Unity講座1.2

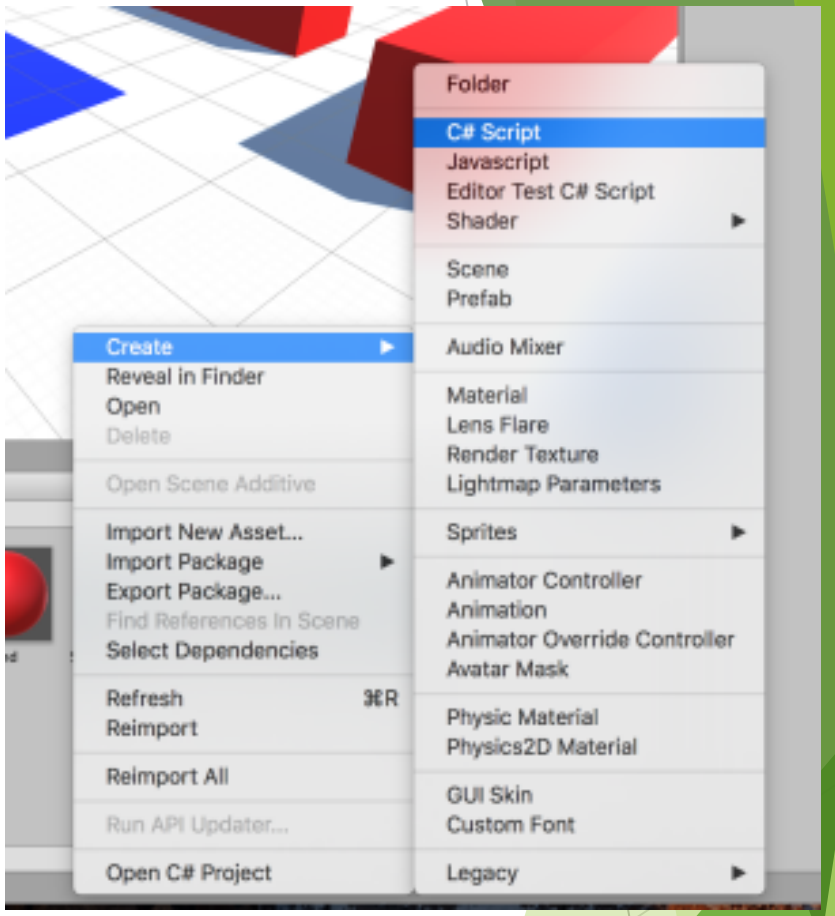
～ナビゲーションで鬼ごっこ2～

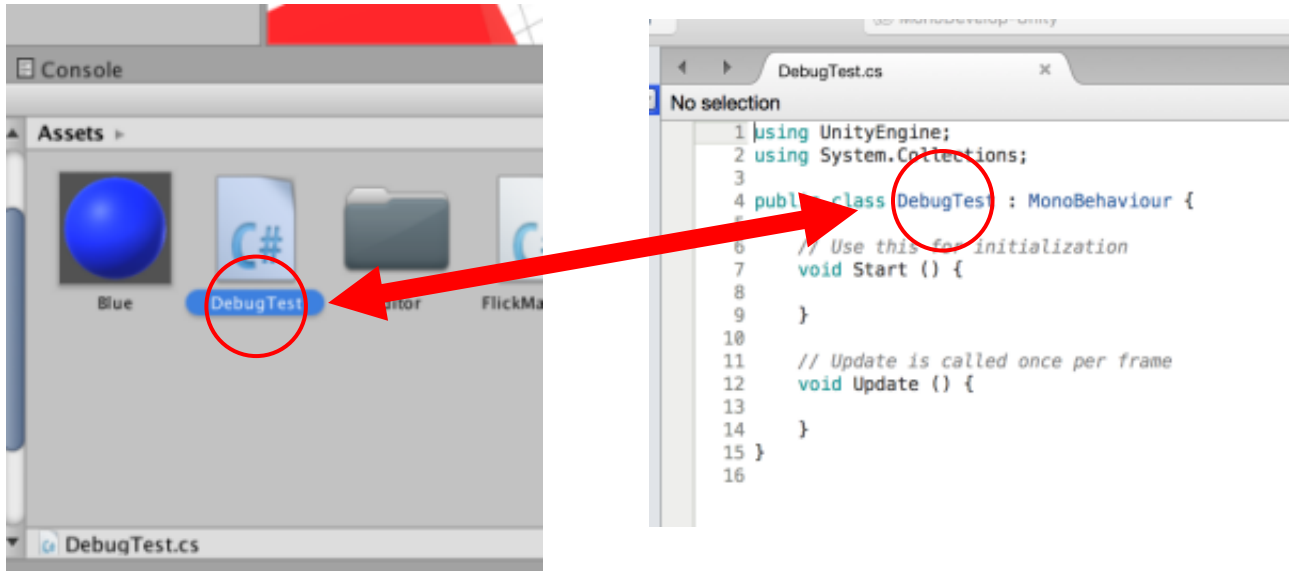
スクリプトを書く

　Unityでプログラムを書く際は、命令をスクリプトファイルに書き込み、それをオブジェクトに持たせるという形式をとります。

　画面下側のプロジェクトウィンドウで右クリックし、Create → C#Scriptと選択し、ファイルを作成してください。UnityではJavaScriptとC#が使えますが、今回はC#を使います。

　ファイル名とクラス名が一致していないとエラーが起こり、実行できません。片方で変更してももう片方に反映されないことがあるので、名前を変えたら必ず下図のように一致していることを確認しましょう。



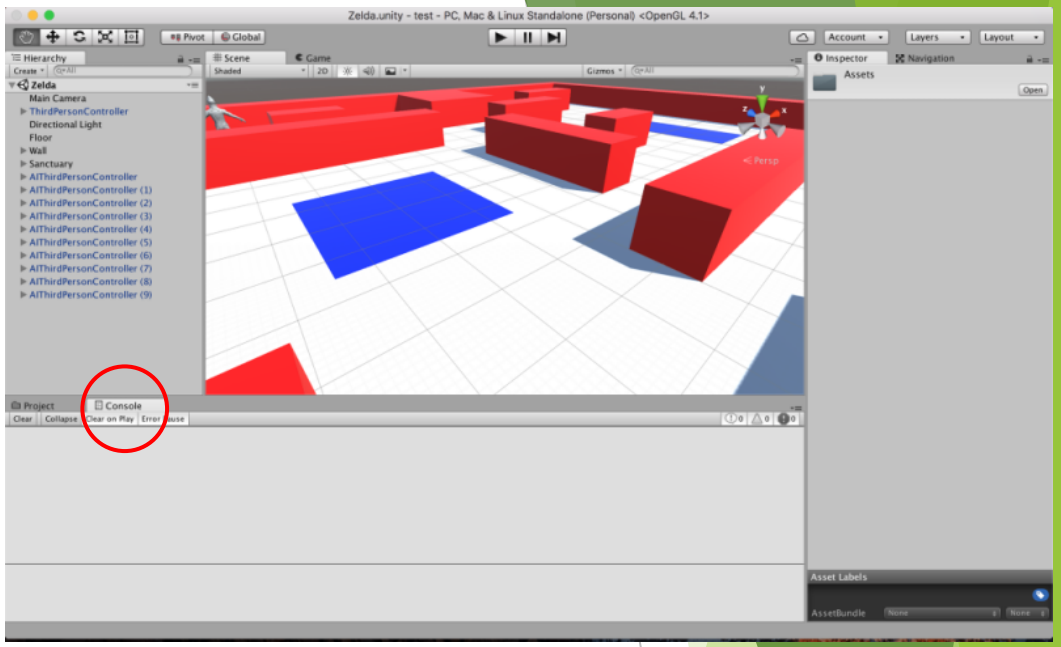


コンソール

　スクリプトを書き始めると、エラーが発生することが非常に多くなります。エラー文は、コンソールウィンドウに表示されます。デフォルトの配置では、プロジェクトウィンドウと同じスペースに配置されています。

　Unityで作成したスクリプトには、はじめからStart関数とUpdate関数が用意されています。Startはシーンの開始時に呼び出され、Updateは毎フレーム呼び出されます。

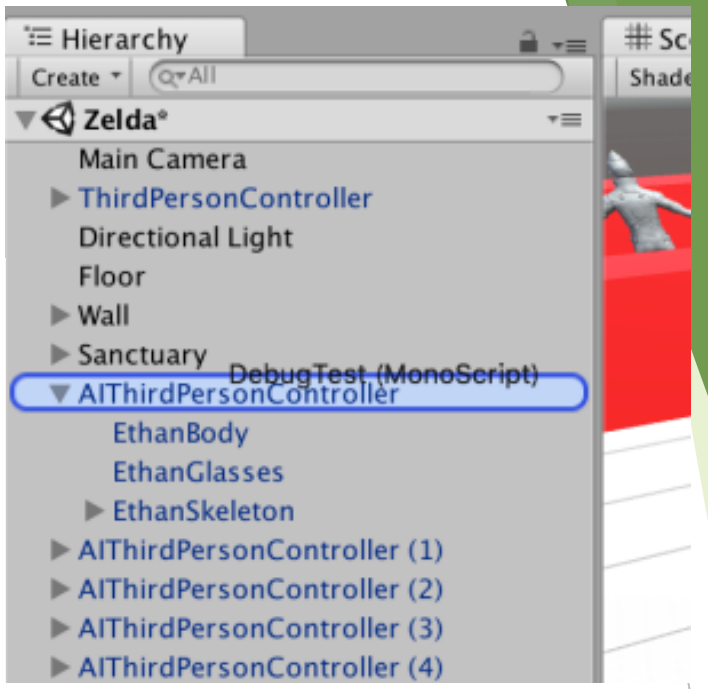
　Unityでは、Debug.Logという関数を使うと、任意の文字列をコンソールに表示することができます。先ほど作成したファイルのスタート関数を、右図のように編集してください。

