書き足し

```
void Update () {
    if(Input.GetKey(KeyCode.LeftArrow))
    {
        if (transform.position.x > -22.0f)
        {
            transform.position += new Vector3(-10.0f, 0.0f, 0.0f) * Time.deltaTime;
        }
    }
    else if(Input.GetKey(KeyCode.RightArrow))
    {
        if (transform.position.x < 22.0f)
        {
            transform.position += new Vector3(10.0f, 0.0f, 0.0f) * Time.deltaTime;
        }
    }
}
```

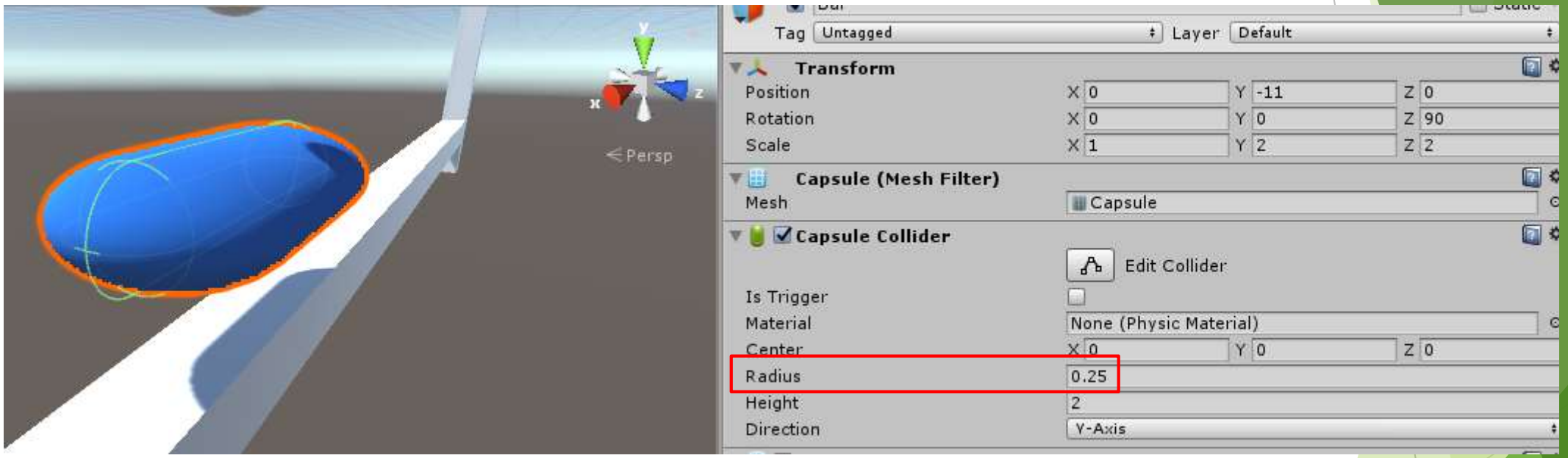
- それぞれ左右の壁より内側までに移動を制限する
- これでゲーム実行するとバーが壁をすり抜けなくなる

- 続いて②
- Barの当たり判定の大きさを変える
- ヒエラルキにあるBarをダブルクリック
- シーンビューにBarがアップされる



- →の緑線が当たり判定ライン
- バーに比べて大きいのが分かる
- この緑線をバーに合ったサイズにする

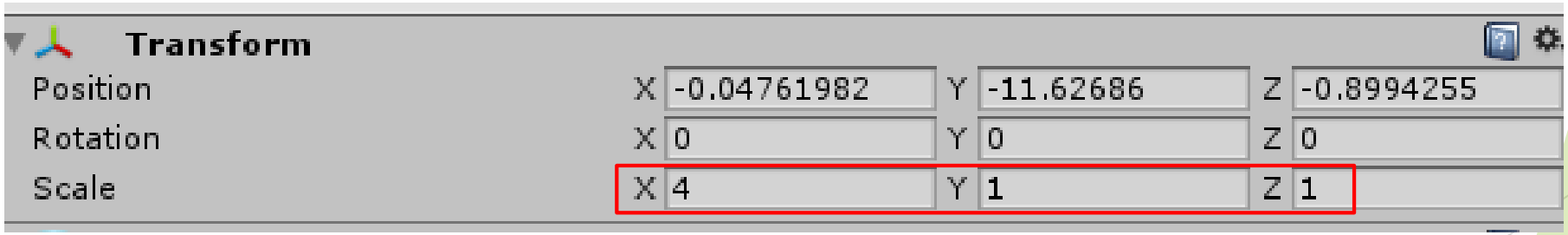
- バーのインスペクタを開く
- 「Capsule Collider」の「Radius」を0.25に
- 結構良い感じになる



