本書に掲載したプログラムは、以下の環境で動作を確認しています。

検証環境

- OS X Yosemite 10.10.4（MacBook Pro Retina、13-inch、Mid 2014）
- Unity 5.1.0f3
- Xcode 6.4
- Android Studio 1.2.2（SDK API 21）

なお、本書に掲載した画面のハードコピーは、一部を除き「Unity5.0」のものです。「Unity5.1」でご利用の場合、画面上部右側の「Account」メニューが本書の画面ではないなどの違いはありますが、本書の操作や解説に関しては「Unity5.1」でもまったく同様にご利用いただけます。

はじめに

スマートフォン向けのゲーム開発は、本当にエキサイティングです。ゲームをアプリとして実現すれば、誰でもそれを世界中のストアに配信することができます。そして時には何万人もの人が、自分が考え作ったゲームを遊び、レビューをくれることもあります。この「想像もつかないほどの多くの人が楽しんでくれている」という状況には、自分自身がゲームで遊んでいる時とはまた違った面白さやワクワク感があります。

Unityは、あなたの奥底で眠っているゲームアイディアを実現する非常に優れたツールです。少し前まで、ゲーム開発は、技術的な面、資金的な面、ライセンス的な面で高いハードルがありました。しかし、Unityを使えばこれらのハードルを限りなく低くし、あなたが思い描いているアイディアを最短で動く形に落とし込めます。

ゲーム開発のプロも利用しているUnityは、極めれば高度な処理や表現が可能ですが、その一方で誰でも簡単にゲームを作れるという特徴も持っています。これは、Unityの唯一無二の洗練されたエディターにより、複雑な内部ロジックをほとんど意識しなくても、さまざまな機能を利用できてしまうためです。

また、Unityは無料でも利用することができ、作ったゲームは自由に公開することができます。これはもはや「革命」と言っても言い過ぎではないでしょう。

この本は、Unityを学びたい人だけでなく、「スマートフォンのゲームを作ってみたい」、「アイディアはあるけど何をすればいいかわからない」といった、はじめてゲーム開発をする人のことも想定して書いています。そのため、Unityの操作方法や機能だけでなく、それ以前のゲームを作る上で必要な知識や、基本もなるべく解説するようにしています。

また、実際に制作するゲームサンプルは、画面サイズや操作方法など、基本的にスマートフォンのストアで公開することを意識しています。さらに、途中で紹介するプログラムコードは極力シンプルにしていますが、ゲームとしてそれなりに成立します。そのため「スマートフォンゲームとしてちゃんと遊べるものを作る」ことを前提に、実践を通してUnityの使い方を学べるはずです。

さらに、この本は入門レベルの解説本ですが、「Unityユーザー」として一人立ちできるような内容を目指しました。手順に沿ってチュートリアルを進めて、Unityの雰囲気を掴むだけでなく、どうしてその過程が必要なのか、その作り方になるのかも説明しています。またこれらは、今後も応用が利く機能や使い方をできるだけピックアップしています。

Unityは巨大なゲームエンジンであり、ゲーム開発自体もすぐにマスターできるものではありません。しかし、これからUnityを使ってアイディアを実現したいという人を、少しでも手助けできたらと思い、この本を執筆しました。また、これらは本当に小さなきっかけかもしれませんが、ゲーム開発の面白さや、アイディアを実現することの楽しさを少しでも共感いただけたら、本当に嬉しく思います。

2015年7月　吉谷 幹人

本書の構成

本書は、以下の章で構成されています。本書のサンプルプログラムは、C#で記述していますが、C#の文法などプログラミングの入門については解説していませんので、別途書籍などを参照してください。

また、本書の内容は執筆時点（2015年6月）の情報です。Unityの最新版の情報やライセンス形態などは、変更される可能性がありますので、ご利用にあたっては以下のUnityのWebサイトで最新情報を確認してください。

Unityの公式Webサイト

➡ **http://japan.unity3d.com/**

序章　Unityの特徴と概要

Unityは世界中で使われているゲームエンジンで、これにより多数のゲームが開発、公開されています。最初に、Unityの特徴と概要をまとめておきます。

1章　Unityでゲーム開発を行う前の準備

Unityは、無償で利用することができます。この章では、Unityの入手方法とインストールについて解説します。

2章　Unity はじめの一歩－Unityの画面構成や基本操作を覚える

Unityはゲームの統合開発環境で、さまざまなツールや機能が用意されています。まずは、Unityエディターで3Dオブジェクトを扱うための基本操作を覚えます。

3章　ゲーム作成の基本－物理エンジンとコリジョンをマスターする

Unityでのゲーム作成の要である、物理エンジンとコリジョンによるイベント制御を、「ボール転がし」サンプルゲームを使って解説します。

4章　ゲームオブジェクトの制御－プレファブとエフェクトを極める

開発効率をアップするためのポイントであるプレファブと、ゲームの効果を演出するパーティクルとサウンドを、「キャンディ落とし」サンプルゲームを使って解説します。

5章　3Dゲームを作成－キャラクターとGUIをコントロールする

これまで紹介した機能を使って、本格的な3Dゲームを作ります。ここでは、著者が実際に業務で制作した「ゆるロボ製作所」のキャラクターを利用します。

6章　2Dゲームを作成－スプライトと2D物理エンジンを使いこなす

Unityでは2Dゲームも作成できます。ここでは、「空飛ぶアラザラシ」サンプルゲームを使って、2Dのスプライトや物理エンジンの制御方法などを学びます。

7章　ゲームのリリース準備をして、ストアに登録する

Unityでは、マルチプラットフォームの開発を行えます。作成したゲームを、App StoreとPlay Storeで公開する方法を紹介します。

Appendix　Unityをさらに使いこなすために

本文では紹介しきれなかったアプリ内広告の掲載方法や、開発効率を上げる定番のアセット、プラグインを紹介します。

アセットのダウンロード

本書に掲載したサンプルプログラムの各種アセットは、以下のWebページよりダウンロードできます。アセットの利用方法などは、各章の本文を参照してください。

アセットのダウンロードWebサイト

➡ **http://www.socym.co.jp/book/967**

サンプルプログラムの利用について

本書に掲載したサンプルプログラムは、Unityの学習のために作成したもので、実用を保証するものではありません。学習用途以外ではお使いいただけませんので、ご注意ください。また、一部のアセットは、以下のライセンスに準拠してご利用ください。なお、本書に掲載したプログラムの著作権は、すべて著者に帰属します。

ゆるロボアセットライセンスについて

ダウンロードデータのアセットには、ゆるロボ製作所のキャラクターモデルが含まれています。本書の解説に基づいたサンプルプログラムの製作には利用いただけますが、それらをゲームとして一般公開することはできませんのでご注意ください。

ご利用の際は、パッケージ内に含まれている、LICENCE.txtの内容に従ってご利用ください。なお、利用したことをもって、LICENCE.txtの内容に同意したものとみなします。

以下にできることと、できないことを簡単にまとめておきます。

できること

- Unity習熟のためのサンプルプログラムの作成
- サンプルプログラム作成過程のスクリーンショットの撮影、および公開

できないこと

- 作成したプログラムのマーケットへのリリース、一般公開
- アセットの二次配布
- 作成したプログラムのリポジトリの公開

もくじ

6章　2Dゲームを作成－スプライトと2D物理エンジンを使いこなす…280

7章　ゲームのリリース準備をして、ストアに登録する…348

Appendix　Unityをさらに使いこなすために　……364

コラム一覧

TIPS一覧

CHAPTER 3

ゲーム作成の基本―
物理エンジンとコリジョンをマスターする

最初のサンプルでは**物理エンジン**を使ったゲームを作ってみましょう。物理エンジンとは物体の重力による落下や、跳ね返り、摩擦による減速など現実世界の物理的な動きをシミュレーションするための機能や枠組みのことです。

Unityではこれらがすでに用意されているため、複雑な物理計算を行うプログラムを自分で書かなくても、理にかなった自然な動作をオブジェクトに実現させることができます。しかも、この物理エンジンはUnityで簡単に扱うことができ、お手軽にいろいろなゲームでの利用や応用が可能です。

また、オブジェクト同士がぶつかったという判定を**コリジョン（衝突）**と呼びますが、このコリジョンの扱い方を理解することがゲームプログラミングの上達のカギになります。Unityではキャラクターがチェックポイントを通過したことや、爆風の範囲にオブジェクトがあるかどうかなど、さまざまなイベントの検知にコリジョンを使うことが多くあります。つまり、コリジョンの処理をうまく利用することで、Unityならではの開発効率のよさを引き出すことができます。

まずは、前章のおさらいも兼ね、Unityの基礎であり重要な機能をしっかりとマスターしましょう。

この章の目的

- 物理エンジンを扱えるようになる
- コリジョンによるイベント制御ができるようになる

サンプルゲーム：ボール転がし（Illumiball）

スマートフォンを傾けると、画面のなかのボールが実際にその方向に向かって転がります。プレイヤーは赤、青、緑の3つの光るボールをそれぞれの色のスポットライトの位置に同時に運ぶことができればクリアになります。

現実の世界にも木のボードを傾かせてボールをゴールの穴まで運ぶおもちゃがありますが、このゲームはそのコンセプトを少し変えたものです。おもちゃの場合は基本的に平面でのボールの操作でしたが、このゲームでは高さ方向にもボールが移動します。つ

まりスマートフォンを手前に傾けたりして、三次元的に考えなければクリアできません。

また、ゴールの穴の部分は、ただのスポットライトになっていますが、実はこの部分には特殊な重力のような力が働いており、同じ色のボールは吸い付けますが、違うボールは弾き飛ばします。

ぱっとした見た目は光輝くボールが綺麗で、ちょっと近未来的なボール転がしといった感じです。ゲームの名前もイルミネーション（Illumination）とボール（Ball）をくっつけたものにしてみました。シンプルで実装も簡単ですが、クリアは意外と難しく奥が深いゲームです。

作成するサンプルゲーム Illumiball

3-1 ボールの作成

まずは、ゲームの中核部分であるボールの部分を作っていきましょう。球状のオブジェクトの作成は前章でも行いましたが、これらに自由落下など物理的な挙動が働くようにします。また、光っているように見せるためのマテリアルとライトの設定を行います。

大きさ違い、色違いでボールが3つ必要なため、似たような作業を繰り返す手間がありますが、その分オブジェクトが動くようになったり、見た目が綺麗になったりなど、これだけで格段と面白さが増すはずです。

シーンの初期化

まず始めに新規シーンの作成と保存をします。作業後にシーンの保存でも問題はありませんが、完全に初期化されている状態にしておくことと、こまめにセーブするために、作業開始前に意識的に行うクセをつけるとよいかと思います。

FileメニューのNew Sceneで新しいシーンを作成後、Save Scene asからシーンの保存を行いましょう(1)。作成するシーン名は分かりやすければ何でも構いませんが、サンプルではゲーム名と同じ「Illumiball」にしています(2)。