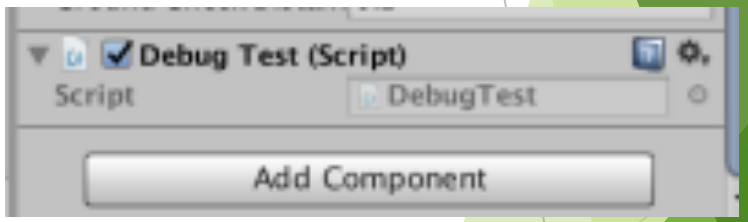


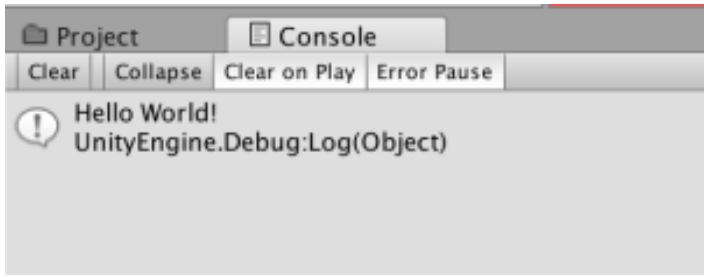

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class DebugTest : MonoBehaviour {
5
6     // Use this for initialization
7     void Start () {
8
9         Debug.Log ("Hello World!");
10
11    }
12
13    // Update is called once per frame
14    void Update () {
15
16    }
17 }
18
```

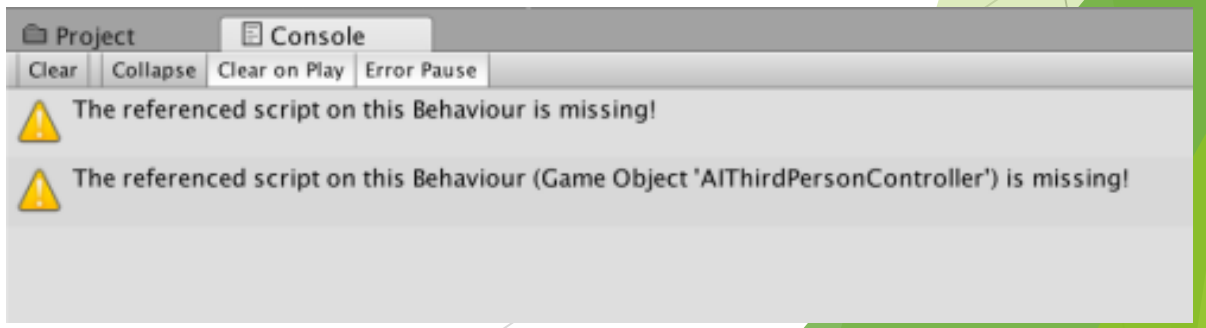
スクリプトの実行方法

　スクリプトは、作成しただけでは素材置き場に置いてあるだけなので、そのままゲームを実行しても何も起こりません。何かのオブジェクトに持たせる(「アタッチ」という)必要があります。
　右図のようにプロジェクトウィンドウからヒエラルキーの適当なオブジェクトにドラッグ&ドロップしてアタッチしましょう。インスペクターの一番下にスクリプトが追加されます。インスペクターのAddComponent→Scriptsからでもアタッチできます。
　ゲームを実行すると、スクリプトのStart関数が実行され、左下の画像のようにコンソールに表示されます。
　ファイル名とクラス名が異なる場合、右下のように Warningが表示され、スクリプトは実行されません。









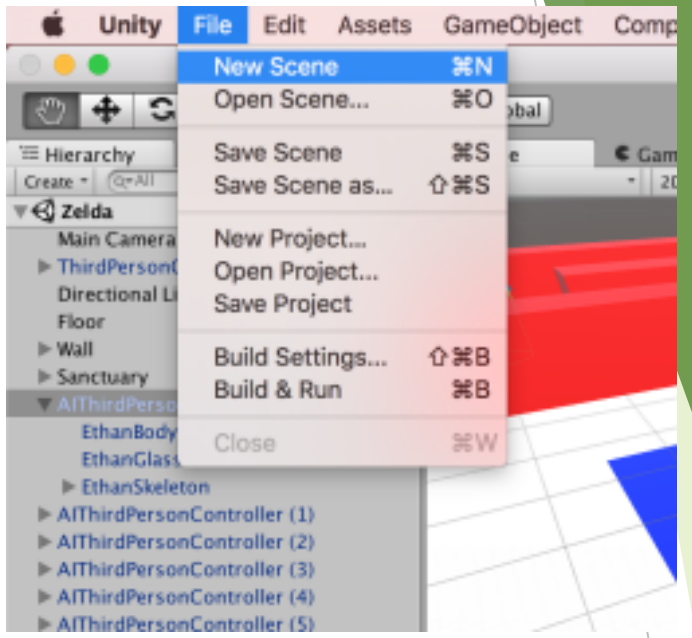
仕様の確認

　スクリプトの実行ができるようになったところで、今回目指す仕様をまとめておきます。

まず、ゲームのルールは、
・中継地点にあるアイテムを拾い、ゴールに集める
・所持できるアイテムは一度に一つだけ
・敵に追いつかれたら負け
とします。

必要な作業は、以下になります。
・ゲーム開始前の画面の作成
・経過時間の表示
・アイテムの設定
・クリア及びゲームオーバーの設定

シーンの追加

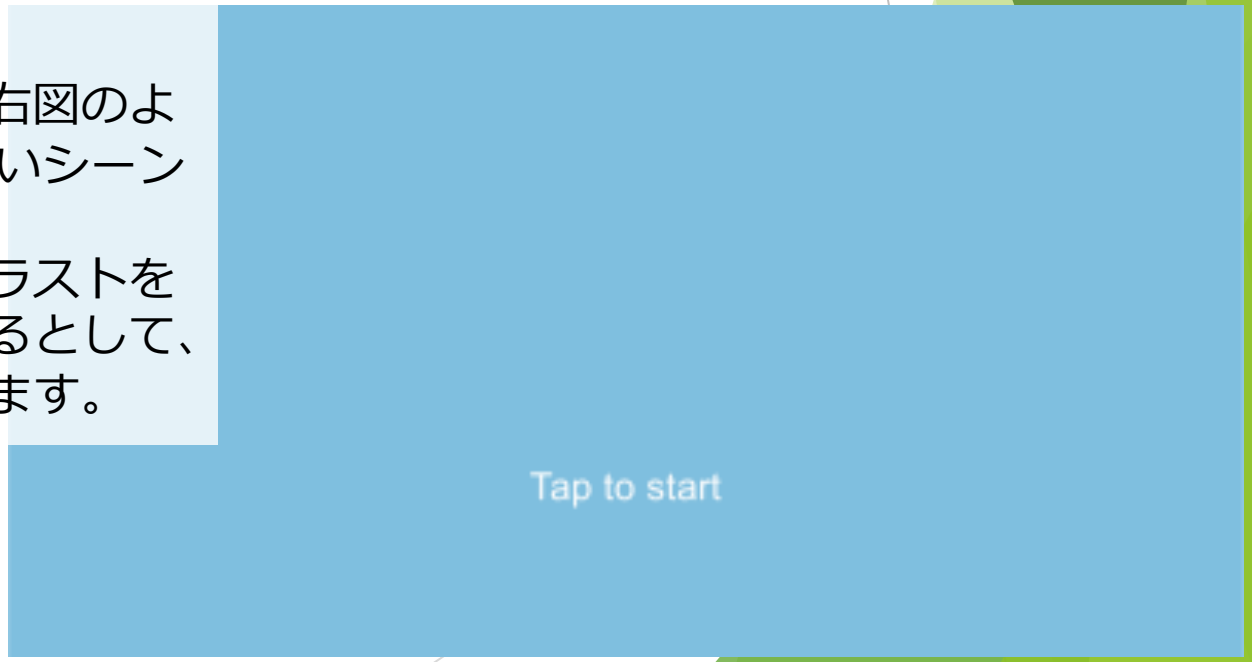


まずはタイトル画面を作成します。これまでの説明でも何度か「シーン」という言葉を使っていましたが、Unityでは、一つのゲームをいくつかのシーンに分割して作成していきます。

タイトル画面は、今まで作っていたゲーム画面とは全く違う構成の画面になるので、新しくシーンを追加することにします。

これまで作業していたシーンを保存したら、右図のようにFileメニューからNewSceneを選択し、新しいシーンを作成してください。

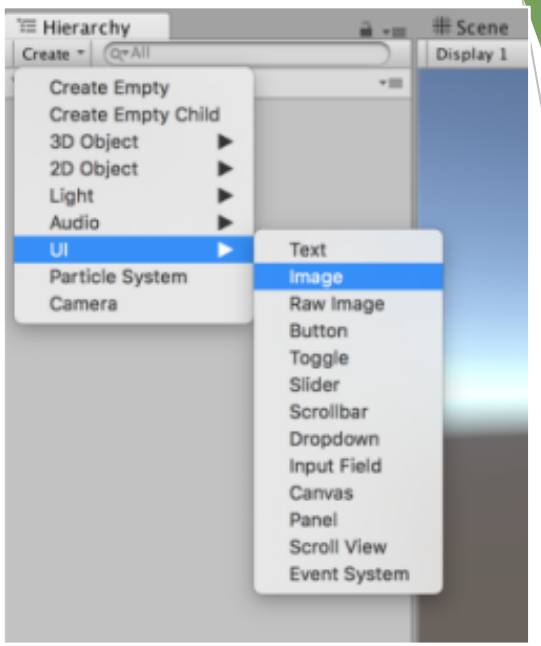
本来はここで背景やタイトルにふさわしいイラストを準備するところですが、それは専門の人に任せるとして、ここでは右下の図のような画面を作ろうと思います。

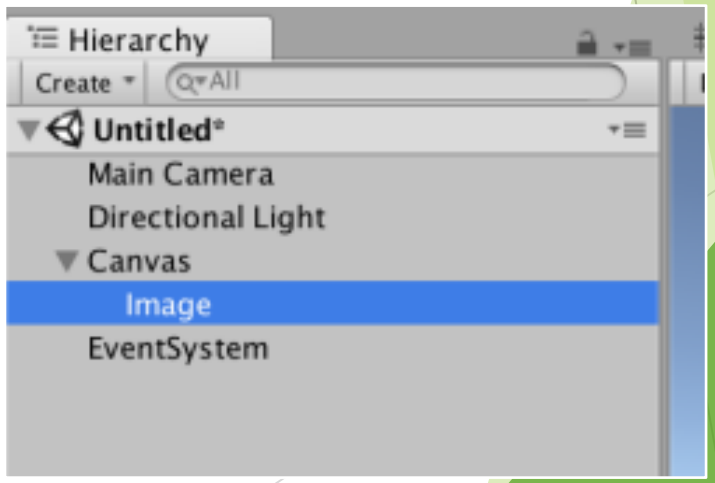


タイトル画面の作成

　まずは背景の画像を作成します。ヒエラルキーのCreateから、UI → Imageと選択して、画像を表示するための枠組みを呼び出します。すると右下図のように、ヒエラルキーにCanvasとEventSystemが自動的に作成されます。

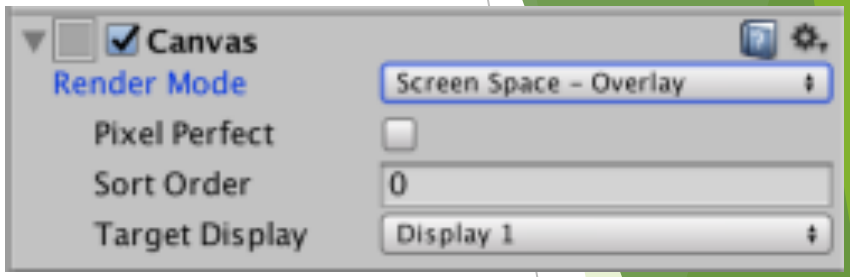
　キャンバスというのは、画像やボタン等のUIを設置するための仕組みで、基本的に2次元のものはキャンバス上に配置します。
　イベントシステムは、「クリックしたら～する」というような処理をする時に必要となるものです。

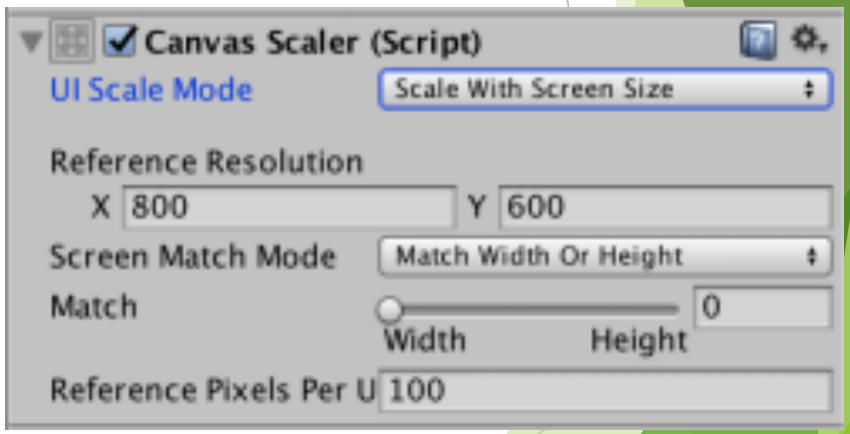




キャンバスの設定

　キャンバスは、ゲーム内での表示と、画面サイズに応じた拡大縮小の設定ができます。
　キャンバスのインスペクターを見ると、Canvasという項目があります。ここのRenderModeから表示方法が選べるのですが、
ScreenSpaceは、画面にそのまま貼りついたような表示
WorldSpaceは、ゲームの世界の中にキャンバスが置いてあるような表示ができます。今回はScreenSpace-Overlayを選択します。
　CanvasScalerは画面サイズに応じた、キャンバスの拡大縮小の設定ができます。今回はScaleWithScreenSizeを選択しておきましょう。

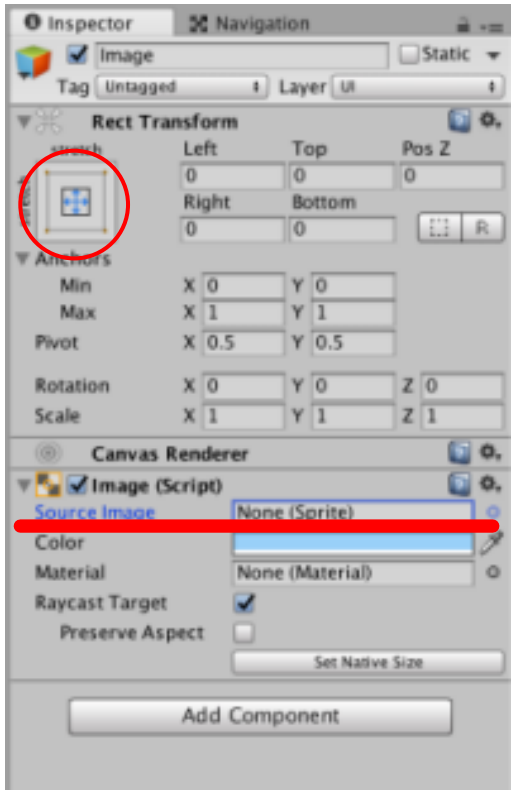


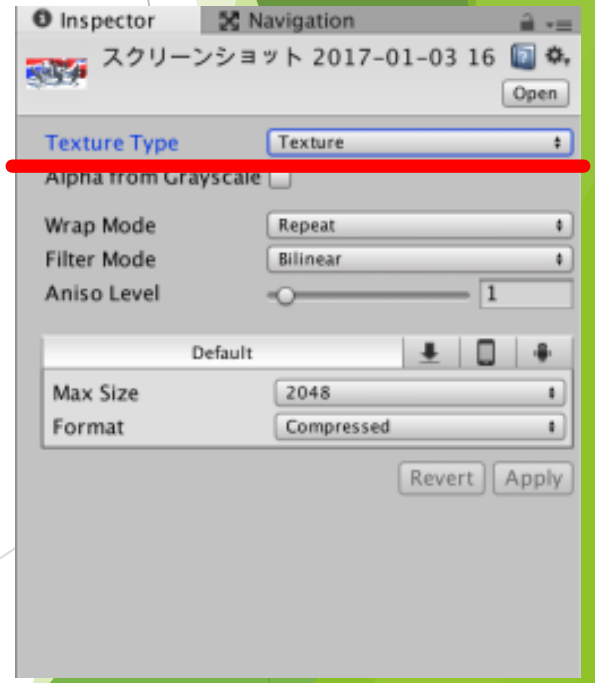


画像の設定

ヒエラルキーでImageを選択し、インスペクターからImageの配置を設定します。