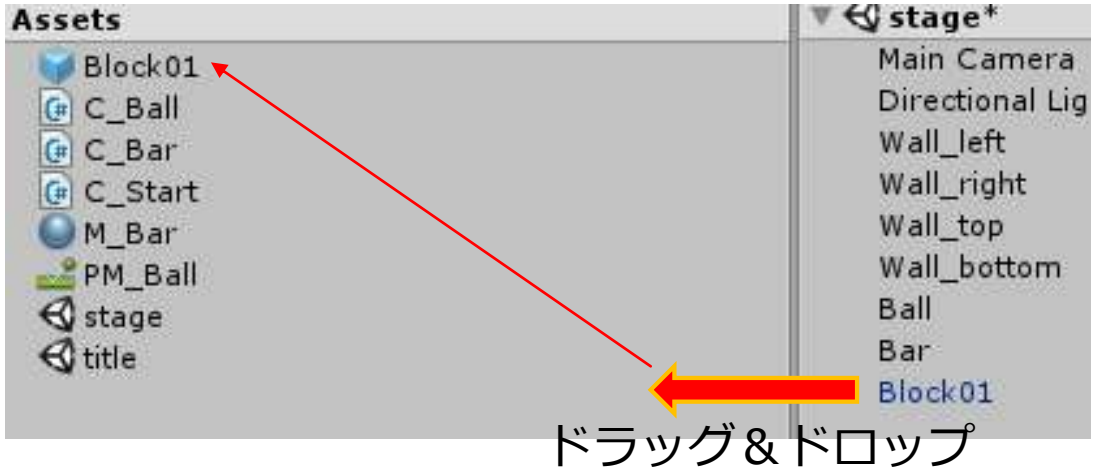
- これでバーに関する問題解決

6.4 ブロック作成

- ここからはブロックを作っていきます
- 「GameObject」→「3D Object」→「Cube」
- 名前は「Block01」
- インスペクタからScaleだけ変更
- 変な場所にあるかもいれないけどとりあえず今は無視。Scaleだけ



- ヒエラルキにある「Block01」をAssetsにドラッグ＆ドロップ
- するとAssetsにオブジェクト「Block01」ができます
- これを「プレハブ化」といいます
- 重要なので覚えときましょう



ドラッグ＆ドロップ

- ブロック崩しのように大量に同じブロックを生成したいといったときにプレハブ化したオブジェクトを使うととても便利です
- ヒエラルキではなくAssetsにあるオブジェクトをコピーして生成しシーン上に配置していきます