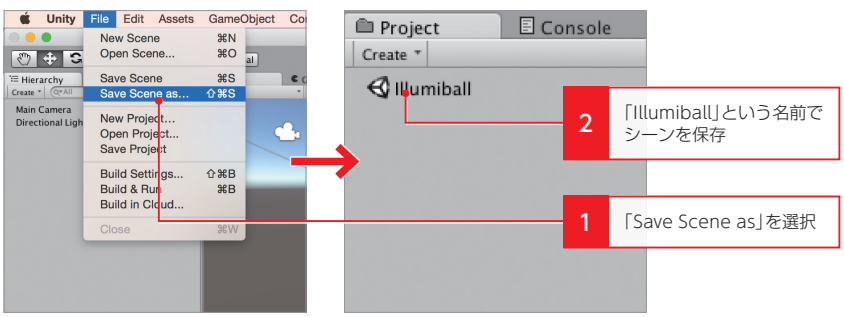

図3-1-1 シーンの作成

シーンの保存ができたら、開発途中のボールを受け止めるための仮ステージを配置しましょう。後のセクションでボックス状のちゃんとしたものを作るので、ここでは大雑把なものを配置します。

HierarchyビューのCreateメニューから3D Object→Cubeを選択し、オブジェクトを生成してください(1)。生成したCubeオブジェクトは、位置が原点の(0, 0, 0)になるようにPositionの値をインスペクタから調整します。

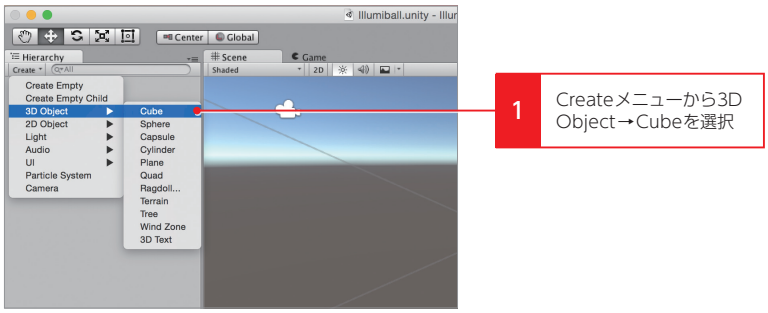

図3-1-2 Cubeオブジェクトの生成

配置ができたら、Scaleツール（1）を使ってCubeのxとzのScaleを引き伸ばし平べったくします（2）。仮ステージなので、細かい値は気にする必要はありません。xとzが25程あれば大丈夫です。

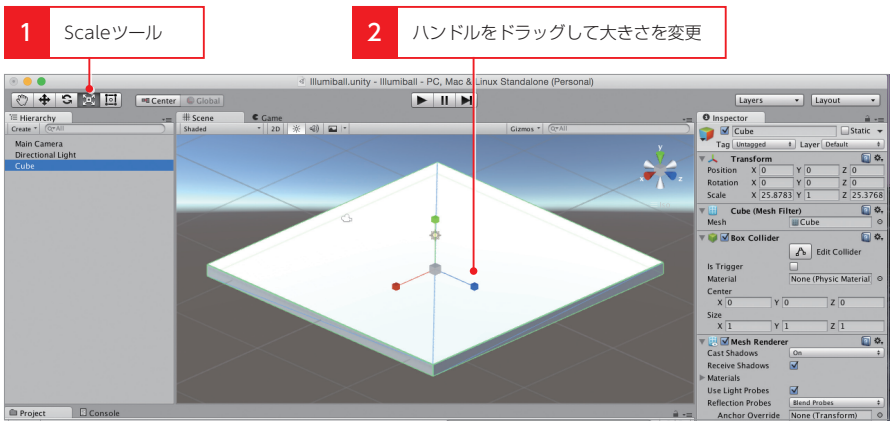

図3-1-3 CubeオブジェクトのScale変更

Ballオブジェクトの生成

次はBallオブジェクトを作成します。HierarchyビューのCreateメニューからSphereを選択してください（1）。生成したSphereオブジェクトの名前を「BallRed」に変更し（2）、Positionツールでステージの見やすい位置に移動させます（3）。

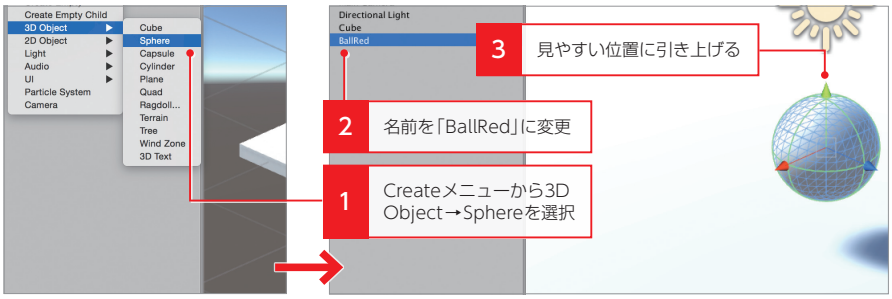

図3-1-4 Sphereオブジェクトの生成

同様に、ほかの2つのBallオブジェクトも同じ手順で作成します（1）。名前はそれぞれ「BallBlue」と「BallGreen」にし、Scaleを(2, 2, 2)と(3, 3, 3)に設定してください（2）。

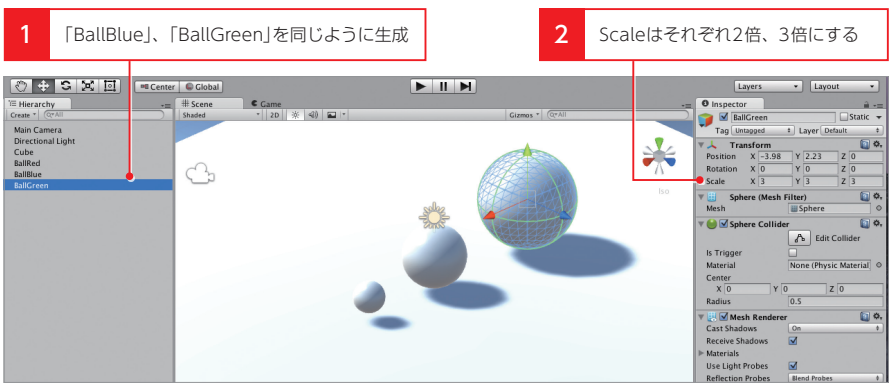

図3-1-5 BallBlue、BallGreenオブジェクトの生成とScaleの設定

Rigidbodyの追加

さて、ボールの見た目はできあがりましたが、ゲームをプレビューしても空中に浮いたまままったく動きません。次は物理的な挙動を設定して、落下させてみましょう。

UnityではHierarchyビュー上に表示される1つの**ゲームオブジェクト**に対して、さまざまな**コンポーネント**を追加することで、機能を上乗せしていくことができます。もっともシンプルなゲームオブジェクトは、3D空間上の位置や大きさを扱う**Transformコンポーネント**のみが設定されている状態です。

一方、SphereをベースしたBallオブジェクトではTransformコンポーネントに加えて、3Dの形を制御する機能のMesh Filterコンポーネント、見た目をカメラに描写する機能のMesh Rendererコンポーネントが設定されており、それらが組み合わさることでシーン上に「丸い形」を「描写する」という仕組みが実現されているわけです。

また、このほかにも新規にコンポーネントを追加することで、独自の動作をさせたりなど機能を追加することができます。

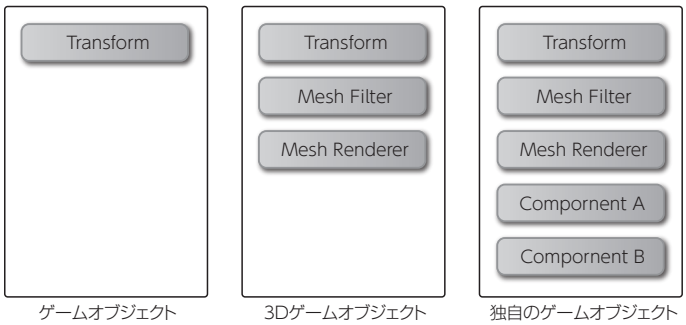

図3-1-6 ゲームオブジェクトとコンポーネント

オブジェクトに物理挙動をさせるためには、**Rigidbodyコンポーネント**を利用します。3つのボールを同時に選択した状態（1）で、ComponentメニューのPhysicsからRigidbodyを選択しましょう（2）。すると、InspectorビューにRigidbodyコンポーネントのパラメータが表示されるようになります（2）。

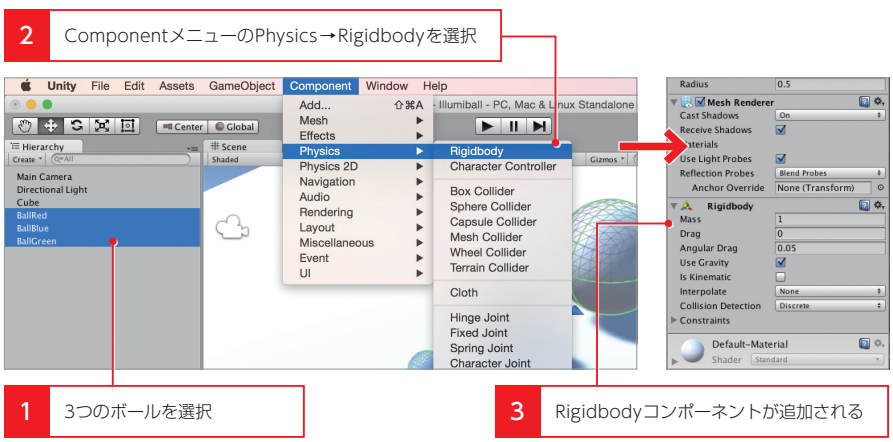

図3-1-7 Rigidbodyコンポーネントの追加

TIPS

InspectorビューのAdd Componentボタン

InspectorビューのAdd Componentボタンからもコンポーネントを追加することができます。こちらではコンポーネント名を検索して絞り込みができるため、目的のコンポーネントがはっきりしている場合は、こちらから追加するほうが早いでしょう。

これだけで、物理挙動の設定は完了です。ものすごく簡単ですね！実際にゲームのプレビューを開始すると（1）、ボールが落下してステージの上で止まります（2）。地味ですが、重力による自由落下とコリジョンによる反発が確認できます。

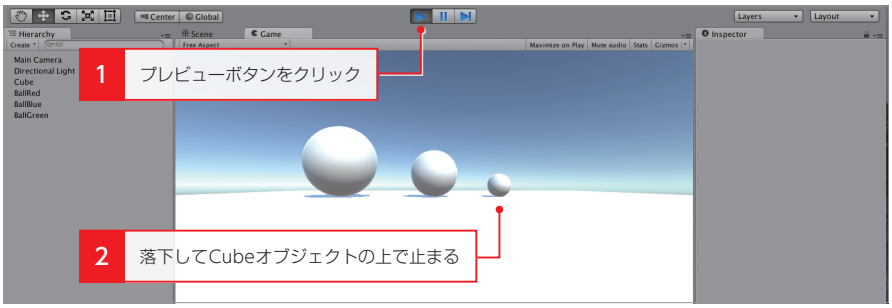

図3-1-8 落下動作のチェック

ステージとボールのコリジョン判定が起こるのは、オブジェクトに対して**Collider(コライダー)**が設定されているためです。コライダーもコンポーネントになっており、CubeオブジェクトにはBox Collider、SphereオブジェクトにはSphere Colliderが自動で追加されています。