　RectTransform内の左上にある四角は、配置の基準点を表しています。これをクリックし、右図にあるように青い十字を選択すると、上下左右の余白の大きさによって配置を決定するようになります。Left, Top, Right, Bottomを全て0にし、キャンバス全体を覆うようにしましょう。

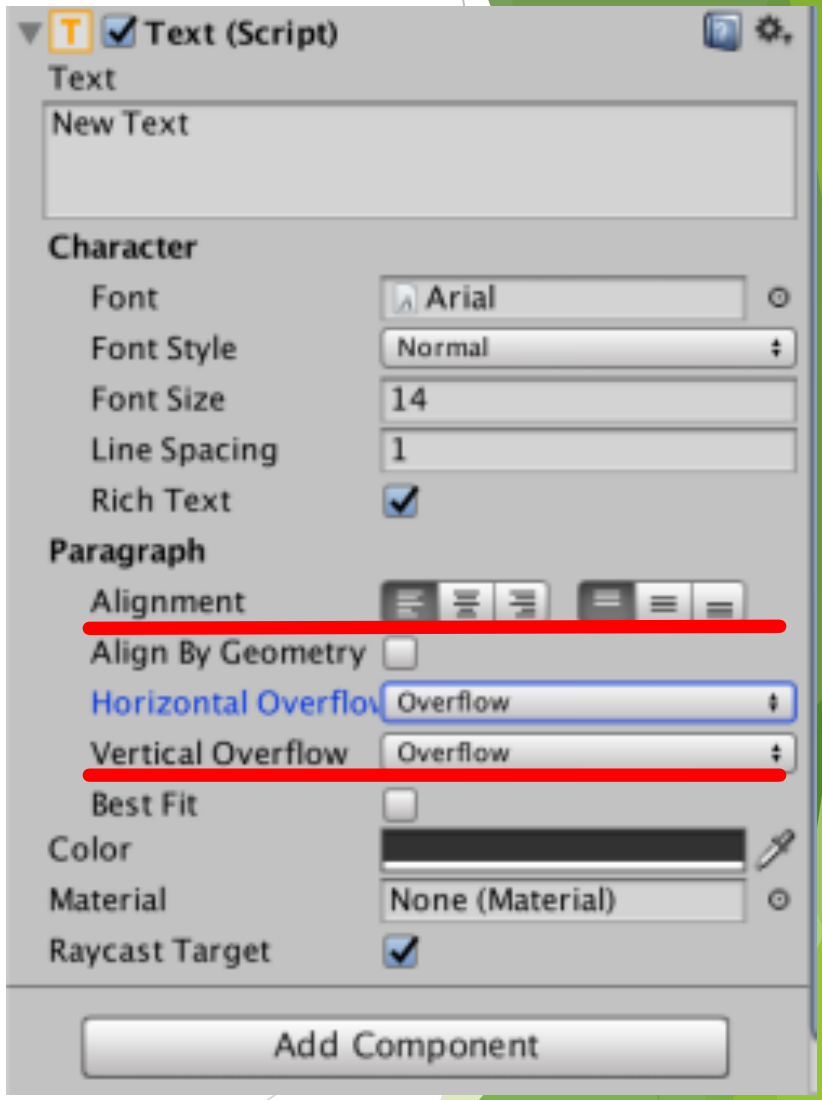
　ImageのSourceImageという項目には、表示したい画像を選択します。Unityに読み込んだ画像は通常、右図のようにTexureTypeがTextureになっていて、そのままではSourceImageに選択できません。これをSpriteに変更することで、SourceImageに選択できるようになります。
　今回は画像が無いので、SourceImageを選択せずにColorだけを変更しました。





テキストの設定

　Imageと同様に、ヒエラルキーのCreateからTextを追加します。Textは、表示する文字列や、文字色、文字の配置などの設定ができます。
　初期値では文字が左上に寄ってしまっているので、Alignmentの項目で中央に揃えておくと扱いやすいです。
　また、Overflowの項目が初期値のままだと、文字が枠をはみ出てしまう時に、何も表示されなくなってしまいます。慣れないうちはHorizontalとVertical両方をOverflowにしておくと良いと思います。

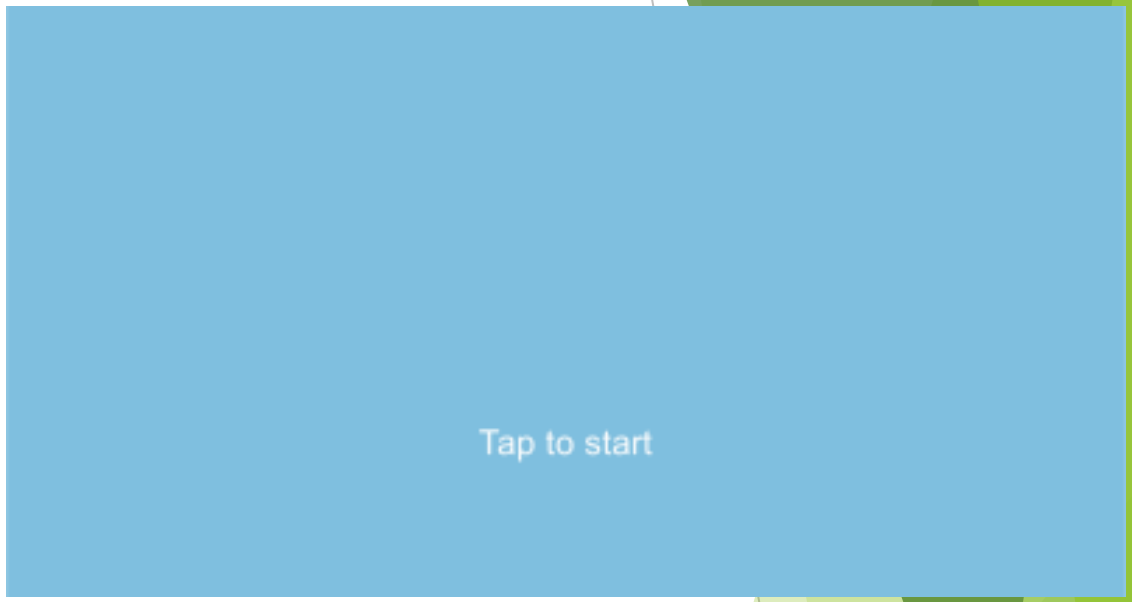


シーンの切り替え

　ここまでの作業で、右図のような画面ができたと思います。次は、このシーンからゲーム画面のステージへ移動するための設定をします。

　今回は、画面のどこかをクリックしたらシーンが切り替わるようにします。unityでは、クリック等をイベントという形で検知します。シーン切り替えの処理などを、特定のイベントに反応するように設定しておくことで、クリック時などのタイミングに処理を呼び出せるようになります。

　シーンを切り替えるためには、プログラムを書く必要があります。プロジェクトウィンドウから新しくスクリプトファイルを作成してください。



シーン切り替えのスクリプト

シーン切り替えのスクリプトは、右のようになります。初期状態から変更した箇所は赤線で示しました。

1~3行目のusingは、あらかじめunityに用意されていた命令の中から、使いたいものを指定して使えるようにするためのものです。今回はSceneManagementの中にあるものを使いたいので、3行目を追加します。

17~19行目は、外部から呼び出す関数です。Unity側で設定をすると、クリック時にこの関数が呼び出されます。18行目のLoadScene関数は、外部から与えられた引数(かっこ内の値)と一致する名前のシーンを探して切り替える関数です。ここでは17~19行目の関数に与えられた引数をそのまま与えてシーン切り替えを行っています。

このスクリプトを保存して、MainCameraにアタッチしてください。

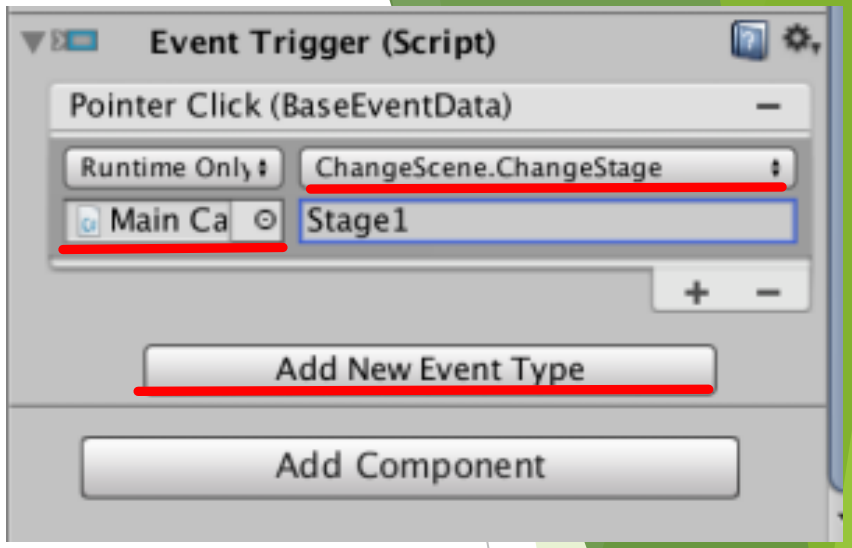
```
1   using UnityEngine;
2   using System.Collections;
3   using UnityEngine.SceneManagement;
4
5   public class ChangeScene : MonoBehaviour {
6
7       // Use this for initialization
8       void Start () {
9
10      }
11
12      // Update is called once per frame
13      void Update () {
14
15      }
16
17      public void ChangeStage(string stageName){
18          SceneManager.LoadScene (stageName);
19      }
20  }
```

※17行目のpublicを付け忘れると外部から呼び出せなくなり、後のイベントの設定ができなくなります。

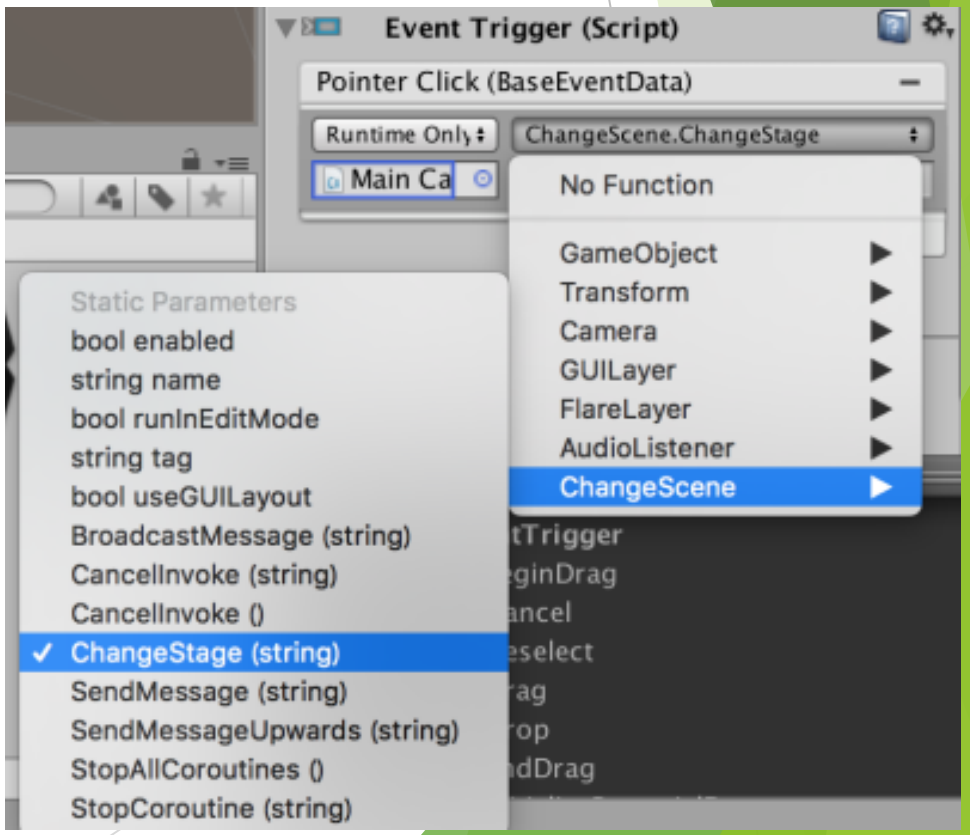
イベントの設定

次に、作成した関数をイベントから呼び出せるようにします。

キャンバスを選択し、インスペクターの一番下のAdd Componentから、Event → EventTriggerと選択し、イベントトリガーを追加します。AddNewEventTypeを押し、PointerClickを選択すると、「これがクリックされた時」のイベントが追加できます。右下の＋マークを押すと、PointerClickイベントが発生した時に実行される命令を指定できます。



左下の枠にMainCameraを指定し、右上のNoFunctionと書かれた部分は、作成したスクリプト(ChangeScene)→シーン切り替え用の関数(ChangeStage)と選択します。



