

# Table of Contents

はじめに	1.1
このチュートリアルの目的	1.1.1
ここでは学ばないこと	1.1.2
Unityについて	1.1.3
開発言語	1.1.4
エディタ	1.1.5
チュートリアル1 「Planet」	2.1
このチュートリアルについて	2.1.1
オブジェクトの基本操作	2.1.2
基本的なマテリアル	2.1.3
基本的なテクスチャ	2.1.4
基本的なカメラと光の演出	2.1.5
物体の回転	2.1.6
見た目のクオリティ向上	2.1.7
チュートリアル2 「Penguin」	3.1
このチュートリアルについて	3.1.1
モデルデザイン用シーン作成	3.1.2
メインシーン作成	3.1.3
ペンギンのスクリプト追加	3.1.4
光を動かすスクリプト	3.1.5
チュートリアル3 「TrenchFighter」	4.1
このチュートリアルについて	4.1.1
メインシーンの作成	4.1.2
モデルシーンの作成	4.1.3
モデル生成処理の作成	4.1.4
サウンドの追加	4.1.5
簡易的なあたり判定	4.1.6
入力の処理	4.1.7
仕上げの絵作り	4.1.8

# 非プログラマー向けUnity入門

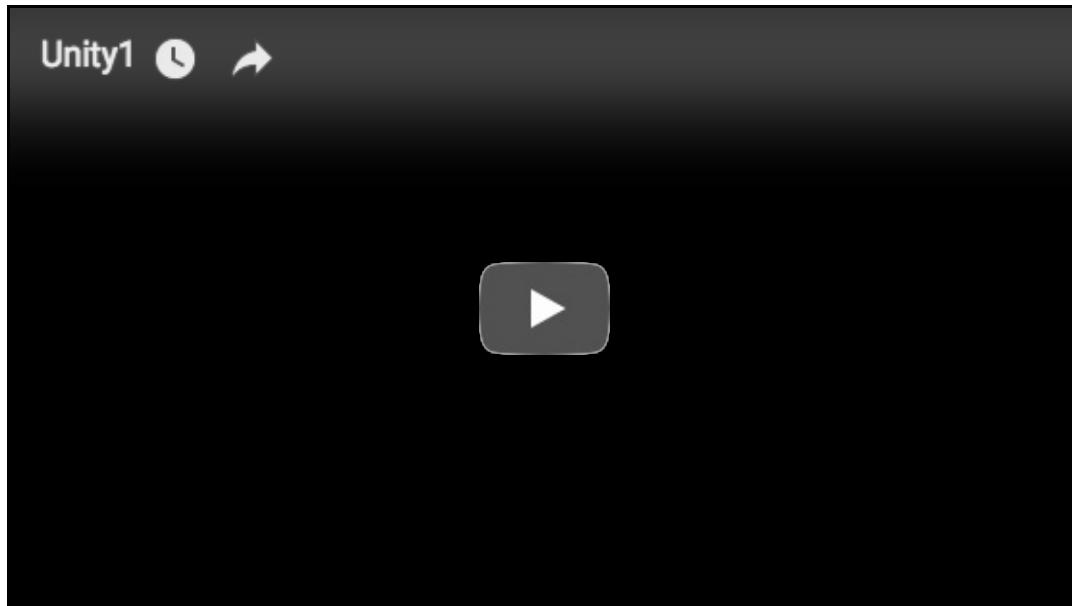
## このチュートリアルの目的

Unityの全機能を網羅的に学ぶには多くの時間がかかります。一般的なUnityのチュートリアルは、簡単なゲームを1本作りきるスタイルが多いですが、デザイナーやアーティストの人が最初に触れるテキストとしては、覚える項目とコーディングする量の多さ、複雑さに圧倒されてしまうように思います。そのため、まずは簡単な物体の動かし方と、見た目の演出方法を身につけて、動く絵（シーン）を比較的自由に作れるようになることを目指し、成果がわかりやすく、挫折しにくいチュートリアルになるようにしました。また、Unity内だけで完結するように、別のツールで作成したモデルを使用せず、Unity内で生成できるオブジェクトやAsset Storeの無料素材を利用するようにしています。

Unityのバージョンは5.6、2017.2、2017.3、プラットフォームはmac OS、Windowsで確認しています。

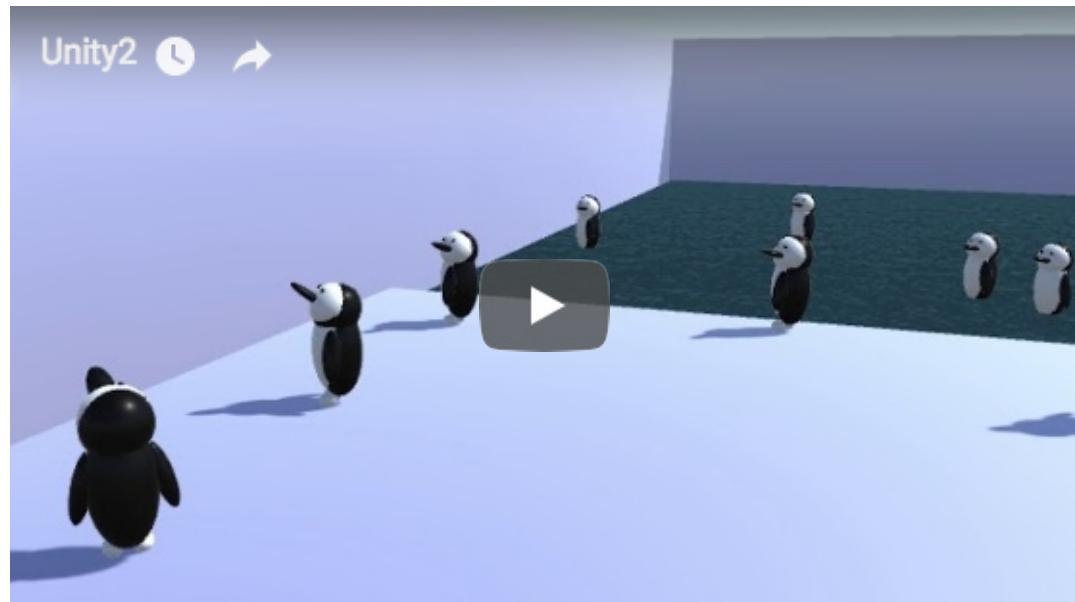
### チュートリアル1 Planet

- 物体の移動と回転
- 基本的なマテリアルとテクスチャ
- 基本的なカメラと光の演出
- アセットストアとの付き合い方
- ポストプロセッシングで絵作りその1



### チュートリアル2 Penguin

- Unityだけでできる、簡易的なモデル作成
- 簡易的な水の表現
- ターゲットの方向を向く処理
- 三角関数を使ったさまざまな物体の移動



### チュートリアル3 TrenchFighter

- プロシージャルなステージ生成
- ポストプロセッシングで絵作りその2
- 簡単なサウンドの使い方
- 物理演算を使わないあたり判定
- 入力の処理



### ここでは学ばないこと

このチュートリアルでは、以下の項目には触れていません。

- 3D数学の詳細（ベクトル、クォータニオン、行列）
- 物理演算
- 本格的なモデル作成
- モデルアニメーション

- ゲーム中のシーン切り替え演出
- 2DのUI(Canvas)
- パーティクルによる演出
- 各種レンダラー詳細
- マテリアルとシェーダーの詳細
- レベルデザイン、面白さの演出
- ゲームAI
- チーム開発

## Unityについて

ここでは、必要最小限の環境設定について説明します。Unity自体の説明や画面の操作方法は以下の公式マニュアルを参照してください。

<https://docs.unity3d.com/ja/current/Manual/UsingTheEditor.html>

## 開発言語

Unityでは数種類の開発言語を選択できますが、ほとんどのプログラマーはC#で開発をおこなっています。

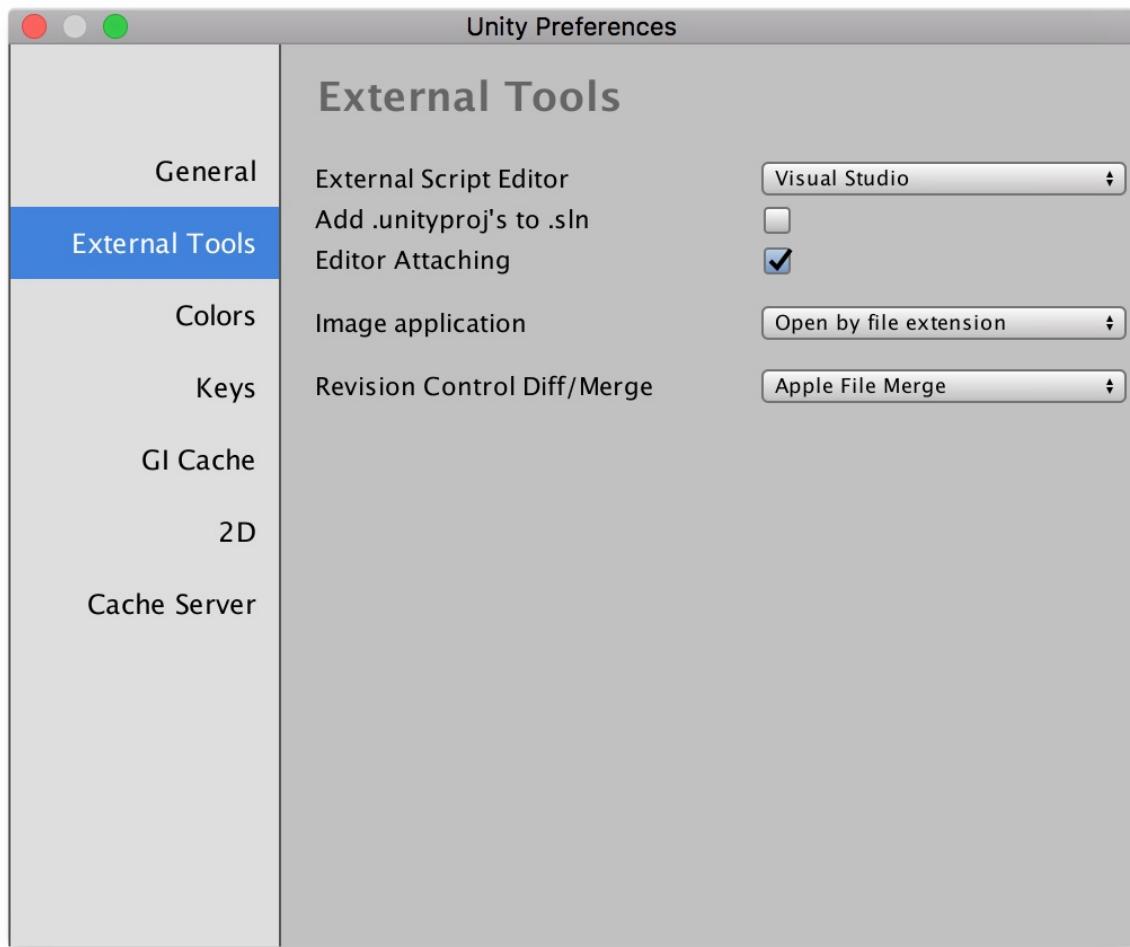
Unityのように多くのソースファイルを行き来する場合、型がしっかりしたC#でおこなうと理解しやすい、C#はC++のようなメモリ管理を自前でしないですむので初心者にもやさしい、UnityのJavaScriptはWeb開発で進化したエコシステムの恩恵をほとんど受けられずメリットが少ない、というのが主な理由です。また、結果的に公開されるサンプルコードやライブラリの多くがC#で書かれているため流用しやすい、という利点もあります。

## エディタ

テキストエディタはVisual Studioがお薦めです。ゲーム開発では、多くのAPIやオブジェクトを利用るので、インテリセンスの入力支援機能が効率的な開発には必須になります。たとえ普段他のエディタを使っていたとしても、Unityの開発のときだけはVisual Studioを使う、ということも検討してみてください。有償も含めるとIntelliJなども良いという話は聞きます。いずれにしても単なるエディタではなく、IDEとしてAPI名の入力補完やクラス構造の探索機能が充実しているものを使用するようにしてください。

注: Visual Studio for Macは、以前は「\_」（アンダースコア）が入力できなかったり、最新版でも日本語入力が若干見づらいという欠点があるようですが、それを差し引いてもVisual Studio for Macを使うようにしてください。

UnityからVisual Studio for Macを起動できるようにするには、Unityメニュー→Preference→External Tools→External Script Editorで「Visual Studio」を選択してください。

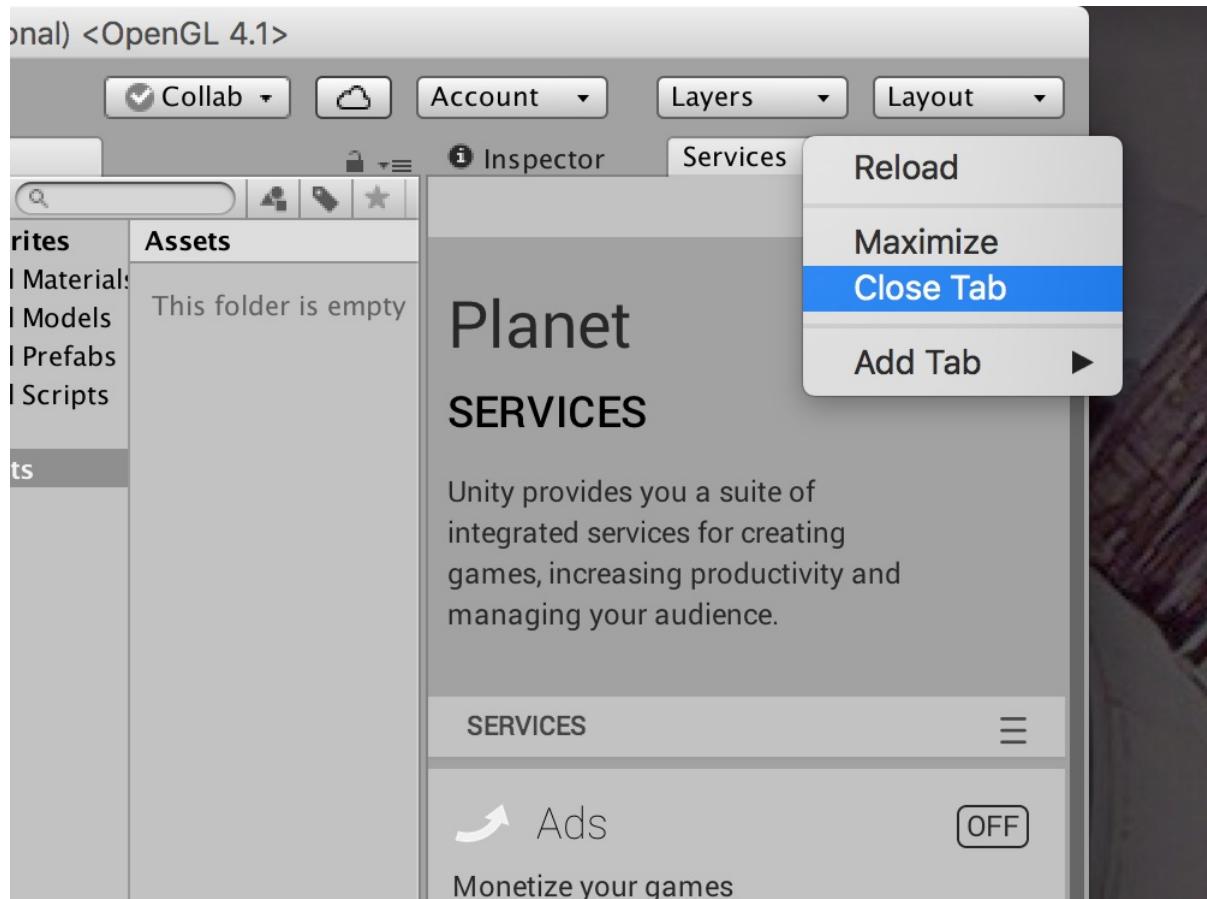


# チュートリアル1 「Planet」

## このチュートリアルについて

最初のチュートリアルです。「Planet」の名前で新規プロジェクトを作ってください。

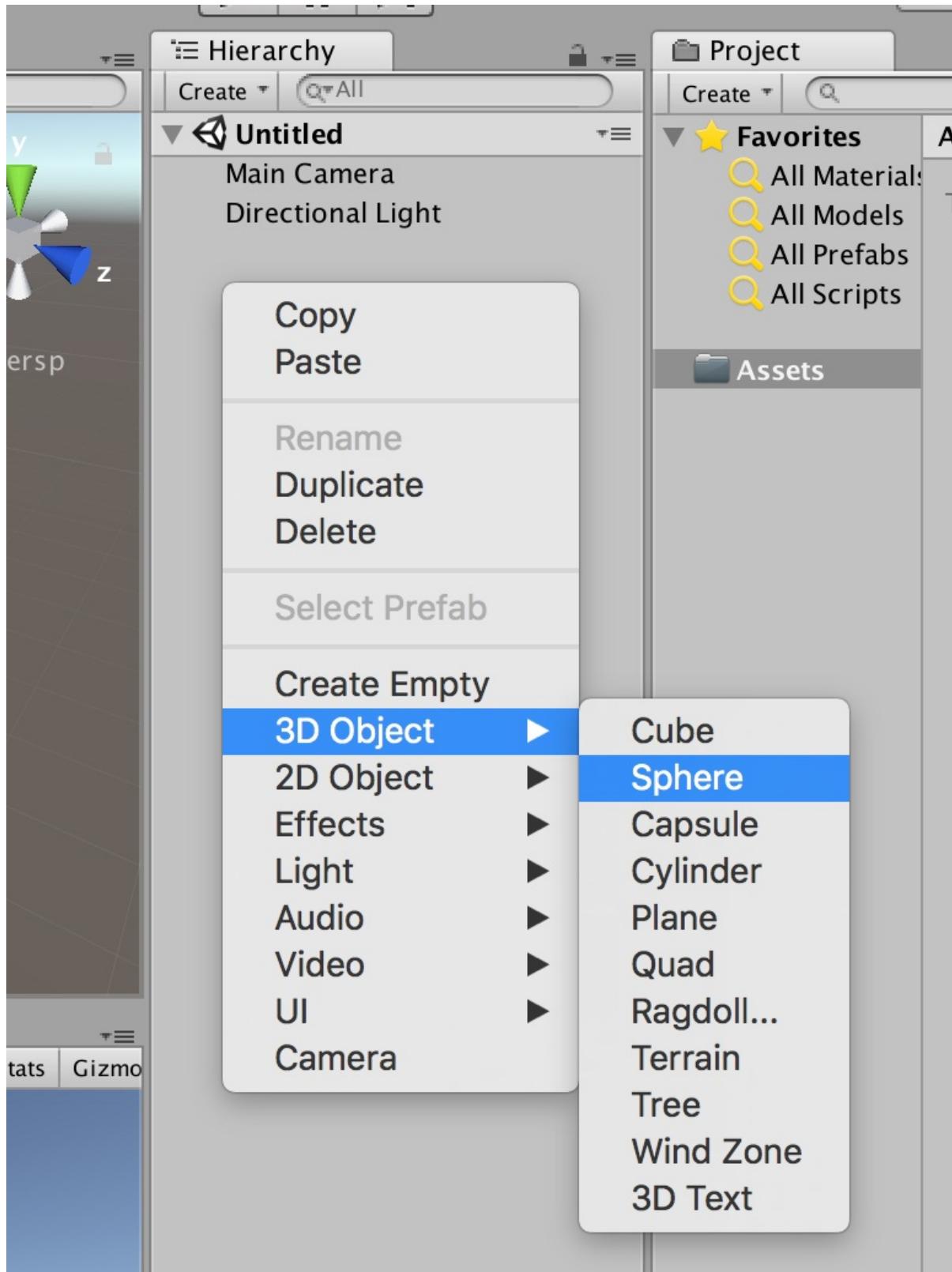
Unity5.2で導入されたServicesタブは今は不要なのでタブを右クリックして閉じましょう。Servicesは、広告の設定やクラウドビルド、マルチプレイヤーの設定をするときなどに使います。



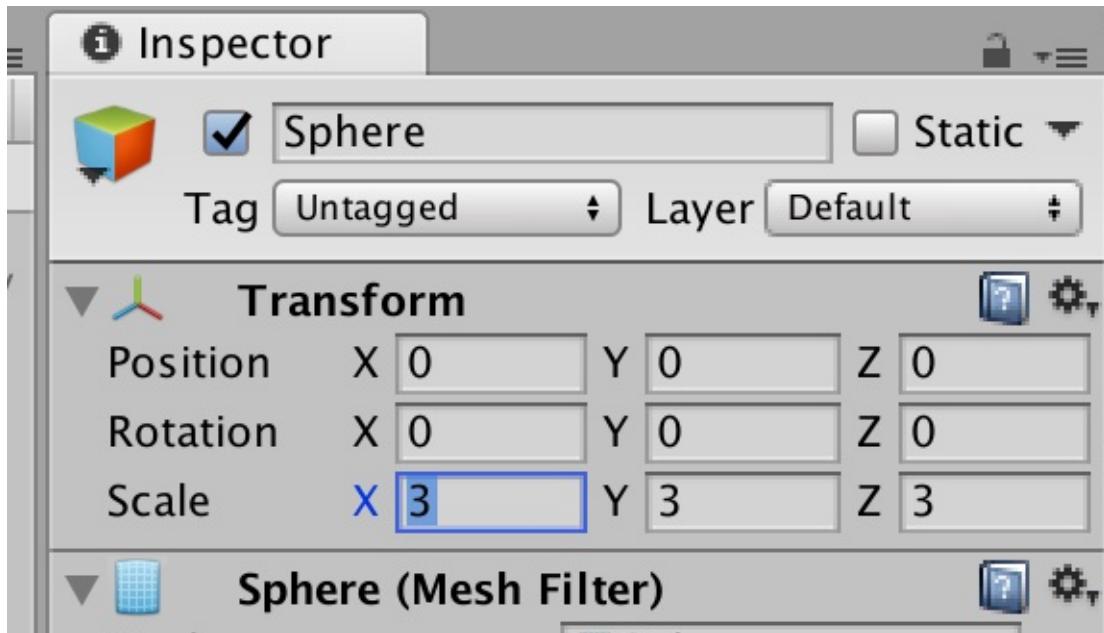
## オブジェクトの基本操作

Planetでは、地球、月、太陽の表現を学んでいきます。

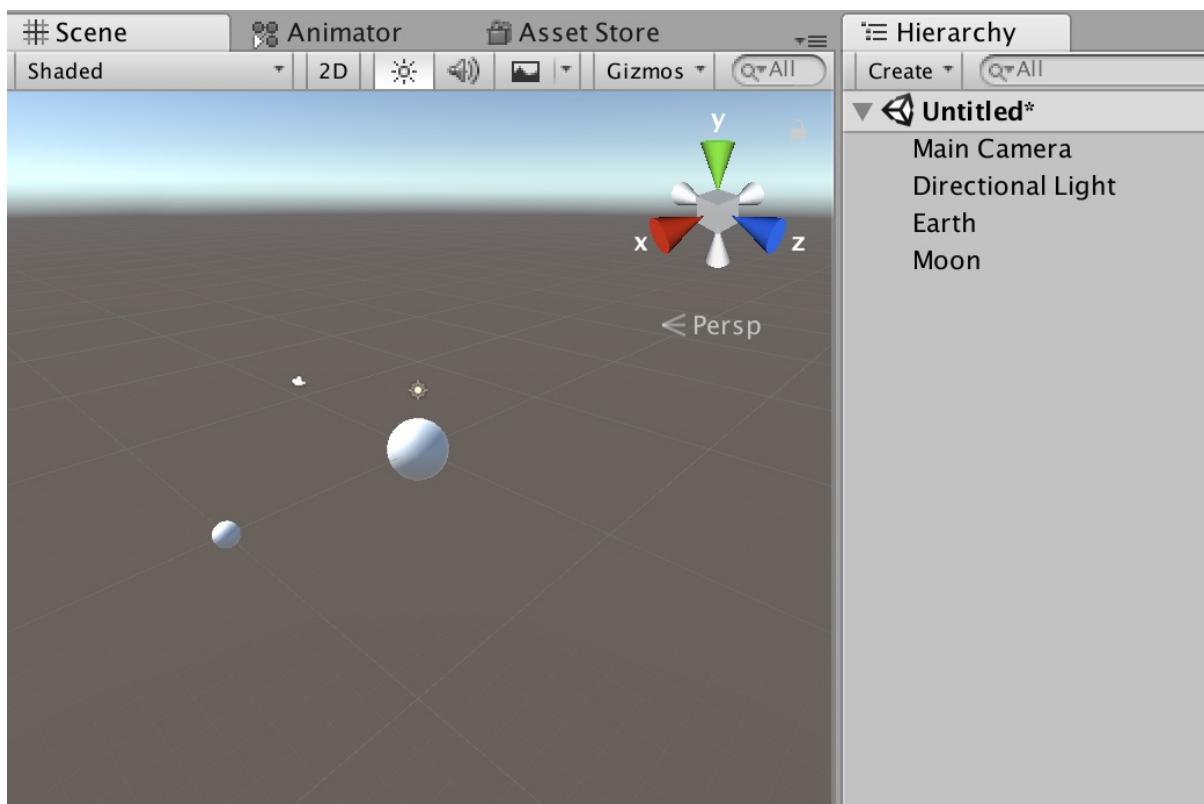
まず、シーンに球体(Sphere)を生成します。Hierarchyで何も選択されていないことを確認し（重要）、Hierarchyウィンドウの空き領域を右クリック（トラックパッドでは二本指クリック）してコンテキストメニューから3D Object→Sphereを選びます。（GameObjectメニュー やツールバーのCreateからも生成できますがコンテキストメニューからが早く楽です）



生成したSphereをHierarchyで選択した状態で、InspectorのPositionを0,0,0、Scaleを3,3,3に変更します。これで座標(0,0,0)に直径3の球体が作成できました。



同じようにもうひとつ球体を生成します。今度はPositionを10,0,0、Scaleを1,1,1にします。これで地球と月の原型ができました。わかりやすいようにそれぞれEarth、Moonと名前を変更しておきましょう。名前の変更は、Hierarchyでオブジェクトが選択されている状態でもう1回クリックするか、Inspectorの一番上の文字列を変更することができます。

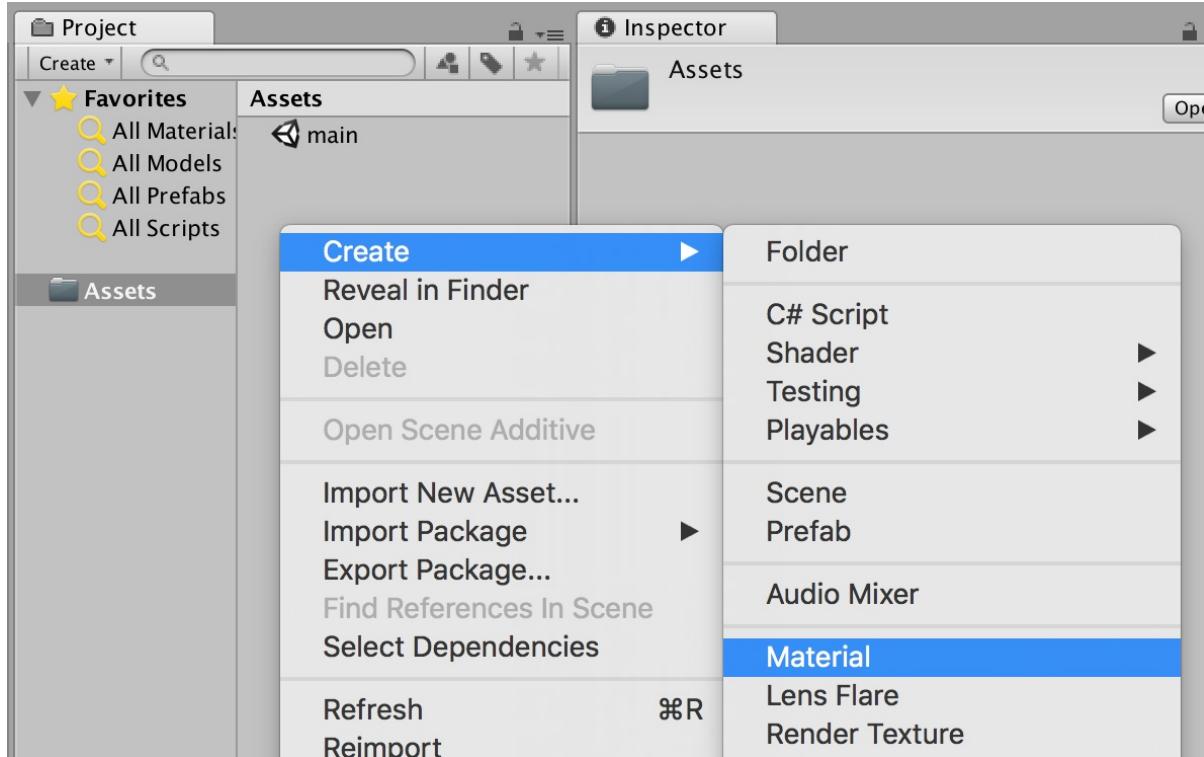


ここまで終わったら、Ctrl+Sで一旦シーンをセーブしておきましょう。名前はなんでも良いですが、おすすめは「main」です。Unityはセーブの概念が少しわかりにくいので注意が必要です。Hierarchy上の変更は原則としてセーブしないと失われるのでこまめにセーブするようにしましょう。