コライダーの範囲は、Sceneビューで緑色のワイヤーフレームで表示されており(1)、Rigidbodyコンポーネントを追加したオブジェクトは、この範囲でぶつかったり跳ね返ったりなどの反応を起こすようになります。

逆にコライダーが設定されていないと、見た目が表示されていてもオブジェクト同士は擦り抜けてしまいます。ちなみにオブジェクトのScaleを変化させると、同じ割合でコライダーの範囲も変化します。

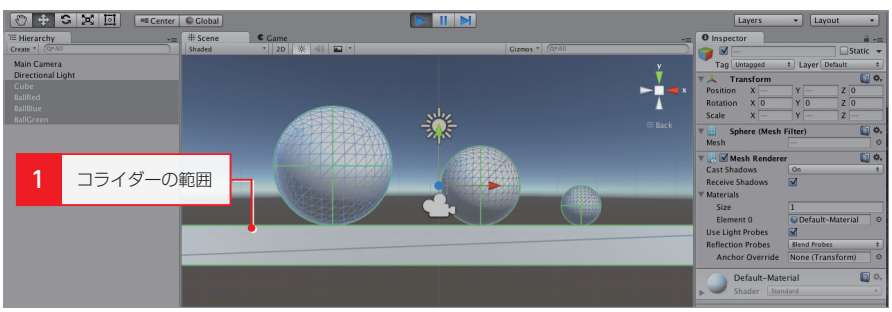

図3-1-9 コライダーの範囲

マテリアルの設定

次はマテリアルを設定して、ボールに色を付けましょう。こちらも赤、青、緑の3種類を用意し、3つの大きさのボールにそれぞれ設定します。また、色だけでなく発光させるパラメータも設定していきます。

まずは、一番小さいボール用の赤色のマテリアルを作成します。ProjectビューのCreateメニューからMaterialを生成し(1)、名前を「BallRed」に変更してください(2)。

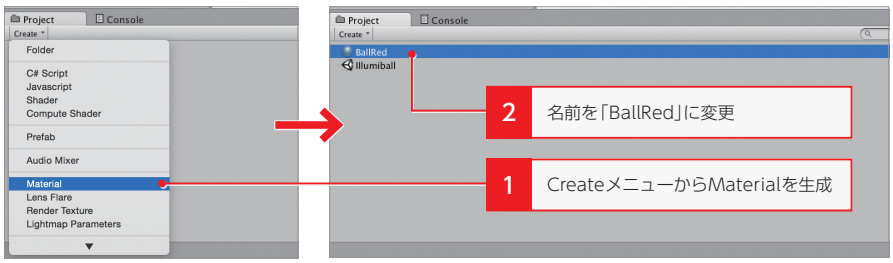

図3-1-10 マテリアルの作成

続いて、マテリアルのパラメータを調整します。まず、Albedoを赤色に設定します(1)。また、Emissionの値を1.0に設定します(2)。このEmissionは発光という意味で、そのマテリアル自身がどのくらい光を放っているかのパラメータになります。

さらに、Emissionの値を0より大きくすると、EmissionのColorを設定できるようになるため、ちょっと薄めの赤色に設定します(3)。RGB値的には(255, 100, 100)のように、赤以外の成分を100程度の値にしておきます。

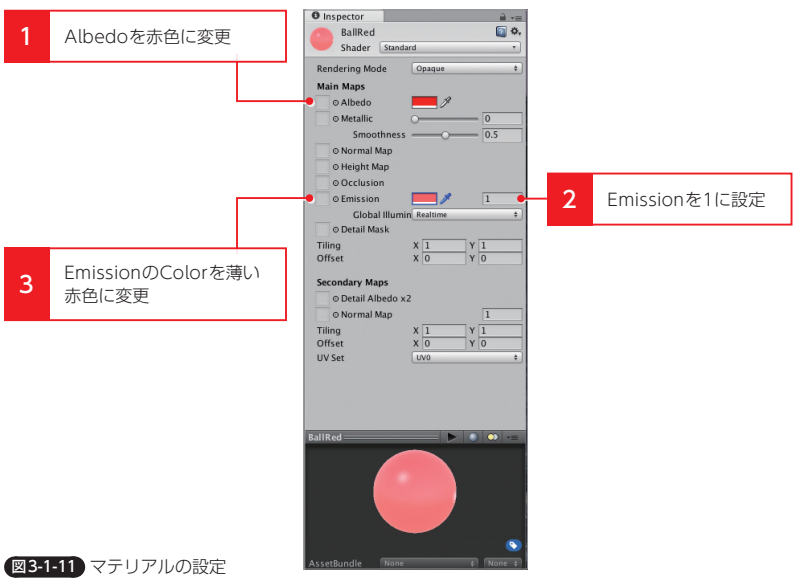

図3-1-11 マテリアルの設定

マテリアルの設定ができたら、BallRedオブジェクトに反映させましょう。BallRedオブジェクトに設定されているMesh RendererのMaterialsに、作成したBallRedマテリアルをドラッグ＆ドロップで設定します(1)。

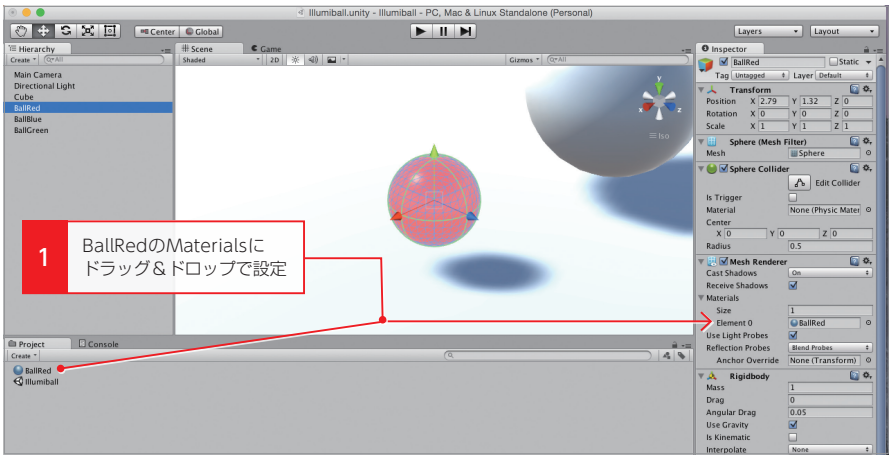

図3-1-12 マテリアルの反映

TIPS

マテリアルの反映

マテリアルは、直接Sceneビューのオブジェクトに対してドラッグ＆ドロップしても設定が可能です。

同じ手順でBallBlueマテリアル、BallGreenマテリアルも作成し各Ballオブジェクトに反映させます（1）。双方ともAlbedoを青と緑にし、EmissionはBallRedマテリアルと同じ1.0に設定してください。また、Emissionの色も薄めの青、薄めの緑を設定します。

このとき、Projectビューが煩雑にならないようにMaterialsというフォルダを新規に作成し、生成したマテリアルをまとめておきます（2）。

図3-1-13 BallBlue、BallGreenマテリアルの作成と反映

これでマテリアルの設定が完了です。しかしよく見てみるとマテリアルのEmission

パラメータを設定したはずなのに、なぜかボールが輝いているように見えません。これは、オブジェクト自身の色は設定されましたが、そこから発せられる光によってステージなどが照らされていないためです。

実はUnityのStandardシェーダーでは、Emissionパラメータによりほかのオブジェクトを照らす機能がありますが、その機能はシーンの再生中にまったく位置を移動させない、スタティックなオブジェクトだけに適用されるという制限があります。そのため、今回のように発光するボールがステージを転がったりする場合は、別の方法を利用する必要があります。この方法は以降で解説していきます。

ライティングの設定

ボールを発光させる前に、シーンのライトを調整しましょう。イルミネーションや夜景など、光輝くものは暗い場所のほうが見栄えがよくなります。そこで、シーン全体の光量を落として薄暗くします。

まずは、シーン全体を照らしているDirectional Lightを調整します。Hierarchyビューにあ るDirectional Lightを選択し(1)、明るさを表すIntensityパラメータを0.1程に下げます(2)。さらに影も必要ないので、Show TypeをNo Shadowに変更します(3)。

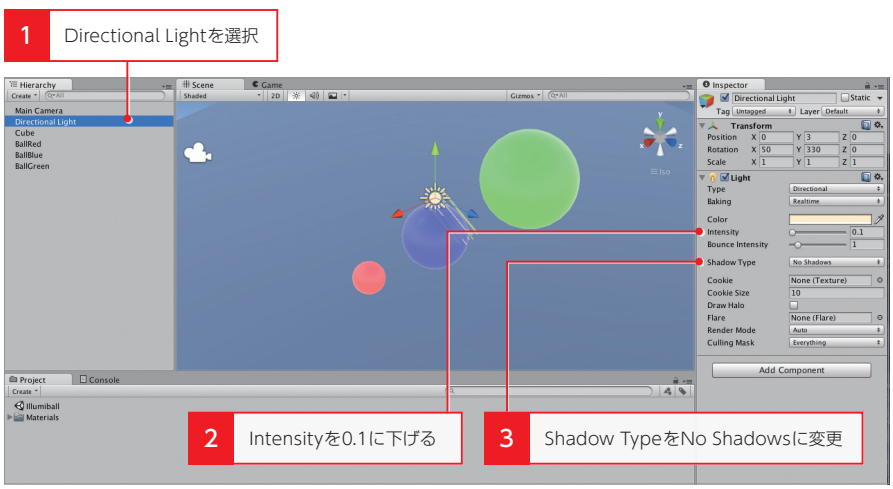

図3-1-14 Directional Lightの調整

さて、これでだいぶ薄暗くなりましたが、シーンに最初から存在している環境光を下げてもう少し暗くします。

まず、WindowメニューのLightingを選択し、Lightingウィンドウを開きましょう(1)。続いて、LightingウィンドウのAmbient Intensityを0.5くらいに下げます(2)。これで、かなり暗くなりましたね。ボールを発光させる準備は万端です。