MainCameraにスクリプトを持たせていない、シーン切り替えの関数にpublicを付けていない等の場合、これが表示されず、選択できません。

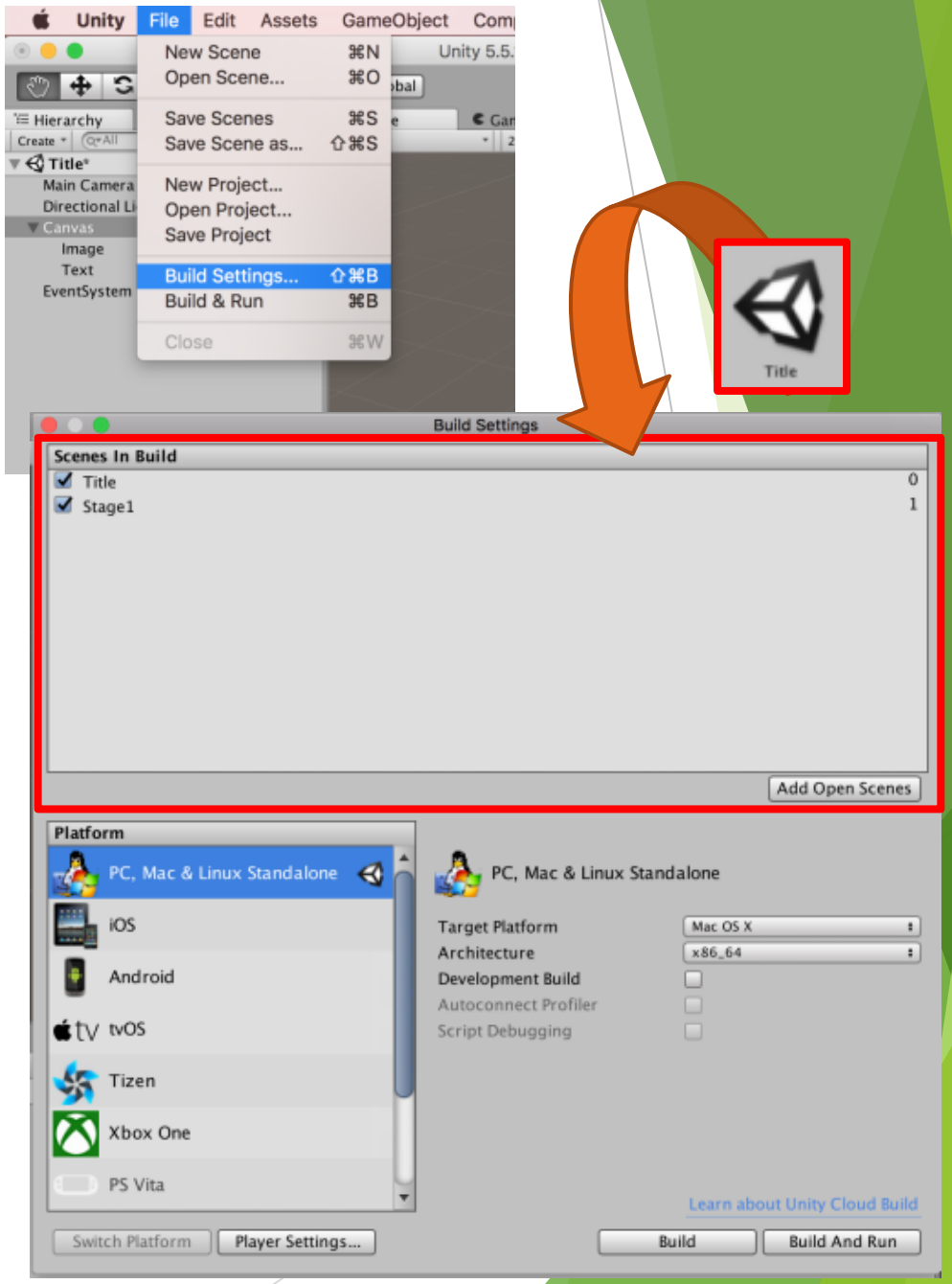
右下の枠に切り替え先のシーンの名前を入力します。

シーン切り替え

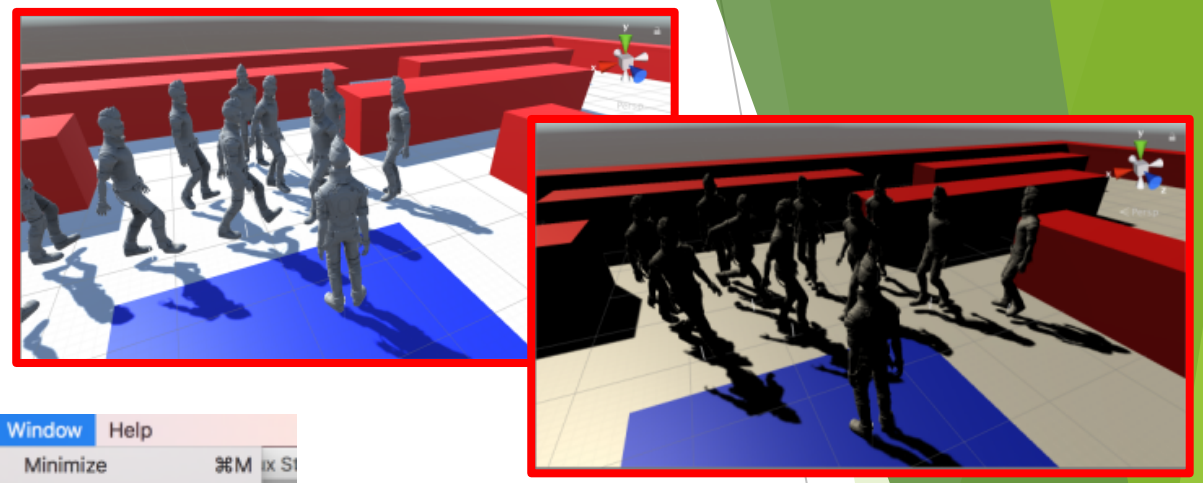
これでイベントの設定は完了しましたが、このままではシーン切り替えはできません。まずはFileメニューからBuildSettingsを開いてください(右図)。

右下の図のような画面が開いたら、上半分のScenesInBuildの部分を見てください。ここには実際にゲームとして出力するときに使用するシーンの一覧が表示されています。ゲームを起動すると一番上のシーンが呼び出され、シーン切り替えの関数が呼び出されると、この一覧から探して切り替えるので、この設定が必要になります。

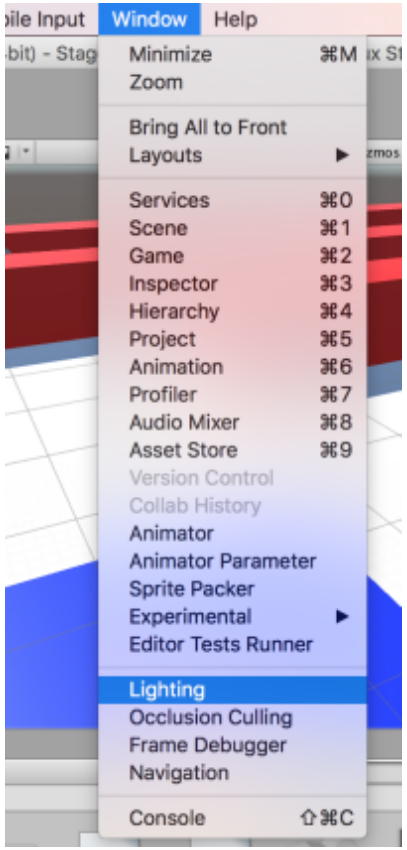
プロジェクトウィンドウのシーンファイルをドラッグ&ドロップで追加しましょう。BuildSettingsの画面を閉じて、タイトルのシーンから実行して試して見てください。画面をクリックするとゲーム画面に切り替えられるはずです。

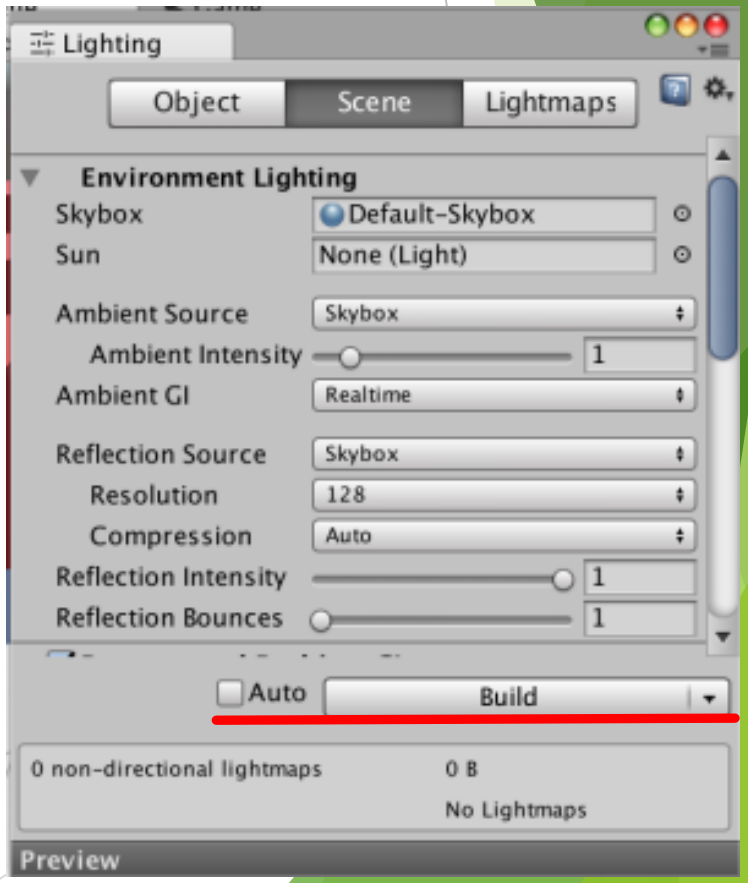


画面が暗い



ゲーム画面のシーンから開くと明るいのに、タイトル画面からシーンを切り替えるとなぜか画面が暗い…ということがよくあります。どうやらデフォルトでは光の当たり方の計算をリアルタイムで行う方式になっているようで、これを事前に計算させておくことで、解消できます。





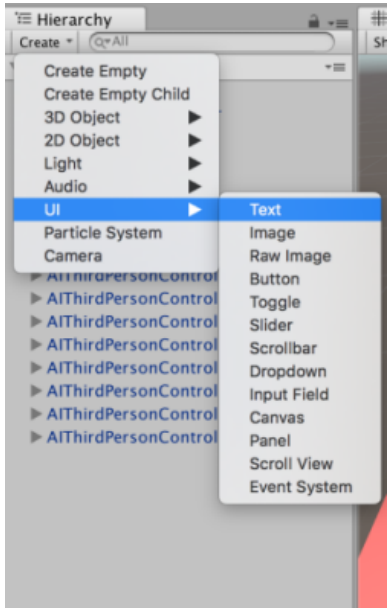
Windowメニューから**Lighting**を選択し、**Lighting**ウィンドウを表示します。画面下方の**Auto**のチェックを外し、**Build**ボタンを押してください。これで、シーン切り替え後も画面が暗くならなくなります。

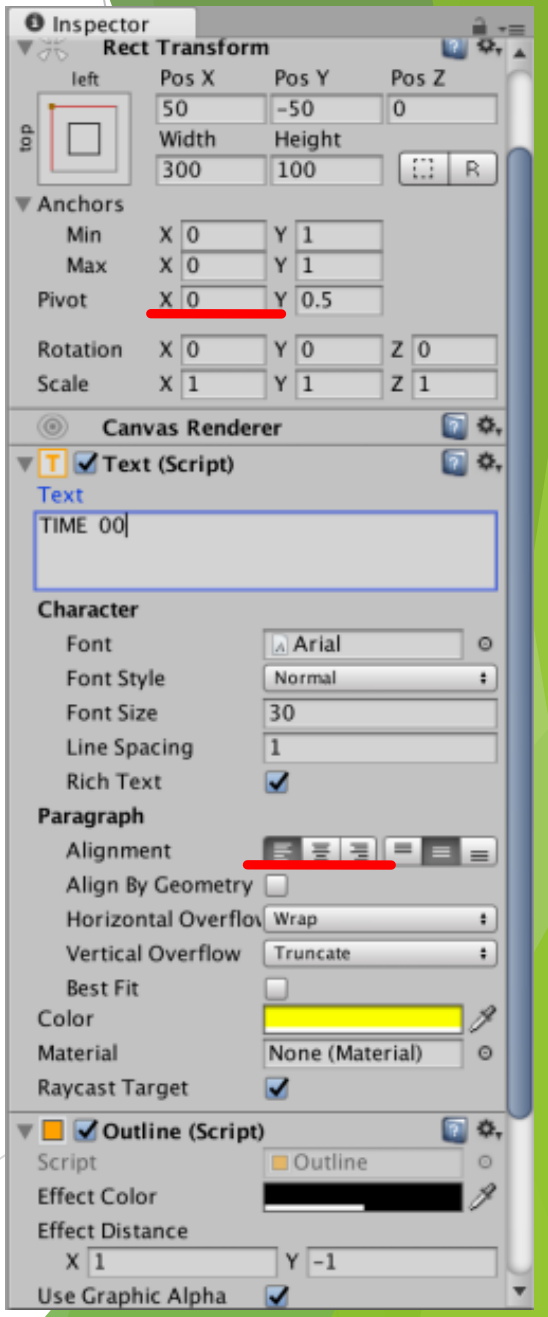
経過時間の表示(1/4)

　次に、経過時間の表示などの設定をしていきます。ヒエラルキーウィンドウでCreate→Textとしてテキストを作成します。タイトル画面の時と同様にキャンバスやテキストを設定しましょう。今回は時間によって文字数が変化するため、PivotとAlignmentの設定によって、テキストの左端が固定されるようにしました(右図)。

　AddComponent→UI→Effects→Outlineと選択すると、テキストの縁取りを追加することができます。

　ゲームビューを見ながらテキストを画面上の好きな位置に配置したら、プロジェクトウィンドウに新しいスクリプトファイルを作成してください。次は経過時間をテキストに反映するプログラムを書きます。





経過時間の表示(2/4)

今回は、C#に用意されているStopwatchというクラスを使って、経過時間を計測しようと思います。

右のプログラムで実装しました。一つずつ説明します。

UnityEngine.UIの中にあるテキストクラスを扱うので、4行目でUnityEngine.UIをインポートします。

8行目は、StopWatchを準備して、swという名前をつけています。StopWatchクラスの名前が長いので難しく見えますが、int a = 10; などと同じことをしています。左辺でStopWatchの容れ物を用意して、右辺でnewによってStopWatchの実体を作って、左辺の容れ物に入れているようなイメージです。

9行目のようにpublicをつけて変数を宣言すると、インスペクターから値を指定できるようになります。

次に続きます。

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.UI;
5
6   public class Timer : MonoBehaviour {
7
8       System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
9       public Text text;
10
11      // Use this for initialization
12      void Start () {
13          sw.Start ();
14      }
15
16      // Update is called once per frame
17      void Update () {
18
19          text.text = "TIME  "+ (int)sw.Elapsed.TotalSeconds;
20
21      }
22  }
```