## 基本的なマテリアル

このままではまだ色が月にも見えないのでマテリアルを新規作成して設定します。

ProjectウィンドウでAssetsを選択し、次にAssetsペインで右クリックメニュー→Create→Materialを選択します。マテリアル名は「matMoon」としておきます。Unityは拡張子が表示されないので、接頭子として種別をつけておくと便利です。



matMoonを選択してInspectorでマテリアルの設定をしていきます。たくさんパラメータがあって戸惑いますが、最初はAlbedoで色またはテクスチャを指定、MetallicとSmoothnessで質感を指定、必要であればEmissionで光させる、くらい覚えておけば十分です。

matMoonの場合

Albedo : R:233, G:230, B:160, A:255

Metallic : 0.36

Smoothness : 0

にします。

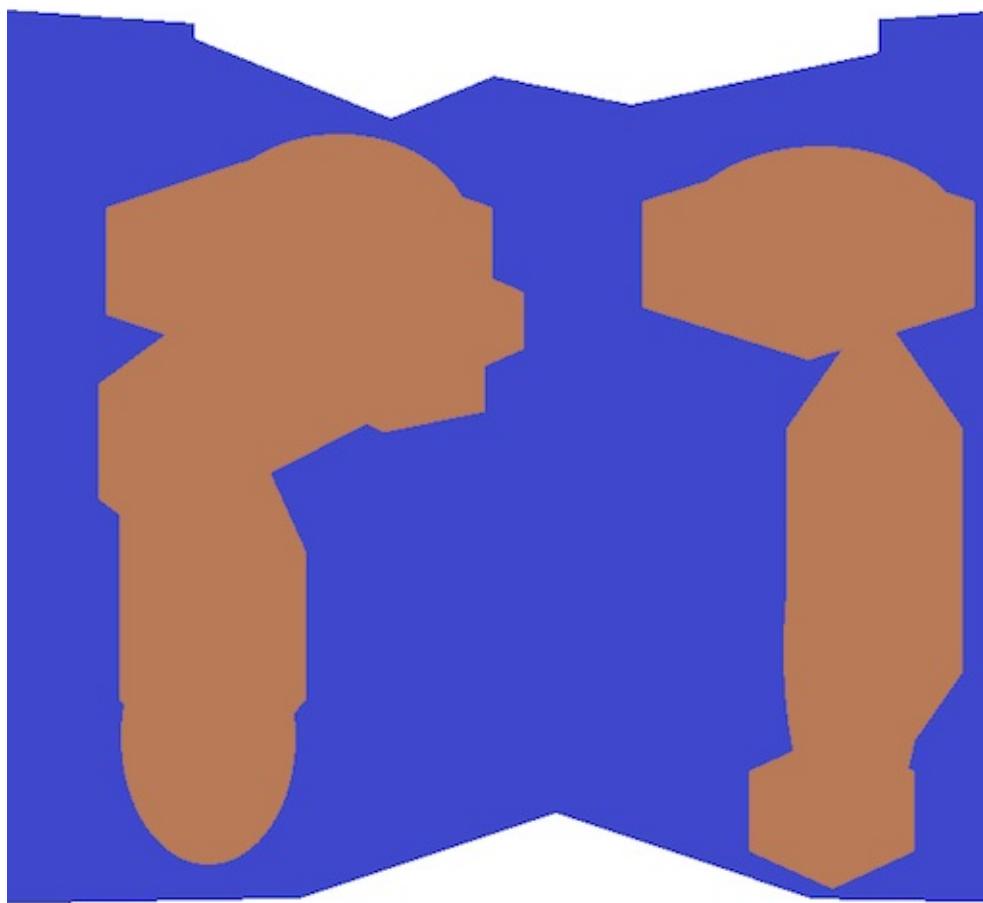
マテリアルをHierarchyのMoonの上にドラッグ&ドロップすると適用されて、Sceneウィンドウの色が変わったのがわかると思います。

同様に地球にもマテリアルmatEarthを作成して適用しましょう。地球の色はあとでテクスチャを貼るのでデフォルトのままにしておきます。

注. マテリアルをInspectorで変更した場合、例外的にセーブ操作をしなくても変更が永続的に残ります。シーンを保存せずに破棄してもマテリアルの変更は破棄されないので注意してください。

## 基本的なテクスチャ

次に地球にテクスチャを適用して陸地を作ってみましょう。サイズは縦横同じ2のn乗ピクセルのpngファイル、こんな雑な感じで大丈夫です。



---

pngファイルをProjectウィンドウのAssetsペインにドラッグ&ドロップしたらアセットとして取り込まれます。

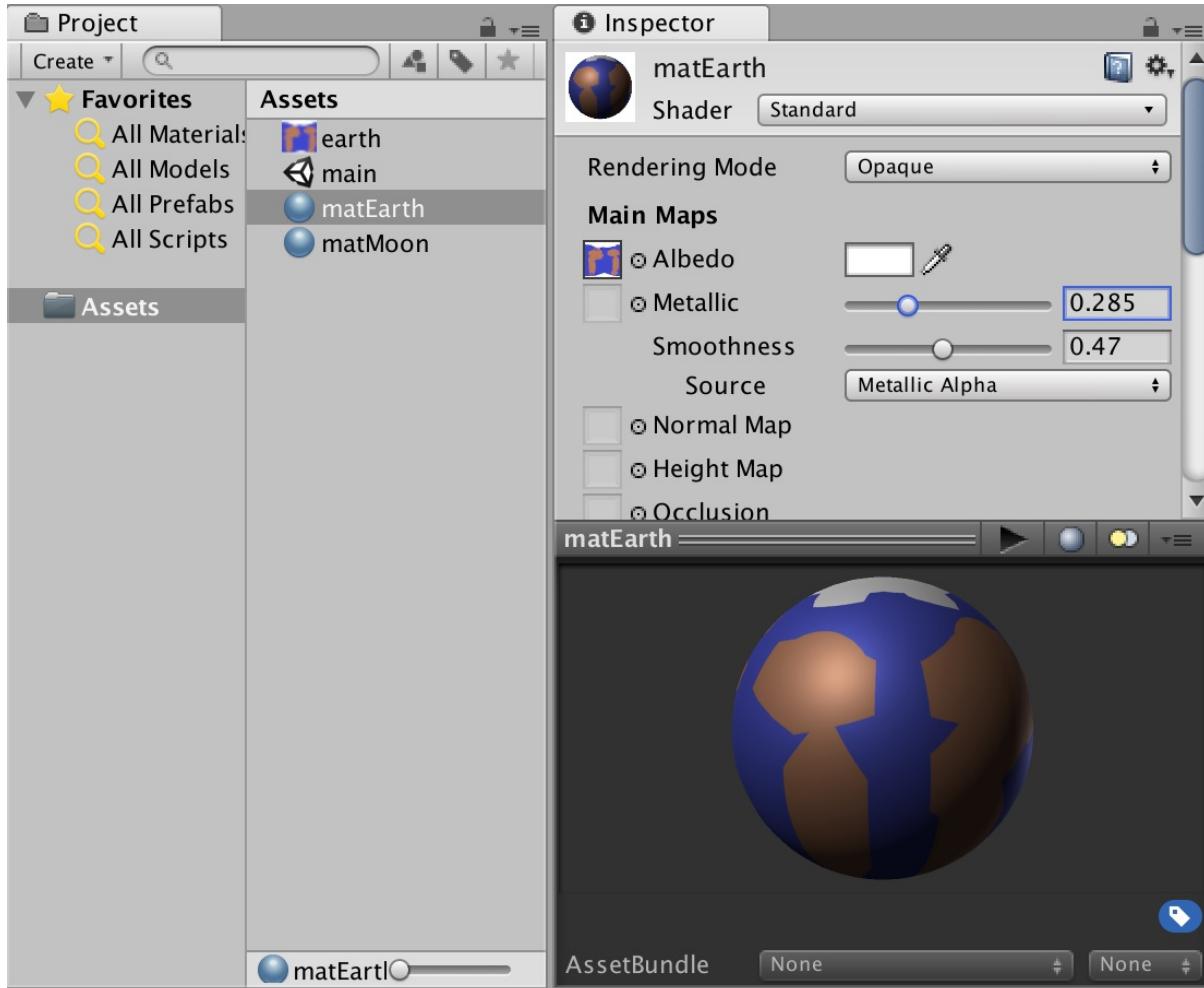
テクスチャはそのままではモデルに貼れず、一旦マテリアルにする必要があります。先ほど作成したmatEarthのAlbedoの左側の正方形にドラッグ&ドロップするか、その隣の小さな円をクリックして選択します。

matEarthの設定：

Albedo : R:255, G:255, B:255, A:255 テクスチャEarthを指定

Metallic : 0.285

Smoothness : 0.47



## 基本的なカメラと光の演出

Unityで光を表現するためには以下の設定をする必要があります。

- Directional Light 降り注ぐ光
- Environment Light 環境光
- Skybox 空や地面などすべての遠景

また、シーンに発光する物体がある場合、上記に加えて以下の設定をおこないます。

- マテリアルのEmission 発光する物体の場合
- Point Light 発光物近辺のオブジェクトへの影響
- Bloom 発光物近辺の空気への影響

これらの各種光の設定が、それぞれ別の画面にあるところがUnityのわかりにくさでもあります。一度慣れてしまえば効率よく作業できるので、頑張って操作を覚えましょう。

### Directional Light 降り注ぐ光

Directional Lightはデフォルトで配置されています。通常はInspectorでRotation、Color、Intensityを調整するくらいです。今回はあとで太陽と差し替えるということもあり、そのままにしておきます。

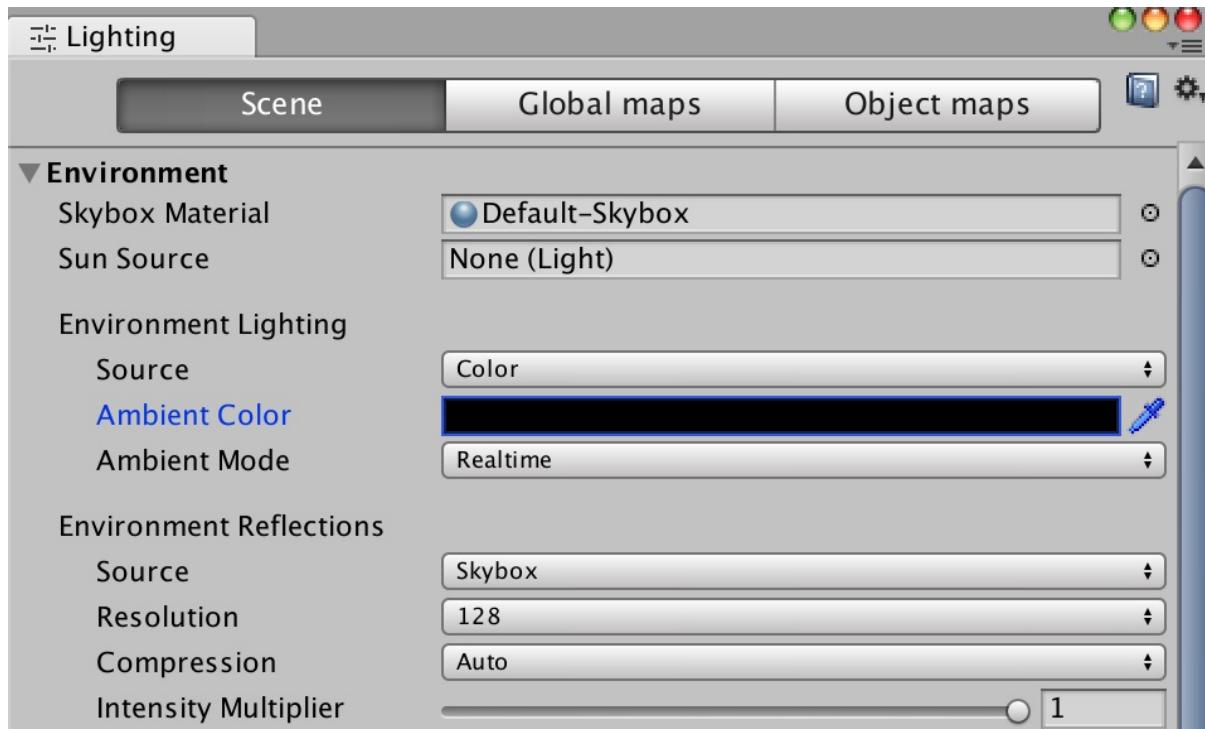
### Environment Light 環境光

今回は舞台が宇宙であり、光のあたらないところは真っ暗にしたいので環境光を暗くします。

Windowメニュー→Lighting→Settingsでダイアログを開きます。

上段のボタンで「Scene」が選択されていなければ選択してください。

Environment Lightingで、SourceをColor、Ambient ColorをR:0,B:0,G:0に設定します。

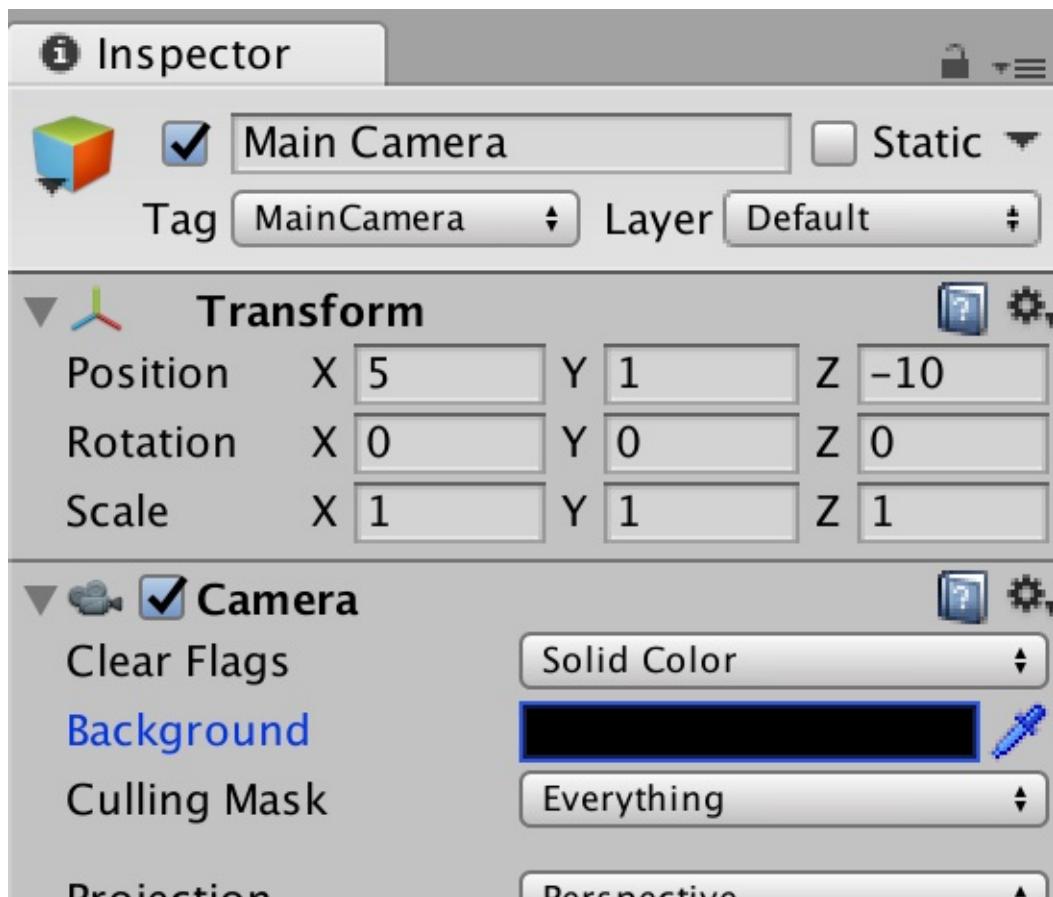


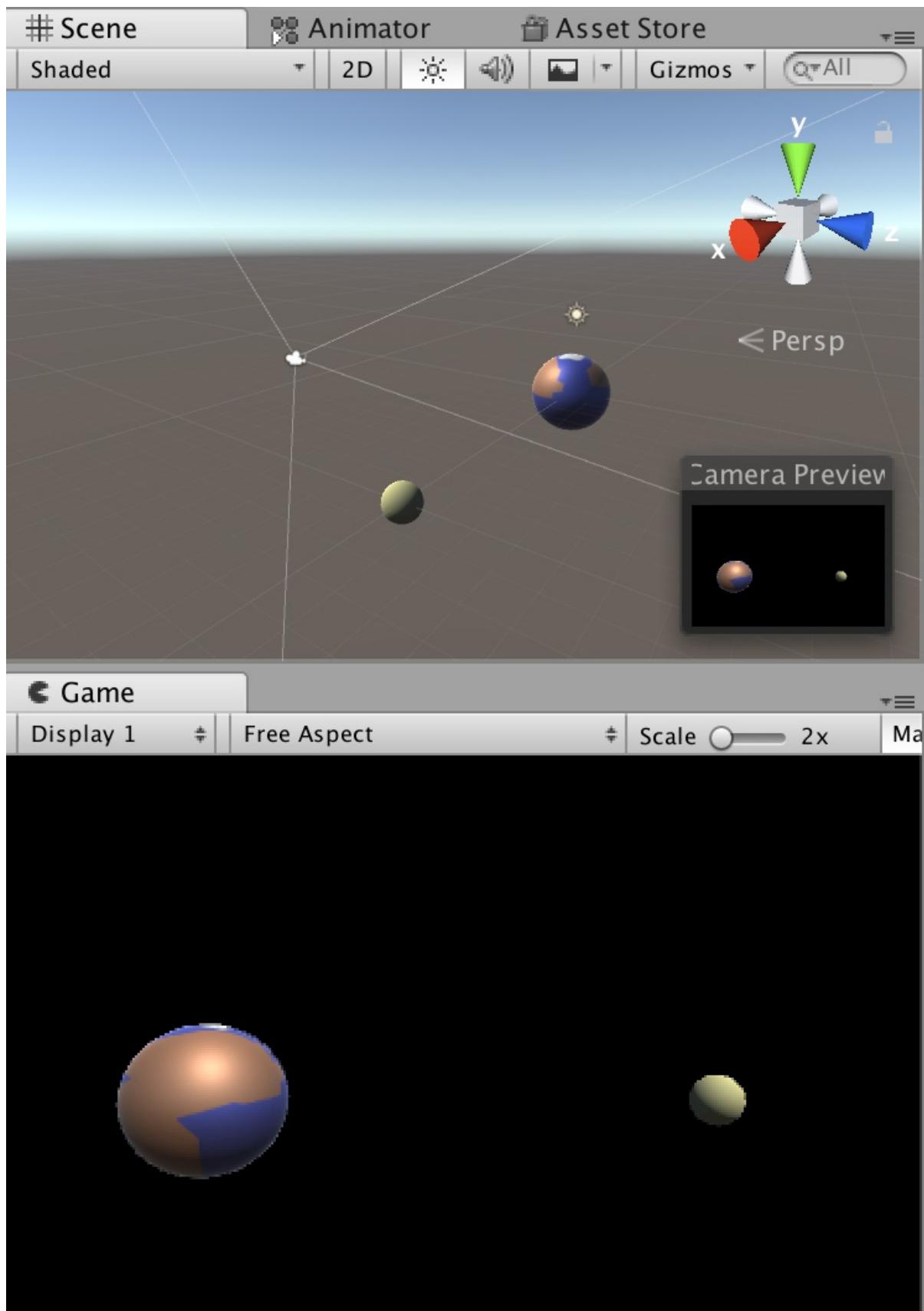
## Skybox

SkyboxはカメラのInspectorにあります。Main Cameraを選択してInspectorで、Clear FlagsをSolid Color、BackgroundをR:0,G:0,B:0に設定します。

ついでに、そのままだと月がちょっと見づらい位置にあるので、Main CameraのPositionを5,1,-10に変更します。

Sceneビューはそのままですが、Gameビューの背景色が変わったのがわかると思います。





## 物体の回転

## 自転

次に、スクリプトを書いて地球を自転させましょう。

HierarchyでEarthを選択してInspectorの最下段のAdd Componentボタンを押します。New Scriptを選択してスクリプトを生成します。スクリプト名は任意ですが、ここでは「Spinner」としておきます。

Spinner.cs

```
using UnityEngine;

public class Spinner : MonoBehaviour {

    float angle = 0;

    void Start () {

    }

    void Update () {
        angle = Time.deltaTime * 360;
        transform.Rotate(Vector3.up, -angle);
    }
}
```

Update関数で毎フレーム角度を計算して、このスクリプトがアタッチされたオブジェクトを回転します。`angle = Time.deltaTime * 360;` で角度を計算しています。Unityで移動や回転のスクリプトを書くときは、必ず Time.deltaTimeを使って速度を決めるようにしてください。これは1フレーム分の時間が格納されている変数で、たとえば30fpsの場合は、 $1 \div 30 = 0.03333\dots$ というような値が入っています。この値を毎回使うことで、CPUの遅いマシンや、重たい処理でフレームレートが十分出ないときでも一定の速度で動かすことができます。逆にこの値を使わないと、高速なPCで実行したときに速すぎてゲームにならないこともあります。

ここでは、Time.deltaTimeに360（度）をかけているので、1秒間に1回転することになります。つまり現実の1秒がこの宇宙での1日を表しています。

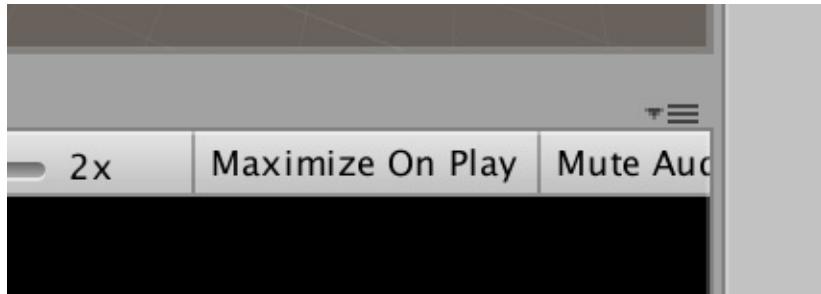
Rotate関数のVector3.upは真上に向いたベクトルで、回転運動の軸がy軸であることを指定しています。`new Vector3(0,1,0)`と書いても同じです。

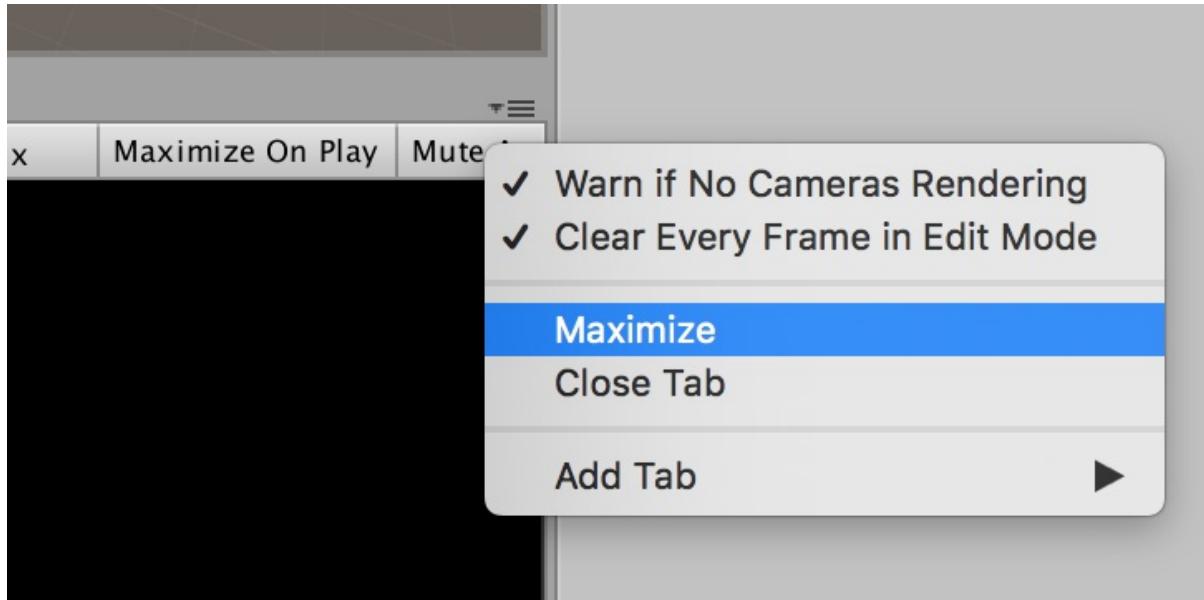
`-angle`のようにマイナス符号をつけると反時計まわりの回転となります。

## 実行確認

ここまでできたらスタートボタンを押して実行してみましょう。地球が1秒間に1回まわっていれば成功です。

実行にあたりおすすめの機能があります。GameウィンドウのMaximize On PlayをクリックするとUnityエディタのサイズで実行画面を表示するので見やすくなります。もう一度クリックするとGameウィンドウのサイズのまま実行するため、今度は動いている最中のHierarchyの状態を確認しやすくなります。実行中にサイズを変えたいときは、右上にある小さな横線のようなメニューアイコンをクリックしてMaximizeを選択します。





## 公転

次に、月の公転を実装して地球のまわりを回してみましょう。先ほどと同じようにMoonを選択してAdd Component→New Script、スクリプト名はOrbiterとします。

Orbiter.cs

```
using UnityEngine;

public class Orbiter : MonoBehaviour {

    float radius = 10; // 回転半径
    float cycle = 27; // 回転周期 27秒
    float angle = 0;

    void Start () {

    }

    void Update () {
        angle -= Time.deltaTime * 2 * Mathf.PI / cycle;
        float x = Mathf.Sin(angle) * radius;
        float z = Mathf.Cos(angle) * radius;

        transform.position = new Vector3(x, 0, z);
    }
}
```

ここでも、Time.deltaTimeを使って角度を計算しています。今回は度ではなく、ラジアンなので $360$ の代わりに $2\pi$ をかけています。使用する関数によって、同じ角度でも度だったりラジアンだったりするので混乱しないように注意してください。

計算した角度をSin関数とCos関数に渡して位置を計算します。サイン、コサインは通常、三角関数と呼ばれます  
が、これは本質的には回転と往復の関数だと考えた方が良いです。自然界で目にするような回転運動や往復運動のほとんどはサインとコサインで表現することができるので、ぜひマスターするようにしましょう。ここではサインをx軸、コサインをz軸にしています。この場合は(0,0,0)を中心とした円運動になります。

cycleは回転周期です。月は27日かけて地球のまわりを1回転するので27としています。1秒に1回転する回転速度をcycleで割ることで、回転速度を $1/27$ にしています。

ここまでできたら、また実行確認してみてください。地球が自転して、そのまわりを月がゆっくり周回しているでしょうか。画面からはみでて見づらいようなら、Main Cameraの位置を少し動かして調整してみてください。

## 太陽を作る

今度は太陽を作つて地球が太陽のまわりを回るようにしましょう。中心に太陽を置けるようにSceneウィンドウで地球と月の位置を少し動かして、HierarchyにSphereを追加してください。Positionは0,0,0、Scaleは4,4,4、名前はSunにします。※このサンプルで作成する太陽、地球、月の大きさや距離の比率は実際とは異なります