- プレハブ化したブロックを使うのでヒエラルキの「Block01」は削除
- ブロックにも色を付けるので

  「Assets」→「Create」→「Material」

  名前は「M_Block01」
- M_Block01のインスペクタから色を適当に設定
- M_Block01をプレハブ化したBlock01にドラッグ＆ドロップ

- 続けてブロック生成を行うプログラムを書きます
- 「Assets」→「Create」→「C# Script」
- 名前は「C_Stage」

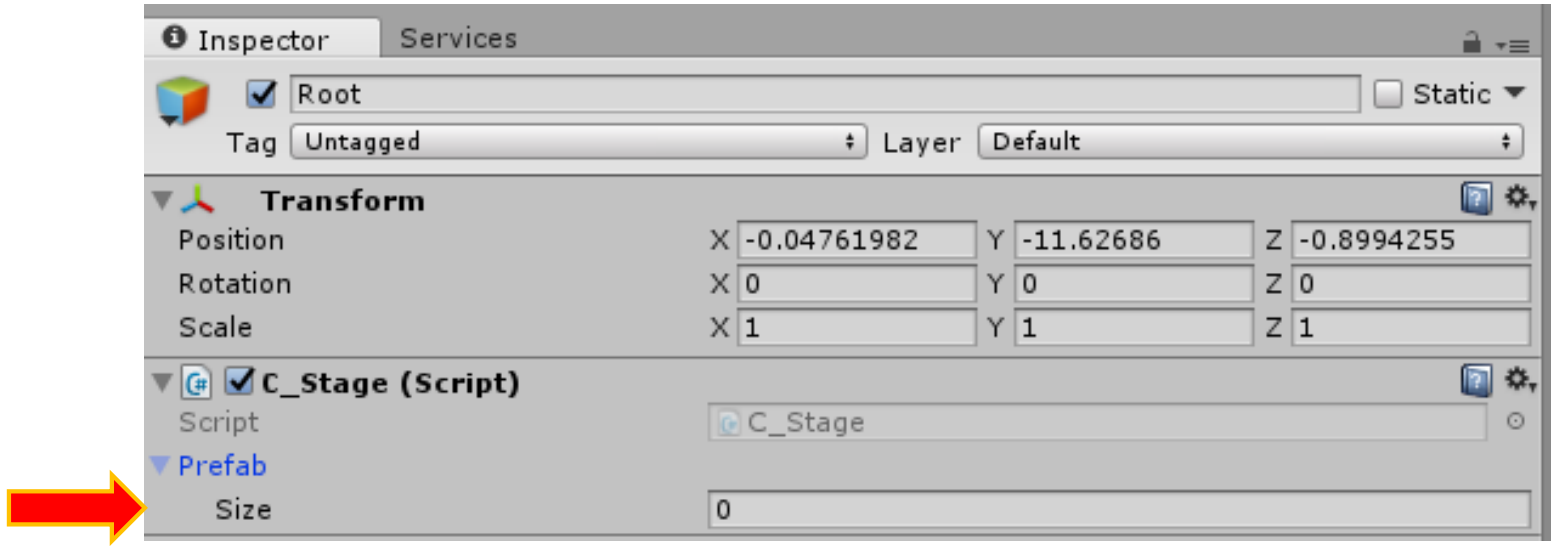
- 「C_Stage」をどのオブジェクトに持たせるか？
- Tileシーンのときと同様にそれ用のオブジェクトを作ります
- 「GameObject」→「Create Empty」
- 名前は「Root」
- C_StageをRootにドラッグ＆ドロップ

- C_Stageを開く
- ↓と同じように書く

```
public class C_Stage : MonoBehaviour {
    public GameObject[] prefab;
```

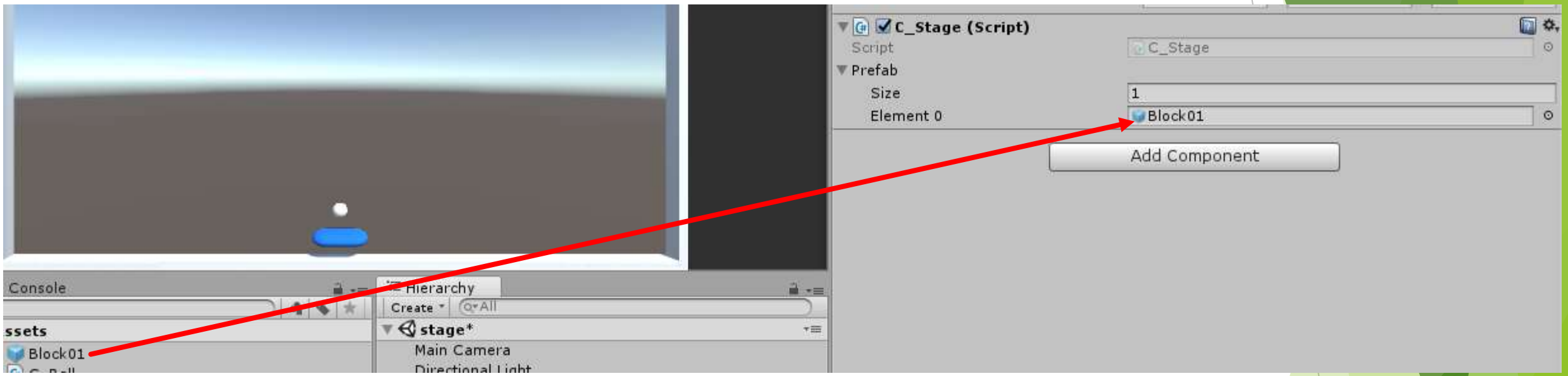
- 今までと書く場所が違うので注意！
- StartとかUpdateの関数内ではなくクラス名の真下に書きます
- 書けたら保存してUnityの画面に戻る