経過時間の表示(3/4)

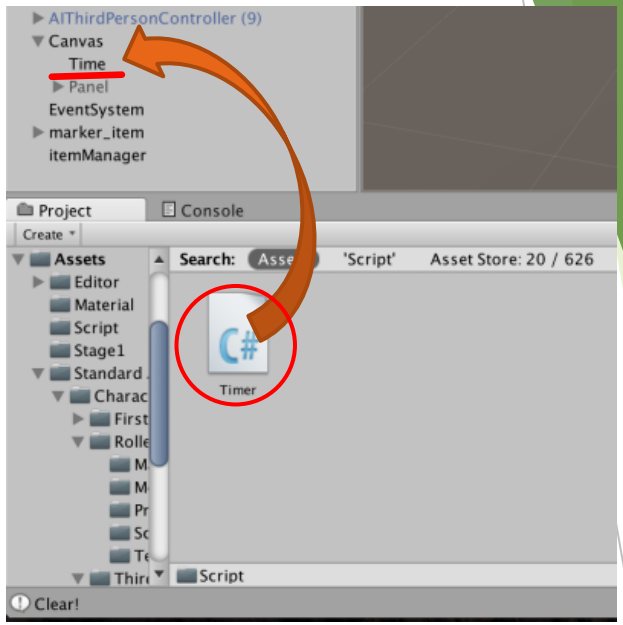
StopWatchには、4つの基本的な関数があります。
・Start(計測開始/再開)
・Stop(計測停止)
・Reset(計測結果の消去)
・Erapsed(計測結果の出力)
Reset以外ではリセットされないため、Stop→Startとすると一時停止と再開も実現できます。また、ErapsedはStopしなくても使えるため、測り続けながら出力することもできます。

19行目で、表示文字列の更新を行います。C#では、各要素を+で連結して文字列を構成します。また、(型名)を変数の前につけると型の変換が行えることを利用して、小数点以下の切り捨てを行います。
「TIME 」という文字列と、swから出力された秒数をint型(整数)に変換したものを連結して表示します。

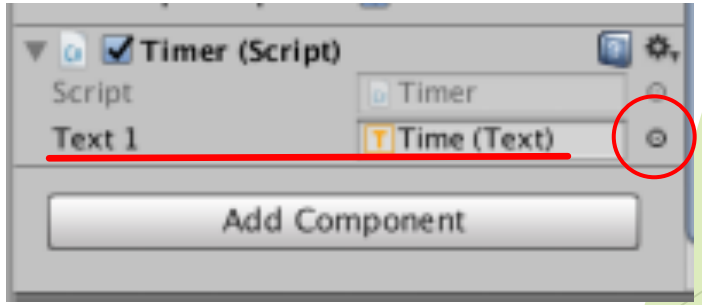
```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.UI;
5
6   public class Timer : MonoBehaviour {
7
8       System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
9       public Text text1;
10
11      // Use this for initialization
12      void Start () {
13          sw.Start ();
14      }
15
16      // Update is called once per frame
17      void Update () {
18
19          text1.text = "TIME  "+ (int)sw.Elapsed.TotalSeconds;
20
21      }
22  }
```

経過時間の表示(4/4)

　作ったスクリプトを、ヒエラルキー上のテキストにドラッグ&ドロップしてください。右図のように、インスペクターにスクリプトが追加されます。



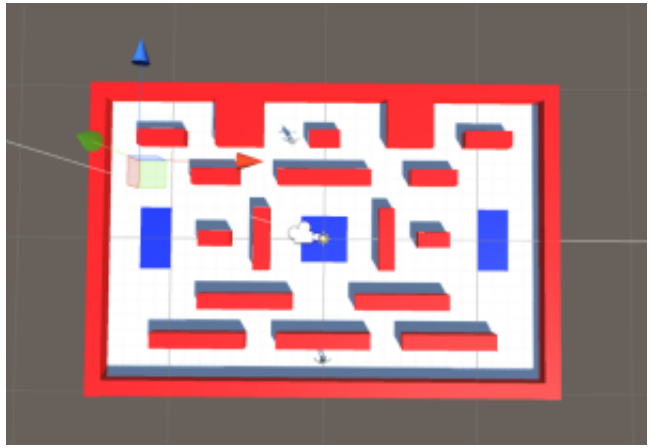
　publicで宣言したので、text1変数が表示されています。右の丸いマークをクリックし、このオブジェクト(デフォルトではText)を選択してください。
　これで、このtext1変数には、このオブジェクトのテキストコンポーネントが指定され、表示する文字列を変更できるようになりました。

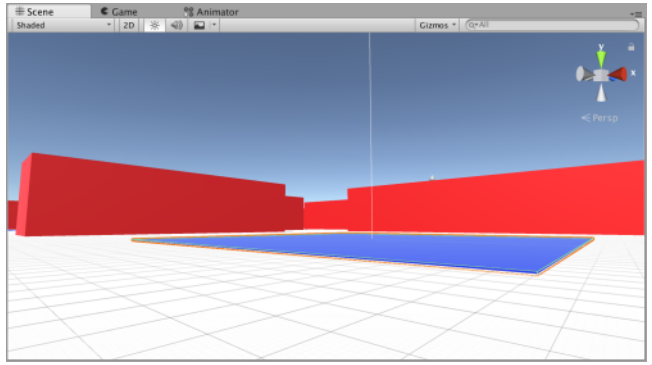


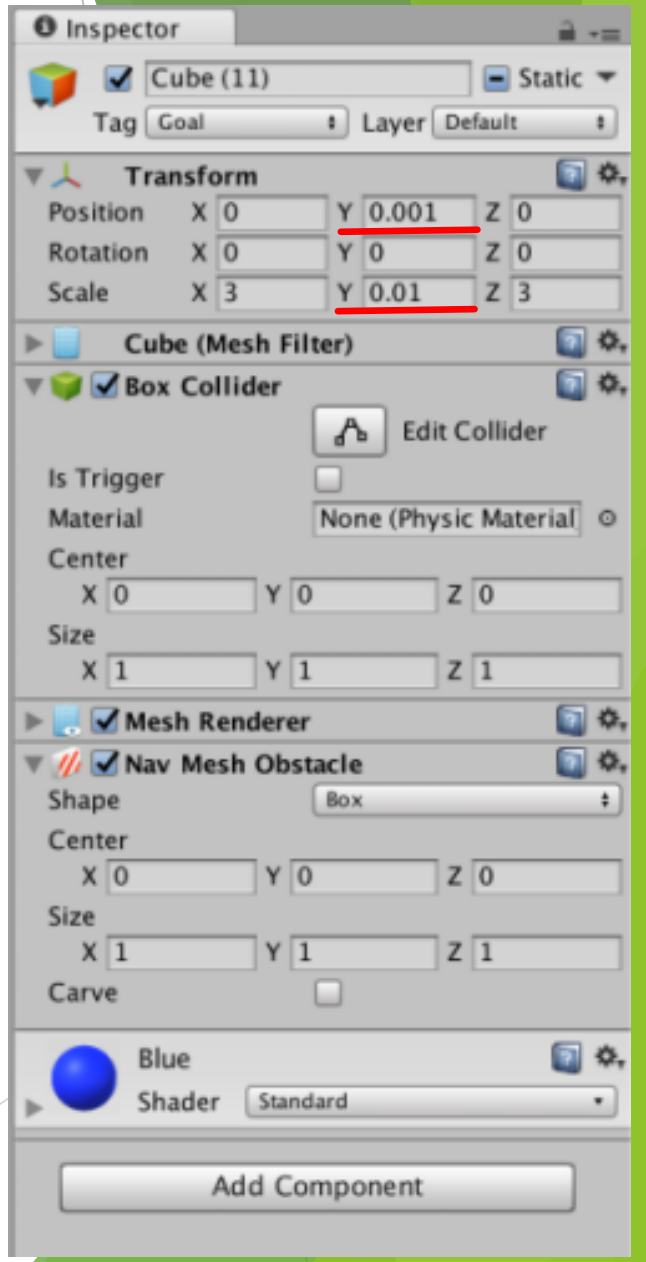
安全地帯の作成

　今の所ステージ上には壁しかなく、逃げ続けるのにも限界があるので、プレイヤーしか入れない安全地帯を作ることにしました。と言ってもこれはとても簡単で、今まで壁として使っていたものを薄くするだけで実現できます。

　TransformのY方向のScaleを小さい値にし、床からわずかに上に置きます。







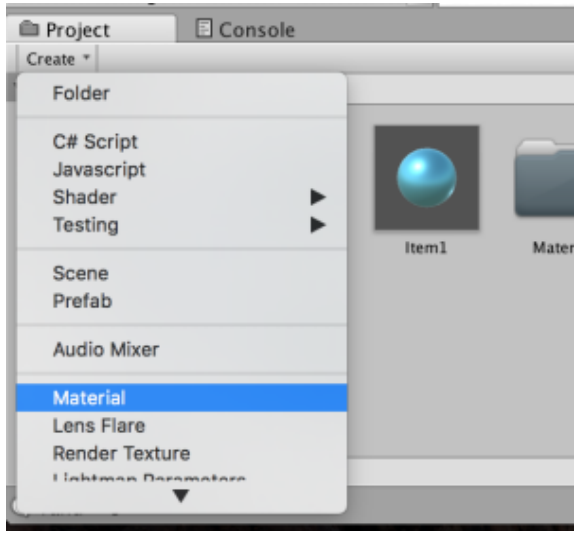
　プレイヤーはある程度小さい段差なら無視して移動できるので上に乗ることができます。一方、敵はNavMeshを基準に移動するので、NavMeshObtacleを持つこのオブジェクトは障害物として扱われ、侵入することができません。

　今回は右上の画像の青いエリアを安全地帯にし、中央の安全地帯をゴールにすることにしました。

マテリアル

　壁や安全地帯が床と同じ色ではわかりにくいので、色をつけることにします。Unityでは、マテリアルというものをオブジェクトに持たせることで、質感や色を表現できます。プロジェクトウィンドウのCreateからMaterialを作成してください。

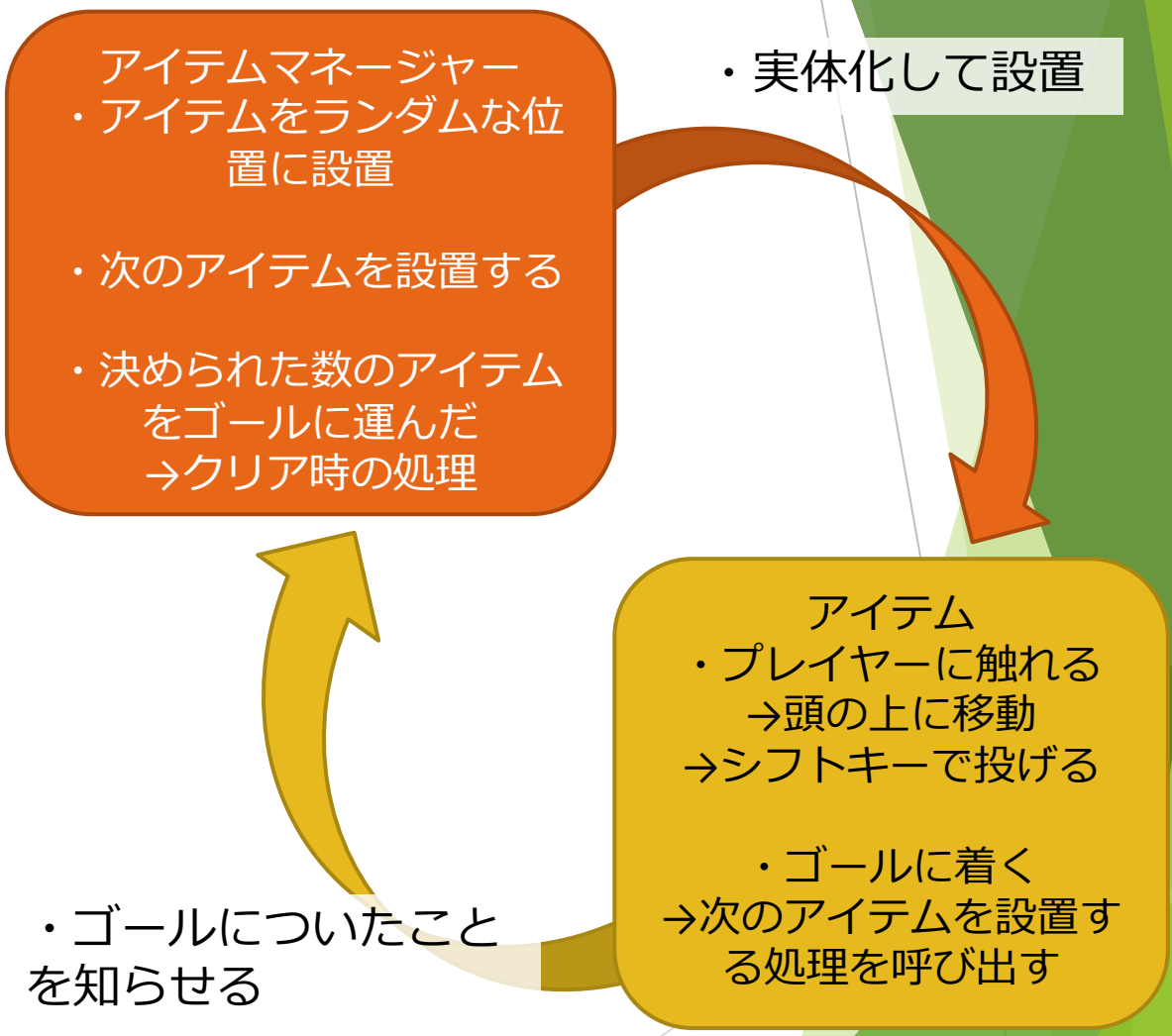
　作成したマテリアルを選択すると、インスペクターで色々な設定ができます。上の方の赤線で囲った部分で基本的な設定ができます。Albedoはオブジェクト全体にかかる色を選択できます。Metallicは金属らしさ、Smoothnessは表面の滑らかさを表すパラメータです。
　他のソフトで作成したモデルを使用する場合は、Albedoの左側の四角にテクスチャを指定すると適用できます。
　マテリアルをオブジェクトにドラッグ&ドロップして色を付けてましょう。