次に太陽のマテリアルmatSunを作成します。マテリアルのパラメータを次のように設定してください。

Albedo : R:255, G:0, B:0, A255

Metallic : 0

Smoothness : 0.5

これだけではまだ発光した感じになりません。Emissionにチェックを入れてEmissionのColorを次のように設定します。

Current Brightness : 5

R:5, G:2.7, B:2.7

こうすることによりColorにHDRの文字が表示され、HDR対応ディスプレイの場合高輝度で表示されます。

しかし、まだ太陽はただの白い球体に見えます。これから光り輝く恒星にするためには、まだ次節以降の何段階かの手順を必要とします。



## ポイントライトで他のオブジェクトに影響を与える

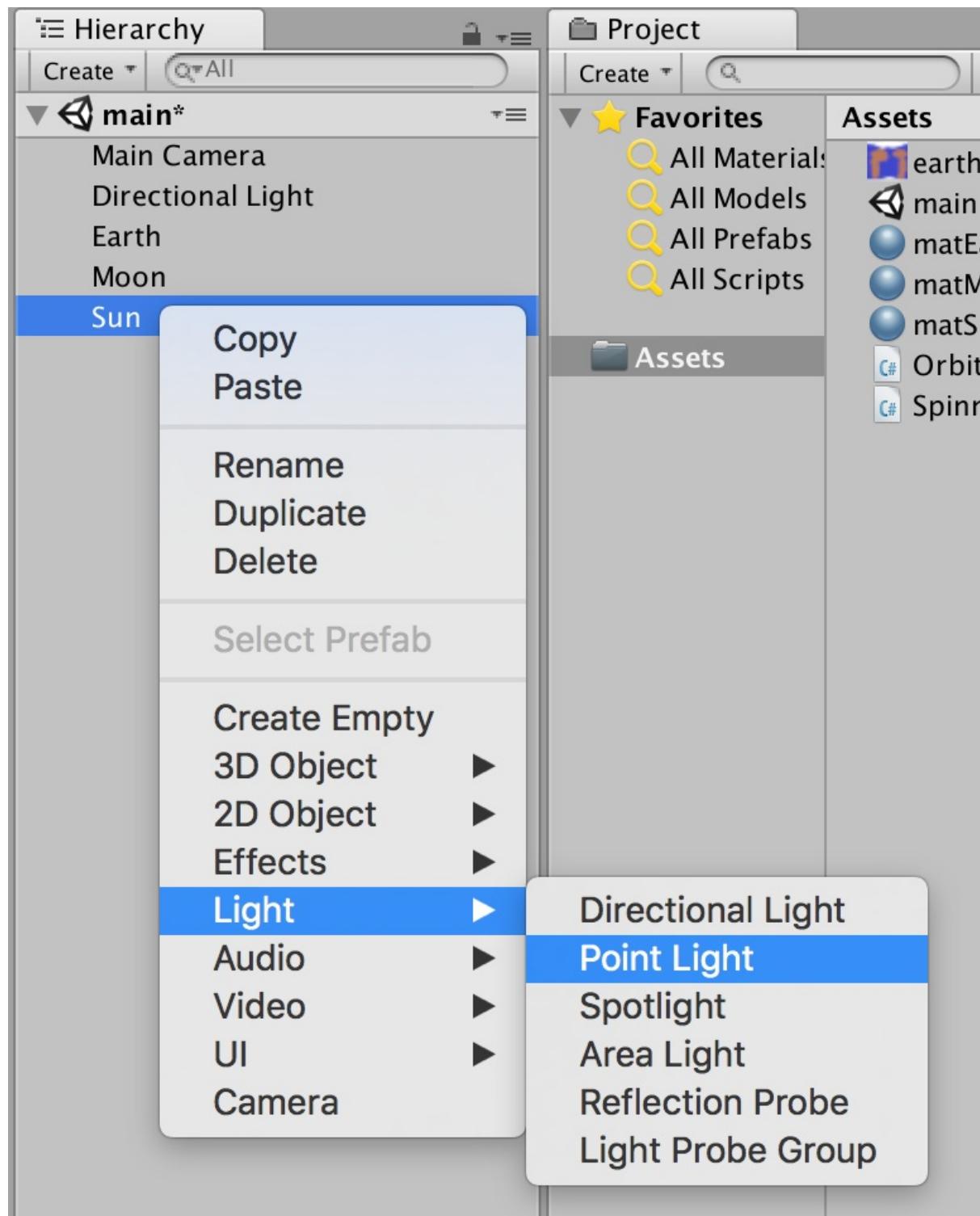
以前、[基本的なカメラと光の演出](#)の項の冒頭で以下のように書きました。

シーンに発光する物体がある場合、上記に加えて以下の設定をおこないます。

- マテリアルのEmission 発光する物体の場合
- Point Light 発光物近辺のオブジェクトへの影響
- Bloom 発光物近辺の空気への影響

現在、太陽は上記のマテリアルの設定をしただけの状態です。続いてポイントライトを設定していきます。

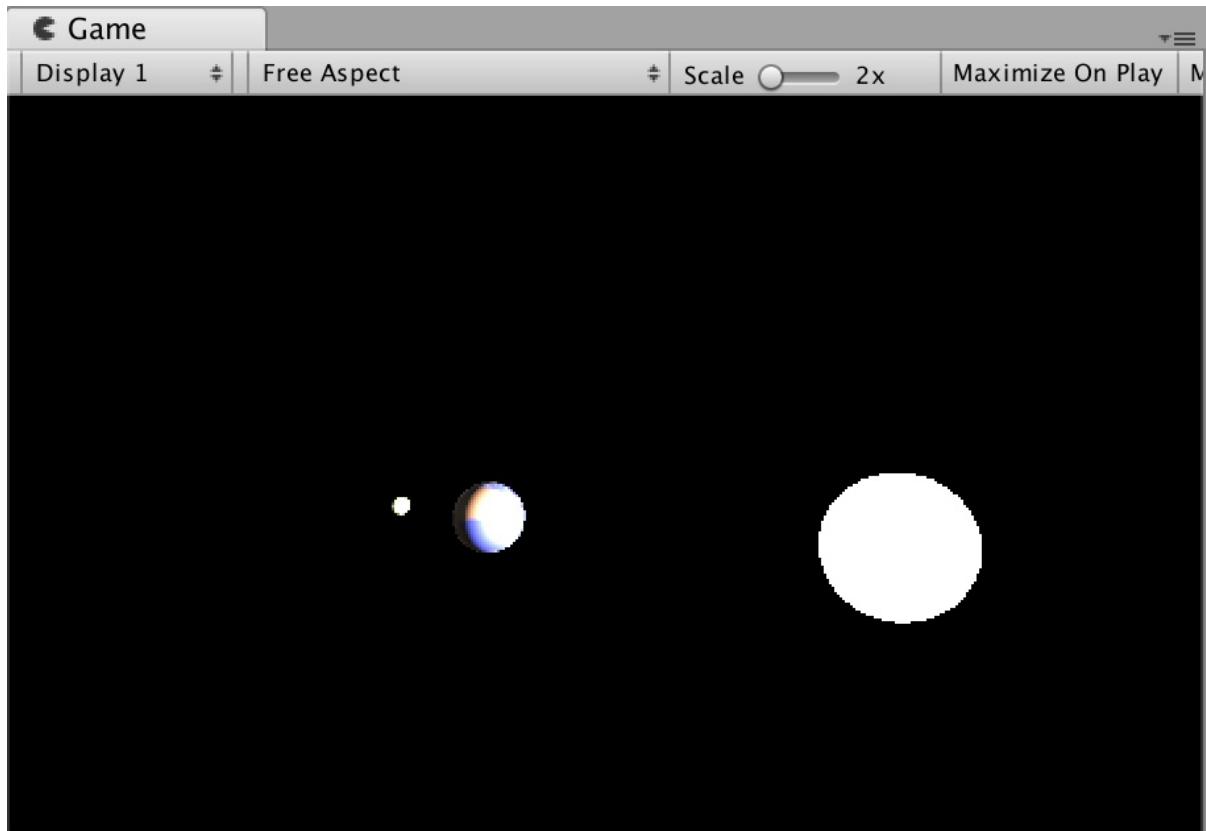
HierarchyでSunを選択した状態で、右クリックからLight→Point Lightを選択してください。



次にPoint LightのInspectorから、Range:100、Intensity:10に設定します。これで距離100まで届く、明るい光となりました。Colorはデフォルトの白いまま大丈夫です。

シーンにPoint Lightを置いたので、Directional Lightは不要になります。Hierarchyで選択して削除するか、Inspectorの一番上のチェックボックスを外して無効にします。

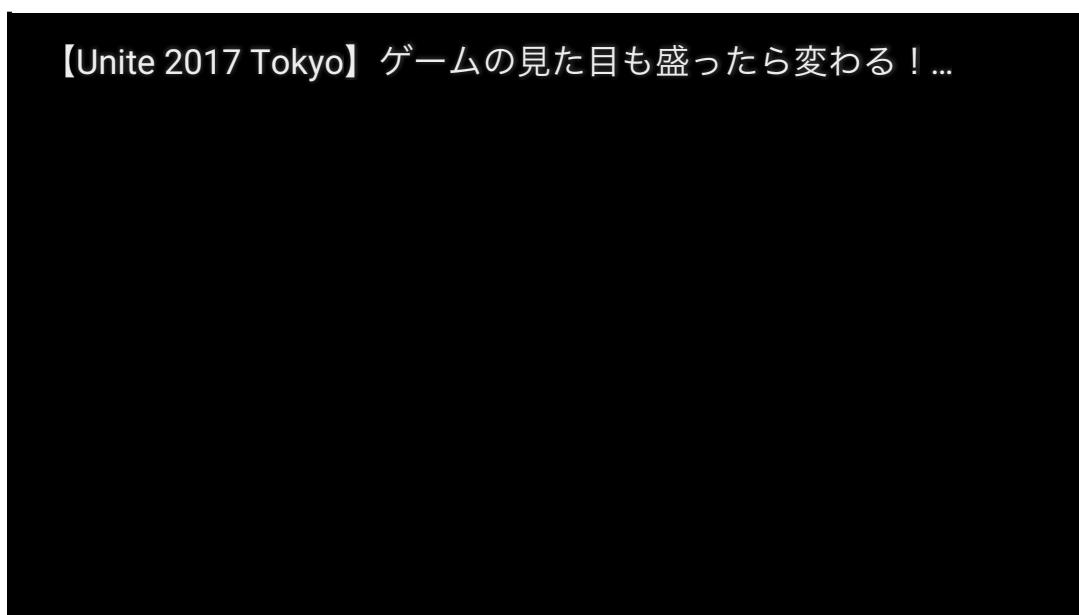
カメラやオブジェクトの位置を調整してGameウィンドウに全ておさまるように調整してください、以下のような表示になったでしょうか。太陽の光が月と地球の太陽側の面を照らしだしているのがわかると思います。



## ポストプロセッシング

太陽の仕上げに、ポストプロセッシングでブルーム効果を付け加えます。Unityのポストプロセッシングは導入するのにちょっと手間がかかりますが、非常に強力な各種ポストプロセス処理が可能で、美しい画面を作るのに必須なのでぜひマスターするようにしてください。

Unityのポストプロセスには、比較的新しいPost Processing Stackというアセットを使用します。Post Processing Stackは非常に多機能なので、機能の全貌や使用方法は以下の動画を見てください。50分ある長い動画ですが必見です。



ここでは、今回必要になる機能と手順だけ説明します。

Post Processing Stackはデフォルトの機能ではないのでアセットストアからダウンロードする必要があります。Sceneウィンドウのタブに並んでいるAsset Storeタブを選択してください。Asset Storeタブがない場合はWindowメニュー→Asset Storeから開くことができます。

検索フォームに「Post Processing Stack」と入力して検索して見つかったらダウンロードします。アセットストアのアセットから使いやすいものかどうかを見極めるのはなかなか難しいですが、今回はUnity公式から提供されており、利用者の評判の高い定番アセットなので比較的安心して導入できます。

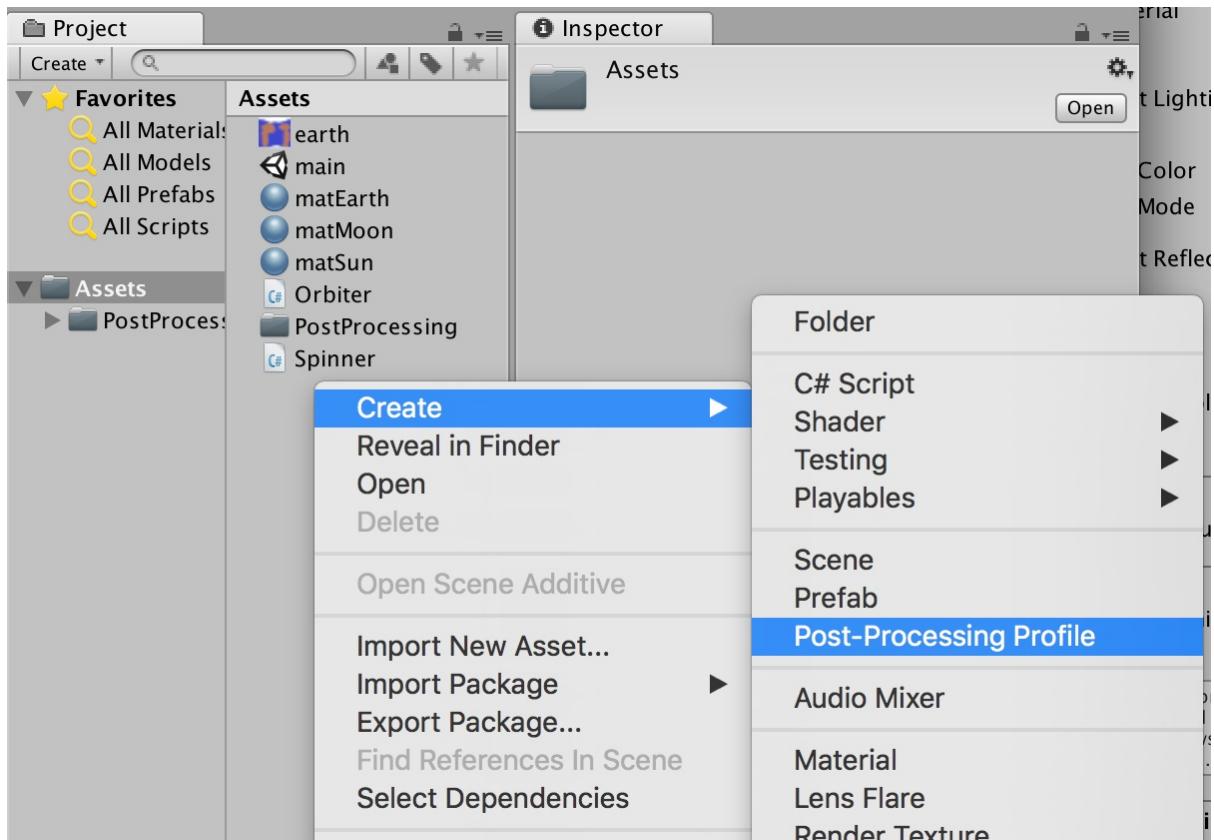
ダウンロードが終わったらプロジェクトにインポートします。（この辺追記するかも）アセットはプロジェクト内にダウンロードされるのではなく、ディスクの専用の場所にマスターとしてダウンロードされています。それをプロジェクトにインポートするとマスターの複製がプロジェクトフォルダにコピーされます。

インポートしたアセットは、Assetsフォルダの下にアセットごとにフォルダを作って格納されます。もし、あるアセットをインポートした後にやっぱりやめたくなった場合、そのフォルダごと削除すればインポートを取り消すことができます。その場合でもダウンロードしたマスターは残っているので、後からまた別のプロジェクトにインポートしたりすることも可能です。

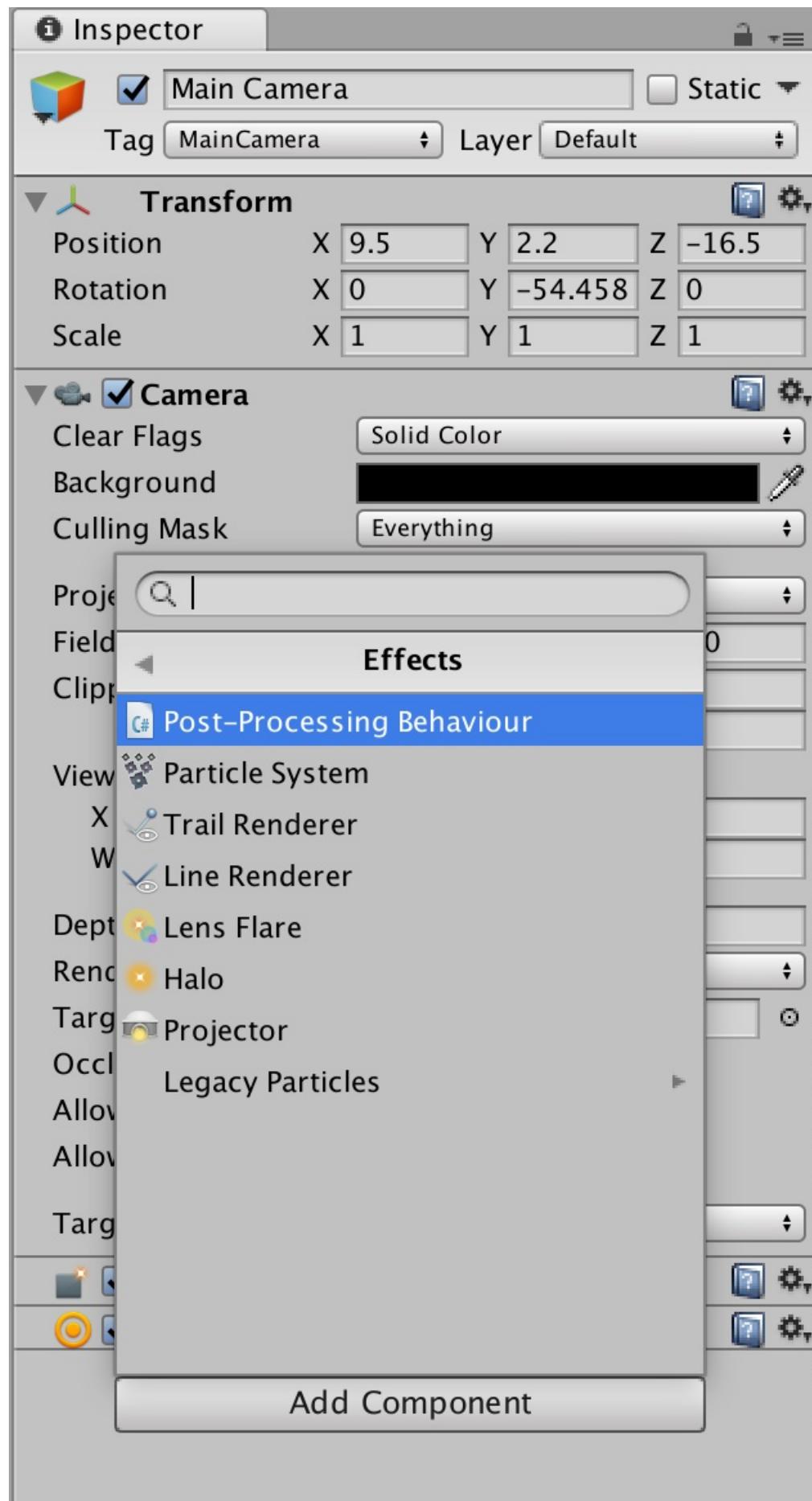
アセットのインポート時に、項目ひとつひとつに対してインポートするかどうか選択できる画面が出てきます。知識がないとここで必要か不要か判断するのは難しいので、最初のうちはすべて選択した状態でインポートしてください。余計なものをインポートしてしまっても、シーンに含まれないものはビルド後の実行ファイルには含まれないので問題ありません。

Post Processing Stackを利用するには、プロファイルの作成・設定と、メインカメラの設定が必要です。順を追って説明します。

まず、ProjectウィンドウのAssetsで右クリックメニューからCreate→Post-Processing Profileを選択します。プロファイル名は「Post-Processing Profile」にしておきます。



次にMain Cameraを選択して、InspectorからAdd Component→Effects→Post-Processing Behaviourと選択しています。



そしてMain Cameraに追加されたPost Processing BehaviourコンポーネントのProfileに、先ほど作成したPost-Processing ProfileをAssetsからドラッグ＆ドロップすれば準備完了です。

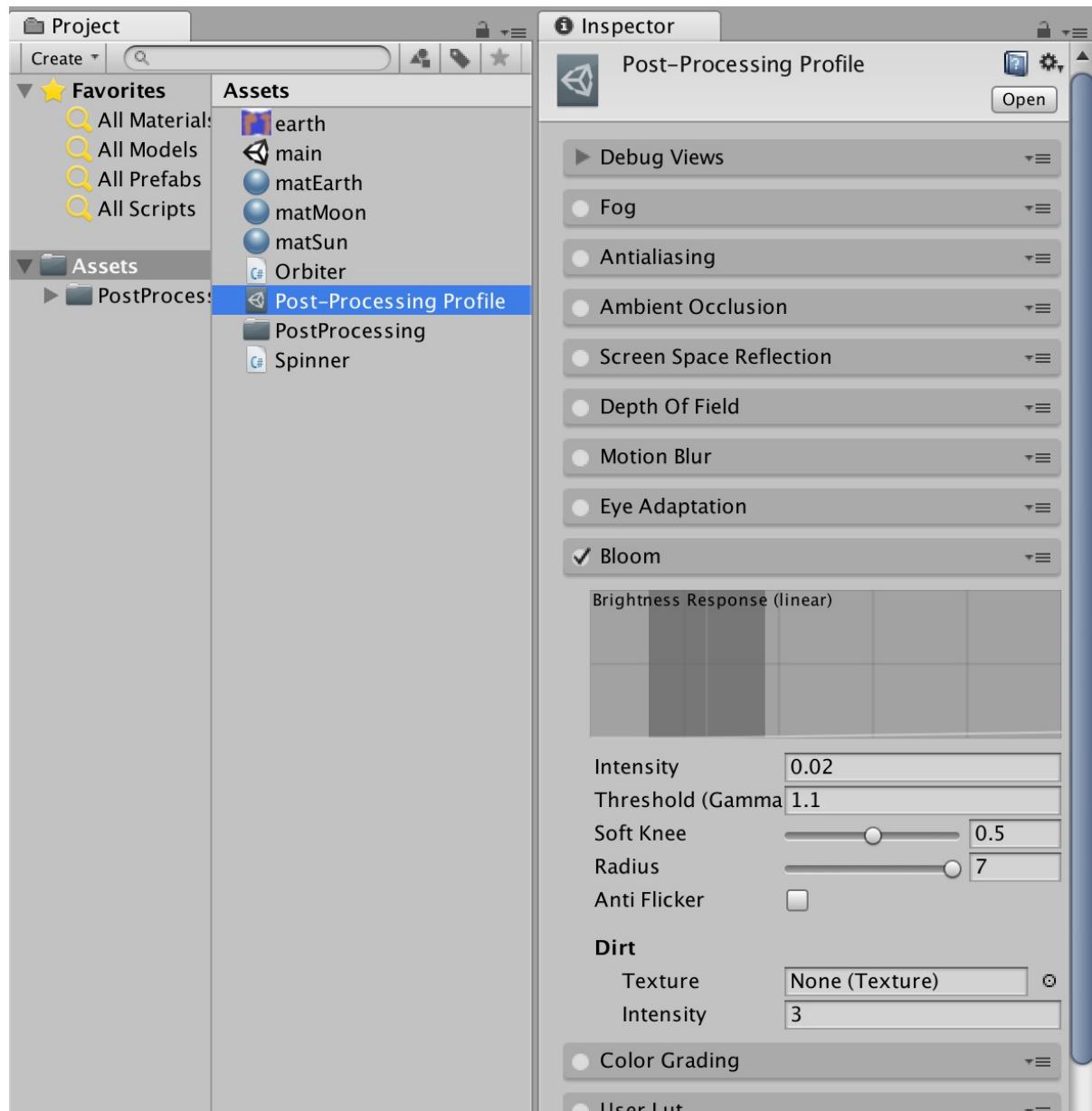
Post-Processing Profileを選択してInspectorを見ると、さまざまな項目が設定できるのがわかると思います。今回はBloomを使っていきます。 Bloomの項目にチェックを入れてパラメータを次のように設定します。

Intensity : 0.02

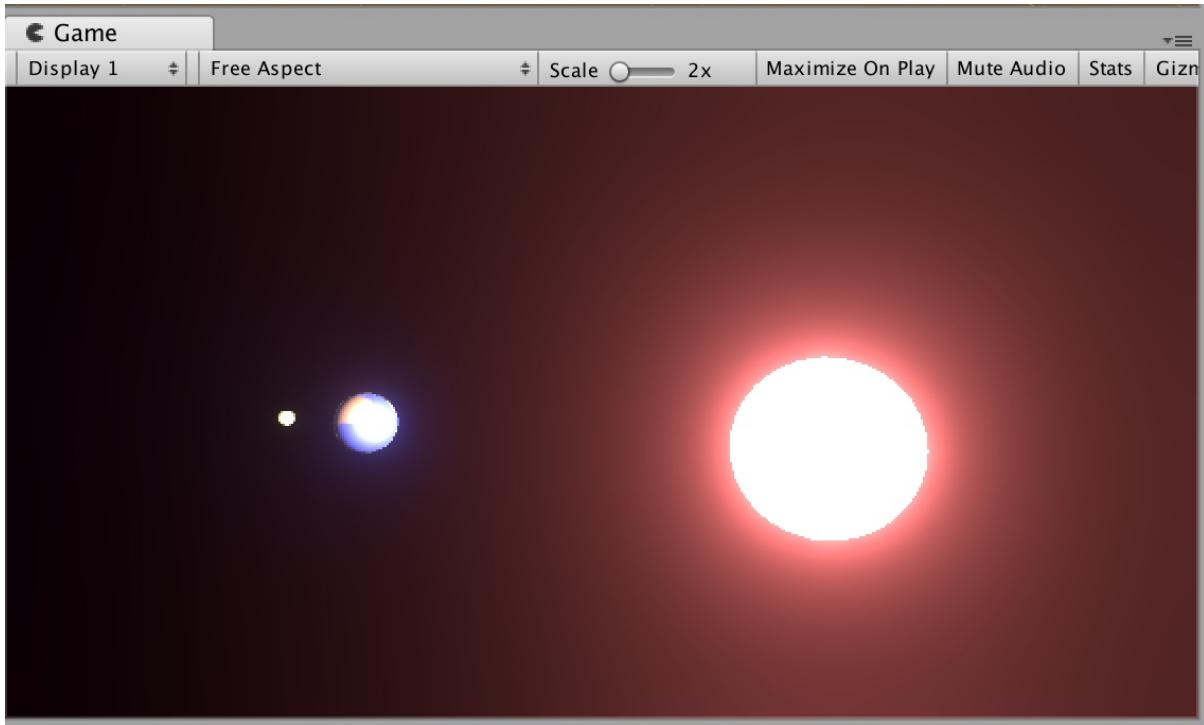
Threshold (Gamma) : 1.1

Soft Knee : 0.5

Radius : 7



これで太陽光の設定は完了です。ポストプロセスはこの後もさまざまな用途で使用するので少しづつおぼえてください。



## 地球の公転

いよいよ太陽のまわりを地球が回転するようにスクリプトを設定します。書いたOrbiter.csを改造して地球の公転、月の公転の両方で使えるようにして見ましょう。

Orbiter.cs

```
using UnityEngine;

public class Orbiter : MonoBehaviour {

    public GameObject centerObject; // 回転中心のオブジェクト
    public float radius; // 回転半径
    public float cycle; // 回転周期

    float angle = 0;

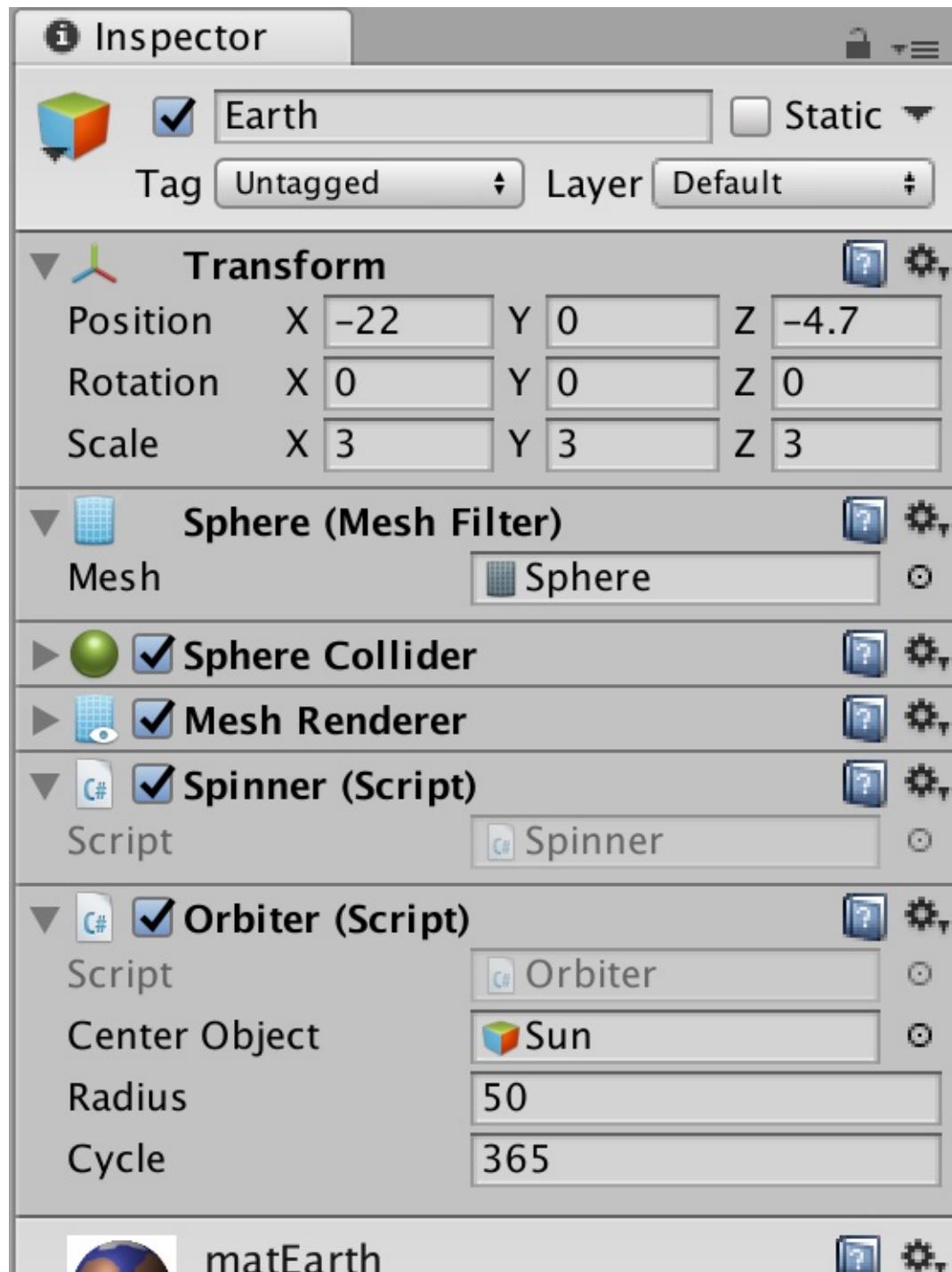
    void Start () {
    }

    void Update () {
        angle -= Time.deltaTime * 2 * Mathf.PI / cycle;
        float x = Mathf.Sin(angle) * radius;
        float z = Mathf.Cos(angle) * radius;

        Vector3 center = centerObject.transform.position;
        transform.position = center + new Vector3(x, 0, z);
    }
}
```