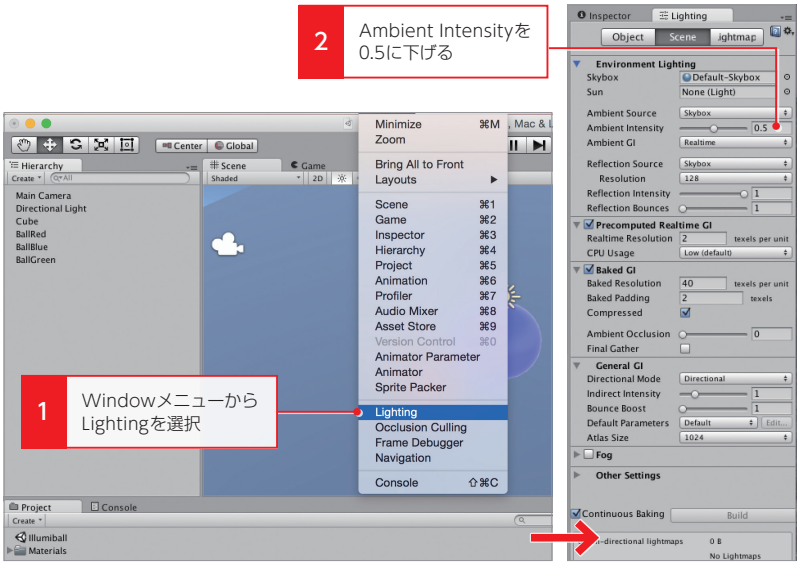

図3-1-15 Abmient Intensityの調整

Point Lightと親子関係の設定

次は、ボールをちゃんと発光させる仕組みを作りましょう。前述したように、シーン上を動き回るオブジェクトの場合、MaterialのEmissionパラメータを設定しただけでは、ほかのオブジェクトを照らすことができません。この場合は**Point Light**を利用します。

Point Lightは、Directional Lightと違い、ある1点の位置から球状に物体を照らすことができます。HierarchyビューのCreateメニューから、Light→Point Lightを選択しオブジェクトを生成してください(1)。

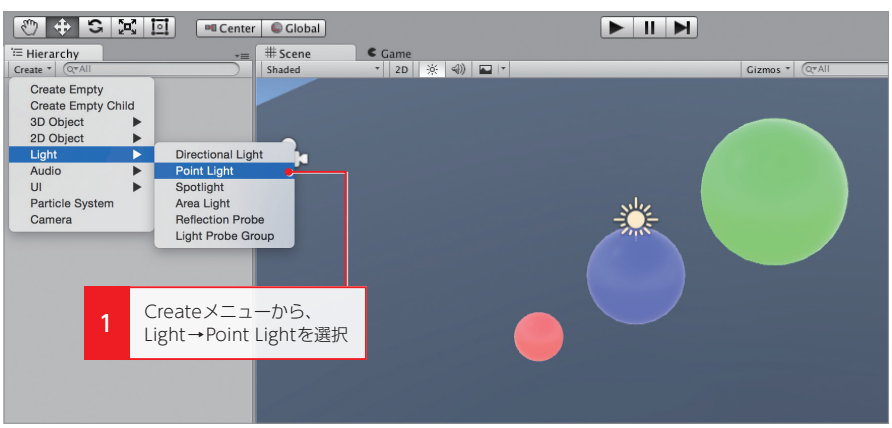

図3-1-16 Point Lightの生成

Point Lightオブジェクトを生成したら、BallRedオブジェクトとの間に**親子関係**を設定します。BallRedオブジェクトを「親」、Point Lightオブジェクトを「子」にしておくことにより、物理シミュレーションで動くBallRedオブジェクトに追従して動くようになります。

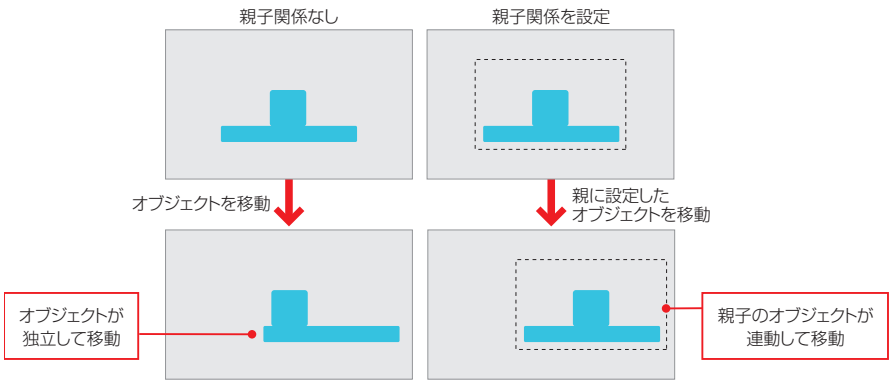

図3-1-17 親子関係を設定したオブジェクトの動き

つまり、親子関係を設定すれば、子は親のTransformの影響を受けるようになり、複数のオブジェクトを1つの大きいオブジェクトのように一括して扱うことができるようになります。

Point Lightオブジェクトを子に設定するには、HierarchyビューでBallRedオブジェクトにドラッグ＆ドロップします（1）。これにより、Point Lightオブジェクトが BallRedオブジェクトの下の階層に入り「子」として設定されます（2）。

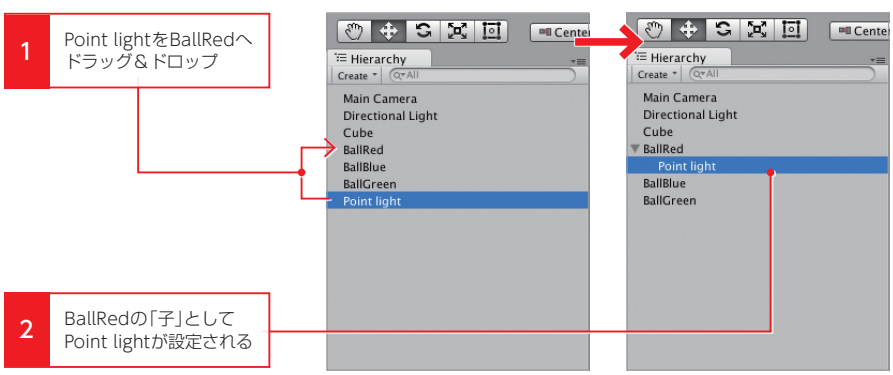

図3-1-18 親子関係の設定

親子関係が設定できたら、Point Lightオブジェクトのパラメータを調整しましょう。このとき、「子」に設定したPoint LightオブジェクトのInspectorビューのPositionの値は**Local Position（ローカルポジション）**を表しています。ローカルポジションは、

親のポジションからみた相対位置のことです。

反対に、シーン上の原点から見た絶対位置を**Global Position（グローバルポジション）**と呼びます。

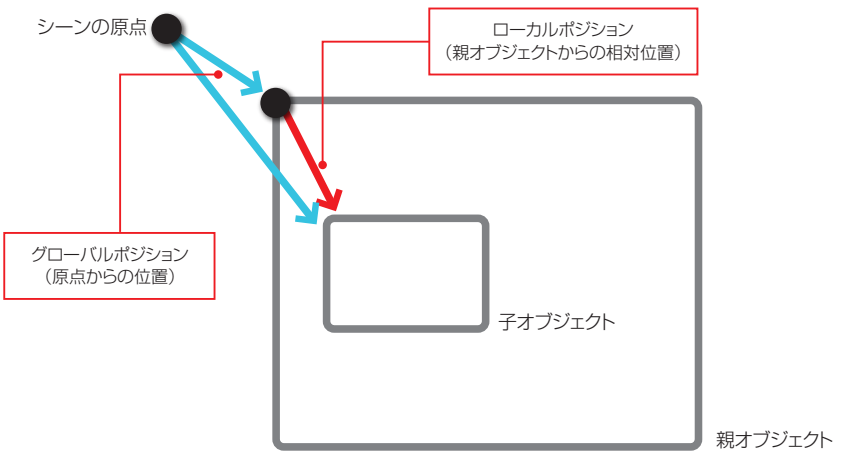

図3-1-19 グローバルポジションとローカルポジション

今回は、Point Lightオブジェクトを親のBallRedオブジェクトとまったく同じ位置に設定するため、InspectorビューでPositionを (0, 0, 0)に合わせます（1）。これにより、子であるPoint Lightオブジェクトの相対的な距離が0になり、親と同じ位置になります。

続いて、Lightコンポーネントのパラメータも変更します。Colorに関しては赤色を設定するのですが、完全な赤だとドギツイ色になってしまうので、マテリアルの設定時と同じように，少し薄めの赤に設定します（2）。さらに、照明の範囲を表すRangeの値は8を設定します（3）。

最後にRender Modeの設定を行います。**Render Mode**はそのライトが重要か重要でないかの設定です。一般的にライティングはかなり高負荷な処理になります。そのため、実行時の環境によっては、重要でないライトに関しては計算を簡略化することで、負荷を下げる対策がとられます。

デフォルトの設定ではAutoになっていますが、今回のゲームでは、ボールのライトは常に必要であるためImportantにしておきます（4）。

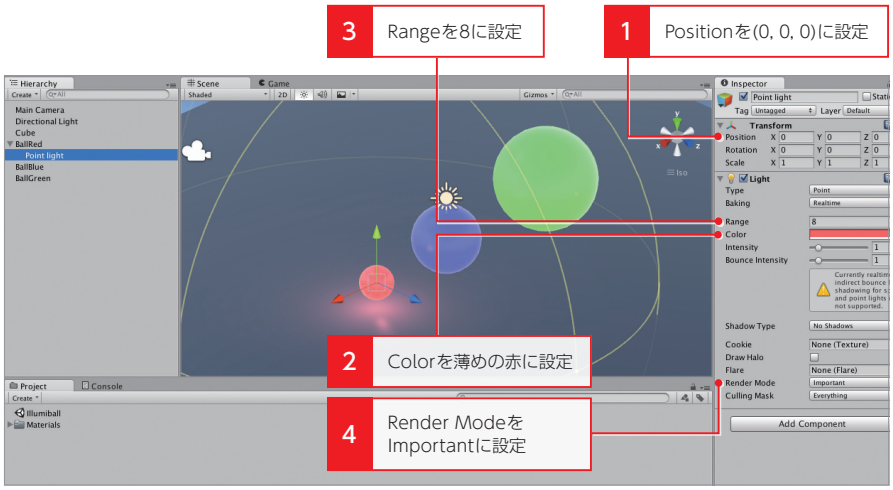

図3-1-20 Point Lightの設定

BallRedのライトの設定ができたら、BallBlueとBallGreenのライトも新規に作成します(1)。基本的なパラメータは同じですが、大きいボールの方がより広範囲を照らすように、BallBlueのPoint LightのRangeの値は16、BallGreenの方は24を設定しましょう(2)。