アイテムの設定(1/2)

次に、集めるアイテムの設定を行います。
・普段は床に置いてある
・触れると取得、頭上に表示
・shiftキーでアイテムを投げる
・一つ目をゴールに置くと次が出現
・一定数集めるとクリアー
という仕様を目指すことにします。

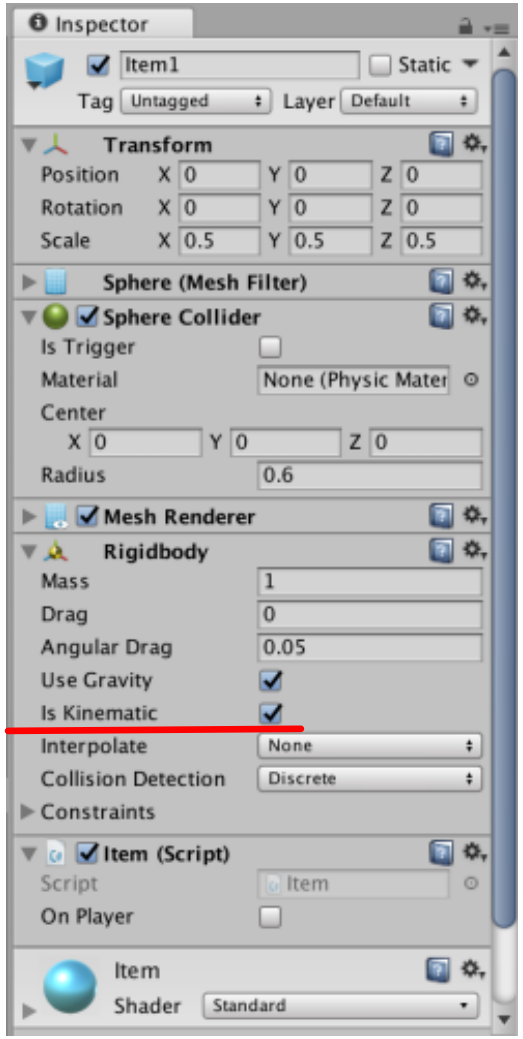
集めたアイテムの数や、アイテムの設置の処理を行うものが必要なため、空のオブジェクトを作ってアイテムマネージャーと名付け、そのような処理をさせることにします。
アイテムマネージャーとアイテムにはそれぞれ右図のような処理が必要になります。お互いが相手の処理を呼び出す必要があるため、複雑になってしまいます。

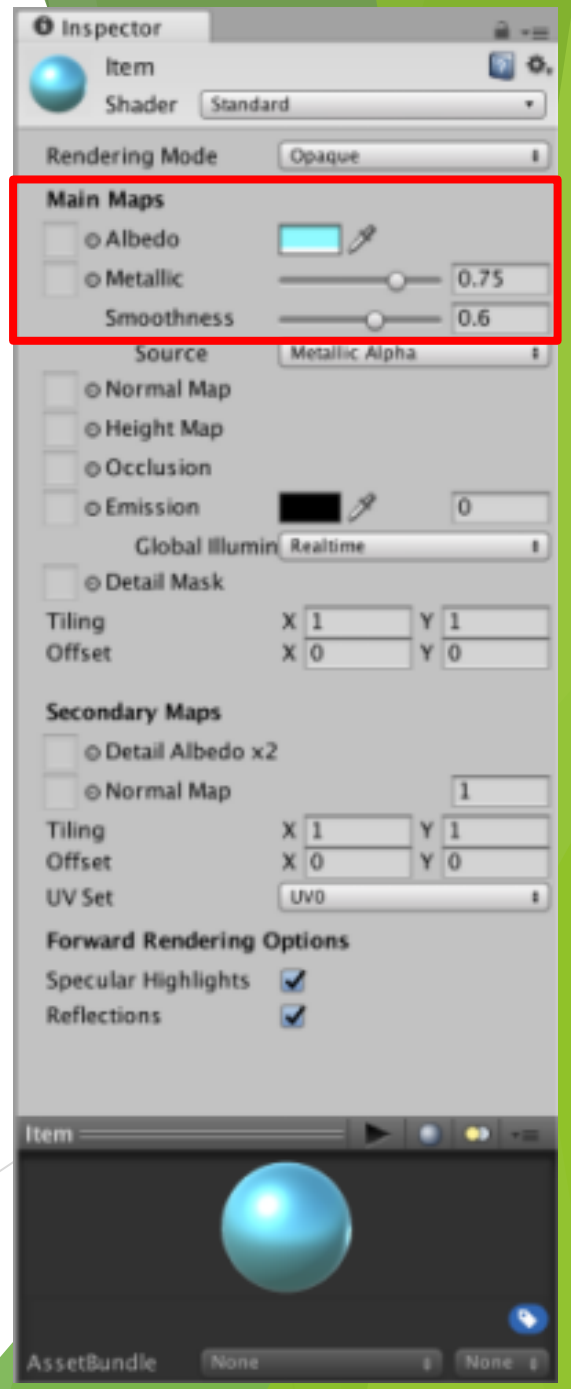


アイテムの設定(2/2)

　今回は右図のように、適当にアイテムを作成しました。実際にゲームを作る際は、Blenderなどのソフトで作成したモデルを使いましょう。
　プロジェクトウィンドウのCreateから新しいマテリアルを作成しました。右図の赤線で囲った部分を好きなように設定したら、プレハブにドラッグ&ドロップして付けましょう。

　アイテムを選択して、インスペクター下のAddComponent→Physicsから、Rigidbodyを追加します。これをつければ、重力などの物理法則に従って運動させることができます。
　しかし、プレイヤーに触れられる前に敵に蹴られてしまうと嫌なので、isKinematicをオンにして一時的に物理法則を無視するようにしておきます。



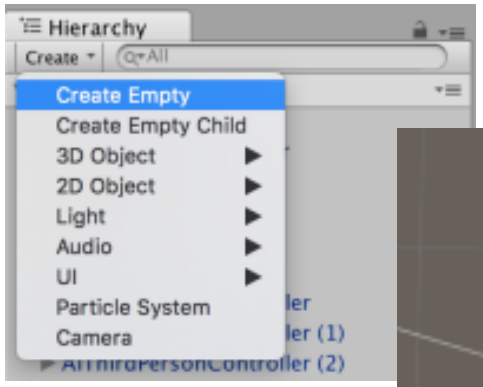


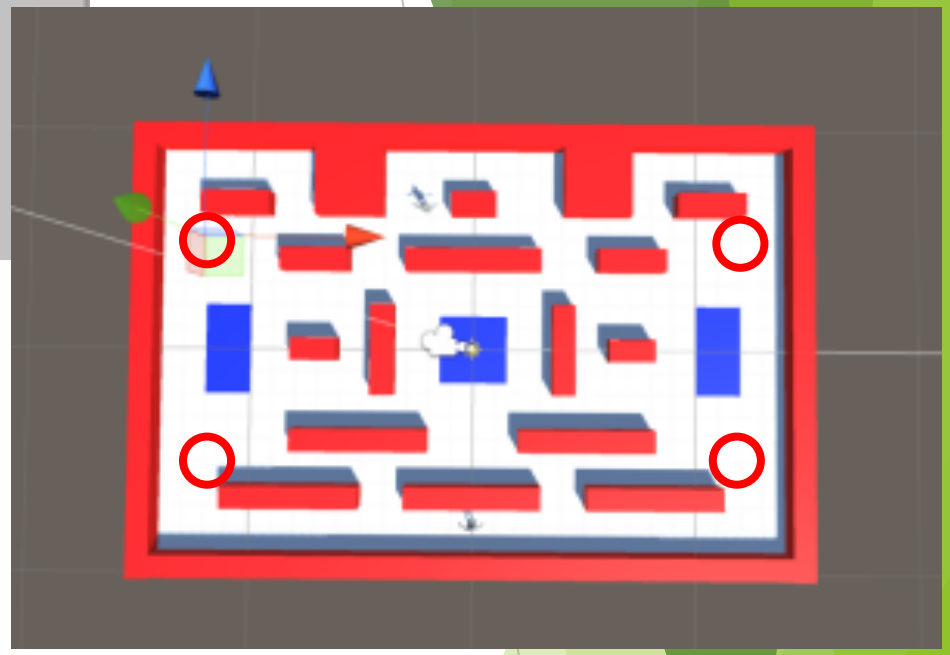
プレハブ

Unityでは、「プレハブ」というオブジェクトの雛形のようなものを作成し、これを元にオブジェクトを作成(インスタンス化という)することで、同じ特性を持ったオブジェクトをいくつも生成することができます。ステージ開始時にランダムな位置にアイテムを一つ設置し、それをゴールへ運ぶと次のアイテムが設置されるようにしたいため、プレハブ化することにします。

プレハブ化したいオブジェクト(今回はitem1)を、ヒエラルキーからプロジェクトウィンドウにドラッグ&ドロップすると、プレハブが作成できます。プレハブを作成したら、元のオブジェクトは削除してしまいましょう。ヒエラルキー上で右クリック→Deleteで削除します。

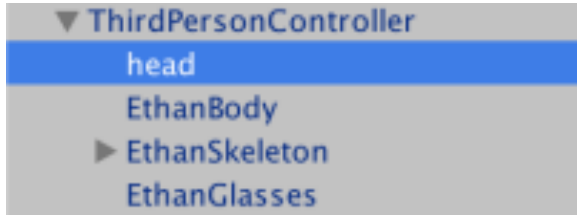
アイテム設置の準備



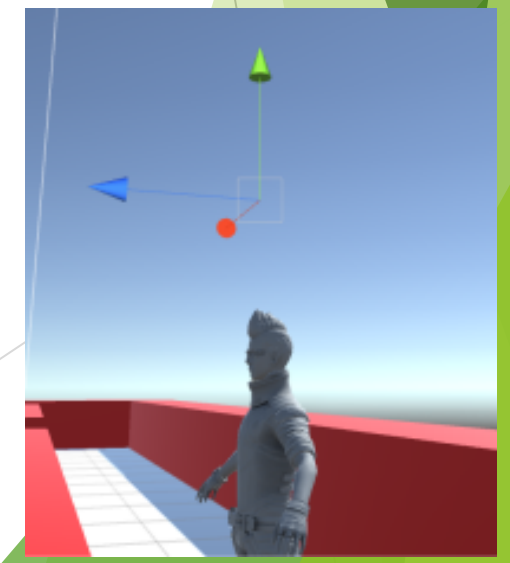


続いて、アイテムの初期配置の設定をします。今回は、プレイヤーのスタート地点をステージ中央にして、四隅のどこかにランダムにアイテムが出現するようにしたいと思います。

ヒエラルキーからCreateEmptyを選択して、空のオブジェクトを作成します。今回は右図の赤丸の位置のあたり4箇所に、y座標0.5くらいで配置しました。この空オブジェクトの位置を基準にアイテムを出現させます。



プレイヤーがアイテムを取得した際に頭上に表示するための基準点も用意しておきます。右図のように、空のオブジェクトにheadという名前をつけて、ThirdPersonController(プレイヤー)の子として置いておきます。今回はy座標を2にしました。

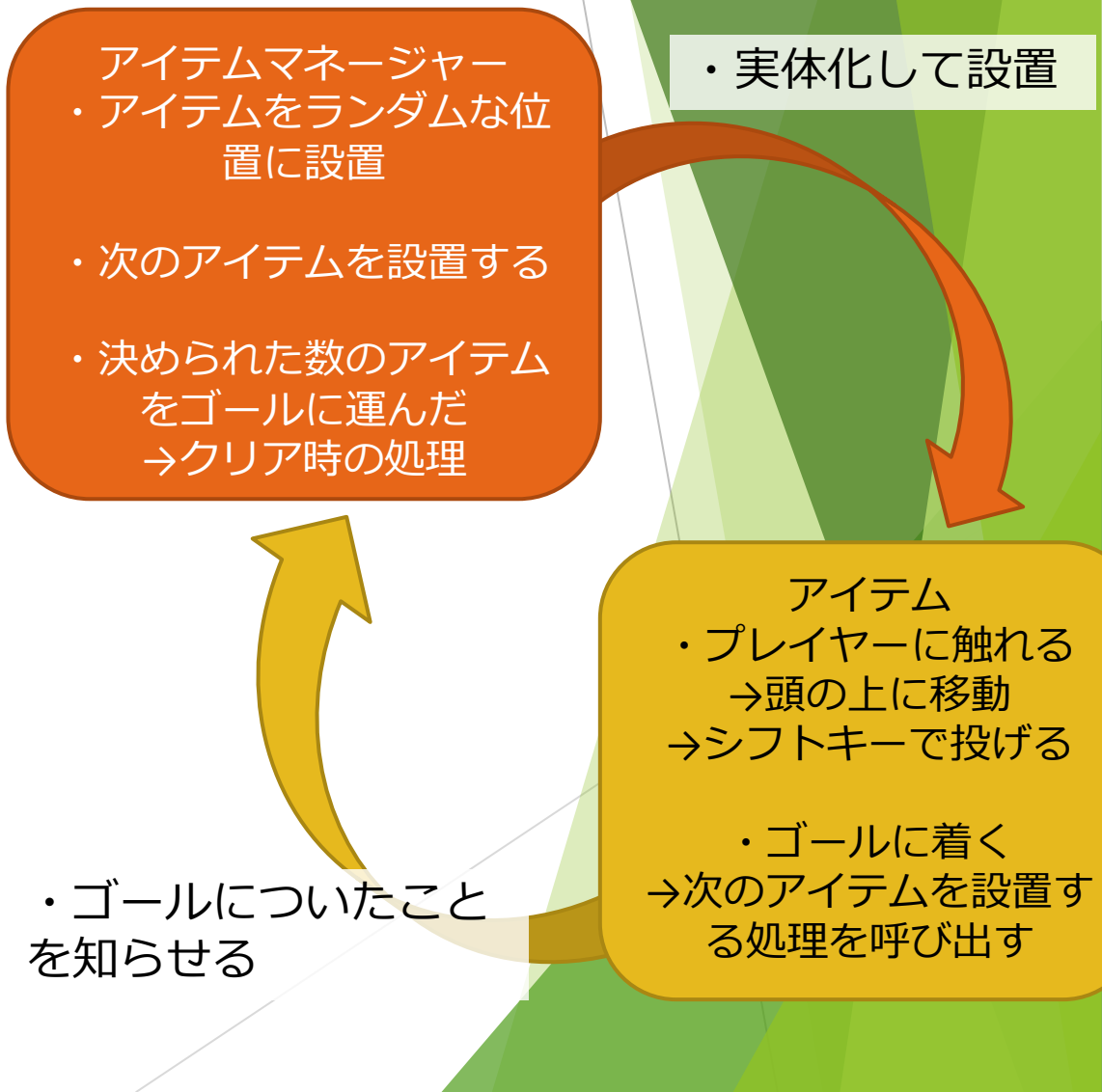


アイテムマネージャー (1/3)