回転半径と回転周期をpublic変数にしてInspectorで設定できるようにしました。また、centerObjectというpublic変数を用意して回転の中心を指定できるようにしています。つまり月の公転では回転中心は地球、地球の公転では回転中心は太陽となります。地球のように常に移動している回転中心もあるので、Update関数内で毎フレーム中心位置を取得しています。

EarthのInspectorにOrbiterスクリプトをドラッグ&ドロップしてアタッチしてください。アタッチできたら、Center ObjectにSunを選択、Radius : 50、Cycle : 365と設定します。



月はすでにOrbiterスクリプトがアタッチされているはずなのでパラメータだけ以下のように設定します。  
Center Object : Earth      Radius : 10      Cycle : 27

全体が見やすい位置にMain Cameraを移動して実行してみましょう。Main Cameraの位置はたとえば以下の設定がおすすめです。

Position : 20, 10, 10

Rotation : 20, -50, 0

RotationのXを少し上げることで下のものを見下ろす感じになります。RotationのZを0以外にすると不自然に傾いた絵になるので必ず0にしておきます。

太陽の周囲を自転しながら地球がまわり、さらに地球の周囲を月がまわっているでしょうか。

## 見た目のクオリティ向上

### 背景の変更

背景が真っ黒だと少し味気ないので星空に変更しましょう。アセットストアでSkyboxを検索するとさまざまな背景が見つかります。今回はシンプルな星空のこちらを使用することにします。

*Ursa Major*

Stellar Sky

Textures & Materials/Skies

Wolfnley

★★★★★ (13)

FREE

Download

Share

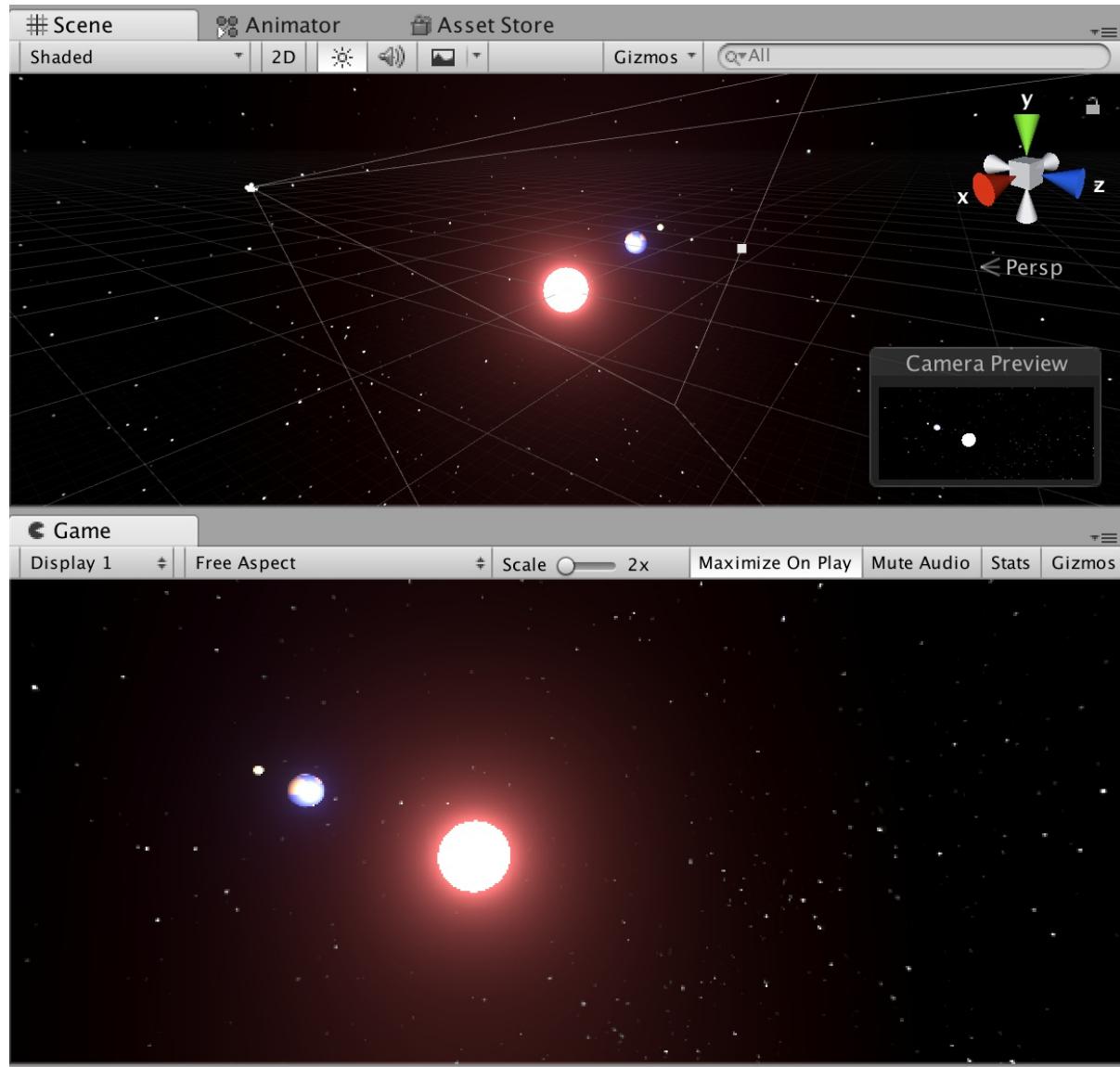
Stellar Sky (SS) is the fully real skybox model of the starry sky. You can use this package for any game with night sky. Contains 512x512, 1024x1024, 2048x2048 version of skybox.

アセットストアで「Stellar Sky」を検索、ダウンロード、インポートしてください。

インポートしたらフォルダの中身をざっくり確認します。Stellar Sky/Skyboxesフォルダの下に512、1024、2048とフォルダがあり、SS\_512、SS\_1024、SS\_2048と3種類のSkyboxが含まれているようです。

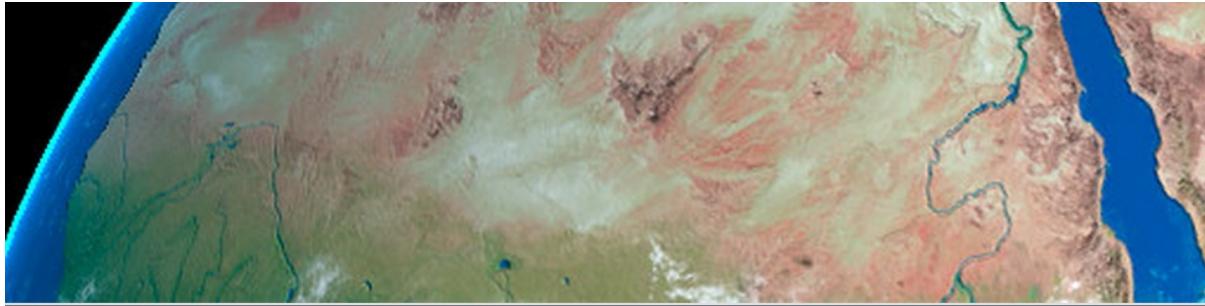
Skyboxの設定はLightingウィンドウとMain Cameraの2箇所でおこないます。まず、Windowメニュー→Lighting→Settingsでウィンドウを開いて、一番上のSkybox Materialに先ほどインポートしたSS\_2048を指定します。

次にMain CameraのInspectorでCameraコンポーネントのClear FlagsをSkyboxに変更します。これでSceneウィンドウとGameウィンドウの両方とも星空の画面になりました。



## 地球モデルの差し替え

これまで雑に作ったテクスチャで地球を表現していましたが、さすがにもうちょっと格好良くなります。クオリティの高い地球のモデルがアセットストアに何種類もあるので、それに差し替えてみましょう。



Planet Earth Free  
3D Models/Environments/Sci-Fi  
headwards  
★★★★ (180)  
**FREE**

[Download](#)  

The #1 Planet Earth Package just got even more awesome (and it's still free!).

アセットストアで「Planet Earth Free」を検索、ダウンロード、インポートします。

インポートしたら、またざっくり中身を確認。Planet Earth Free/Prefabsの下にEarthHigh、EarthLow、EarthMediumの3種類のモデルが入っているようです。EarthHighをHierarchyにドラッグ＆ドロップして配置します。

しかしながら、予想に反して巨大な水色の球が表示されました。HierarchyのEarthHighを見ると、子オブジェクトとしてPlanet16128TrisとEarthGlowの2個のオブジェクトがぶら下がっています。このEarthGlowが怪しいので、Inspectorで表示して一番上のチェックを外して非表示にしてしまいましょう。

巨大な水色の球はなくなったものの、まだサイズ的に大きいようです。Planet16128Trisを選択して、Scaleを0.07,0.07,0.07に変更します。このとき、親のEarthHighのScaleではなく、子のPlanet16128TrisのScaleを変更するのがコツです。トップレベルのオブジェクトはスケールが1になっている方が何かと楽です。

続いてコンポーネントを設定します。EarthHighを選択してInspectorを見てみます。Spin Freeという自転用コンポーネントが最初からアタッチされていますがこれは不要なので、右クリックメニューからRemove Componentで削除します。

代わりに自作のSpinnerとOrbiterをドラッグ＆ドロップでアタッチします。OrbiterのパラメータをEarthと同じく以下のように設定します。

Center Object : Sun

Radius : 50

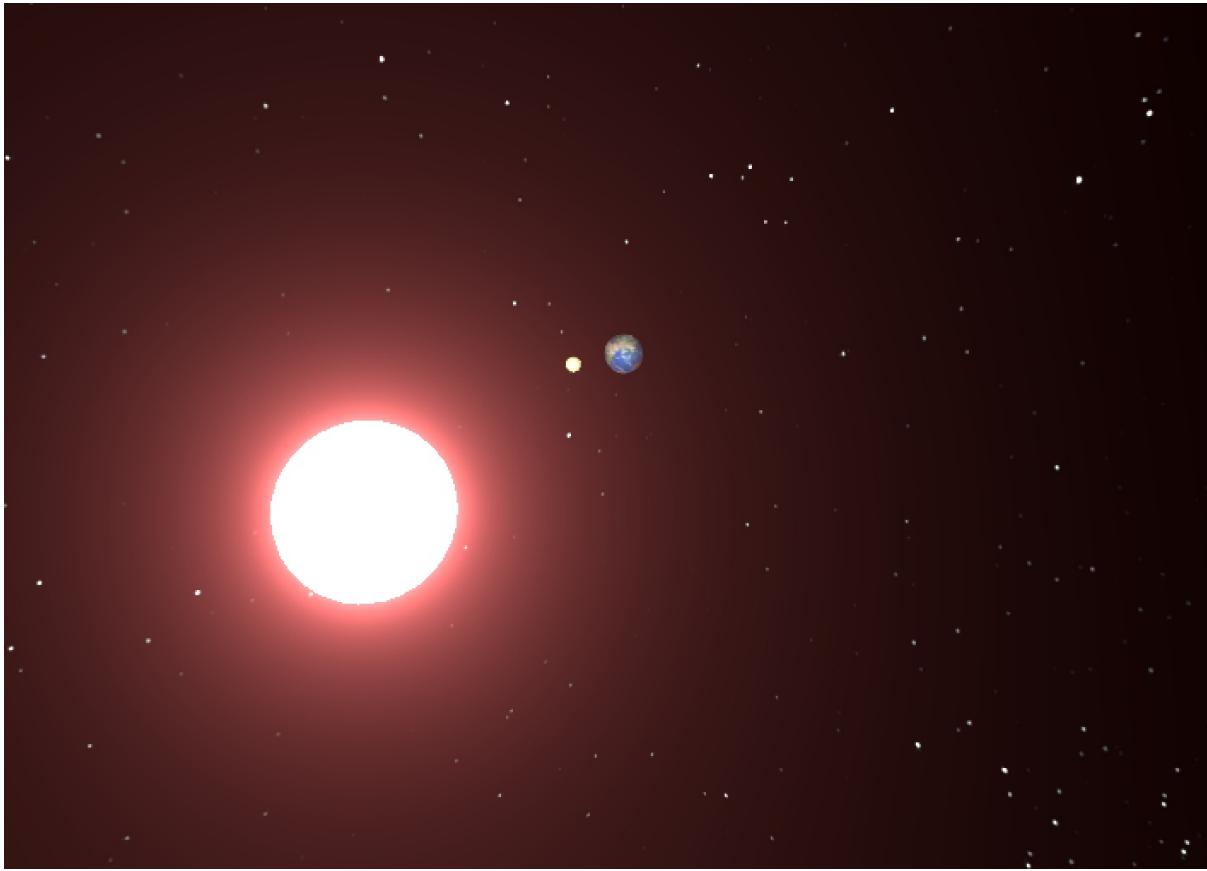
Cycle : 365

次にHierarchyでMoonを選択、OrbiterコンポーネントのCenter ObjectをEarthからEarthHighに変更します。

最後に、もともとあったEarthはもう不要なのでInspector最上段のチェックボックスを外して非表示にします。

これで地球のモデル差し替えは完了です。見た目がぐっと良くなりました。

ここまででシーンの作成は一旦完成です。次項ではこのシーンをもとに別視点のシーンを作ってみます。



## 別視点の演出

今度はMain Cameraを移動して地球から宇宙を見た風景を作っていきましょう。

その前に現在のシーンをmainとしてセーブしておきます。

次にFileメニュー→Save Scene Asで別名保存します。シーン名は「EarthView」とします。

Hierarchy内でMain CameraをドラッグしてEarthHighの子オブジェクトにします。次にカメラの座標を地表に近く、地表と地平線と同じ向きになるように、以下のようにパラメータを設定します。

Position : -1.8, 0, 1.7

Rotation : 0, 180, -90

それから地球の大きさを演出するためにカメラの画角を調整します。Main CameraのCameraコンポーネントにあるField of Viewをデフォルトの60から40に変更します。

実行してみると、あまりにも高速回転して目がまわりそうになってしまいます。少し自転と公転の回転速度を調整した方が良さそうです。Spinner.csとOrbiter.csのUpdate関数のそれぞれ1行目に「/ 50」を書き加えて、回転速度を1/50に落とします。

### Spinner.cs

```
void Update () {
    angle = Time.deltaTime * 360 / 50;
    transform.Rotate(Vector3.up, -angle);
}
```

## Orbiter.cs

```
void Update () {
    angle -= Time.deltaTime * 2 * Mathf.PI / cycle / 50;
    float x = Mathf.Sin(angle) * radius;
    float z = Mathf.Cos(angle) * radius;

    Vector3 center = centerObject.transform.position;
    transform.position = center + new Vector3(x, 0, z);
}
```

これでチュートリアル「Planet」は終了です。

この先は、たとえば2. 5. 6項で導入したポストプロセッシングのパラメータを調整して、さらに高品質な絵を追求していくこともできます。Post-Processing ProfileのAntialiasing、Motion Blur、Color Grading、Vignetteあたりを調整してみてください。



## チュートリアル2 「Penguin」

### このチュートリアルについて

今回のチュートリアルでは光を追いかけるように顔を向けるペンギンを作っていきます。項目としては以下のような内容を学んでいきます。

- プレハブの使い方
- 乱数の使い方
- 水の表現
- ターゲットの方向を向く処理
- 三角関数を使ったさまざまな動き



### モデルデザイン用シーン作成

このテーマではmainとmodelのふたつのシーンを作成します。mainは通常のゲーム画面用のシーン、modelはゲーム用のモデルを準備するためのシーンです。実際のゲーム開発でもこのようにシーンをふたつ使うことがよくあります。ゲーム画面ではスタートしてしばらく経ってからキャラクターや背景のモデルが出現することがあります。また出現したモデルも高速に動きまわっていたり、複数同時に現れたりして、モデルの見え方をさまざまな方向からじっくり確認するのには向いていません。そのため、モデルを一覧表示して確認・調整するための専用のシーンを開発用に用意します。また、こうしてシーンを別にすることで、ゲームを実装するプログラマーと、モデルを作るデザイナーが分業しやすくなる、という大きなメリットもあります。

Unityの起動画面、あるいはFileメニュー→New Projectから新規プロジェクトを作成します。プロジェクト名は「Penguin」にします。

### ステージの作成

最初にmodelシーンを作っていきます。modelシーンには、ステージ（動物園のペンギン舎）のモデルと、ペンギンのモデルひとつだけが置かれます。ここでは、MayaやBlenderといった外部のDCCツール(Digital Content Creation Tool)を使用せず、Unityのプリミティブの組み合わせだけでモデルを作ります。

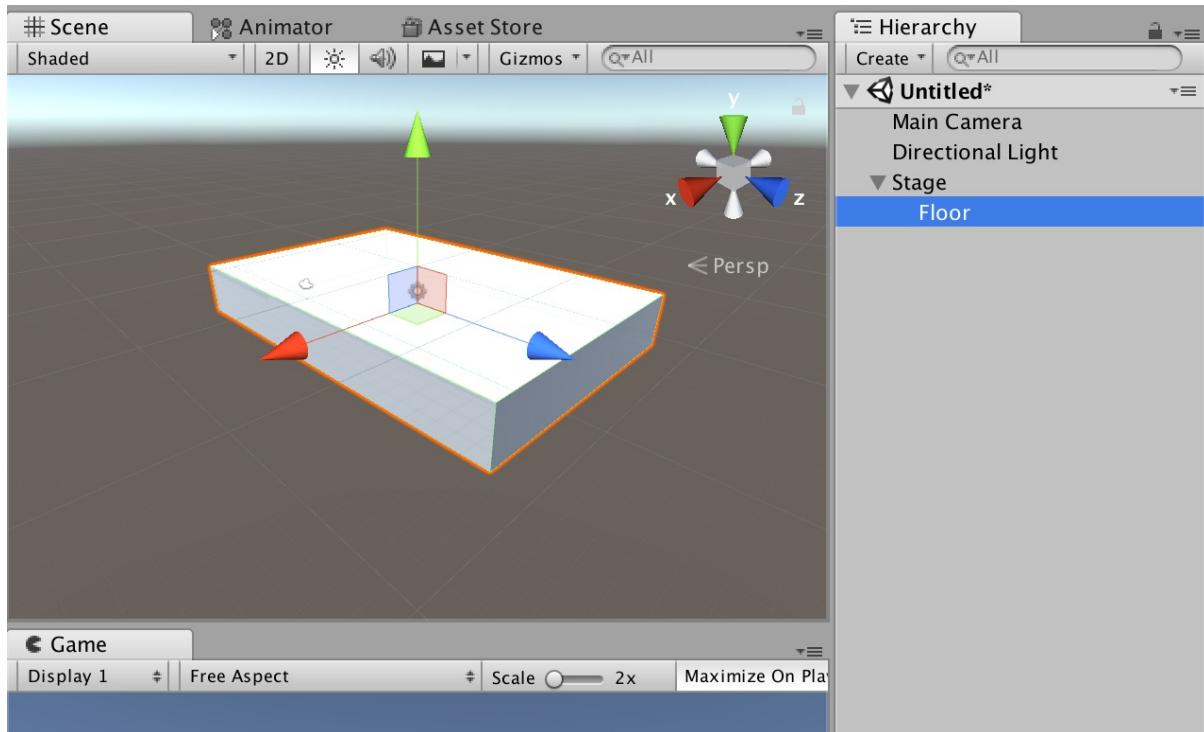
Hierarchyで何も選択されていないことを確認し、右クリックメニュー→Create Emptyで空のオブジェクトを作成し、名前を「Stage」と変更します。この下にステージのパーツを子オブジェクトとしてぶら下げていきます。親となるオブジェクトを空ではないキューブや球体にすることもできますが、後々必要になる個別の移動やスケール変更といった調整作業を考えると一番親は空オブジェクトにしておくのが楽です。

次にHierarchyでStageを選択した状態で、右クリックメニュー→3D Object→Cubeを選択してキューブを作成し、名前を「Floor」にします。生成したFloorを選択してInspectorで以下のようにパラメータを設定します。

Position : -10, -3, -10

Scale : 20, 4, 30

ここまでできたらCtrl+S(Macは⌘+S)でシーンを一旦セーブしましょう。シーン名は「model」とします。



同様にStageを選択してCubeをあと2個作ります。名前はWall1、Wall2としてそれぞれ以下のパラメータを設定します。

Wall1

Position : 10, 3, 5

Scale : 60,10,1

Wall2

Position : 40, 3, -14

Scale : 1,10,40

次にこれらのマテリアルを作成します。ProjectウィンドウのAssetsの空き領域を右クリック、メニューから→Create→Materialという操作は前のチュートリアルでおこなった手順と同じです。名前はmatIceとします。このマテリアルは氷をイメージするような水色に設定します。本物の氷というよりも動物園のペンギン舎の壁に近い質感にすると良い感じです。

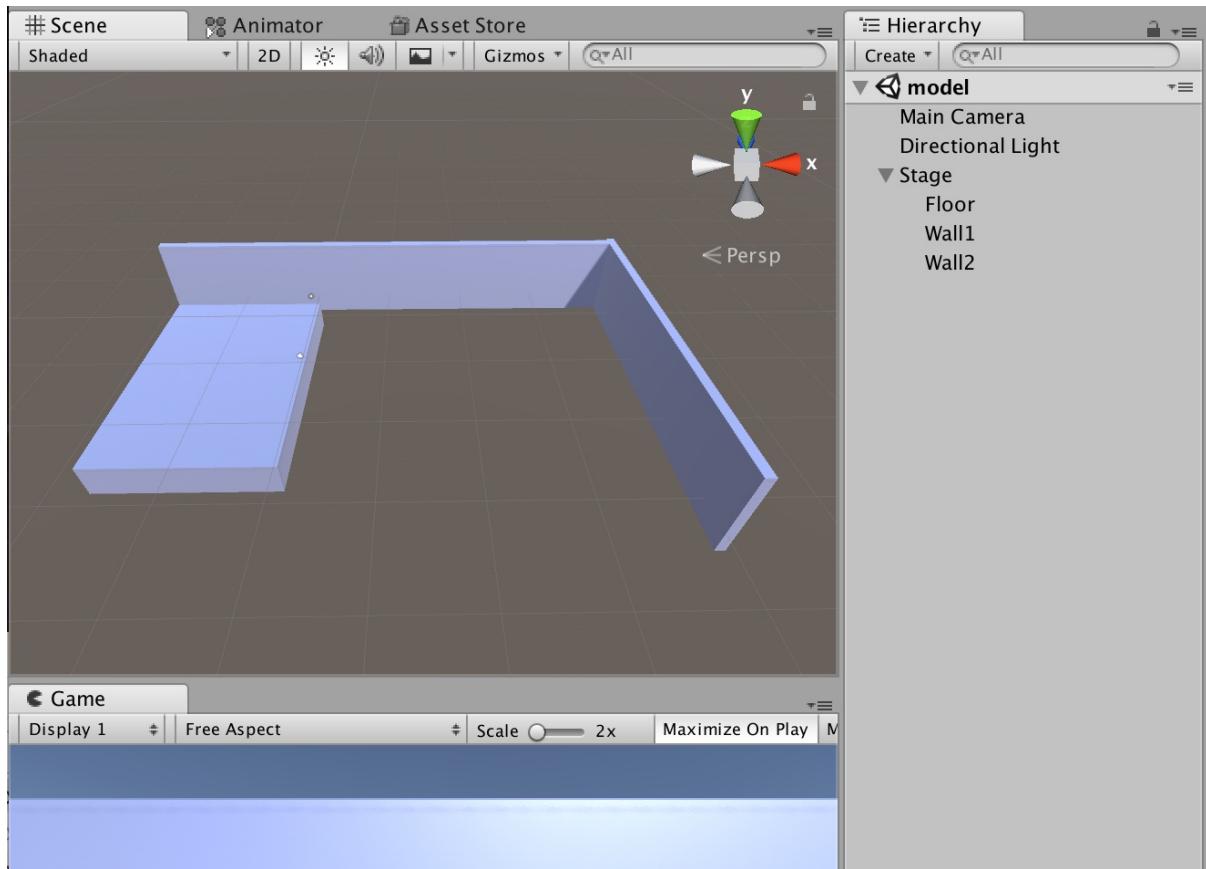
マテリアル matIce

Albedo : R:196,G:200,B:255,A:255

Metallic : 0.4

Smoothness : 0.6

このマテリアルをFloor、Wall1、Wall2にそれぞれドラッグ＆ドロップで適用したらステージは完成です。



## ペンギンモデルの作成

続いてペンギンのモデルを作成していきます。ペンギンのマテリアルは白と黒の2種類、オブジェクトはすべてSphereを変形して組み合わせることで作成します。

先ほどと同様に新規マテリアルを作成してください。名前はそれぞれmatWhite、matBlackとします。

**matWhite**

Albedo : R:255,G:255,B:255,A:255

Metallic : 0.2

Smoothness : 0.2

**matBlack**

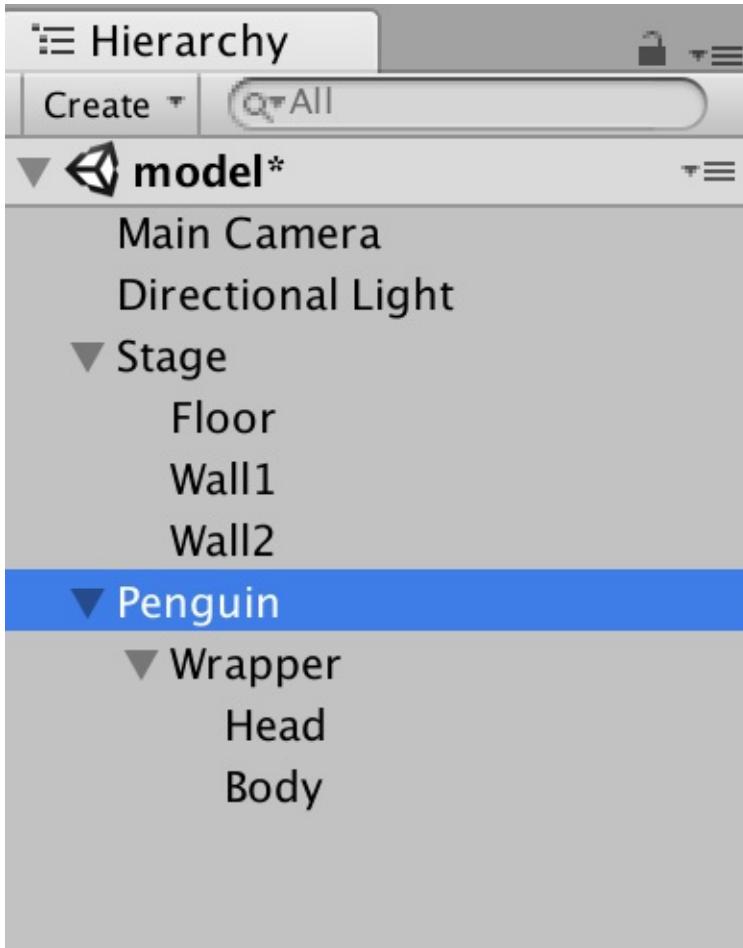
Albedo : R:28,G:28,B:28,A:255

Metallic : 0.9

Smoothness : 0.7

次にHierarchyで何も選択されていないことを確認し、右クリックメニュー→Create Emptyで空のオブジェクトを作成し、名前を「Penguin」と変更します。