また、Colorにはそれぞれ、薄めの青と薄めの緑を設定し(3)、RenderModeも忘れずに変更します(4)。

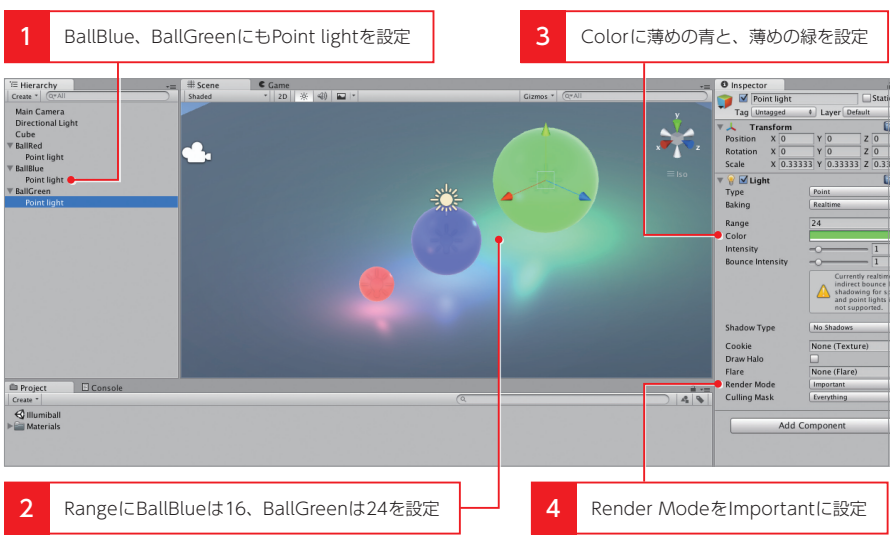

図3-1-21 BallBlue、BallGreenのPoint Lightの設定

これで、ボールの基本部分の作成は完了です。まだゲームとして操作はできませんが、Point Lightによるライティングがとても綺麗かと思います。

3-2 ステージの作成

次は、ゲームの土台となるステージを作っていきましょう。と言っても特殊なことはせず、Cubeオブジェクトを組み合わせて四角い箱を作ります。オブジェクトの大きさや位置などを細かく調整する必要があるため、Sceneビューの移動ツールよりも、主にInspectorビューでTransformを操作したほうが効率がよいかと思います。

ステージ作成の準備

まずは、床となる部分を作成します。すでに仮のステージとして配置したCubeオブジェクトがシーンに存在しているので、これを「Floor」と名前を変えて再利用することにします。

Cubeオブジェクトの名前を変更し(1)、Scaleを(10, 1, 15)に設定してください(2)。その後、Sceneビューの視点を真上から見下ろすようにして、Floorオブジェクトの範囲に収まるように、Ballオブジェクトの位置を調整しておきます(3)。

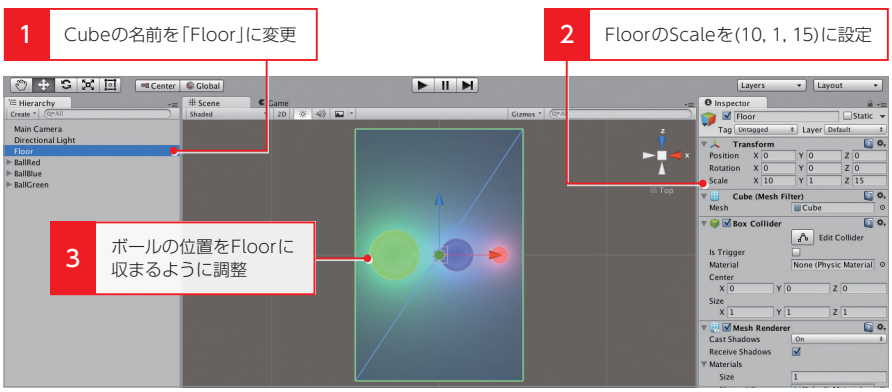

図3-2-1 Floorオブジェクトの設定

この後に，壁などのオブジェクトを作っていくのですが、このままだとステージに使うオブジェクトが増えて、Hierarchyビューが見づらくなってしまいます。ステージに使うオブジェクトを一括で管理できるように、ある1つのオブジェクトの子にまとめてしまいましょう。

ここでは**空のゲームオブジェクト**を利用します。空のゲームオブジェクトとは、Trasnformコンポーネントのみが存在しているもっとも基本的なゲームオブジェクトのことです。ここから必要なコンポーネントを追加して、目的の機能を組み立てることもできますし､今回のように複数のオブジェクトをまとめる時などに頻繁に利用します。

空のゲームオブジェクトは、HierarchyビューのCreateメニューからCreate Emptyを選択すると生成できます(1)。生成したら名前を「Stage」に変更してください(2)。また、空のゲームオブジェクトを利用するときは、子に何かを設定する前に

Positionが(0, 0, 0)、Rotationが(0, 0, 0)、Scaleが(1, 1, 1)の何も変更がない**Transformが初期化**されている状態にしておくことを推奨します(3)。こうすることで、相対的な位置や大きさを崩すことなく後の作業を行うことができます。調整ができたら、Floorオブジェクトをドラッグ&ドロップで子に設定してください(4)。

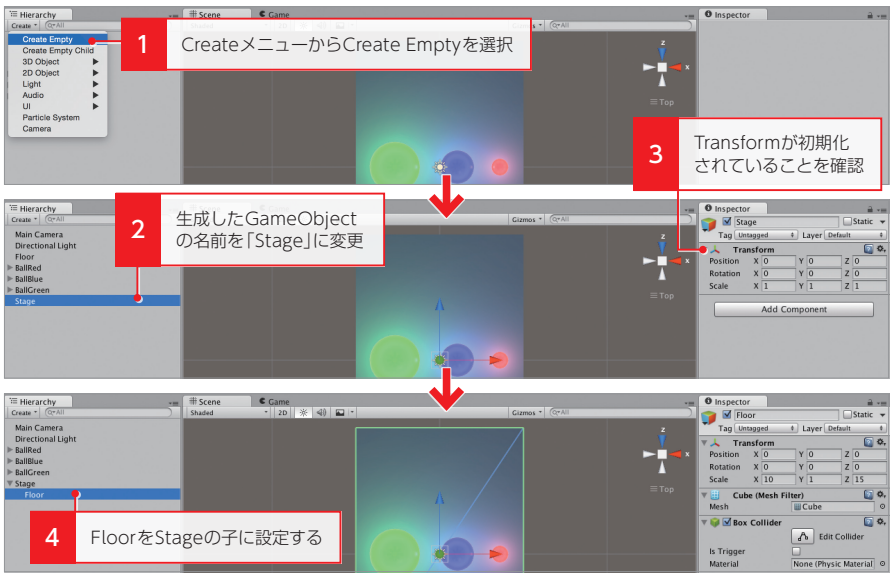

図3-2-2 空のゲームオブジェクトによるグループ化

周囲の壁の作成

Stageオブジェクトができたら、左右の壁を作りましょう。まず、Cubeオブジェクトを新たに生成し、名前を「WallRight」に変更後、Stageオブジェクトの子に設定します(1)。Positionは(5.5, 3.5, 0)、Scaleは(1, 8, 17)に設定し、Floorオブジェクトの横にぴったりとくっつくようにします(2)。

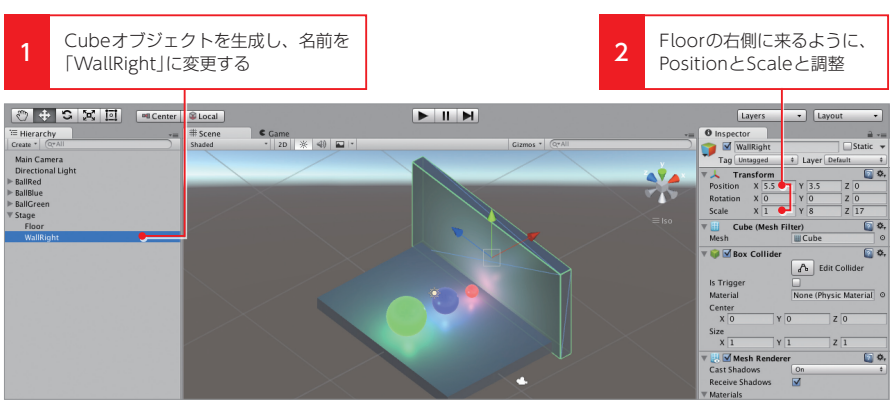

図3-2-3 Wallオブジェクトの設定

反対側の壁も同様に作成します。この時、1からオブジェクトを作成してもよいのですが、WallRightオブジェクトを選択後、EditメニューのDuplicate、もしくは「Command（Ctrl）+D」で複製できます（1）。位置が変わるだけで、大きさなどは同じなので、こちらのほうが楽でしょう。名前は「WallLeft」にし、Positionは(-5.5, 3.5, 0)に設定します（2）。

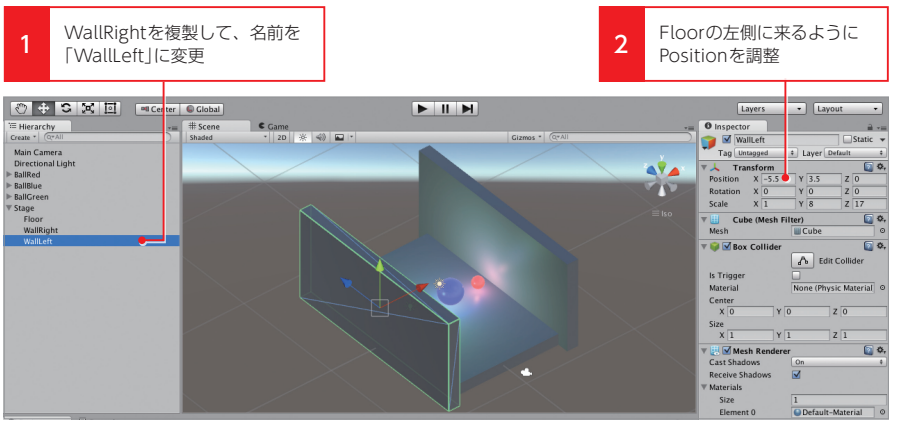

図3-2-4 Wallオブジェクトの複製

続いて、手前と奥の壁を作ります。こちらもCubeオブジェクトをベースに名前は「WallFront」と「WallBack」としてStageオブジェクトの子に設定します（1）。ともにScaleは(10, 8, 1)にし、PositionをWallFrontは(0, 3.5, -8)、WallBackは(0, 3.5, 8)に設定します（2）。設定が合っていれば、すっぽりと収まるはずです。

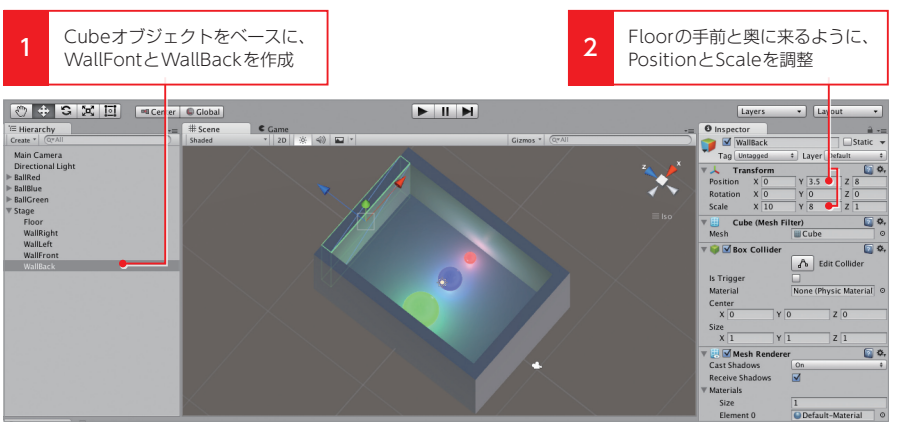

図3-2-5 手前と奥の Wallオブジェクトの設定

見えない天井の作成

最後に、天井を作ります。このゲームではボールが視点の手前、つまりシーンのy軸方向にも転がってくるため、フタをしておかないとボールがこぼれてしまいます。しかし天井をCubeオブジェクトで作ると、まったく中身が見えなくなってしまうため、「見えないがぶつかる物」を配置しておく必要があります。