- Rootのインスペクタを開く



- Prefab にSizeという項目が増えるのでクリックで開き、0の部分を1にする

- するとElement0という項目が増えるので
  AssetsにあるBlock01をドラッグ＆ドロップ



- これでプログラム上でブロックオブジェクトを扱うことが出来ます

- C_Stageを開いて↓のコードを書く

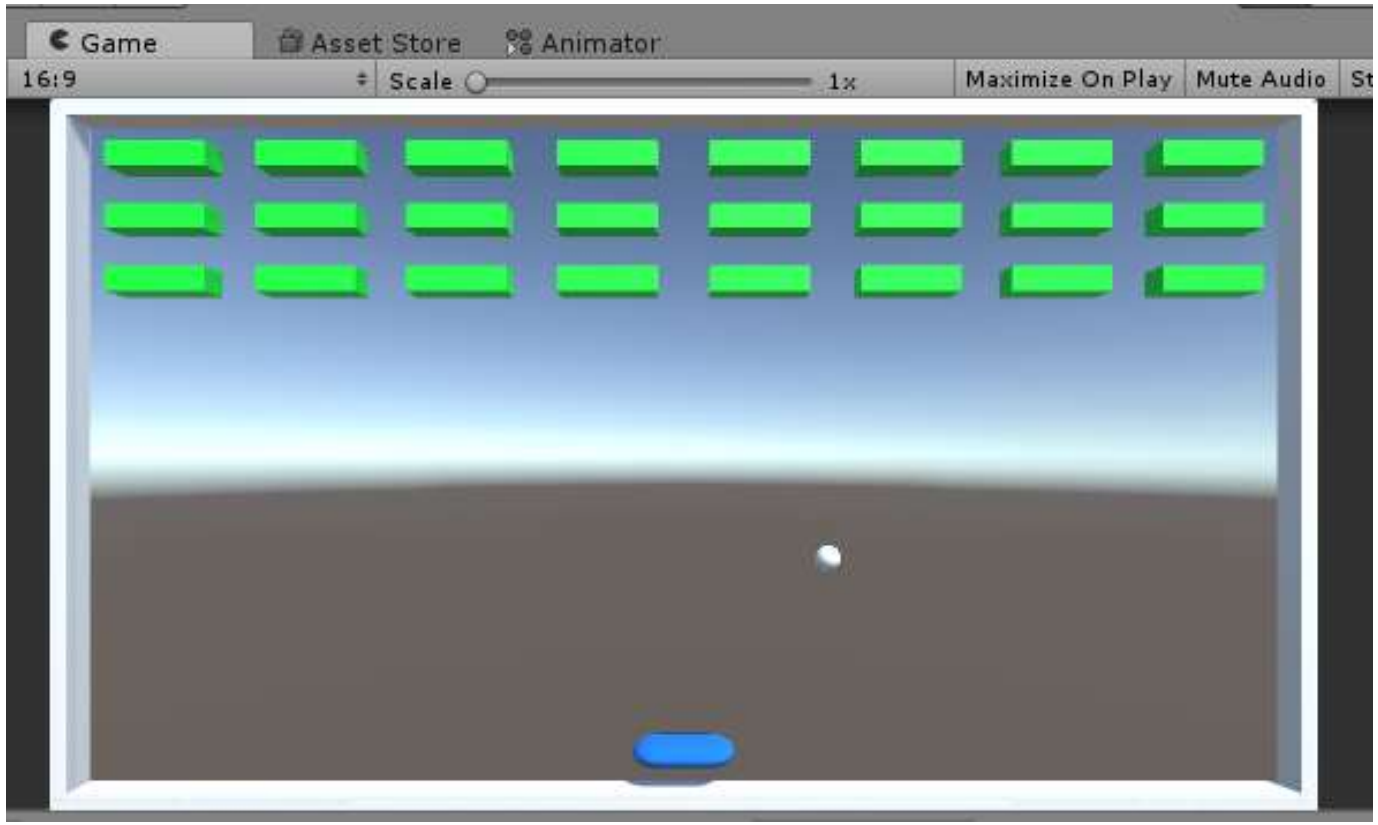
```
public class C_Stage : MonoBehaviour {
    public GameObject[] prefab;

    // Use this for initialization
    0 個の参照
    void Start()
    {
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 8; j++)
            {
                GameObject block_1 = GameObject.Instantiate(prefab[0]) as GameObject;
                block_1.transform.position = new Vector3(-21.0f + (6.0f * j), 13.0f - (2.5f * i), 0.0f);
            }
        }
    }
}
```

- prefab[0]にはさきほど設定したBlock01が入ってます
- プログラム上からオブジェクトを生成するとき

  GameObject 変数名 = GameObject.Instantiate(使用オブジェクト) as GameObject

  で生成します
- また生成したオブジェクトは元のオブジェクトと同じ情報を持つため全て同じ位置にできます
- そのため生成するオブジェクト24個に対しnew Vector3で異なる位置座標を与えます