Unityでは、Random.value や Random.Range() を用いて、ランダムな値を取得することができます。valueでは0.0~1.0のランダムな値、Rangeでは最小値と最大値を指定してランダムな値を取得できます。

プレハブは、Instantiateという関数でシーンに呼び出すことができます。単に
Instantiate(prefab);
としてもいいのですが、
GameObject obj = Instantiate(prefab) as GameObject;
というようにGameObjectとして変数に代入すれば、呼び出した後で初速を与えるなどの操作ができます。

Unityでは、あるオブジェクトから別のオブジェクトの変数や関数を利用するのが少し面倒なのですが、SendMessageという関数を用いると、それが簡単に行えます。どこから何を動かしているのか分かりにくい欠点があり、バグの元になりやすいのですが、簡単なので今回はこれを用います。これらを用いてスクリプトを書きました。



アイテムマネージャー (2/3)

アイテムの配置と、指定された個数のアイテムをゴールに運ぶとクリアー処理を呼び出す機能を誰かに持たせなければいけないため、アイテムマネージャーという空オブジェクトを用意することにします。

実装したスクリプトが右になります。
7行目以降の変数の宣言では、いくつかの変数にpublicをつけてインスペクターに表示させています。

markersはアイテムを出現させる位置の基準点、headはアイテムを取得した時にプレイヤーの頭上に表示するための基準点です。

numItemsにはクリアに必要なアイテムの個数を指定します。

9行目で宣言するのは変数の配列なのでただの変数とは違う書き方になります。注意しましょう。

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class ItemManager : MonoBehaviour {
6
7       public int numItems;
8       public GameObject itemPrefab;
9       public GameObject[] markers = new GameObject[4];
10      public GameObject head;
11      int num = 0;
12
13      // Use this for initialization
14      void Start () {
15          SetItem ();
16      }
17      // Update is called once per frame
18      void Update () {
19      }
20      public void SetItem(){
21          if (num < numItems) {
22              int rand = (int)(Random.value * 100) % 4;
23              Debug.Log ("rand = " + rand);
24              GameObject item = Instantiate (itemPrefab,markers [rand].transfor
    m.position,transform.rotation) as GameObject;
25              item.SendMessage ("SetManager",gameObject);
26              item.SendMessage ("SetHead",head);
27
28              num++;
29          } else {
30              //クリアー時の処理を呼び出し
31              Debug.Log("Clear!");
32          }
33      }
34  }
```

アイテムマネージャー (3/3)

基準点からランダムに一箇所選んでアイテムを出現させ、必要な変数を設定する。今までに作成したアイテムの個数が、あらかじめ指定された数に達していたら、クリアー時の処理を呼び出す。という処理をまとめてSetItem関数を作りました。

15行目でスタート時に自分で1回呼び出し、以降はアイテムがゴールに触れた時にアイテム側のスクリプトから呼び出すようにします。

22行目は、0~100のランダムな数値を4で割った余りを使って、アイテムの位置を決めています。

24行目でアイテムを実体化させ、25,26行目でSendMessageを使ってアイテム側の関数を呼び出しています。アイテムのスクリプトを書くところで解説します。

Instantiateは色々な書き方がありますが、今回は、Instantiate(プレハブ,位置,角度)の書き方にし、makersの位置にアイテムを出現させています。

28行目のnum++で作成したアイテム数を数えます。

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class ItemManager : MonoBehaviour {
6
7       public int numItems;
8       public GameObject itemPrefab;
9       public GameObject[] markers = new GameObject[4];
10      public GameObject head;
11      int num = 0;
12
13      // Use this for initialization
14      void Start () {
15          SetItem ();
16      }
17      // Update is called once per frame
18      void Update () {
19      }
20      public void SetItem(){
21          if (num < numItems) {
22              int rand = (int)(Random.value * 100) % 4;
23              Debug.Log ("rand = " + rand);
24              GameObject item = Instantiate (itemPrefab,markers [rand].transfor
    m.position,transform.rotation) as GameObject;
25              item.SendMessage ("SetManager",gameObject);
26              item.SendMessage ("SetHead",head);
27
28              num++;
29          } else {
30              //クリアー時の処理を呼び出し
31              Debug.Log("Clear!");
32          }
33      }
34  }
```

アイテムマネージャーの動作確認(1/3)

　アイテムマネージャーとアイテムのスクリプトは複雑なので、ここで一旦動作確認をしておこうと思います。

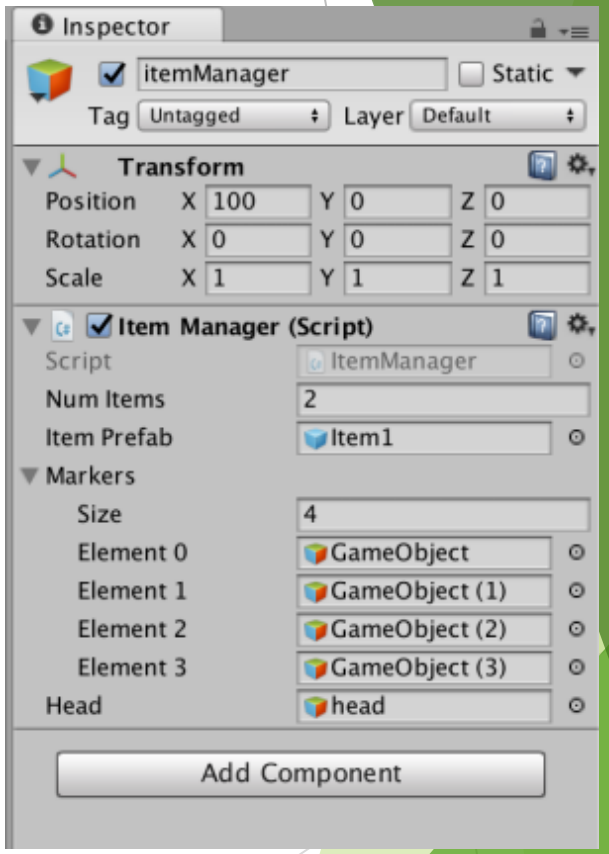
　アイテムマネージャーの25,26行目(アイテムにSendMessageする部分)の行のはじめに // を付けてコメントアウトしてください。

　これでエラーが出なくなるはずなので、ここまでの部分が正常に動くか試してみましょう。

```
public void SetItem(){
    if (num < numItems) {
        int rand = (int)(Random.value * 100) % 4;
        Debug.Log ("rand = " + rand);
        GameObject item = Instantiate (itemPrefab,markers [ran
        //item.SendMessage ("SetManager",gameObject);
        //item.SendMessage ("SetHead",head);
        num++;
    } else {
        //クリアー時の処理を呼び出し
        Debug.Log("Clear!");
    }
}
```

アイテムマネージャーの動作確認(2/3)

　ヒエラルキーのCreateからCreate Emptyを選択して、空のオブジェクトを作成し、アイテムマネージャー用のスクリプトを持たせましょう。インスペクターから、publicにした変数に持たせる値を指定していきます。
　numItemsはクリアに必要なアイテム数、itemPrefabはアイテムのプレハブです。
　markersという配列にはステージ四隅に配置した空オブジェクトを、headにはプレイヤーの頭上に配置した空オブジェクトを登録します。
　このオブジェクトのTransformの値には特に意味はありません。



アイテムマネージャーの動作確認(3/3)

再生ボタンを押して実行してみましょう。
23行目のDebug.Logによってランダムに生成した値がコンソールに表示されるので、その数値に対応する基準点にアイテムが設置されているか確認しましょう。また、何度実行しても同じ値が出てしまうバグがないかなども確認しましょう。
今はアイテムに触れても何も起こりません。

必ず今コメントアウトした25,26行目の // を外してから先に進んでください。

```
public void SetItem(){
    if (num < numItems) {
        int rand = (int)(Random.value * 100) % 4;
        Debug.Log ("rand = " + rand);
        GameObject item = Instantiate (itemPrefab,markers [ran
        //item.SendMessage ("SetManager",gameObject);
        //item.SendMessage ("SetHead",head);
        num++;
    } else {
        //クリアー時の処理を呼び出し
        Debug.Log("Clear!");
    }
}
```

アイテムのスクリプト(1/5)

次に、アイテムに持たせるスクリプトを書きます。
プレハブの変数には事前にオブジェクトを登録しておくことができないので、アイテムマネージャーやプレイヤーの頭上の基準点などの変数をゲーム中にスクリプトから取得してやる必要があります。

```
item.SendMessage ("SetManager",gameObject);
item.SendMessage ("SetHead",head);
```

アイテムマネージャー

SendMessageは、関数名を指定して他のオブジェクトの持つ関数を実行できる関数です。呼び出す関数に与える引数を一つだけ指定できるので、ここでアイテムマネージャーとheadを与えることにします。

アイテムに持たせるスクリプトの方では、引数に与えられたオブジェクトを変数に代入するだけの関数を用意しておき、これをアイテムマネージャーから呼び出します。

```
public void SetManager(GameObject obj){
    itemManager = obj;
}
public void SetHead(GameObject obj){
    head = obj;
}
```

アイテム

アイテムのスクリプト(2/5)

右が、アイテムに持たせるスクリプトです。長すぎてはみ出してしまったので、半分に分けて説明します。

・プレイヤーに触れると頭の上に乗る
・頭に乗っている時のみシフトキーで投げられる
・ゴールに触れると、アイテムマネージャーに次のアイテムを設置する処理を要求する

主にこの3つの処理を持たせます

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Item : MonoBehaviour {
6
7       bool onGoal = false;      //ゴールに入ったか否か
8       bool onPlayer = false;    //プレイヤーに持たれているか
9       Rigidbody rigid;
10      GameObject itemManager;
11      GameObject head;
12      Collider collItem;        //自分のコライダー(当たり判定)
13
14      // Use this for initialization
15      void Start () {
16          //自分の持つコンポーネントの中から指定した種類のものを取得
17          rigid = gameObject.GetComponent<Rigidbody> ();
18          collItem = gameObject.GetComponent<Collider> ();
19      }
20
21      // Update is called once per frame
22      void Update () {
23          //アイテムを投げる
24          if( Input.GetKeyDown(KeyCode.LeftShift) && onPlayer){
25              Debug.Log ("Shoot Item");
26              collItem.isTrigger = false;
27              onPlayer = false;
28              gameObject.transform.SetParent (null);
29              rigid.isKinematic = false;
30              rigid.AddForce (4.0f * transform.forward, ForceMode.VelocityChange);
31          }
32      }
33
        void OnCollisionEnter(Collision coll){

            if (coll.transform.tag == "Player" && !onGoal) {
```

アイテムのスクリプト(3/5)

スタート関数では自身のRigidbodyとColliderを取得し、以降自由に使えるように変数に入れています。GetComponentやFind系の関数は重いので、Updateなどの繰り返し行われる部分ではなくStartに書きましょう。

24行目～はアイテムを投げる処理です。このアイテムがPlayerに持たれている時にシフトキーを押すとこの部分が実行されます。

26~29行目に関しては、この後で説明する「プレイヤーに触れられた時」の処理で行う設定を解除しています。

30行目は、AddForce関数によってアイテムのRigidbodyに力を加え、頭の上から発射しています。1つ目の引数は加える力、2つ目は力の加え方を指定します。transform.forwardはオブジェクトの正面方向のベクトルで、これに力の大きさ(4.0)を適当にかけています。ForceModeは、重さを無視して速度を直接変更するVelocityChangeが扱いやすいです。

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Item : MonoBehaviour {
6
7       bool onGoal = false;      //ゴールに入ったか否か
8       bool onPlayer = false;    //プレイヤーに持たれているか
9       Rigidbody rigid;
10      GameObject itemManager;
11      GameObject head;
12      Collider collItem;        //自分のコライダー(当たり判定)
13
14      // Use this for initialization
15      void Start () {
16          //自分の持つコンポーネントの中から指定した種類のものを取得
17          rigid = gameObject.GetComponent<Rigidbody> ();
18          collItem = gameObject.GetComponent<Collider> ();
19      }
20
21      // Update is called once per frame
22      void Update () {
23          //アイテムを投げる
24          if( Input.GetKeyDown(KeyCode.LeftShift) && onPlayer){
25              Debug.Log ("Shoot Item");
26              collItem.isTrigger = false;
27              onPlayer = false;
28              gameObject.transform.SetParent (null);
29              rigid.isKinematic = false;
30              rigid.AddForce (4.0f * transform.forward, ForceMode.VelocityChange);
31          }
32      }
33
        void OnCollisionEnter(Collision coll){

            if (coll.transform.tag == "Player" && !onGoal) {
```