このようなときは、単純にコライダーだけが追加されているオブジェクトを利用するのがよいでしょう。まずは、Stageオブジェクトを作成したときと同じように、空のオブジェクトを生成し、名前を「Top」に変更後、ステージオブジェクトの子に設定します(1)。そして、Topオブジェクトを選択した状態で、Componentメニューの Physics→Box Colliderをクリックし、コライダーを追加してください(2)。

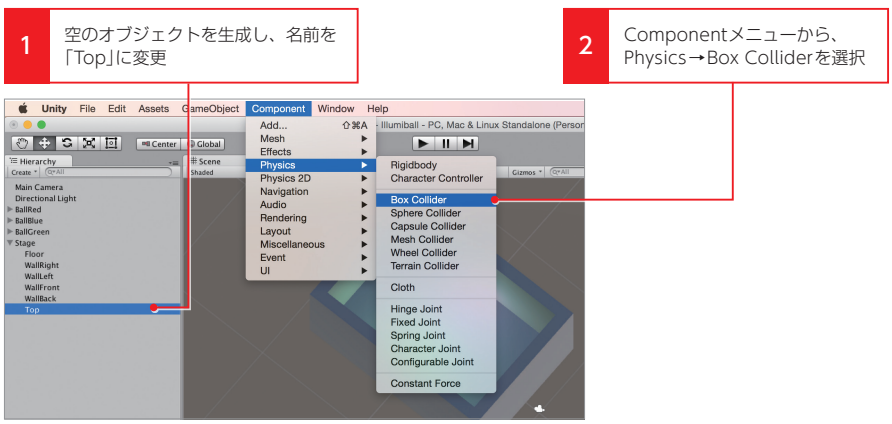

図3-2-6 Topオブジェクトの生成

Topオブジェクトの設定ができたら、大きさと位置を調整します。Scaleを床と同じ(10, 1, 15)に設定し、Positionをステージ上部にすっぽりと収まるように、(0, 7, 0)に設定しましょう(1)。これでボールがこぼれる心配はなくなりました。

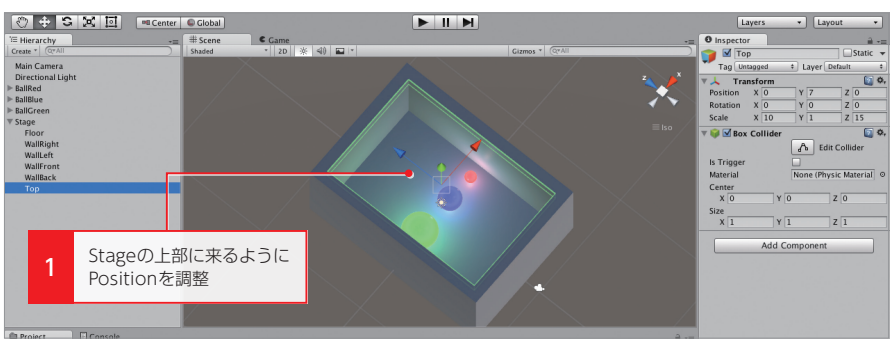

図3-2-7 Topオブジェクトの設定

障害物の設置

次は、ステージの中の障害物を作成します。具体的にはステージを二分する高さが低めの仕切りと、ボールの運び先を高い位置にするための柱の2つを設置します。

まずは、ゲーム開始時のことを考えてボールの初期位置を調整します。BallRedオブジェクトはステージの右側中央に、BallBlueオブジェクトは中央奥に、BallGreenオブジェクトは左手前に移動させます(1)。Sceneビューを真上から見るようにして、Positionツールを利用して微調整するとやりやすいかと思います。

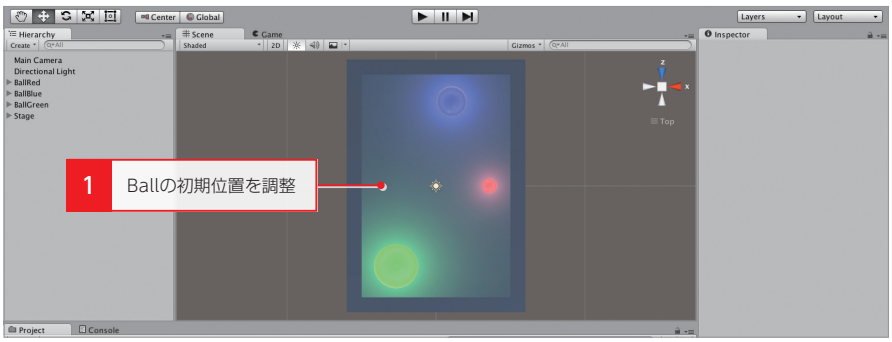

図3-2-8 Ballオブジェクトの初期位置の設定

次に仕切りを作ります。再度Cubeオブジェクトを作成し、名前を「Fence」に変更します(1)。Scaleは(10, 3, 2)に設定し、ちょうどBallRedとBallGreenを区切るように位置を調整します(2)。

図3-2-9 Fenceオブジェクトの設定

続いて柱を作ります。こちらもCubeオブジェクトを作成し、名前を「Post」にします。Scaleは (2, 4, 2)に設定し、Positionはステージの右奥にぴったりとくっつくように配置します。

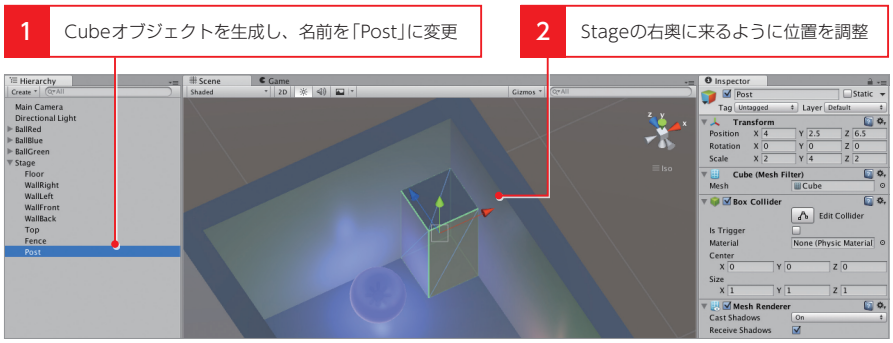

図3-2-10 Postオブジェクトの設定

カメラの調整

最後に、カメラの調整を行います。今回はカメラワークなど複雑なことは考えず、単純にステージを見下ろす形で固定することにします。

その前に実機での見え方を考慮するため、GameビューのAspect設定を縦長のiPhone5 TallやWSVGA Portraitなどに変更します。これらの解像度がメニューに出るように、1章を参考にしてBuild SettingsからiOSかAndroidにSwitch Platformしてください。

Gameビューの設定ができたら、カメラの調整を行います。まず、Main Cameraオブジェクトの Positionをステージの座標と同じ原点の(0, 0, 0)にし、真下を映すようにRotationを(90, 0, 0)に設定します(1)。その状態から、Positionツールを利用して、Main Cameraオブジェクトのy軸成分のハンドルを上方向にドラッグしてください。

Sceneビューの左下に表示されるカメラのプレビューやGameビューを見ながら、ステージが映りきる位置までカメラを引き上げます(2)。

図3-2-11 カメラの調整

3-3 重力の操作

次はいよいよ、スクリプトを作成してボールを動かす処理を作ります。実装の方針としては、個々のボールに処理を加えるのではなく、シーン中の重力をプログラムで変化させることで、全体を一度に操作することにします。

また、入力の方法としては、実際のおもちゃを再現するように、スマートフォンならではの**加速度センサー**を利用することにしましょう。

物理エンジンの設定と操作

物理演算処理は、それなりに負荷が高い処理です。そのため、速度が極小となり、しばらく動かないと判定されたオブジェクトは、CPUリソースの節約のため演算の対象外とされ、外部から力を加えられない限りは物理的な挙動をしなくなります。これを**スリープ**と呼びます。

Rigidbodyを持つオブジェクトがスリープ状態に入るのは、速度がある一定の閾値を下回ったときです。今回はオブジェクトの数も少ないですし、スリープするのは都合が悪いので、この設定を外してしまいましょう。EditメニューのProject Settings→Physicsをクリックします(1)。すると、InspectorビューにPhysics Managerのプロパティが表示されるため、Sleep Thresholdを0に変更します(2)。

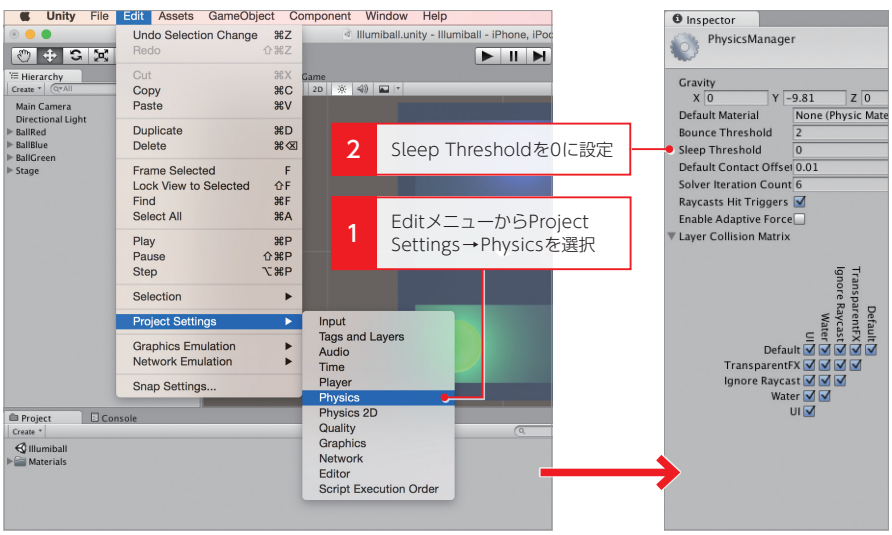

図3-3-1 Sleep Thresholdの設定

このPhysics Managerの設定項目では、ほかにも物理エンジンに関するいくつかの設定を行うことができます。そのなかに重力加速度を表すGravityという項目がありますが、スクリプトで制御する前に手動でこの値を変更し、どういった動きをするかを確

かめてみましょう。

まず、ゲームのプレビューを開始します（1）。そのまま、Physics ManagerのGravityのx成分やz成分を動かしてみます（2）。すると、ボールが再設定した量に応じて動くことが分かります（3）。これで、Gravityの値を変更することで、それなりにボールを操作できそうです。あとは、いま手動で行った操作をスクリプトで実現すればよいことになります。

ちなみに、ゲームのプレビュー後にGravityの値を操作していれば、プレビューを終了したときにGravityはデフォルトの (0, -9.81, 0)に戻ります。