今度はPenguinを選択して右クリックメニュー→Create Emptyで空のオブジェクトを作成し、名前を「Wrapper」と変更します。さらに、Wrapperを選択して空オブジェクトを2個作成し、名前をそれぞれ「Head」「Body」とします。



このような構造にした理由を説明します。スクリプトを使ってPenguinをmainシーンに配置したときに、自分の意図した向きや大きさとは違っていたりするようなことがあります。その場合、配置してからPenguinのRotationやScaleを変更することも可能ですが、以前にも少し述べたように親のScaleは1であると扱いやすく、また親のRotationも0である方が何かと便利だったりします。そのためWrapperというノードを一段かませて、配置するサイズや向きのデフォルト値をWrapperに設定する、というテクニックがよく使われます。

また、今回作るペンギンモデルは、光の方向に追従して回転する顔のパートと、何も動かさない体のパートから構成されます。そのためそれをグループ化するためにHead、Bodyというノードを用意して、その下に実際のオブジェクトをぶら下げていきます。

それでは実際にオブジェクトを配置していきます。まずはHeadを選択し、右クリックメニュー→3D Object→Sphereとして、Sphereを生成します。同様にBodyを選択してもうひとつSphereを生成します。それぞれのPositionとScaleなどを以下のように設定します。

```
Penguin
Position : 0, 0, 0
Rotation : 0, 180, 0
Scale : 1, 1, 1
```

```
Wrapper
Position : 0, 0, 0
Scale : 1, 1, 1
```

```
Head
Position : 0, 1.04, 0
Scale : 1, 1, 1
```

Headの下のSphere

Position : 0, 0, 0

Scale : 1, 1, 1

Body

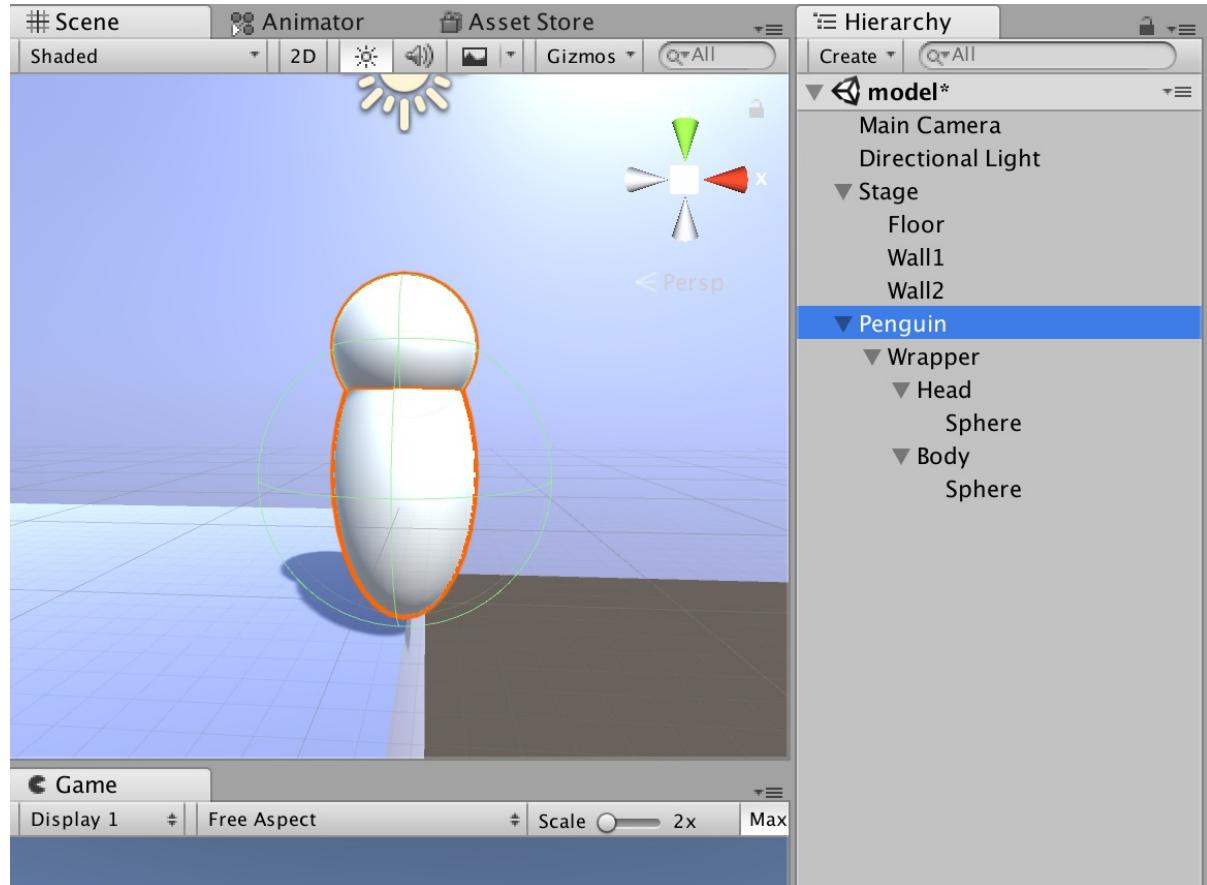
Position : 0, 0, 0

Scale : 1, 1, 1

Bodyの下のSphere

Position : 0, 0.15, 0

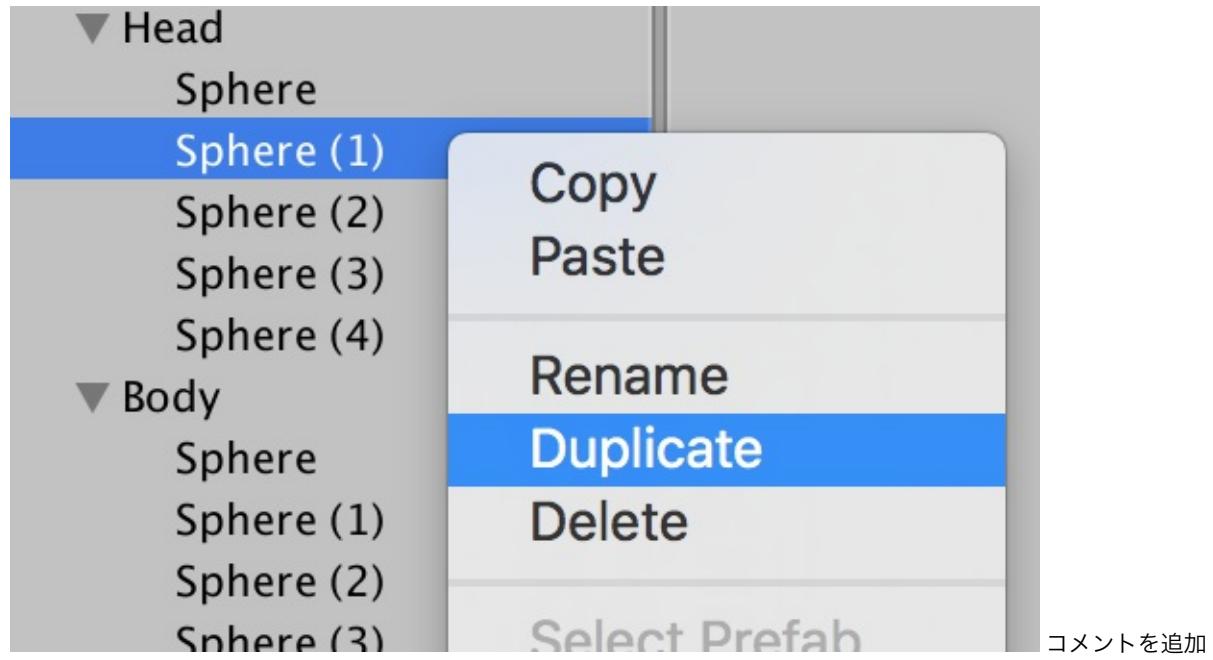
Scale : 1, 2, 1



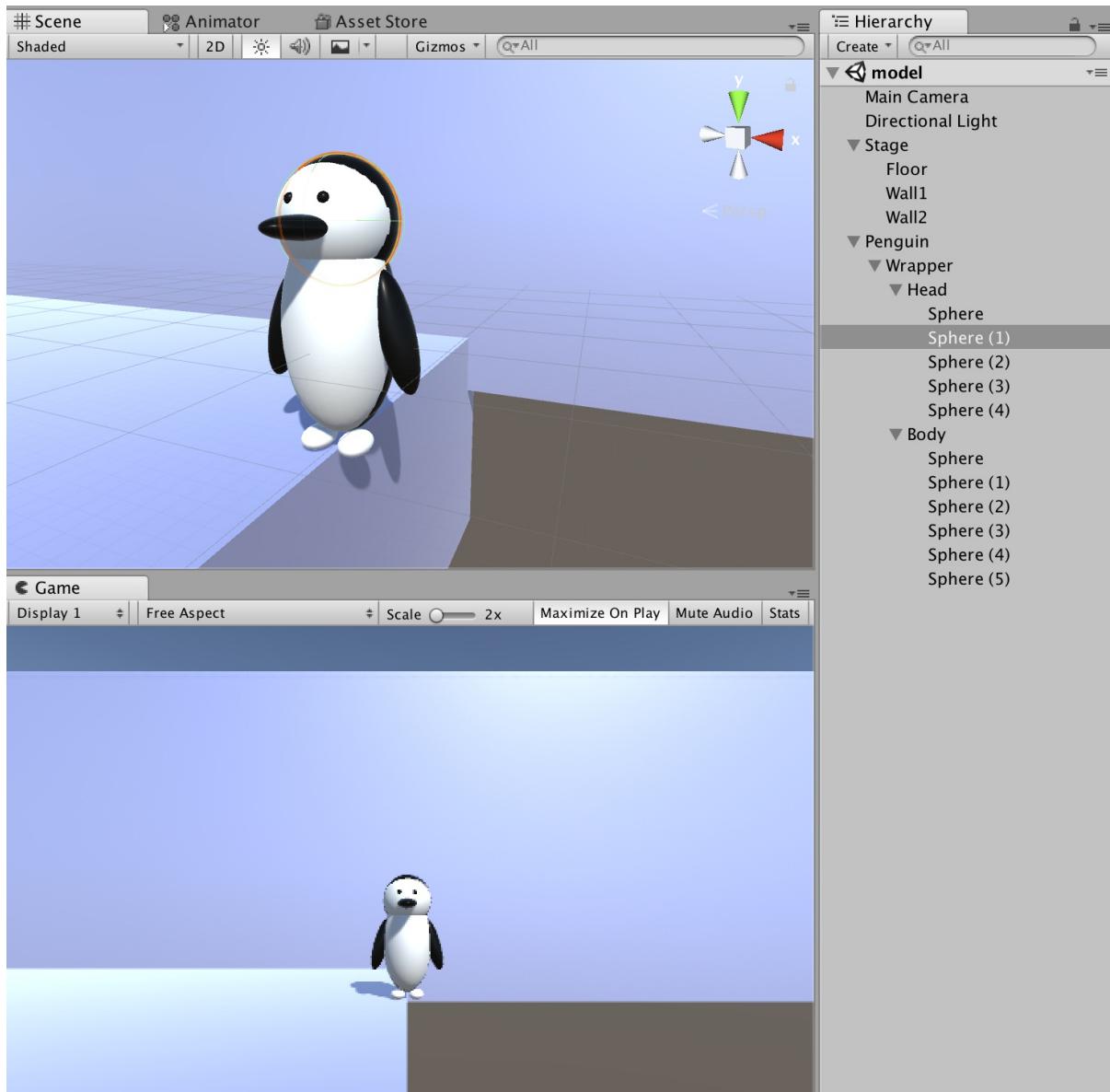
同様にSphereを生成し、位置とスケールを調整して目やくちばし、羽などを作っていきます。顔のパーツはHeadの下に、体のパーツはBodyの下にぶら下げます。

生成したSphereにはmatWhiteまたはmatBlackをドラッグ&ドロップで適用していきます。頭や胴体など白黒が途中で切り替わるようなパーツは同じ大きさか少しだけサイズの違う白と黒のパーツをわずかにずらして配置することができます。

目や足など同じ大きさのパーツを作るときは、オブジェクトを選択して右クリックメニュー→Duplicateとすると複製できるので効率的です。



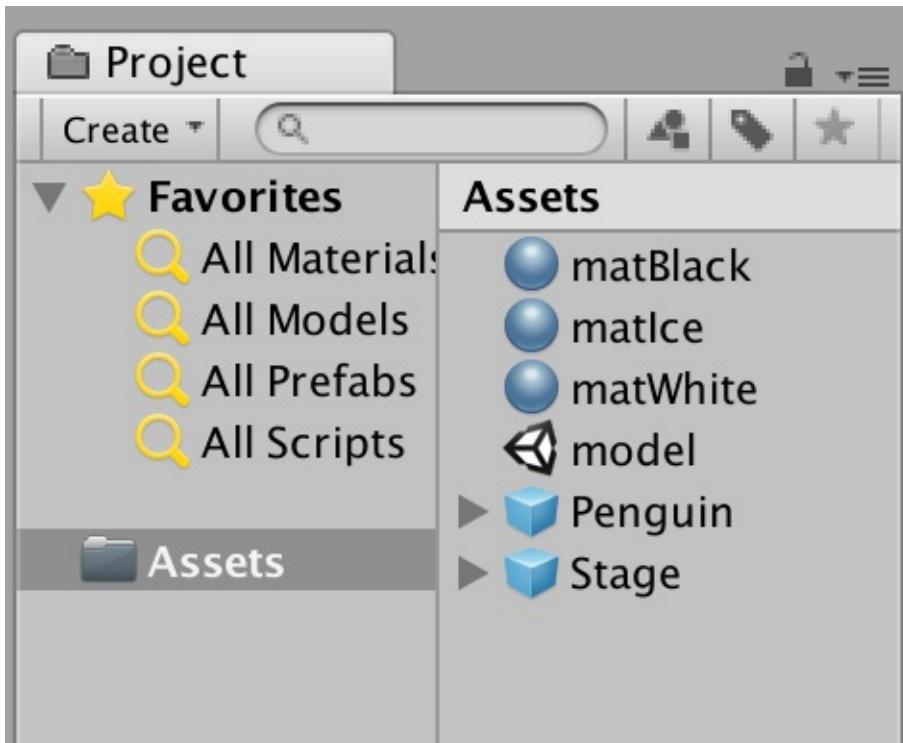
こんな感じになればペンギンモデルの完成です。



## モデルのプレハブ化

作成したモデルを別のシーンから使うためにはプレハブと呼ばれるアセットにする必要があります。方法は簡単で、HierarchyのオブジェクトをProjectウィンドウのAssetsにドラッグ&ドロップするだけです。

StageとPenguinをプレハブ化してみましょう。



## メインシーン作成

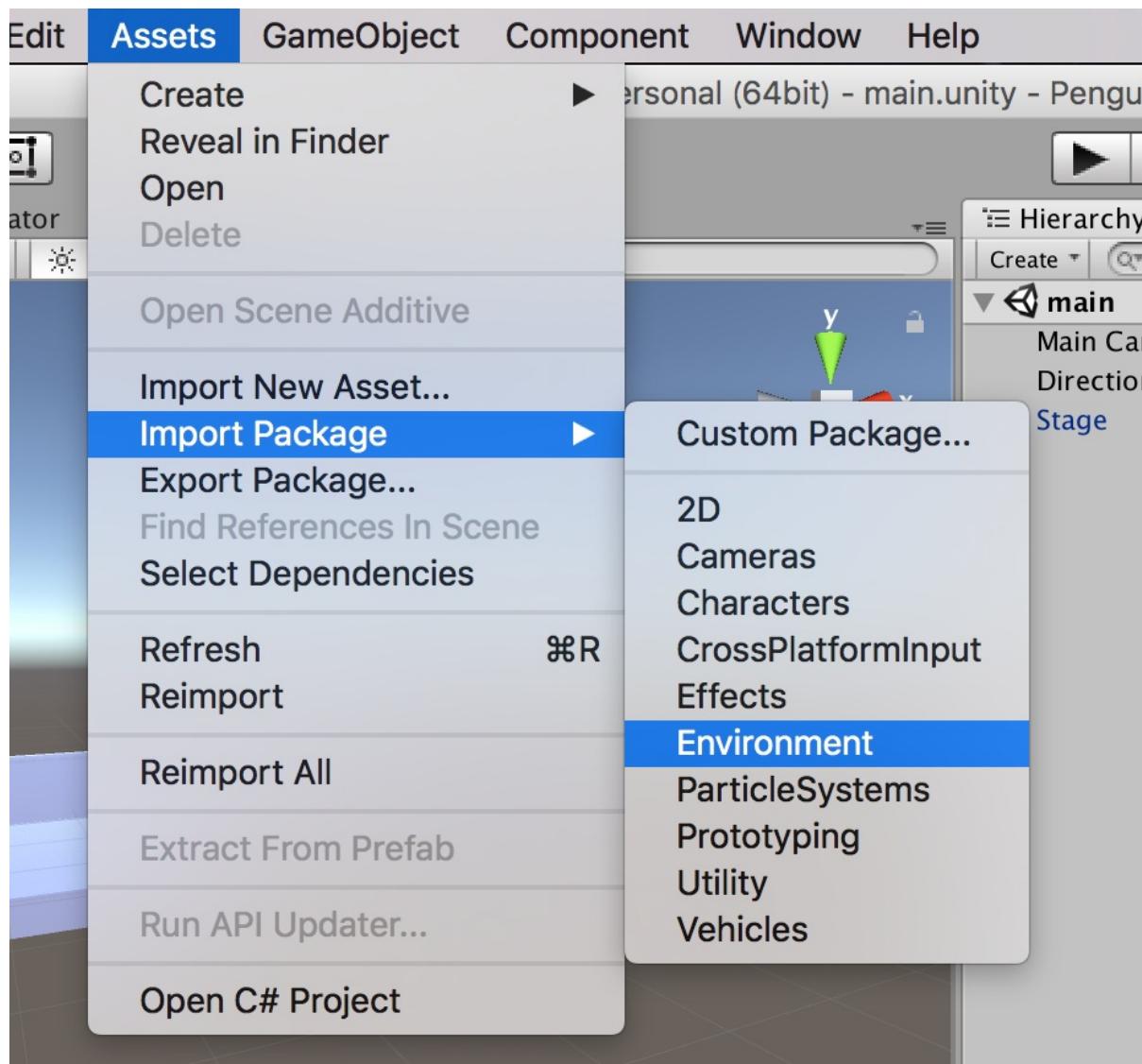
ここまでではモデルデザイン用のシーンで作業していましたが、いよいよメインシーンを作成します。一旦modelシーンをセーブして、FileメニューからNew Sceneを選択してください。何もないシーンが表示されるのでこれを「main」の名前でセーブします。

### ステージの配置

メインシーンにステージを配置します。先ほど作ったStageプレハブがAssetsにあるのでそれをHierarchyにドラッグ&ドロップします。StageのInspectorを見てPositionが0, 0, 0になっていなければ全てゼロにしてください。

### 水面の配置

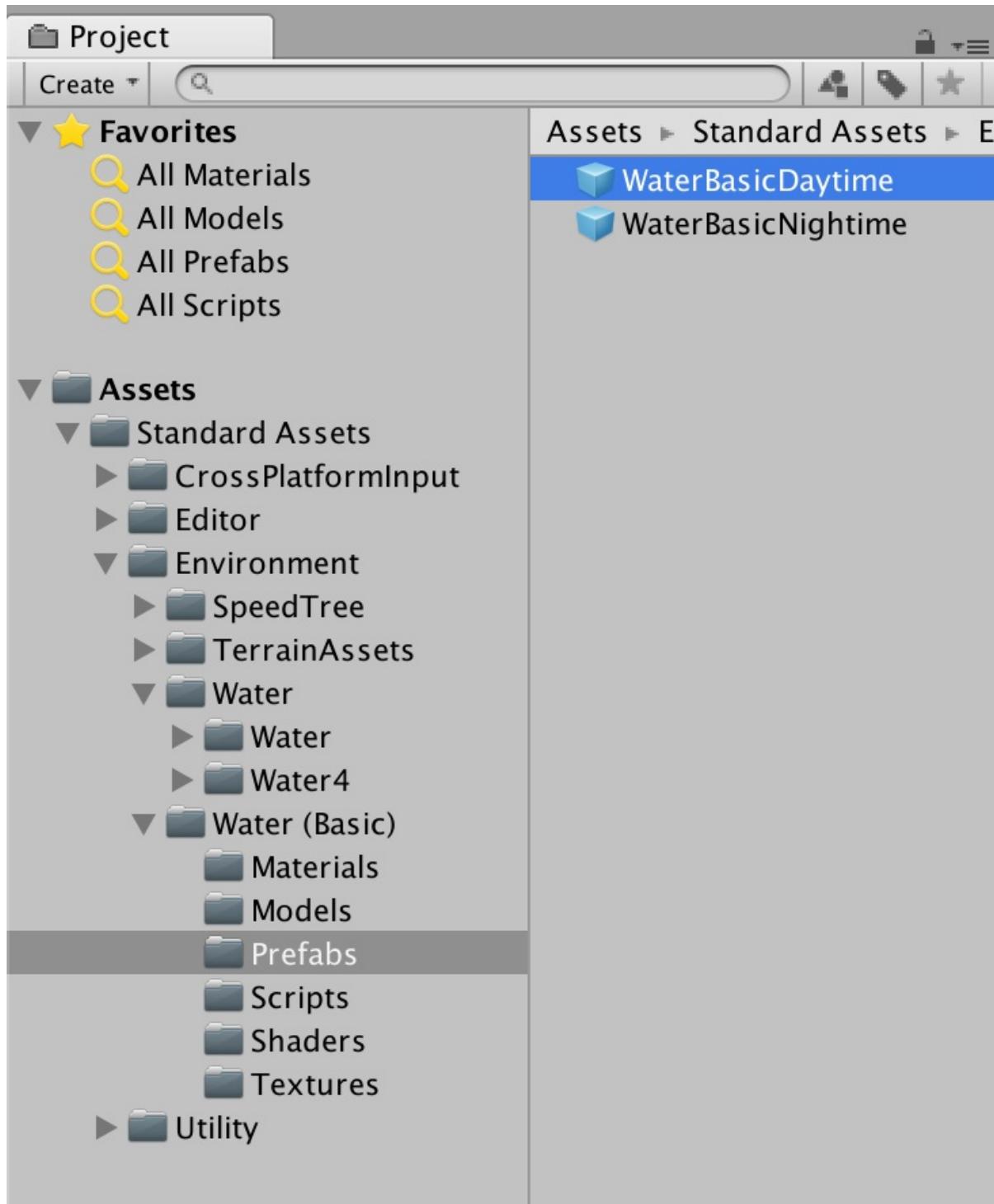
次はステージに水を張りましょう。Unityは標準アセットとしてリアルな水面が用意されています。Assetsメニュー→Import Package→Environmentを選択してください。インポート対象の選択ウィンドウが表示されたら、全部の項目にチェックが入っているそのままの状態でImportボタンを押します。

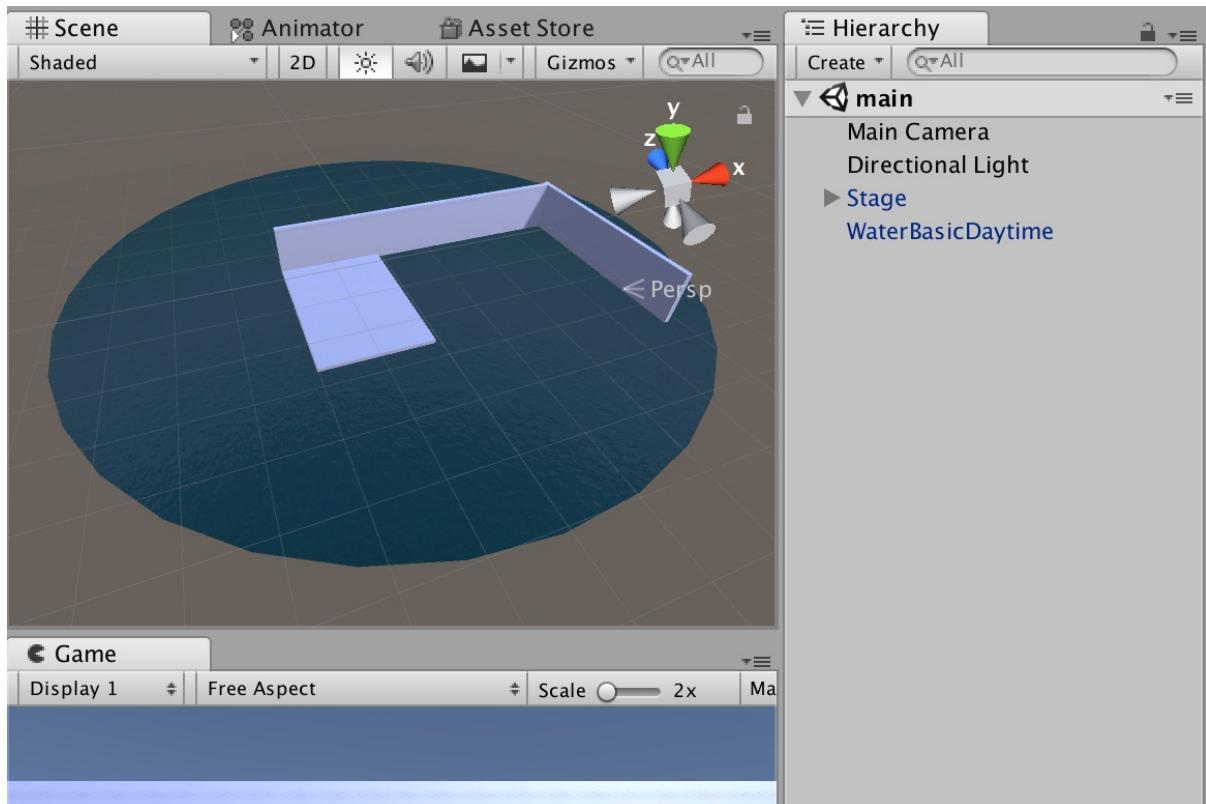


標準アセットには水の表現用にいくつかのアセットが用意されています。今回は一番シンプルな WaterBasicDaytimeを使用します。Assets/Standard Assets/Environment/Water(Basic)/Prefabs の下にある WaterBasicDaytimeをHierarchyにドラッグ&ドロップして、以下のようにパラメータを調整してください。