- 書けたら保存してゲーム実行



- こんな感じに表示されればおっけー
- ただこのブロックは消えないので、ボール衝突時に消えるようにします

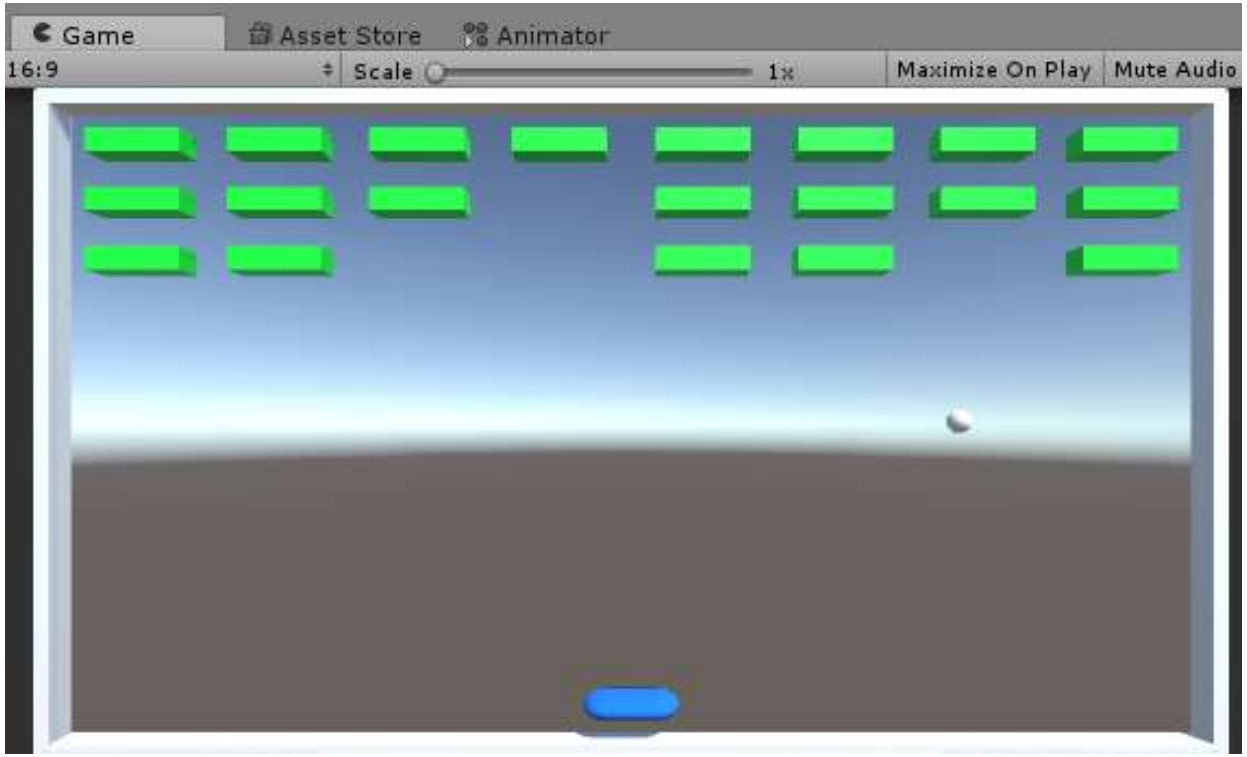
- 「Assets」→「Create」→「C# Script」
- 名前は「C_Block」、C_BlockをBlock01にドラッグ＆ドロップ
- 下のコードを書く

```
using System.Collections.Generic;
using UnityEngine;

0 個の参照
public class C_Block : MonoBehaviour {

    0 個の参照
    void OnCollisionEnter(Collision other)
    {
        GameObject.Destroy(this.gameObject);
    }
}
```

- OnCollisionEnterもStartやUpdateと同様に特別な関数で
  このプログラムを持つオブジェクトが他のオブジェクトを接触した時に
  呼び出される関数になっています

- GameObject.Destroy(オブジェクト)でオブジェクトを削除する
- this.gameObjectはこのプログラムを持つオブジェクト
- よって、他のオブジェクト（ボール）と衝突時に自身（ブロック）が消える

- ゲーム実行



- こんな感じでボールに当たったブロックだけ消えていく
- ブロック崩しっぽいものになってきた

- 今回はここまで

- 次回（来週）はもうちょっとゲームらしさを出すために
  色々機能を追加したりする予定です

- というわけでみなさんお疲れ様でした！