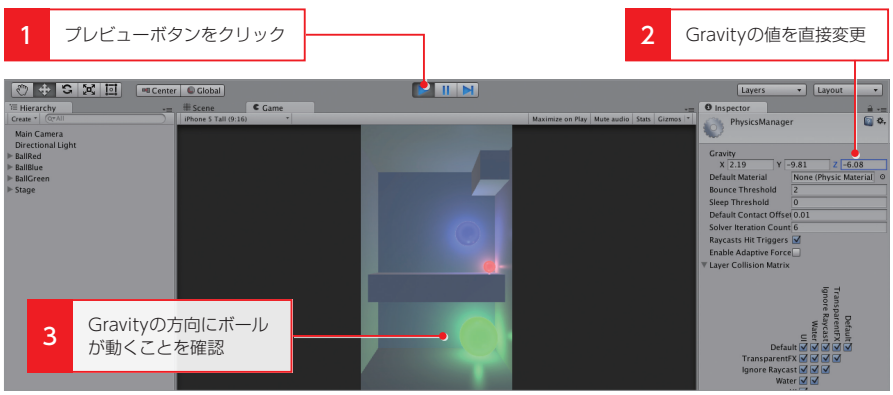

図3-3-2 Gravity操作のプレビュー

Scriptの作成

では、実際に重力を操作するスクリプトを作成してみます。ProjectビューのCreateメニューからC# Sciriptを選択し、新規スクリプトを生成します（1）。生成後は名前を「GravityController」に変更します。

このとき、スクリプトのファイルに関しても、後々増えることを考えて、マテリアルと同様にScriptsフォルダを作って整理しておきましょう（2）。

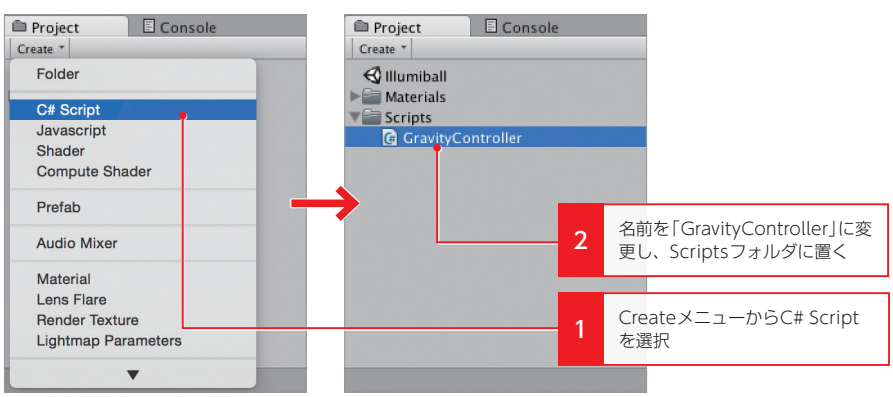

図3-3-3 GravityControllerスクリプトの作成

GravityControllerスクリプトが作成できたら、ダブルクリックしてスクリプトエディターを起動します。デフォルトでは、Unity標準の**MonoDevelop**が起動します。MonoDevelopを利用すると、コードの補完、プロジェクトを横断した検索や置換、Unityと統合されたデバッグなど豊富な機能を利用することが可能です。

ただし、Unity付属のMonoDevelopは**日本語の入力ができません**。コピー&ペーストで貼り付けることはできるのですが、支障がでる場合があるかと思います。日本語を入力するための作業は、巻末のAppendixで紹介していますので、参照してください。

TIPS

エディターの変更

MonoDevelopではなく使い慣れたエディターを使用したい場合は、Preferences→External Tools→External Script Editorから変更できます。

スクリプトを開くと、スクリプトと同名の**MonoBehaviour**を継承したクラスがすでに定義されています(1)。MonoBehaviourはUnityの根本を担うクラスで、シーン内で用いるクラスは基本的にMonoBehaviourを継承します*。最初から定義されている**Start関数**と**Update関数**は、MonoBehaviourが提供する特殊な関数です。

特に開発者が意識しなくても、Start関数はスクリプトが起動した後の初めてのフレームで一度、またUpdate関数は毎フレーム継続的にコールされます。

＊通常のObject型のクラスも定義できます。

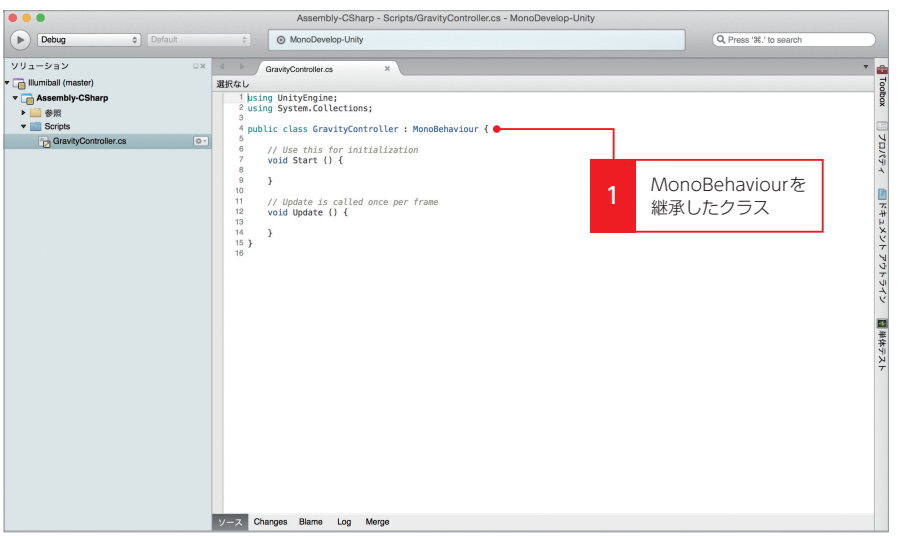

図3-3-4 MonoDevelop(スクリプトエディター)

COLUMN

フレームを理解する

ゲーム開発を行う上で、ゲーム内の時間と密接に関わる**フレーム**の概念を理解しておくことは、とても重要です。時間の経過に応じてどのようにオブジェクトが動くかを知らなければ、適切なプログラムを書くことはできません。

さて、ゲームが再生されているとき、その裏側ではどんな処理が行われているのでしょうか。実は、1秒間に何回も画面をリフレッシュして、少しずつオブジェクトの位置を変えた新しい画像を描写し直しています。オブジェクトだけを見ていると、移動というよりは、位置を少しずつワープさせる処理を繰り返しているだけなのですが、距離が短く、また時間も短いためなめらかに動いているように見えます。

これらのオブジェクトを制御する処理から画面描写までの一連の流れの単位をフレームと呼び、ゲームはこれらの膨大なフレームの積み重ねで進行していきます。

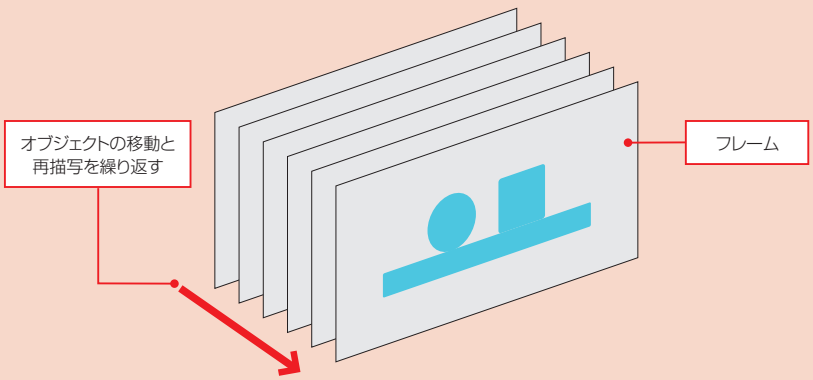

図 フレーム単位でのオブジェクトの描画

1フレームのなかでは、物理演算処理やオブジェクトの制御処理、画面の描写処理などのさまざまな処理が行われます。そのなかでシーン内にあるMonoBehaviourを継承したクラスでは、1フレームごとにUpdate関数が一度ずつ呼ばれます。つまり、このUpdate関数のなかにオブジェクトのパラメータを更新する処理を実装すれば、フレームの進行に合わせて移動させることなどができます。

またStart関数は、ゲームが起動したときや、そのオブジェクトが生成された最初のフレームの前で一度だけ呼ばれます。そのためStart関数は、オブジェクトの初期化処理などに利用できます。

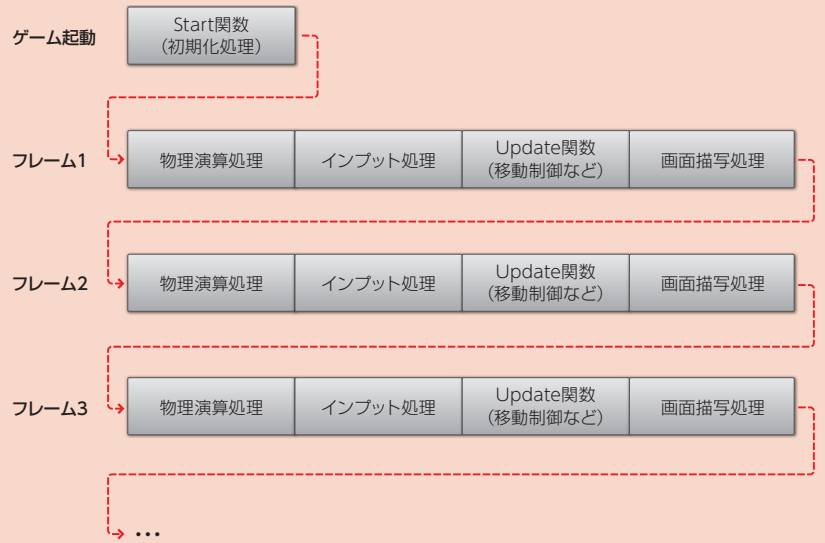

図 フレーム処理の流れ

1秒間に呼び出されるフレームの割合を**フレームレート**と呼び、FPS（フレームパーセコンド）という単位で表します。たいていのゲームでは30FPSあれば、それなりになめらかに動くゲームが再現できます。つまり、1フレームの時間は0.033秒と非常に短い時間であるため、これ以上の時間がかかる処理をUpdate関数内に含めると、処理落ちで画面がカクついてしまいます。

ゲームのなめらかさを追求する場合は、その前提で設計や構成を考えなければ、端末がものすごく熱くなったり、電池の消耗が激しくなるため注意が必要です。

キーボードでの操作

次はプログラムコードの実装に入りますが、いきなり加速度センサーを利用するのではなく、まずはキーボードのカーソルキーで操作ができるようにしましょう。このようにUnityエディター上でも実機と同じ、もしくは代替できる操作方法を用意しておくと、毎回実機でのチェックをせずに効率よく開発を進められます。