Position : 0, -1.67, 0

Scale : 60, 1, 60





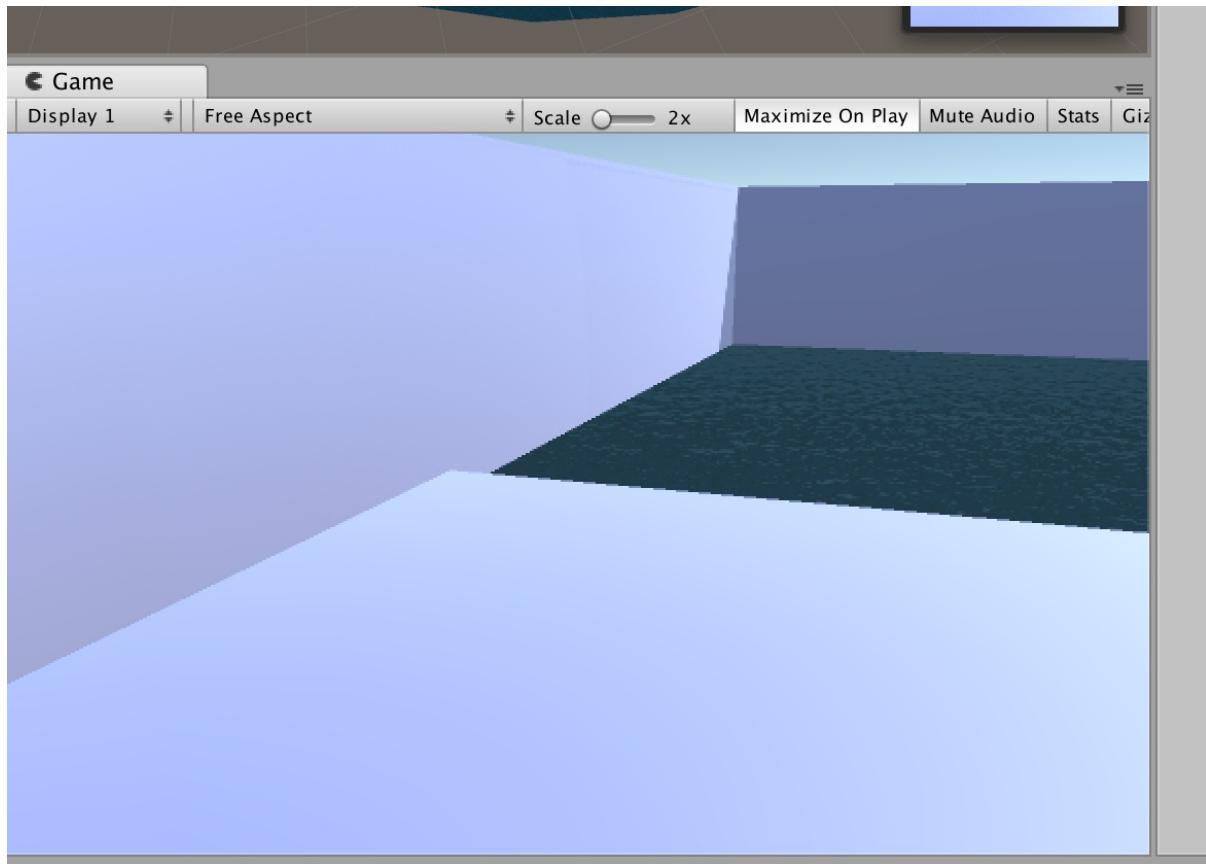
## 構図の調整

メインカメラの位置と角度を調整して構図を決めましょう。左側の方からステージを見るように以下のようにMain Cameraのパラメータを設定します。

Position : -24, 5.5, -8

Rotation : 15, 70, 0

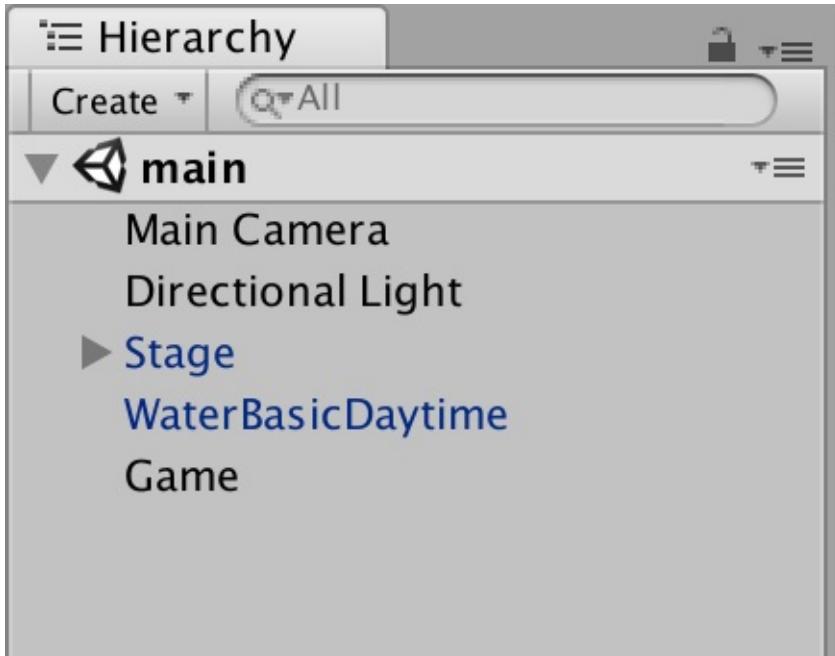
前回のチュートリアルでも書きましたが、こういった斜めからの構図の場合、RotationのZは必ず0にして水平を正しくとるようにしましょう。



## ペンギンの生成

次にペンギンをランダムな位置に生成するスクリプトを作ります。ペンギンを生成するスクリプトはどこにアタッチすると良いと思いますか？わりとこれは悩みどころだったりします。Main Cameraにアタッチすることも多いですが、数が増えてくるとごちゃごちゃしてきます。複数カメラを配置するようなシーンのような場合、ちょっと見つけにくくなったりすることもあります。

自分の場合、Gameという名前の空のオブジェクトをシーンにひとつ置いて、こういったゲームの環境を整えたり、ゲーム全体に関わる処理をするスクリプトを全部そこにアタッチするようにしています。



それでは、Gameオブジェクトを選択し、InspectorのAdd Component→New Scriptを選んで新規スクリプトを作成しましょう。スクリプト名はPenguinGeneratorとします。

PenguinGenerator.cs

```
using UnityEngine;

public class PenguinGenerator : MonoBehaviour {

    public GameObject penguinPrefab; // ペンギンのプレハブ

    void Start () {
        // ゲーム開始時に10羽生成する
        for (int i = 0; i < 10; i++)
        {
            float x = Random.Range(-14f, 20f); // 横方向の位置
            float y = 0; // 高さ方向の位置
            if (x > 0) // ペンギンが画面右側のプールにいる場合少し位置を下げる
            {
                y = -1;
            }
            float z = Random.Range(-10f, 2f); // 奥行き方向の位置
            float angle = Random.Range(-45f, 45f); // 体の向き

            // プレハブをもとにインスタンス化
            GameObject penguin = Instantiate(penguinPrefab, new Vector3(x, y, z), Quaternion.identity);

            // ランダムな向きになるよう角度をつける
            penguin.transform.eulerAngles = new Vector3(0, angle, 0);
        }
    }
}
```

public変数でペンギンのプレハブをInspectorから設定できるようにしています。AssetsにあるPenguinプレハブをInspectorのPenguin Prefabにドラッグ＆ドロップしてください。

このスクリプトでは、for文のループを10回まわしてゲーム開始時にペンギンを10羽生成しています。ペンギンの出現位置は変数xと変数zで指定していますが、これはRandom.Range関数を使って乱数を生成しているので毎回ランダムな位置に出現します。

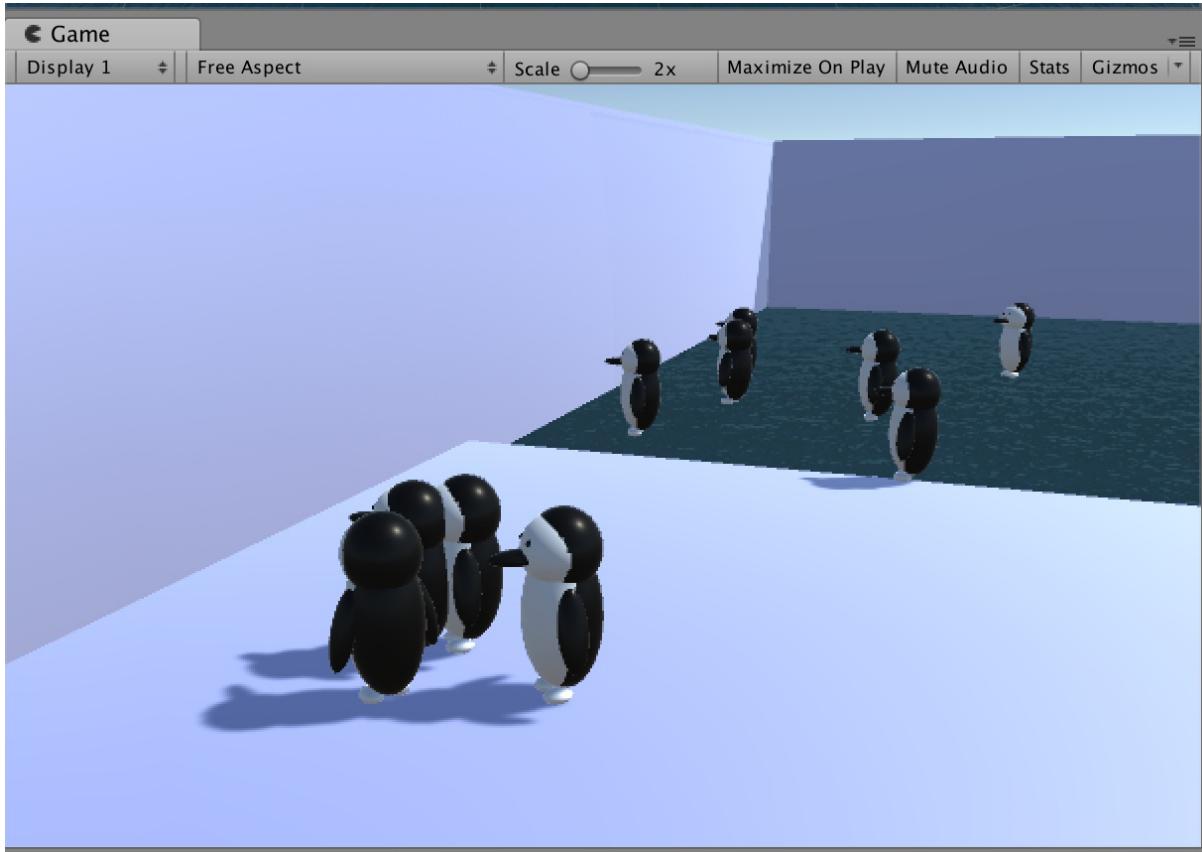
ここでちょっと型について解説します。C#言語では、数値の型によって書き方が異なります。10と書くとint、10.0はdouble、10fはfloatとなります。Unityでは、Position、Scale、Rotationなどの値はfloatで計算されるので、それらに設定する数字の後にはfをつけるのが正しい書き方です。通常はintの書き方で書いても自動的に変換されますが、doubleの書き方をするとエラーになります。スクリプト自体は一見正しいように見えるので注意してください。最初のうちはdoubleの型を使うことはほとんど無いので、「小数点を含む数値には必ず末尾にfをつける」と覚えておきましょう。

Random.Range関数を使うときは型を意識する必要があります。なぜならRandom.Range(intの数値, intの数値)と書いたときとRandom.Range(floatの数値, floatの数値)と書いたときでは挙動が異なるからです。たとえばRandom.Range(0, 5)の場合、0、1、2、3、4のいずれかの整数が返ります。しかしRandom.Range(0f, 5f)の場合、0.1fや2.71f、4.89f……といった無数にある実数の値がひとつ返ります。つまり、実数の乱数がほしい場合は、明示的に引数にfをつけて実数であることを示す必要があるということです。

```
// プレハブをもとにインスタンス化
GameObject penguin = Instantiate(penguinPrefab, new Vector3(x, 0, z), Quaternion.identity);
```

ここでプレハブをもとにペンギンの実体を生成しています。プレハブはあくまでも設計図のようなもので、それをもとにシーンに実体を生成することをインスタンス化と呼びます。ひとつの設計図から大量の製品を作り出すような様子をイメージしてください。

スタートボタンで実行すると以下のようにペンギンが生成されます。乱数で生成しているので出現位置は毎回異なります。



## ターゲットオブジェクトの配置

ペンギンが目で追いかける光を配置します。Hierarchyで何も選択されていないことを確認し、右クリックメニュー→Light→Point Lightで空のオブジェクトを作成し、名前を「Target」と変更します。

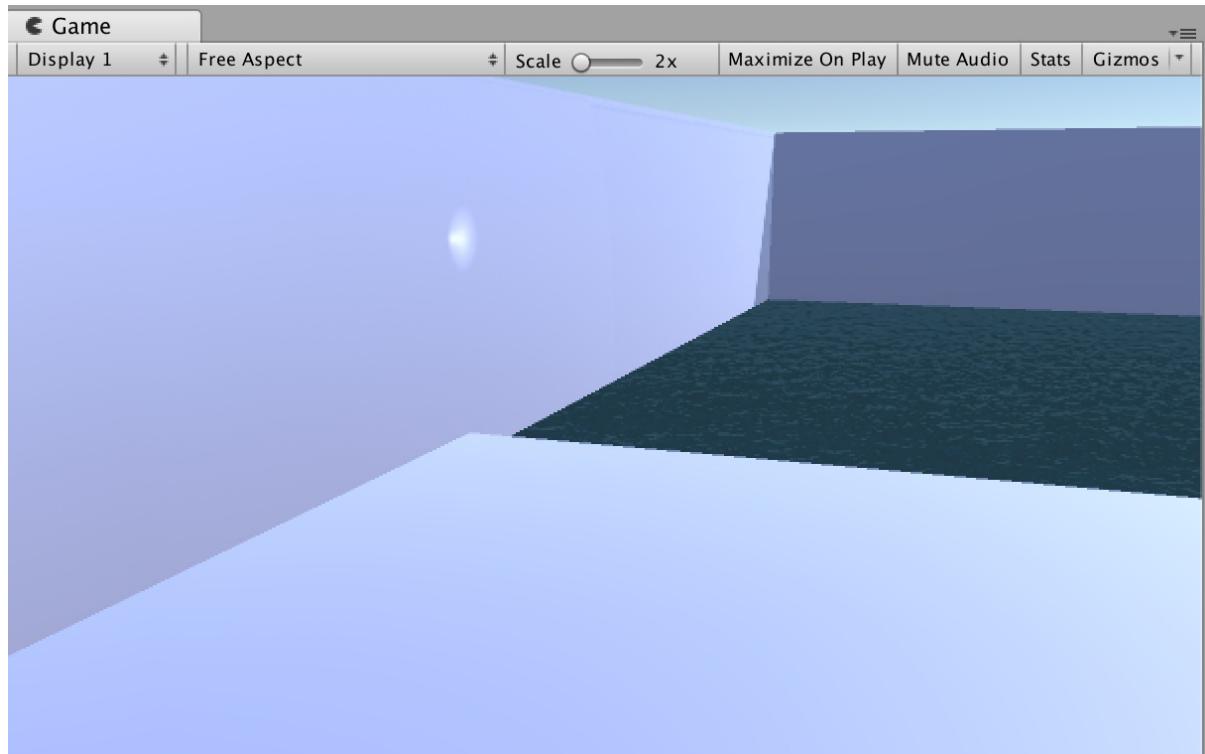
Inspectorからパラメータを設定。壁にギリギリのところに配置します。

Position : 0, 4, 4

小さく強い光にします。

Range : 1

Intensity : 3



## ペンギンのスクリプト追加

ペンギンにスクリプトを追加して光の方向を向くようにします。一旦、mainシーンをセーブした後に、modelシーンを開いてください。

HierarchyのペンギンモデルのHeadを選択してAdd Component→New Scriptとしてスクリプトを追加。名前はHeadRotatorとします。

HeadRotator.cs

```
using UnityEngine;

public class HeadRotator : MonoBehaviour {
    GameObject target;

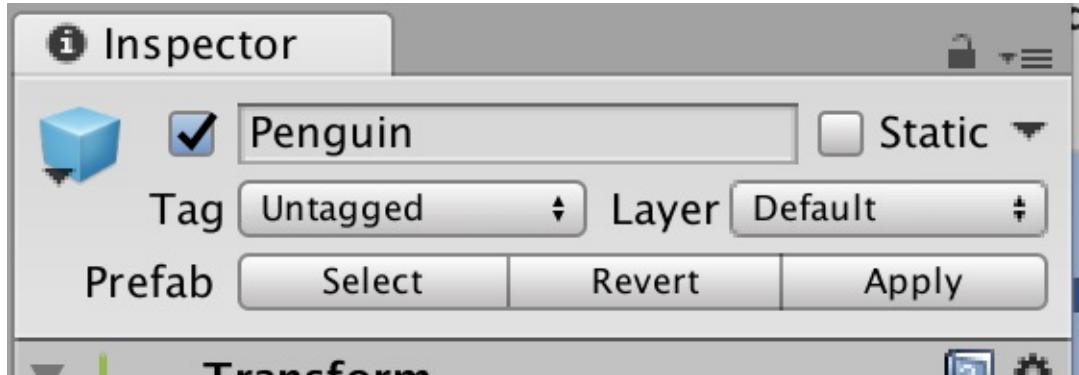
    void Start () {
        target = GameObject.Find("/Target");
    }

    void Update () {
        transform.LookAt(target.transform);
    }
}
```

Start関数では、HierarchyのTargetというオブジェクトを探してtargetという変数に割り当てています。Update関数では毎フレームtargetの位置を取得してLookAt関数でその方向を向くようにしています。

さて、シーン上のPenguinオブジェクトにスクリプトを追加しましたが、それだけではAssetsのPenguinプレハブにはまだ追加されておらず、相違がある状態になっています。mainシーンで参照しているのはAssetsのプレハブの方なので、こちらにも変更を反映する必要があります。

HierarchyのPenguinを選択してInspectorの右上3段目にあるApplyボタンを押します。これで変更がプレハブの方にも反映されて他のシーンからも参照できるようになります。



ここまでできたらmodelシーンをセーブしてmainシーンに戻ってください。

mainシーンでスタートボタンを押すと、出現したペンギンがそれぞれ光の方向を向いていたら成功です。今のところ光は動かないで、ペンギンも動かずに一点を見つめているだけです。

## 光を動かすスクリプト

最後にスクリプトを追加して光を動かします。何種類か異なるアルゴリズムのスクリプトを書いて動きの違いを見てみましょう。

### シンプルな往復

まずはシンプルな往復運動で実装してみます。HierarchyでTargetを選択してAdd Component→New Scriptでスクリプトを追加します。スクリプト名はLightMoverとします。

LightMover.cs

```
using UnityEngine;

public class LightMover : MonoBehaviour
{
    bool flag = true;
    float x = 0;

    void Update()
    {
        if (flag)
            x += Time.deltaTime * 30;
        else
            x -= Time.deltaTime * 30;

        if (Mathf.Abs(x) > 12)
            flag = !flag;

        transform.position = new Vector3(
            x,
            transform.position.y,
            transform.position.z
        );
    }
}
```

このスクリプトでは、flagで移動方向を制御しています。flagがtrueのときは右へ移動、falseのときは左へ移動します。移動速度はTime.deltaTime \* 30、位置xの絶対値をとって位置が12以上あるいは-12以下になったらflagを反転させて移動方向を逆向きにしています。 往復運動は一応できましたが、移動方向の変化が急激すぎてペンギンが首を痛めそうです。

## スムーズな往復

次はペンギンの首の健康に考慮して移動方向の変化がスムーズになるような往復運動を考えてみます。こういった現実にあるようななめらかな動きにすることをイージングと言い、さまざまな手法が考案されています。今回はシンプルな三角関数で実装することにします。LightMover.csを以下のように書きかえて実行してみてください。

LightMover.cs

```
using UnityEngine;

public class LightMover : MonoBehaviour {

    float angle = 0;

    void Update () {
        angle += Time.deltaTime * 2;

        transform.position = new Vector3(
            Mathf.Sin(angle) * 12,
            transform.position.y,
            transform.position.z
        );
    }
}
```

Sin関数を使うと回転運動を真横から見た動きになります。つまり動き自体は往復ですが、往復の端に近づくほど速度が遅くなるためスムーズな動きになりました。

## 橿円形の移動

Sin関数の横方向の往復に加えて、同じタイミングで縦方向の動きも加えると橿円形の運動になります。

LightMover.cs

```
using UnityEngine;

public class LightMover : MonoBehaviour
{
    float angle = 0;

    void Update()
    {
        angle += Time.deltaTime * 2;

        transform.position = new Vector3(
            Mathf.Sin(angle) * 12,
            Mathf.Cos(angle) * 2 + 4,
            transform.position.z
        );
    }
}
```

x方向はSin関数を使い、y方向はCos関数を使うことで、xがゼロのときはyが最大、xが最大のときはyがゼロになる動きをします、つまりこれは円の運動です。x方向の移動距離を最大12、y方向の移動距離を最大2とすることで楕円にしています。y方向は往復の中心位置を4の高さにして壁の少し上方で回転するようにしました。

## 8の字の移動

前項のスクリプトを少し変えるだけで8の字の移動になります。

LightMover.cs

```
using UnityEngine;

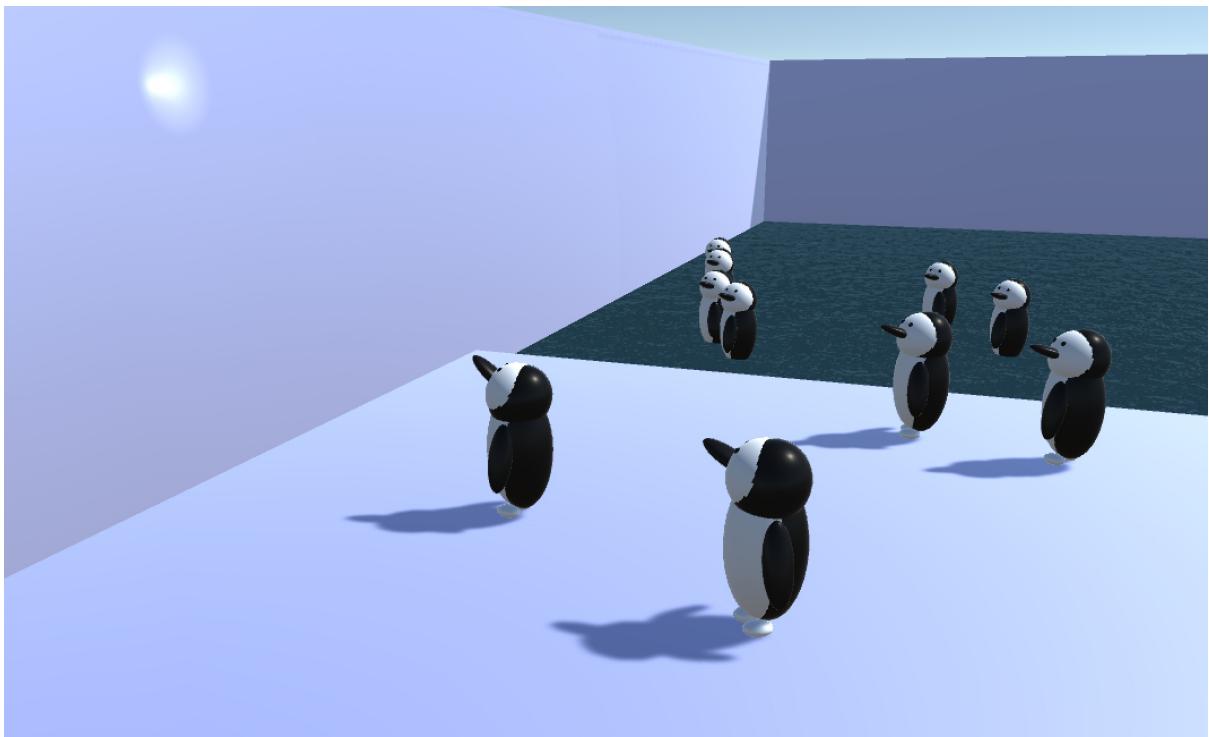
public class LightMover : MonoBehaviour
{
    float angle = 0;

    void Update()
    {
        angle += Time.deltaTime * 2;

        transform.position = new Vector3(
            Mathf.Sin(angle) * 12,
            Mathf.Sin(angle * 2) * 2 + 4,
            transform.position.z
        );
    }
}
```

違いはCos(angle)をSin(angle \* 2)にしただけです。y方向の往復を、x方向の往復の2倍のタイミングでおこなうことで、8の字の動きをするようになりました。`angle`にかける値や、Sin、Cosどちらを使うかによってこれ以外にも複雑な動きをさせることができます。ぜひ試してみてください。

これでチュートリアル「ペンギン」は終了です。短いプログラムでも工夫次第でゲーム的な面白い動きを演出できることが理解できたかと思います。





# チュートリアル3 「TrenchFighter」

## このチュートリアルについて

今回のチュートリアルでは、敵機をよけながら猛スピードで狭い隙間を進んでいくゲーム風の画面を作っていくします。項目としては以下のような内容を学んでいきます。

- プロシージャルなステージ生成
- アセットの再利用方法
- 簡単なサウンドの使い方
- 物理演算を使わないあたり判定
- 入力の処理
- ポストプロセッシングで絵作りその2



## メインシーンの作成

このテーマも、前回同様にmainとmodelのふたつのシーンを用意します。敵機のような常に動き回るものを感じ確認しながら調整するためにmodelシーンを利用します。

Unityの起動画面、あるいはFileメニュー→New Projectから新規プロジェクトを作成します。プロジェクト名は「TrenchFighter」にします。 今回のステージはmainシーンに直接作っていきます。Hierarchyで何も選択していない状態で、右クリックメニューから平面(Plane)1個と直方体(Cube)2個を作成してください。

### Plane

Position : 0, 0, 0

Scale : 50, 1, 50

### Cube

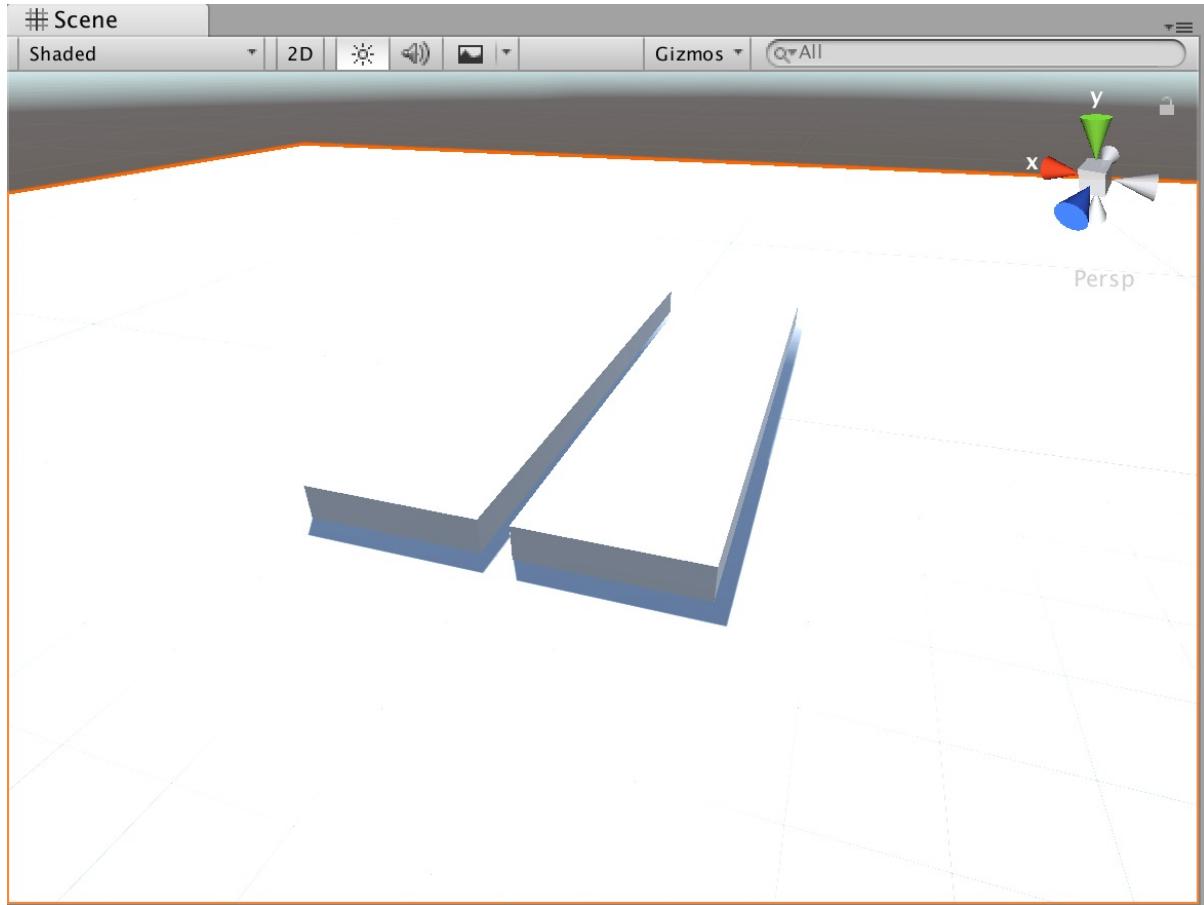
Position : -11.7, 0, 0

Scale : 20, 8, 80

**Cube**

Position : 11.7, 0, 0

Scale : 20, 8, 80



ここまでできたらCtrl+S(Macは⌘+S)でシーンを一旦セーブします。シーン名は「main」とします。

次はマテリアルです。これまで同様、ProjectウィンドウのAssetsの空き領域を右クリック、メニューから→Create→Materialの操作ですね。名前はmatStageとします。建築物の外壁のようなイメージですが、後からさまざまな視覚効果をかけるためここでは白に近いグレーにしておきます。