アイテムのスクリプト(4/5)

コライダー同士が衝突すると、お互いのOnCollisionEnter関数が呼び出されます。衝突の相手を引数で受け取れるので、相手の種類によって処理が場合分けできます。

相手を見分ける方法の1つとして、タグを使う方法があります。プレイヤーには Player というタグを付けることにします(あとで設定します)。37~46行目は、相手のタグがPlayerで、このアイテムがまだゴールしていないなら実行されます。

39行目で、コライダーのisTriggerを有効にしています。コライダーには、コライダーとトリガーの2種類が存在し、isTriggerによってこれを切り替えられます。トリガーは他のオブジェクトと衝突しても反射せずにすり抜けます。また片方がトリガーだとOnCollisionEnterが反応しなくなり、代わりにOnTriggerEnter関数が呼び出されるようになります。

プレイヤーに使っているThirdPersonControllerは、頭上にコライダーがあるとしゃがんでしまうので、一時的にトリガーに変更します。

```
34                rigid.AddForce (4.0f * transform.forward, ForceMode.VelocityChange);
35            }
36        }
37
38        void OnCollisionEnter(Collision coll){
39
40            if (coll.transform.tag == "Player" && !onGoal) {  //プレイヤーに触れると
41                Debug.Log ("Get Item");
42                onPlayer = true;    //プレイヤーに運ばれていることを表す変数
43                collItem.isTrigger = true;
44                rigid.isKinematic = true;           //物理法則を無視する
45                //親の設定
46                gameObject.transform.SetParent (head.transform);
47                //位置と向きを親に合わせる
48                gameObject.transform.localPosition = Vector3.zero;
49                gameObject.transform.localRotation = Quaternion.Euler(0,0,0);
50            }
51
52            if (coll.transform.tag == "Goal") {    //ゴールに触れると
53                rigid.isKinematic = true;
54                onGoal = true;      //ゴールしたことを表す変数
55                itemManager.SendMessage ("SetItem");  //次のアイテムを設置する
56            }
57
58        }
59
60        //インスタンス化したItemに、itemManagerとheadを紐付けるための関数
61        public void SetManager(GameObject obj){
62            itemManager = obj;
63        }
64        public void SetHead(GameObject obj){
65            head = obj;
66        }
67
68    }
```

アイテムのスクリプト(5/5)

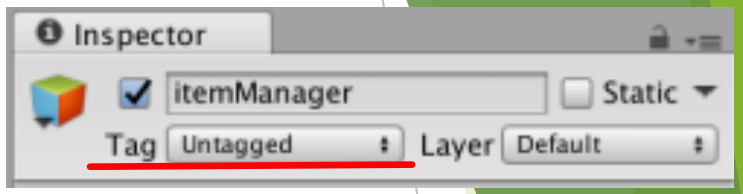
42行目でアイテムをプレイヤーの移動に追従させるために、headを親に設定します。

オブジェクトの位置や回転は、ゲーム内の世界の基準点から見たグローバル座標と、親の位置を基準に見たローカル座標の2種類で扱えます。ローカル座標で見た位置を(0,0,0)にすれば、親に位置を揃えられます。
同様に回転も親に合わせておくと後で便利なのですが、回転はクォータニオンという(x,y,z,w)の4つの数値によって表現されており、これを直接いじっても思い通りに動かないことが多いです。そこで、Quaternion.Eulerによって、オイラー角を使って変更します(45行目)。

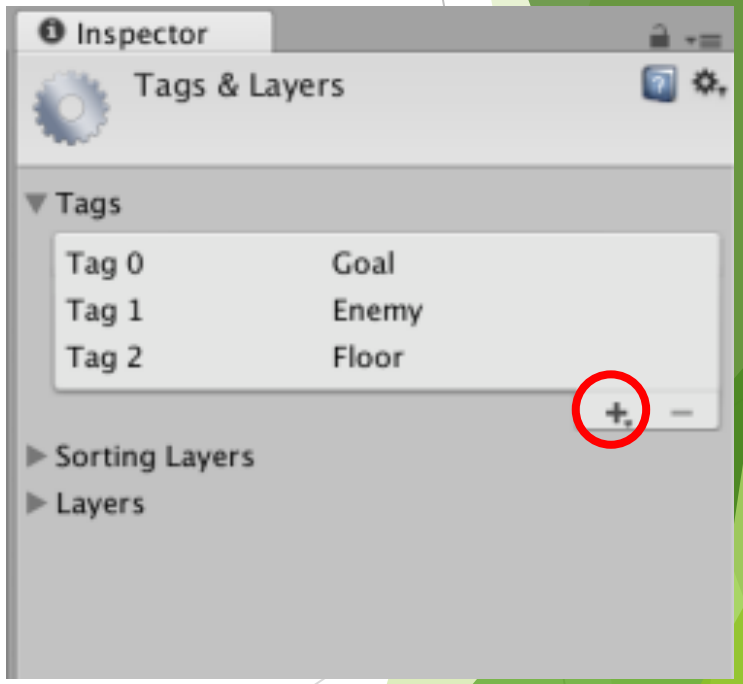
48～51行目のゴールに触れた時の処理では、isKinematicを有効にしてプレイヤーに蹴られても移動しないようにし、onGoalをtrueにしてゴール済みであることがわかるようにしています。36行目ではこの変数を調べてゴール済みのものは拾わないようになっています。また、SendMessageによってアイテムマネージャーに報告し、次のアイテムを設置させます。

```
34                rigid.AddForce (4.0f * transform.forward, ForceMode.VelocityChange);
35            }
36        }
37
38        void OnCollisionEnter(Collision coll){
39
40            if (coll.transform.tag == "Player" && !onGoal) {
41                Debug.Log ("Get Item");
42                onPlayer = true;    //プレイヤーに運ばれていることを表す変数
43                collItem.isTrigger = true;
44                rigid.isKinematic = true;           //物理法則を無視する
45
46                gameObject.transform.SetParent (head.transform);   //親の設定
47                //位置と向きを親に合わせる
48                gameObject.transform.localPosition = Vector3.zero;
49                gameObject.transform.localRotation = Quaternion.Euler(0,0,0);
50            }
51
52            if (coll.transform.tag == "Goal") {    //ゴールに触れると
53                rigid.isKinematic = true;
54                onGoal = true;      //ゴールしたことを表す変数
55                itemManager.SendMessage ("SetItem");  //次のアイテムを設置する
56            }
57
58        }
59
60        //インスタンス化したItemに、itemManagerとheadを紐付けるための関数
61        public void SetManager(GameObject obj){
62            itemManager = obj;
63        }
64        public void SetHead(GameObject obj){
65            head = obj;
66        }
67
68    }
```

アイテムの動作確認(1/2)



　アイテム用のスクリプトを保存してプレハブに持たせたら、次はOnCollisionEnter内で使用するタグの設定をします。
　インスペクター上部にタグを指定する項目があります。Untaggedとなっているボタンを押し、一番下のAddTagを選択してください。
　新しいタグを作成する画面が開くので、右下の+ボタンから、Goalタグを作成します。(Playerタグははじめからあると思います。なければ同様に作成してください)



　タグを設定したいオブジェクトを選択し、インスペクター上部のUntaggedとなっているボタンを押し、作成したタグを設定しましょう。
　プレイヤー用のオブジェクトにPlayerタグ、ゴールに設定したいオブジェクト(今回は中央の安全地帯)にGoalタグを設定します。プレイヤーは階層が別れていますが、コライダーを持つオブジェクトにタグを設定しないと反応しません。注意してください。

アイテムの動作確認(2/2)

実行して正常に動作するか確認してみましょう。

スタートするとアイテムが出現し、触れるとプレイヤーの頭上へ、シフトキーを押すと前方に発射され、ゴールの床に置くと2つ目のアイテムが出現する。という流れで処理が進めば成功です。

上手くいかない場合ですが、コライダーやSendMessageあたりでミスが起こりやすそうです。インスペクターでpublicな変数にきちんと値が設定されているかなども確認してみてください。

ビルド(1/2)

なんとなく遊べるものになってきたので、ビルドしてPC向けのゲームとして出力してみようと思います。ですがその前にスクリプトを1つ追加します。

このままだとゲームを起動した後、強制終了以外でゲームを終了できないため、ゲーム終了の処理を追加します。

プロジェクトウィンドウからスクリプトファイルを作成し、右のようなプログラムを書いてください。

Application.Quit()はゲームを終了するための処理です。これをエスケープキーが押された時に呼び出します。

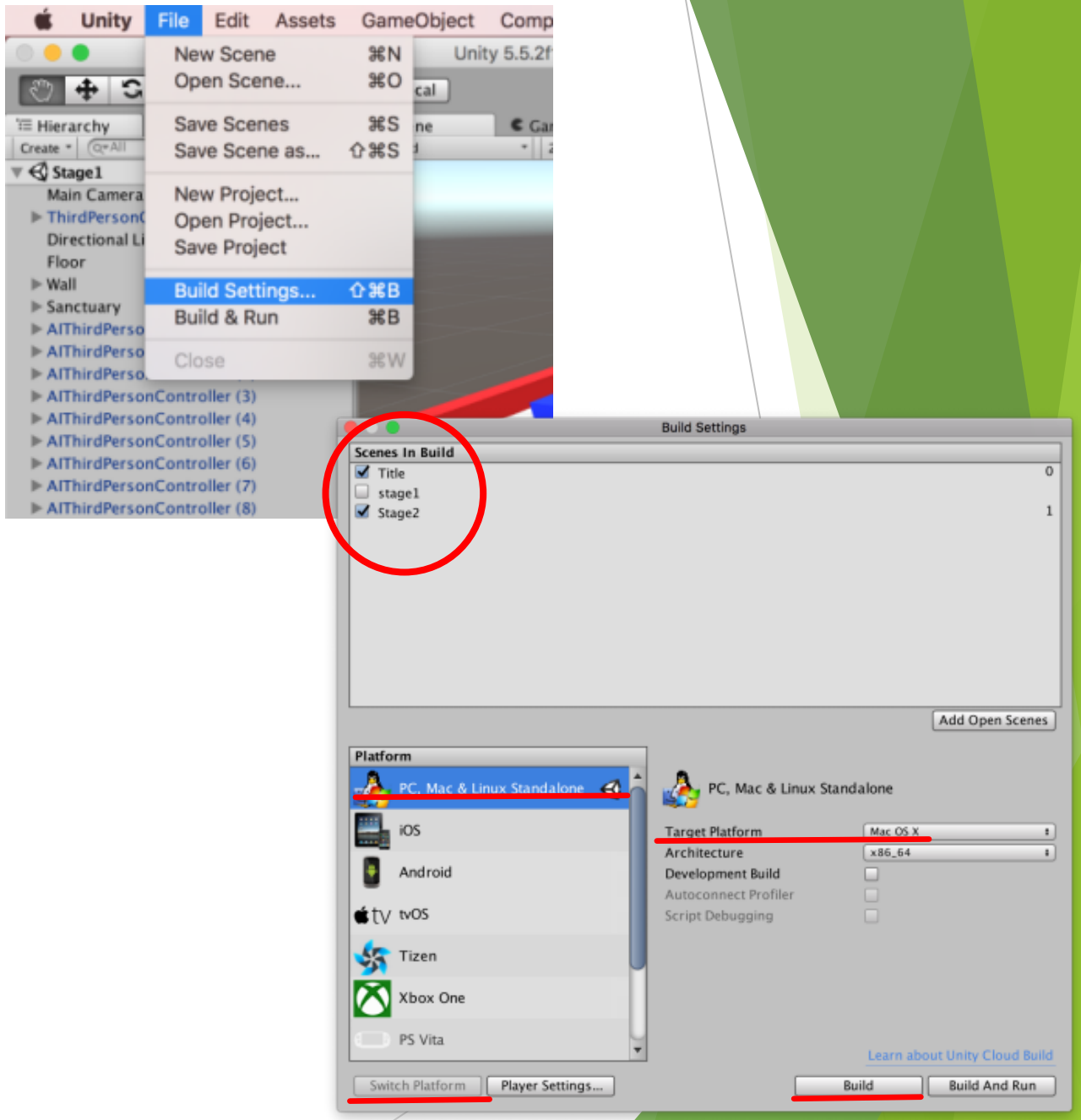
このスクリプトを各シーンのMainCameraに持たせてください。

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class QuitGame : MonoBehaviour {
6
7       // Use this for initialization
8       void Start () {
9
10      }
11
12      // Update is called once per frame
13      void Update () {
14          if (Input.GetKeyDown (KeyCode.Escape)) {
15              Application.Quit ();
16          }
17      }
18  }
```

ビルド(2/2)

　Fileメニューから Build Settingsを開きます。
　プロジェクトウィンドウのシーンファイルをドラッグ&ドロップしてScenes In Build の項目にビルドしたいシーンを登録します。起動して最初に表示させたいシーン(タイトル画面など)は一番上に登録し、ビルドから除外したいシーンはチェックを外します。

　画面左下のPlatformがPCになっていなかったら、PCを選択してSwitchPlatformボタンを押します。
　画面右下のTargetPlatformでどの種類のPCで実行できるように書き出すか設定できます。
　Buildボタンを押してビルドしましょう。



　今回はスクリプトの書き方を中心に、シーン切り替え、テキストの表示、リジッドボディによる運動など、よく使うものをいろいろ使ってみました。
　この資料だけでは説明不足で分からないことがたくさんあると思うので、公式のマニュアルや検索サイトで調べて遊んでみると、理解が深まると思います。

　また、公式サイトにもチュートリアルがあるので、こういったものにも挑戦してみるといいと思います。
(https://unity3d.com/jp/learn/tutorials)