**マテリアル matStage**

Albedo : R:250,G:250,B:250,A:255

Metallic : 0.1

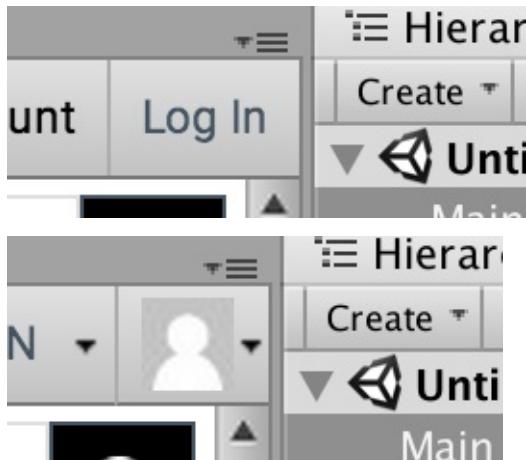
Smoothness : 0.14

このマテリアルを先ほどのPlaneとCubeにそれぞれドラッグ&ドロップで適用します。

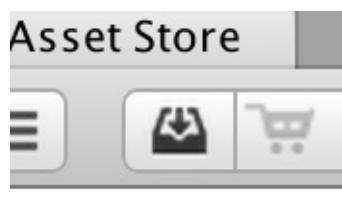
**アセットのインポート**

背景は星空にしたいので、チュートリアル1で使った「Stellar Sky」をここでも利用します。ダウンロードしたオリジナルのアセットは専用のフォルダに保存されているので、今回は再度ダウンロードする必要はありません。以下の手順でインポートしてください。

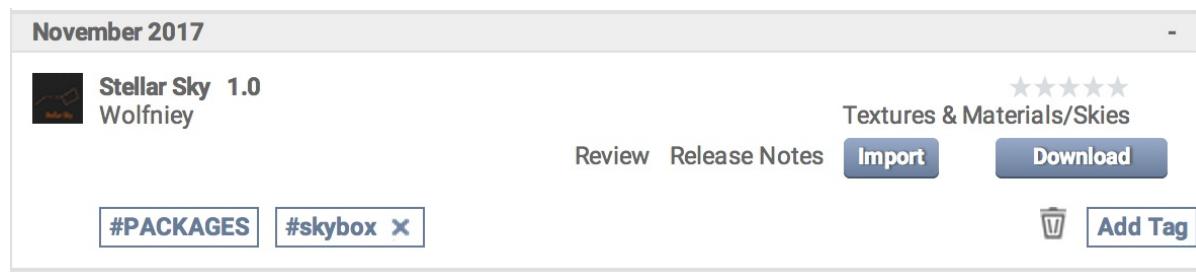
- Windowメニュー→Asset Storeを選択してAsset Storeウィンドウを開く
- Asset Storeウィンドウの右上が「Log In」となっていたらクリックしてログイン
- 右上が人型のアイコンであればログインは不要です



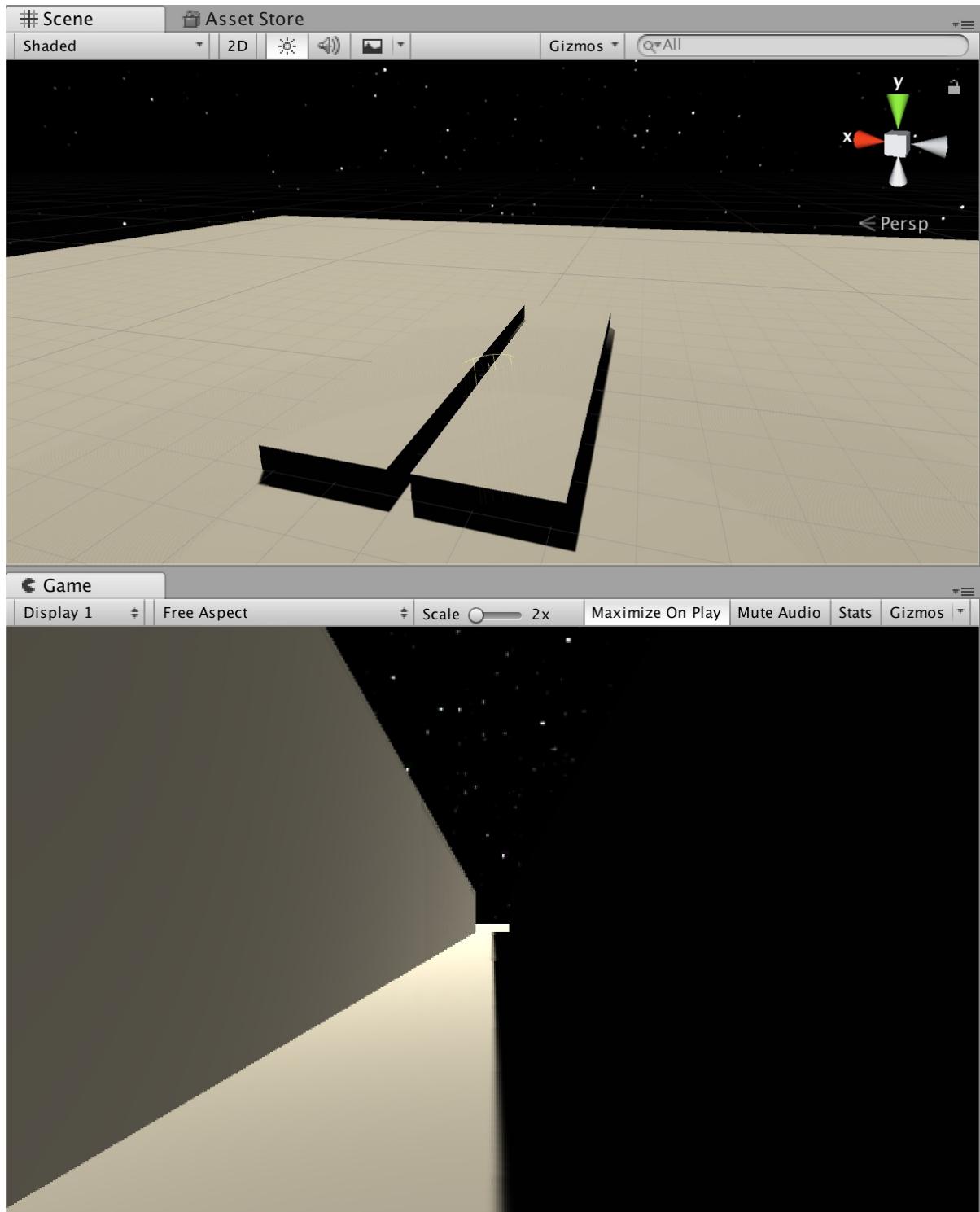
- 次に下向き矢印のついた引き出しのようなアイコンのダウンロードマネージャーを選択します



ここには、過去に入手したアセットが一覧表示されます。Stellar Skyが見つかったら「Import」ボタンを押してプロジェクトにインポートします。インポート対象は例によってデフォルトの全選択のままで良いです。



Skyboxの設定も以前やった通りです。Windowメニュー→Lighting→Settingsでウィンドウを開き、一番上のSkybox MaterialにインポートしたSS\_2048を指定。次にMain CameraのInspectorでCameraコンポーネントのClear FlagsをSkyboxに変更。



## ライトの設定

ここからライトを設定していきます。上のスクリーンショットでは直接光が強く影が濃く出すぎているので、直接光を弱めて環境光を強調します。また色味も環境光で調整します。

HierarchyのDirectional Lightを選択して、InspectorからLightの以下のパラメータを変更します。

Intensity : 0.8

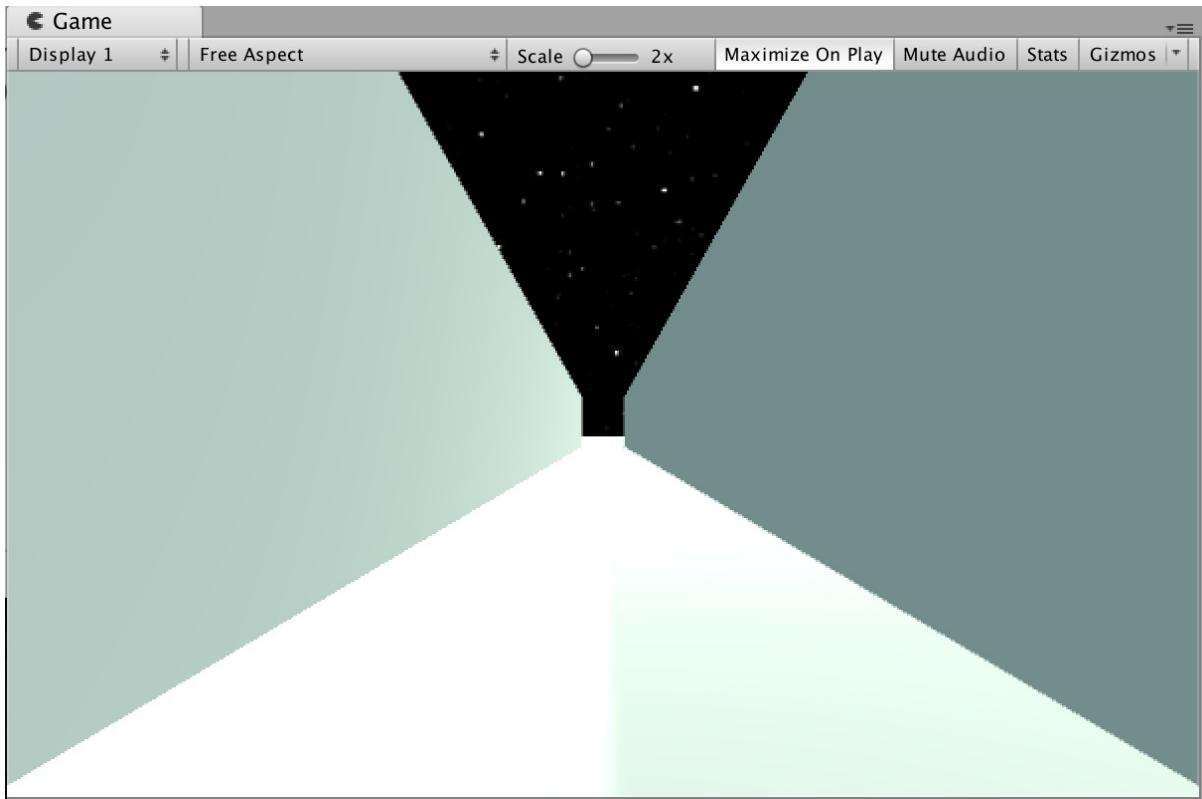
Realtime Shadows

Strength : 0.2

次に環境光です。Windowメニュー→Lighting→Settingsで再度ウィンドウを開き、Environment Lightingを以下のように設定します。

Source : Color

Ambient Color : R:0.65 G:0.8 B:0.8



これでmainシーンはほぼ完成です。Ctrl+S(Macは⌘+S)でシーンをセーブしましょう。

## モデルシーンの作成

次にmodelシーンでモデルを作成していきます。今回はステージに現れるでこぼこした構造物と敵機をモデルとして作成します。

先ほどmainシーンのライトを設定しました。モデル調整時の色味もmainシーンと同じようにした方がわかりやすいので設定をそのまま流用します。 mainシーンを開いた状態でFileメニュー→Save Scene Asでセーブします。シーン名は「model」とします。タイトルバーにシーン名model.unityが表示されていることを確認してください。



mainシーン用に配置したPlaneとCubeは不要なので削除します。Hierarchyで選択、右クリックメニューからDeleteです。

## 構造物モデルの作成

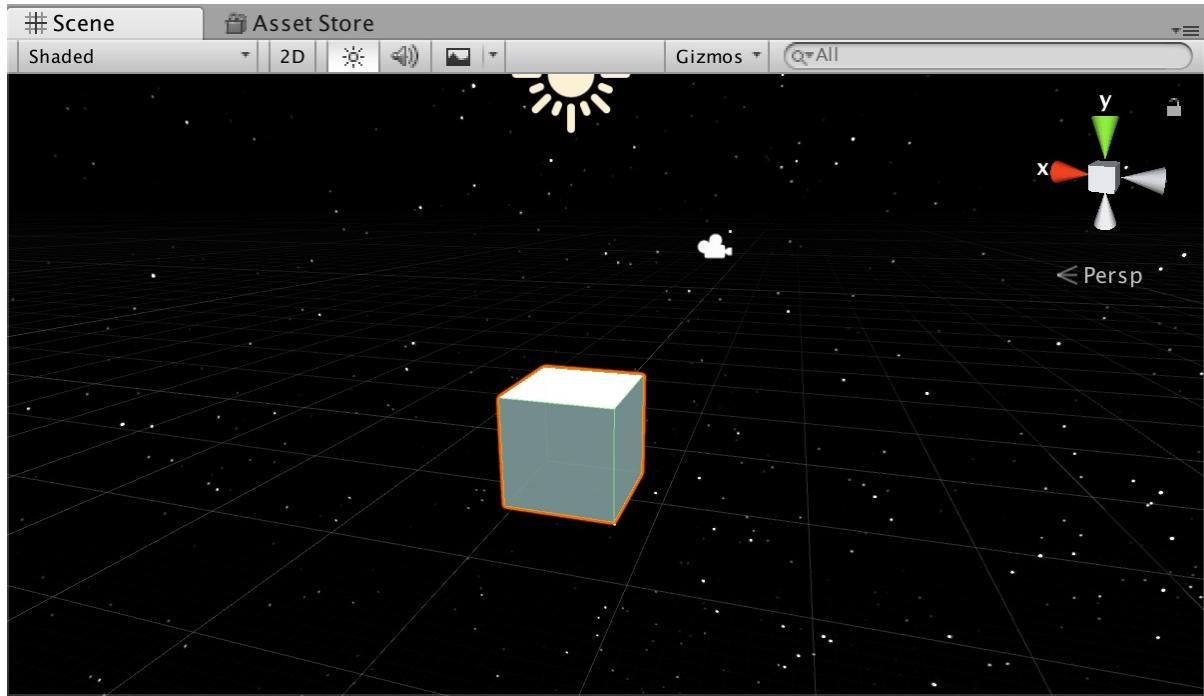
ステージに配置する構造物は非常にシンプルです。HierarchyにCubeをひとつ生成して、「Structure」と名前を付けます。設定値は以下の通りです。

Position : 0, 0, 0

Scale : 1, 1, 1

次にStructureに、先ほど作成したmatStageをドラッグ＆ドロップします。ここまでできたらStructureをプレハブ化するためにProjectウィンドウのAssetsにドラッグ＆ドロップします。

これで構造物のモデルは完成です。この立方体をランダムに生成することで複雑な壁面を表現します。今回作成するプログラムでは、自機が猛スピードで進んでいるように見えますが、実は自機もmainシーンの溝も動いておらず、ランダムに生成した構造物モデルが高速で前から後ろに移動することで自分が進んでいるように見せています。無限に続くステージをUnityのシーンで作成するのは困難なので、見える部分の構造物だけ生成して奥から手前に動かし続け、通り過ぎたらまた、はるか前方から現れるようにして長いステージを表現しています。



## 敵機モデルの作成

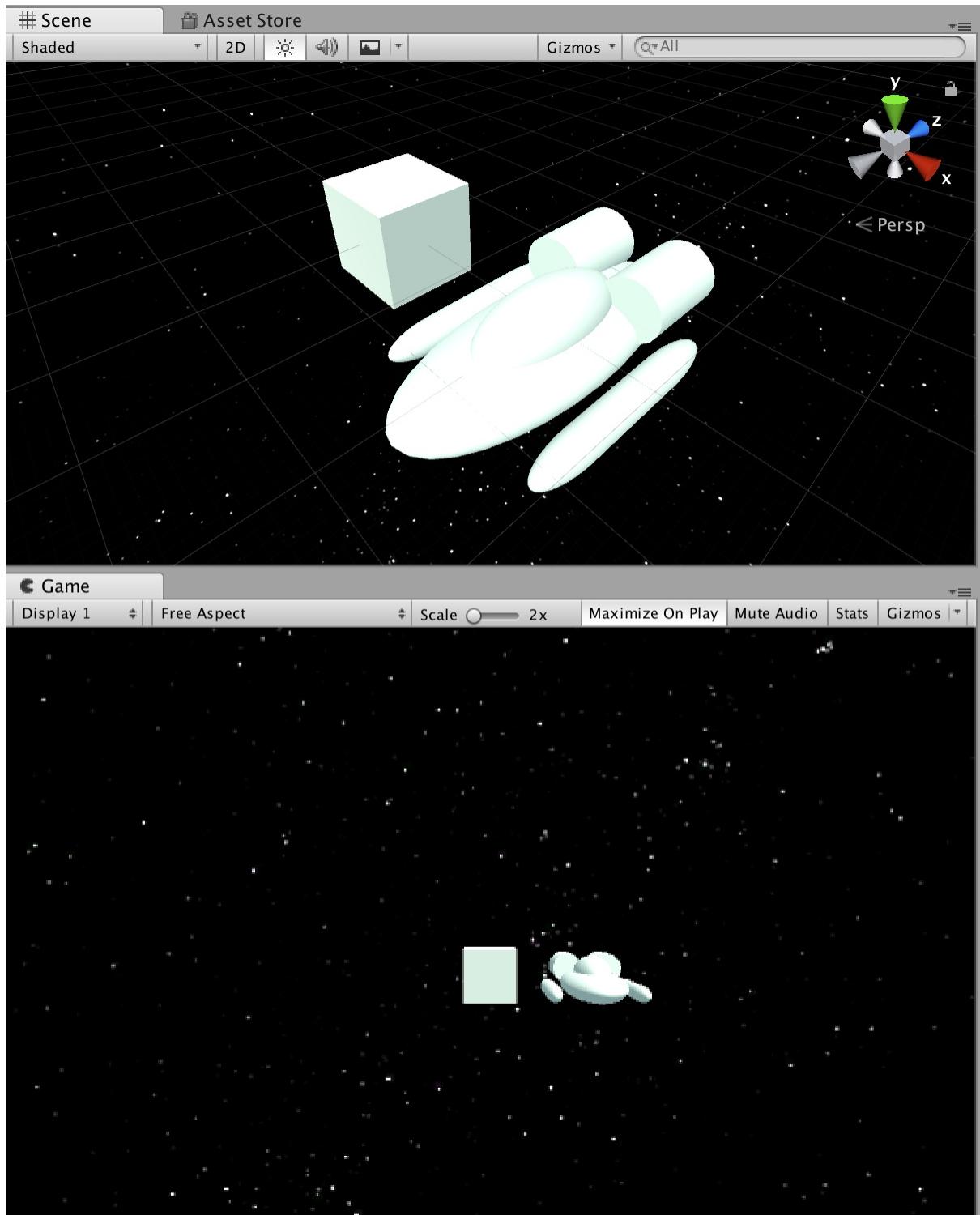
次に敵機モデルを作成します。何も選択されていない状態のHierarchyで右クリックからCreate Empty、名前を「Enemy」とします。パラメータは以下の通り。

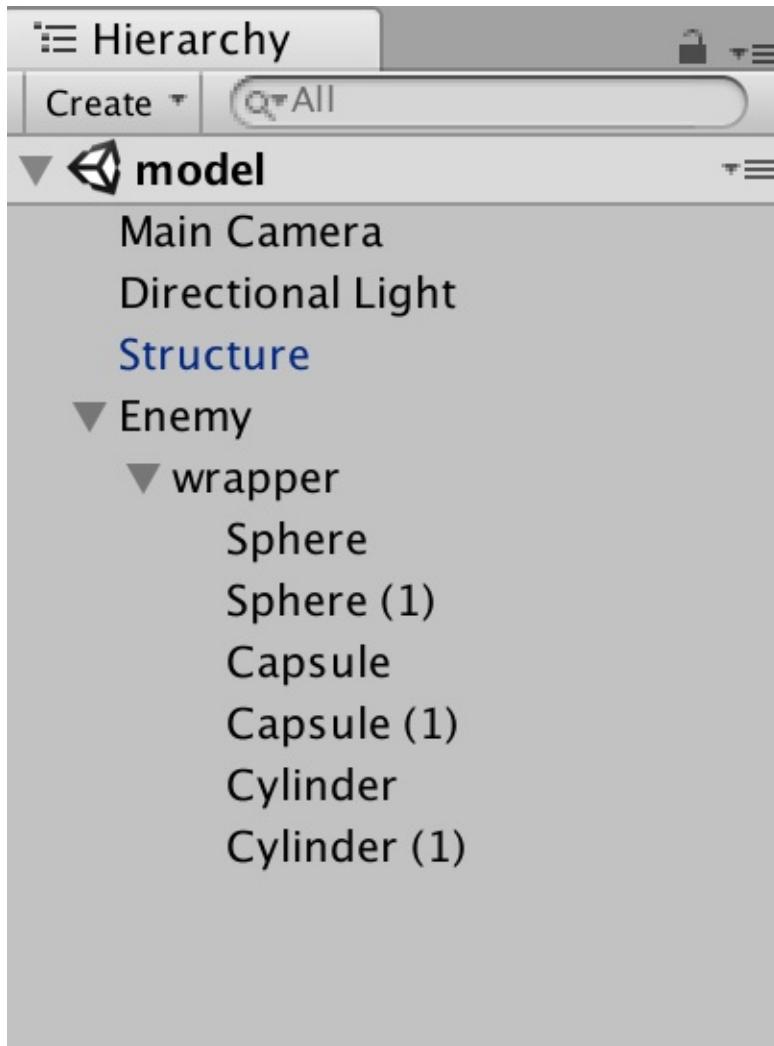
Position : 2, 0, 0

Scale : 1, 1, 1

Enemyの下にもう1階層Create Emptyして名前を「wrapper」とし、さらにその下にScaleで引き伸ばしたSphere、Capsule、Cylinderなどを左右対称に配置して宇宙船っぽい形にします。

このとき宇宙船の前方は、Z軸がマイナスの方向を向くようにします。Gameビューには宇宙船の正面が見えている状態になります。（後から方向が間違っていることに気付いたときや、サイズを変えたくなった場合は、wrapperのRotationやScaleを変更することで調整することができます）





次に敵機のマテリアルを作成します。敵機はステージよりも目立たせたいので、Emissionを有効にして少し発光するマテリアルにします。ProjectウィンドウのAssetsで右クリック→Create→Materialでマテリアルを作成してください。名前はmatEnemyにします。

Albedo : R:206 G:207 B:240 A:255

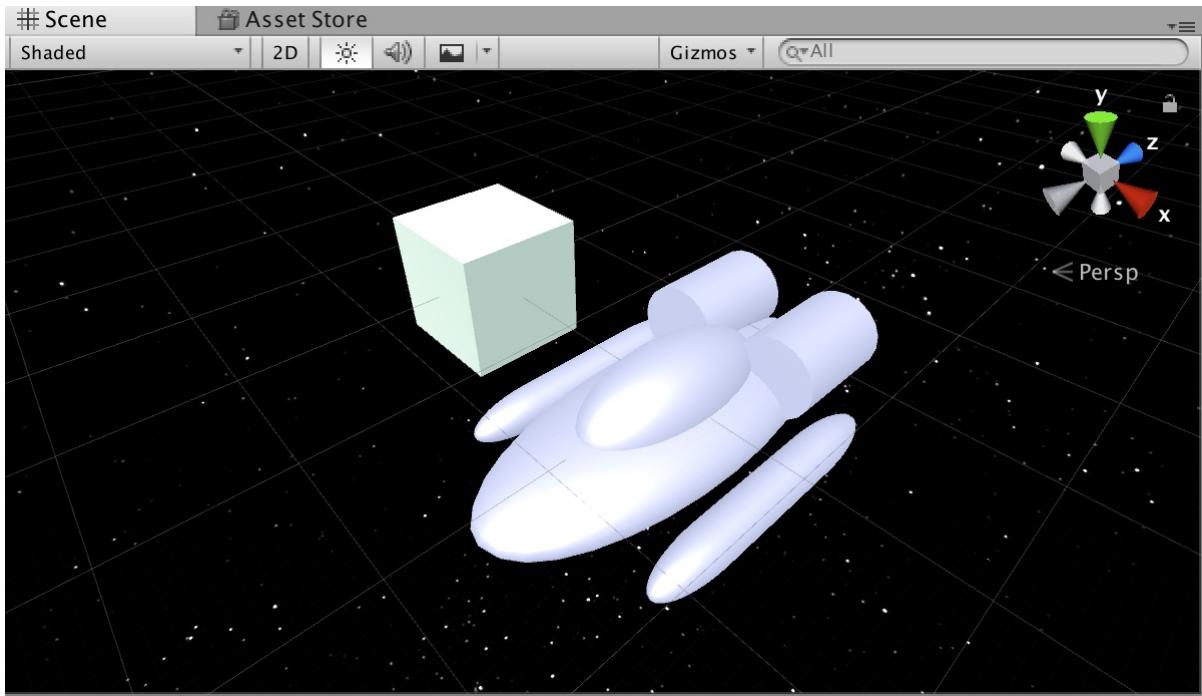
Metallic : 0.7

Smoothness : 0.55

Emissivity : チェックを入れる

Color : Current Brightness : 0.6

R:0.5 G:0.5 B:0.6



ここまでできたらEnemyをProjectウィンドウのAssetsにドラッグ&ドロップしてプレハブ化します。

## 移動スクリプトの作成

次に構造物モデルと敵機モデルを移動するプログラムを作成します。先ほども説明したように前方に現れて後方に移動し、見えなくなったらまた前方から現れる、という処理です。

Structureオブジェクトを選択し、InspectorのAdd Component→New Scriptを選んで新規スクリプトを作成しましょう。スクリプト名はStructureMoverとします。

StructureMover.cs

```
using UnityEngine;

public class StructureMover : MonoBehaviour {

    float speed = 30;

    void Update () {
        float z = Time.deltaTime * speed;

        transform.position -= new Vector3(0, 0, z);
        if (transform.position.z < -10)
        {
            transform.position += new Vector3(0, 0, 50);
        }
    }
}
```

速度のパラメータを30とし、Time.deltaTimeと掛けることでPC性能に関わらず一定の速度で動くようにしています。x、y座標は変化せず、z座標だけをマイナスしていくことで奥から手前へ移動させています。z座標が-10になったら、もう後方で見えなくなっているので、z座標に50を足して再び前方から現れるようにしています。

敵機のスクリプトも同じような構造です。Enemyを選択し、InspectorのAdd Component→New Script。スクリプト名はEnemyMoverとします。

EnemyMover.cs

```

using UnityEngine;

public class EnemyMover : MonoBehaviour {

    float speed;

    void Start () {
        Init();
    }

    void Update () {
        float z = Time.deltaTime * speed;

        transform.position -= new Vector3(0, 0, z);
        if (transform.position.z < -30)
        {
            Init();
        }
    }

    void Init() {
        speed = Random.Range(40f, 60f);
        float x = Random.Range(-1f, 1f);
        float y = Random.Range(0.5f, 5f);
        float z = 50;
        transform.position = new Vector3(x, y, z);
    }
}

```

構造物の移動処理と似ているのがわかりますね。今度は出現位置と速度が都度ランダムになっています。一番最初のStart時と、後ろに消えて再出現するときに乱数で位置と速度を決めるため、共通関数Init()を用意して呼ぶようにしています。構造物はz座標が-10になったら前方に移動して再出現するようにしていましたが、敵機の場合はあとでエンジン音を追加する関係上、通過直後の-10ではなく-30まで達して音が聞こえなくなってから再出現するように調整してあります。

スクリプトを追加したので、Hierarchyとプレハブとの相違が発生しています。変更をプレハブに反映するために、HierarchyのStructureを選択してInspectorからApplyを押してください。Enemyも同様にApplyします。

ここまでできたら、Ctrl+S(Macは⌘+S)でmodelシーンをセーブしましょう。

## モデル生成処理の作成

### 構造物生成処理

それでは作成したモデルを生成する処理を書いていきましょう。mainシーンを開いてください。

ゲーム全体の処理のスクリプトを配置するために、前回も作成した空のオブジェクトを配置しましょう。HierarchyでCreate Emptyして名前を「Game」とします。

次にGameを選択し、InspectorのAdd Component→New Script。スクリプト名はStructureGeneratorとします。

StructureGenerator.cs

```

using UnityEngine;

public class StructureGenerator : MonoBehaviour {

    public GameObject structurePrefab;

    void Start () {
        for (int i = 0; i < 100; i++)
        {
    }
}

```

```

float size;
Vector3 pos;
GameObject obj;

// 右側
size = Random.Range(0.3f, 2f);
pos = new Vector3(
    Random.Range(-0.1f, 0.1f) + 2f,
    Random.Range(0f, 4f),
    Random.Range(-10f, 50f)
);
obj = Instantiate(structurePrefab, pos, Quaternion.identity);
obj.transform.localScale = new Vector3(1f, size, size);

// 左側
size = Random.Range(0.3f, 2f);
pos = new Vector3(
    Random.Range(-0.1f, 0.1f) - 2f,
    Random.Range(0f, 4f),
    Random.Range(-10f, 50f)
);
obj = Instantiate(structurePrefab, pos, Quaternion.identity);
obj.transform.localScale = new Vector3(1f, size, size);

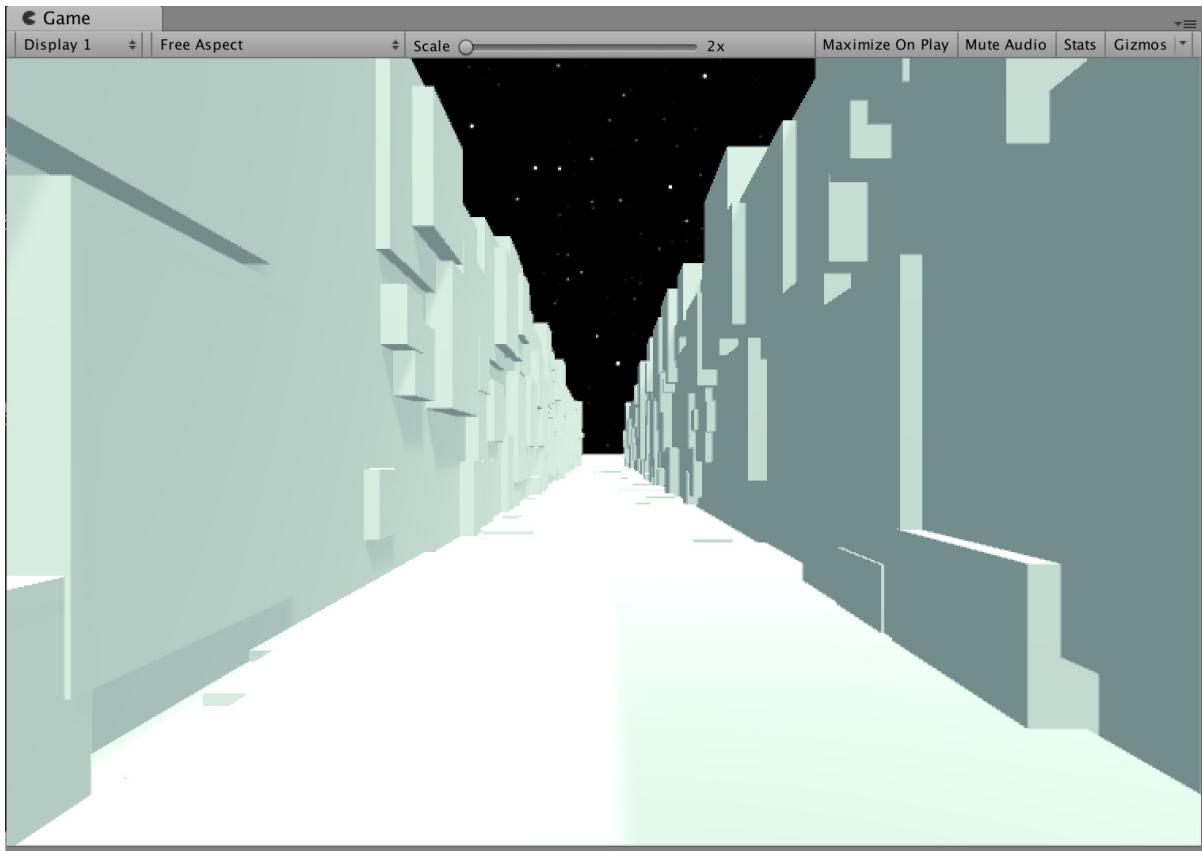
// 下側
size = Random.Range(1f, 2f);
pos = new Vector3(
    Random.Range(-1f, 1f),
    0,
    Random.Range(-10f, 50f)
);
obj = Instantiate(structurePrefab, pos, Quaternion.identity);
obj.transform.localScale = new Vector3(size, 0.1f, size);
}

}

```

ちょっと長いですが、頑張って読んでみてください。やっていることは単純です。右側、左側、下側にそれぞれランダムなサイズ、ランダムな位置に構造物モデルを生成しています。Instantiate関数は、以前も出てきましたがプレハブをシーン上に実体化する処理です。ひとつのプレハブからたくさんのコピーを作成するときに良く使われます。今回は全体が100回ループのfor文で囲まれているので、右、左、下それぞれ100個ずつの構造物オブジェクトが生成されます。このように、あらかじめシーンに配置するのではなく、動的にプログラムでステージを生成することをプロシージャルな手法と呼びます。うまく生成すると複雑で広大な地形を一気に作ることができるので、最近のゲーム制作でもよく使用される手法です。なお、実行時にフレームレートが十分出ないときは、この100の数字を下げることで処理を軽くすることができます。画面上のオブジェクトの数と処理速度はトレードオフなので仕上げの段階にならこういったところを調整するようにしてみてください。

このスクリプトは、public変数で構造物のプレハブをInspectorから設定できるようにしてあります。AssetsにあるStructureプレハブをInspectorのStructurePrefabにドラッグ＆ドロップしてから実行してください。



## 敵機生成処理

敵機生成処理も同様に、Gameを選択し、InspectorのAdd Component→New Scriptです。スクリプト名はEnemyGeneratorとします。

EnemyGenerator.cs

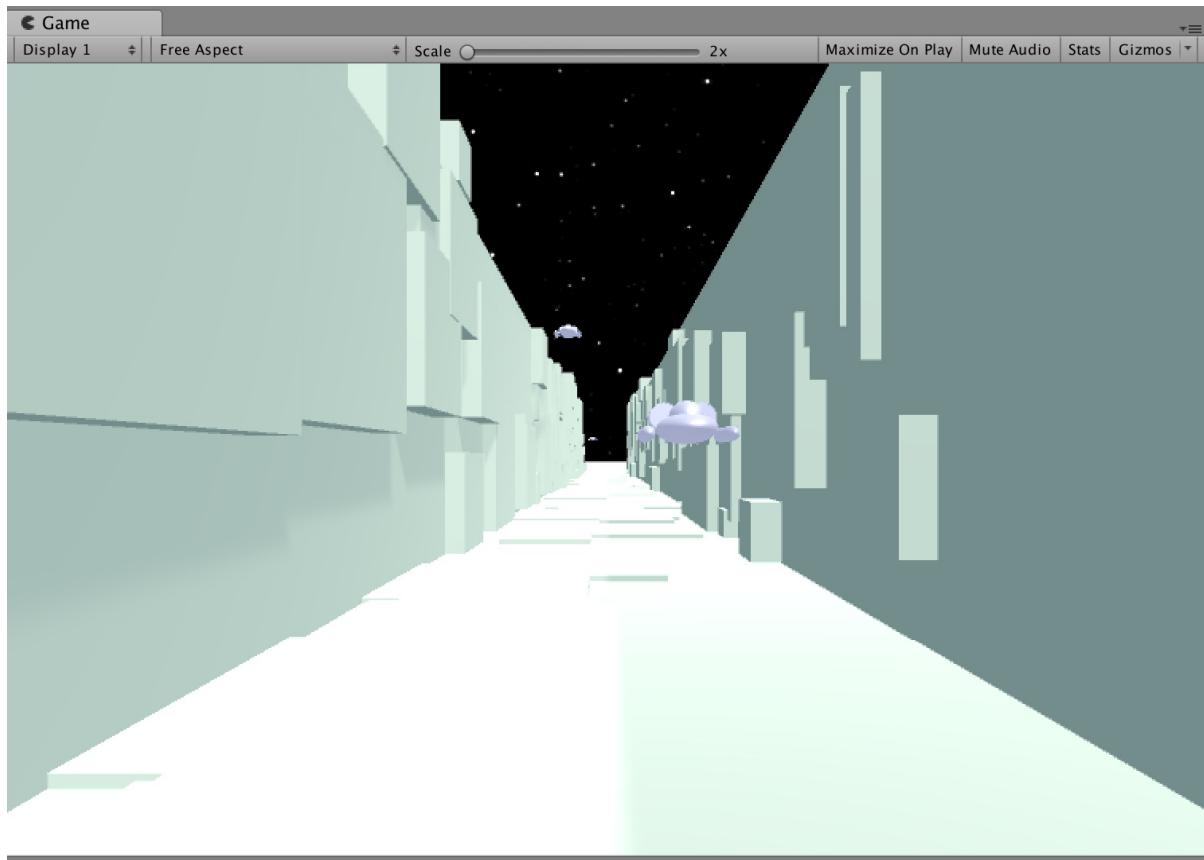
```
using UnityEngine;

public class EnemyGenerator : MonoBehaviour {

    public GameObject enemyPrefab;

    void Start () {
        for (int i = 0; i < 3; i++)
        {
            GameObject enemy = Instantiate(enemyPrefab, Vector3.zero, Quaternion.identity);
        }
    }
}
```

敵機生成は非常にシンプルです。出現位置は敵機自身がランダムに決める処理を先ほど書きましたし、サイズも一定程度良いので、単純にInstantiateするだけです。for文で3回ループしているので敵機は3機出現します。これも、AssetsにあるEnemyプレハブをInspectorのEnemyPrefabにドラッグ&ドロップしてから実行してください。



## サウンドの追加

次はサウンドを追加してみましょう。Asset StoreでそれっぽいフリーSE集を探してみます。



Asset StoreでSci-Fi Sfxを検索してダウンロード、インポートしてください。

インポートしたら、中身をのぞいてみます。サウンドファイルを選択してInspector下部の波形表示の右上に再生ボタンがあります。なんか、どれも思ってたんと違う感じ。まあ、でも妥協して使っていきましょう。イメージぴったりのアセットが見つかることはなかなか無いので、とりあえず手元にある素材でなんとかするのも腕の見せ所です。

今回導入したいSEは、自機のエンジン音、敵機のエンジン音、自機と敵機の衝突音の3種類です。まずは自機のエンジン音から設定していきます。

## 自機のエンジン音

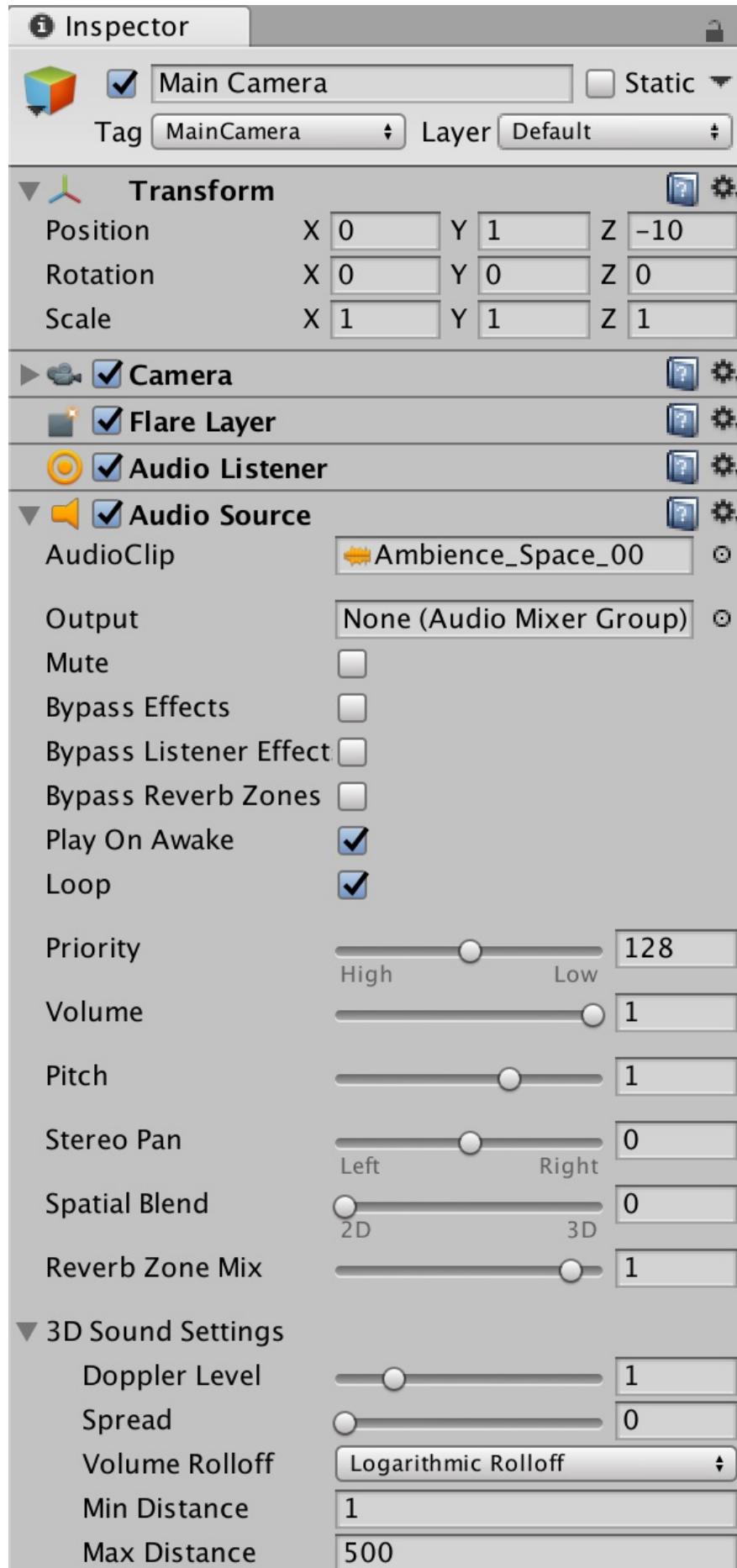
自機のエンジン音はAssets/Sci-Fi-Sfx/Wav/Ambience\_Space\_00が良さそうです。ゴーという低い小さな音が鳴り続ける感じです。Sci-Fi SfxにはMP3のサウンドファイルとWavのサウンドファイルが両方収録されているようです。注意点としては、環境音のように無限ループで鳴らす音はMP3はNGということがあります。MP3は構造上先頭に空白が入ってしまうため、ループするときにブツッと音が途切れてしまいます。Wavの場合は空白が入らないのでこちらを使うようにしましょう。

自機のサウンドはMain Cameraにアタッチします。これは、音を受けるAudio ListenerコンポーネントがMain Cameraにアタッチされているため、方向もなく移動もしない音はそれと同じ場所に配置するのが適切であるためです。Main CameraのInspectorからAdd Component→Audio Sourceを選択します。次にAudio SourceコンポーネントのAudioClipにAmbience\_Space\_00をドラッグ&ドロップします。その他、以下のパラメータを設定してください。

Play On Awake : 有効

Loop : 有効

Spatial Blend : 0 (2D)



## 敵機のエンジン音

敵機のエンジン音はAssets/Sci-Fi-Sfx/Wav/SpaceShip\_Engine\_Large\_Loop\_00を使います。今度は自機と敵機の位置や速度によって音が変化する3Dオーディオを活用しましょう。 AssetsのEnemyプレハブを選択してAdd Component→Audio Sourceを選択します。次にAudio SourceコンポーネントのAudioClipにSpaceShip\_Engine\_Large\_Loop\_00をドラッグ＆ドロップします。パラメータは以下のようにします。

Play On Awake : 有効

Loop : 有効

Spatial Blend : 1 (3D)

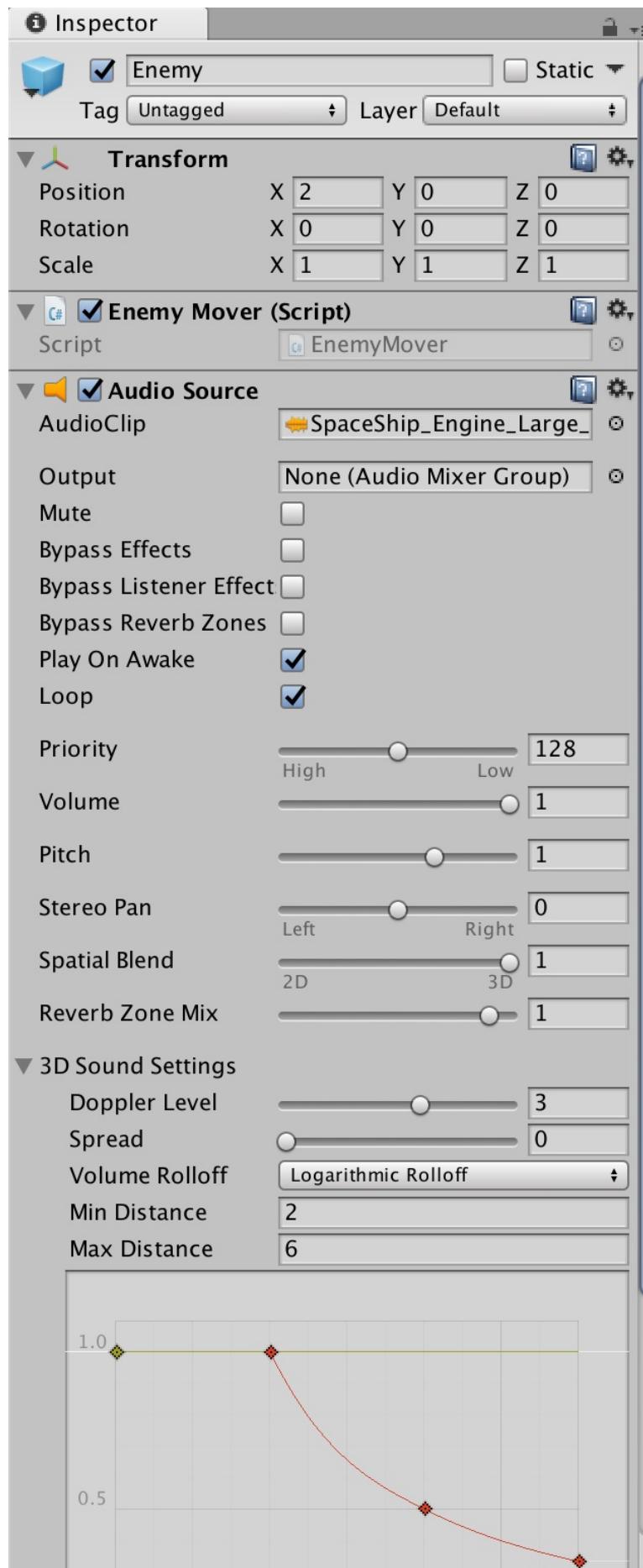
3D Sound Setting

Doppler Level : 3

Min Distance : 2

Max Distance : 6

Doppler Levelを設定することでドッpler効果をシミュレートできます。近づいてくるときは高い音、離れていくときは低い音になり、それ違うときにビューウンという音程変化が生まれます。またMin Distance、Max Distanceを小さめにすることで近づいたときに急に音が大きくなり、スピード感が強調されます。



## 衝突音

後で使うときのために衝突音も準備しておきましょう。衝突音はAssets/Sci-Fi-Sfx/Wav/Laser/Laser\_02を使います。このサウンドファイルは音量が大きいので少し調整した方が良さそうです。先ほどと同様に、AssetsのEnemyプレハブを選択してAdd Component→Audio Sourceを選択します。次にAudio Sourceコンポーネントの AudioClipにLaser\_02をドラッグ&ドロップします。パラメータは以下のようにします。

Play On Awake：無効

Loop：無効

Volume：0.16

Spatial Blend：0 (2D)

Play On Awakeを無効にしたので、そのままでは音は鳴りません。後でスクリプトに鳴らす処理を追加します。

ここまでできたら実行してみましょう。低い自機のエンジン音と、敵機とすれ違うときの音が聞こえたでしょうか。

## 簡易的なあたり判定

Unityでは物理演算を使ってあたり判定をするのが定番ですが、物理演算はかなり癖があって初心者にはあまりおすすめできないと思っています。たとえば、今回のような高速で移動するオブジェクトは、物理演算だと意図せぬずり抜けてしまうことが起こってしまい、それらに対処するために直感的とは言えないいくつかのバッドノウハウを必要とします。そのため、最初はシンプルなあたり判定を実装し、その後本格的なゲームを作成するときに物理演算を学んでいくような順番が良いと考えています。

今回実装するあたり判定処理は、敵機と自機（メインカメラ）との距離を毎フレーム計算し、一定距離より近づいたら衝突したと判断するものです。

AssetsのEnemyプレハブを選択し、InspectorのAdd Component→New Script。スクリプト名はHitDetectorとします。

HitDetector.cs

```
using UnityEngine;

public class HitDetector : MonoBehaviour {

    GameObject target;
    AudioSource se;

    void Start () {
        target = Camera.main.gameObject;
        se = GetComponents<

```

targetが自機（メインカメラ）です。これはCamera.main.gameObjectとして取得することができます。また、衝突音seはGetComponents()としてEnemyにアタッチされている複数のAudio Sourceを取得しています。先ほど敵機のエンジン音と衝突音の2個のAudio Sourceをアタッチしました。そのため、[0]と指定するとエンジン音、[1]と指定すると衝突音が得られます。Update関数の中で、自機とEnemyの距離Distanceを取得して、距離が0.5未満の場合、衝突音を鳴らしています。

## 入力の処理

敵機と衝突するようになったので、キー操作で自機を移動できるようにしましょう。基本的にはGetKeyでキーの状態を読んで、positionを移動するだけです。

HierarchyのMain Cameraを選択し、Add Component→New Script。スクリプト名はCameraMoverとします。

CameraMover.cs

```
using UnityEngine;

public class CameraMover : MonoBehaviour {
    void Update () {
        float delta = Time.deltaTime * 1.5f;

        if (Input.GetKey(KeyCode.LeftArrow))
        {
            if (transform.position.x > -1)
            {
                transform.position -= new Vector3(delta, 0f, 0f);
            }
        }
        if (Input.GetKey(KeyCode.RightArrow))
        {
            if (transform.position.x < 1)
            {
                transform.position += new Vector3(delta, 0f, 0f);
            }
        }
        if (Input.GetKey(KeyCode.DownArrow))
        {
            if (transform.position.y > 0.5f)
            {
                transform.position -= new Vector3(0f, delta, 0f);
            }
        }
        if (Input.GetKey(KeyCode.UpArrow))
        {
            if (transform.position.y < 5)
            {
                transform.position += new Vector3(0f, delta, 0f);
            }
        }
    }
}
```

キーコードを見て矢印キーなら上下左右に動かしているだけです。それぞれリミットに達したらそれ以上は動かないようく制限しています。Unityのキー状態取得関数にはGetKeyの他にもGetKeyDown、GetKeyUpなどがあります。GetKeyは押している間ずっとtrueが返りますが、GetKeyDown、GetKeyUpはキーのダウン、アップの1回だけtrueが返ります。今回のように連続的に移動するような場合はGetKeyが適しています。

## 仕上げの絵作り

ゲームの構造は以上で完成です。最後にまたポストプロセスを使って画面をリアルにしていきましょう。

Asset Storeからダウンロードマネージャーを選び、Post Processing Stackをインポート。Assetsで右クリックメニュー→Create→Post-Processing Profileを選択してプロファイルを作成します。

次に、HierarchyのMain Cameraを選択してInspectorからAdd Component→Effects→Post Processing Behaviourを選択します。プロファイルをPost Processing BehaviourのProfileにドラッグ＆ドロップします。あとはAssetsのPost-Processing Profileを選択してInspectorで調整していきます。

## Ambient Occlusion

壁と床の交わるような角を暗くする効果です。古びた雰囲気やリアルな質感が簡単に出せるのでおすすめです。

Intensity : 0.64

Radius : 0.72

他はデフォルト

## Motion Blur

今回の構造物や敵機のような高速で移動するものは、よく見るとフレーム間でパラパラと細かくジャンプしているように見えます。モーションブラーは、動いている物体に微妙にブラーをかけることでパラパラした感じを無くして、滑らかに高速で動いている効果を出すことができます。

パラメータは全部デフォルト

## Bloom

輝度の高いものの周囲をボワっと柔らかく光らせる効果です。SF風のCGをてつりばやく作ることができます。

パラメータは全部デフォルト

## Vignette

Instagram等でよく見るような画面の外周付近を暗くする効果です。レトロな雰囲気が出るほか、高速な乗り物を操縦しているときにスピードで視界が狭くなる効果にも使えると思います。

Intensity : 0.43

Smoothness : 0.43

他はデフォルト

最後にフォグ（霧）の効果をかけます。距離的に遠いものを暗くするエフェクトで、視認性を悪くする効果だけでなく、遠近感やスケール感の演出に使えます。フォグはPost-Processing Profileにもありますが、これはパラメータが調整できないので、ライトの設定にある方のフォグをかけるのがおすすめです。

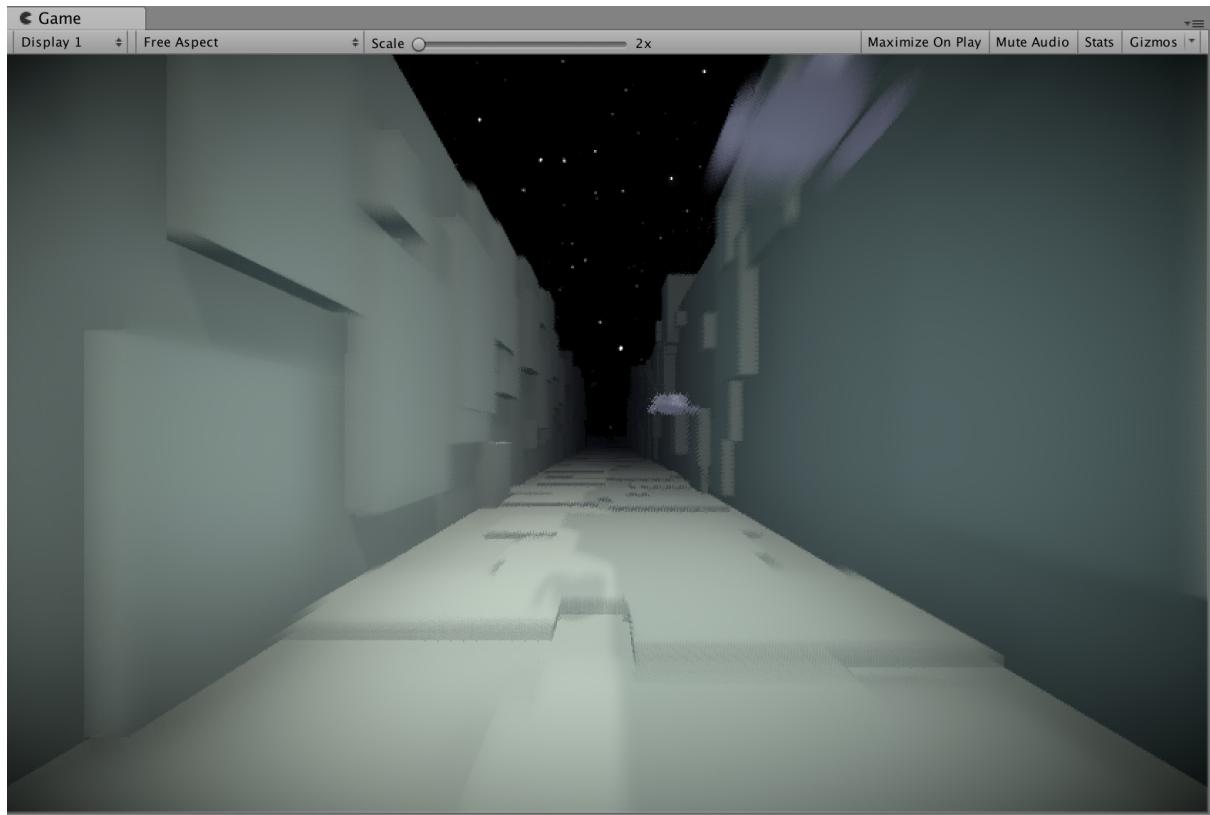
Windowメニュー→Lighting→Settingsでウィンドウを開いて、下の方にあるOther Settingsで設定します。

Fog : 有効

Color : R:15 G:15 B:20

Mode : Exponential

Density : 0.1



おつかれさまでした。これでチュートリアル「トレンチファイター」は完了です。