では、GravityControllerスクリプトに、以下のコードを実装します。

リスト3-3-1 GravityController.cs

```
using UnityEngine;
using System.Collections;

public class GravityController : MonoBehaviour
{
```

```
    //重力加速度
    const float Gravity = 9.81f;  ①重力加速度定数

    //重力の適用具合
    public float gravityScale = 1.0f;  ②重力のスケールパラメータ

    void Update ()
    {
        Vector3 vector = new Vector3();  ③重力ベクトルの初期化

        //キーの入力を検知しベクトルを設定
        vector.x = Input.GetAxis("Horizontal");
        vector.z = Input.GetAxis("Vertical");  ④カーソルキー入力の取得

        //高さ方向の判定はキーのzとする
        if (Input.GetKey("z"))
        {
            vector.y = 1.0f;
        } else {
            vector.y = -1.0f;
        }  ⑤高さ方向の入力の取得

        //シーンの重力を入力ベクトルの方向に合わせて変化させる
        Physics.gravity = Gravity * vector.normalized * gravityScale;  ⑥重力の変更
    }
}
```

キーボードの入力取得はInputクラスで、重力の操作はPhysicsクラスを介して行っています。これらをUpdate関数のなかで行うことで、各フレームごとに重力の方向を更新します。以下で、コードの内容をもう少し詳しく見てみましょう。

①重力加速度定数

重力の大きさを定義したものになります。リアルな物理シミュレーションをするわけではないので、値にあまり深い意味はありませんが、現実の世界と同じ9.81に設定しています。

②重力のスケールパラメータ

どのくらいの強さで重力を反映させるかの割合です。シーン上のオブジェクトのスケールにもよるのですが、ゲームの世界では少し重力を強めにかけたほうが見栄えや操作感がよい場合があります。その割合をInspectorビューで編集できるように、public修飾子で外部から設定できるようにしています。

③重力ベクトルの初期化

重力方向の決定のためのベクトルを初期化します。Vector3はx, y, zの3方向の軸を持つ、3D空間で頻繁に利用する基本的なベクトル構造体になります。x, y, zはそれぞれfloat値で、3つの値をいっぺんに扱える変数だと思えばよいでしょう。

④カーソルキー入力の取得

Inputクラスは入力全般の取得を行うクラスです。GetAxis関数では縦もしくは横に属す

る入力を、キーボードでもジョイスティックでも、その強さに応じて-1から1の範囲で取得できます。「Horizontal」の文字列を渡すと左右の値、「Vertical」の文字列で上下の値になるため、それぞれをxとzのベクトルに振り分けます。

⑤高さ方向の入力の取得

縦横の入力はGetAxis関数で取得できますが、高さ方向に割り当てるカーソルキーはありません。そこでキーボードの「z」キーを押したときに高さ方向に入力、つまり端末をひっくり反したということにします。

キーの入力の取得はGetKey関数でできるため、入力があったときにベクトルのyを反転させます。ちなみに入力がないときは、通常の重力がかかっている状態にするため-1を設定します。

⑥重力の変更

Physics.gravityの値は、以前手動で操作したPhysics ManagerのGravityパラメータと紐付いています。Physics.gravityはVector3であるため、事前に計算した入力による重力の方向ベクトルと、重力の定数、重力のスケールを掛け合わせたものを設定します。

このとき、キーボードからの入力ベクトルは、normalizedパラメータを介すことにより大きさを1に正規化します。これは、入力は重力の方向のみを規定し、重力の大きさは、事前に設定した一定の値にするためです。

スクリプトの利用とパラメータの設定

スクリプトが完成したら、実際に利用してみましょう。Unityではスクリプトは書いただけでは動きません。MonoBehaviourを継承したスクリプトを動かすためには、何かのゲームオブジェクトに**アタッチ**する必要があります。

まずは、空のゲームオブジェクトを生成し、スクリプトと同名の「GravityController」と名前を変更します（1）。次にGravityControllerスクリプトをドラッグして、GravityControllerオブジェクトにドロップします（2）。これでアタッチが完了し、シーンのプレビューを開始すると、自動で毎フレームUpdate関数が呼ばれるようになります。

アタッチ後は、スクリプト内でpublicとして定義した変数をInspectorビュー上で設定できます。Gravity Scaleを変更可能にしてあるため、値を6に設定します（3）。この値はゲームのプレビュー中でも即座に反映されます。プレビューを終了すると値がもとに戻ってしまいますが、実際の動きを見ながら調整が可能です。起動して動作チェックしてみましょう。

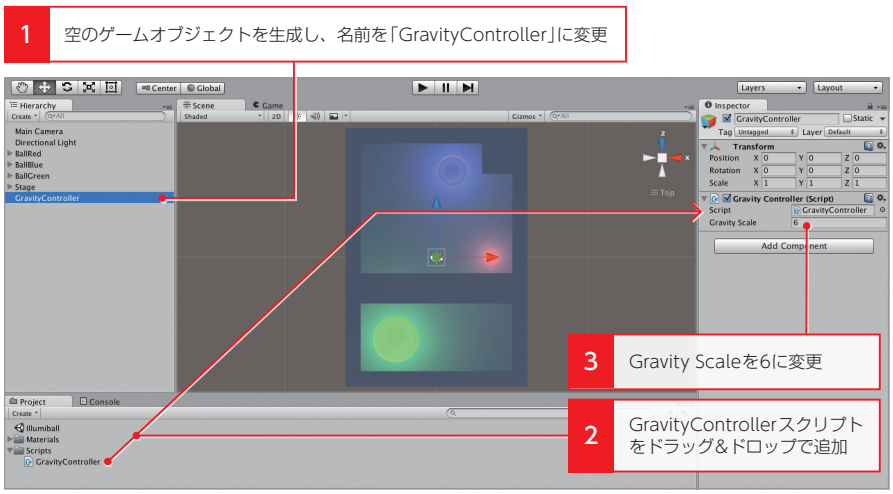

図3-3-5 GravityControllerの設定

すでに、お気づきの方もいるかもしれませんが、いま実装したスクリプトはRigidbodyやCollider、Lightなどのコンポーネントとまったく同じような使い方ができています。つまりスクリプトの実装により、自作のコンポーネントを作ったと言えます。このようにUnityでは、すでに用意されているコンポーネントや自作のスクリプトを複数組み合わせることで、さまざまな特徴を持つオブジェクトを実現していきます。

COLUMN

スクリプトのプログラミング言語

Unityでは複数の言語でのプログラミングが可能です。主にC#とJavaScriptで、Unity4まではBoo言語もありましたが、利用率の低さからサポート外になりました。

C#とJavaScriptですが、断然C#を利用することをお勧めします。UnityのJavaScriptは、純正なJavaScriptではなく、Unity界隈ではUnityスクリプトと呼ばれています。そのため、Webなどで利用するJavaScriptと互換性はまったくありません。JavaScriptで記述するほうが、やや簡単で1〜2割ほどコード量が少なくなる傾向がありますが構文などに大差はありません。

C#であればドキュメントも豊富ですし、Unity以外でも利用できる可能性があります。Unity公式の利用率調査でも 80%以上の人がC#を選択しており、サンプルコードやコミュニティにおいてもC#で語られることが多くなっています。

加速度センサーの利用

次は、実際にスマートフォンで加速度センサーを利用するように、Gravity Controllerを改良します。Input.accelerationで取得できるデバイスの傾きはInput.accelerationプロパティを利用することで簡単に取得できます。しかし、Unityの3D

空間上の座標軸と、デバイスの加速度ベクトルの座標軸は異なるため、ゲームごとに対応する向きに合わせて値を付け替えることが必要となります。

今回はデバイスの加速度のy軸とz軸がそれぞれ、ゲーム上のz軸とy軸に相当するため、そのように値をマッピングします。

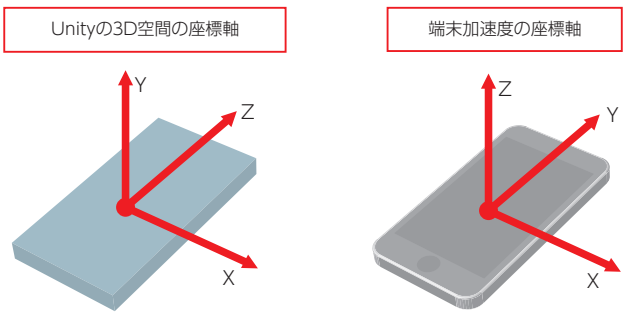

図3-3-6 端末の加速度センサーのベクトル

また、入力方法を加速度センサーに単純に置き換えてしまうと、今度はUnityエディター上でのデバッグができなくなってしまいます。そこで、Unityエディター上ではキーボードの入力、実機上では加速度センサーを使うといった具合に処理を切り分けることにします。これはApplicationクラスのisEditorプロパティを参照することで実現できます。

以上を踏まえて、以下のようにGravityControllerスクリプトを修正します。

TIPS

プラットフォームごとの処理の切り分け

Application.platformの値を見ることで、より細かい処理の切り分けができます。また、プラットフォームに依存したコンパイルを行うためのマクロ定義も利用できます。詳しくはUnity公式ドキュメント（http://docs.unity3d.com/ja/current/Manual/PlatformDependentCompilation.html）を参照ください。

リスト3-3-2 GravityController.cs（加速度センサーの利用）

```
using UnityEngine;
using System.Collections;

public class GravityController : MonoBehaviour
{
    //重力加速度
    const float Gravity = 9.81f;

    //重力の適用具合
    public float gravityScale = 1.0f;
```

```
    void Update ()
    {
        Vector3 vector = new Vector3();

        // Unityエディターと実機で処理を分ける
        if (Application.isEditor)
        {
            //キーの入力を検知しベクトルを設定
            vector.x = Input.GetAxis ("Horizontal");
            vector.z = Input.GetAxis ("Vertical");

            //高さ方向の判定はキーのzとする
            if (Input.GetKey ("z"))
            {
                vector.y = 1.0f;
            } else {
                vector.y = .1.0f;
            }
        } else {
            //加速度センサーの入力をUnity空間の軸にマッピングする
            vector.x = Input.acceleration.x;
            vector.z = Input.acceleration.y;
            vector.y = Input.acceleration.z;
        }

        //シーンの重力を入力ベクトルの方向に合わせて変化させる
        Physics.gravity = Gravity * vector.normalized * gravityScale;
    }
}
```

実機検証

GaravityControllerを加速度センサーに対応させたので、実機での動作チェックを行います。実機へビルドするためには、プロジェクト全体に対して最低限設定する項目があります。これらは**Player Settings**から設定ができるため、Editメニューの Project Settings→Playerをクリックし、Inspectorビューに設定を表示します(1)。その後、ビルドするプラットフォームのアイコンのタブに切り替えてください。

Player Settingsが表示されたら、まずOrientationを設定します。Orientationとは端末の向きのことで、横持ちで遊ぶか縦持ちで遊ぶかの設定になります。デフォルトではAuto Rotation、つまり、現在の端末の向きによって自動で変更する設定になっています。

今回のボールを転がすゲームは、自動でビューが回転すると都合が悪いため、縦持ち限定にしておきます。OrientationはResolution and Presentationメニューの Default Orientationパラメータで設定ができるため、縦持ちを表すPortraitに変更します(2)。

次にBundle Identifierを設定します（3）。Bundle Identifierは世界中のアプリを一意に識別するためのIDです。Other SettingsメニューのBundle Identifierから設定できるため、ほかのアプリと重複しないように好きな文字列を設定します。

ちなみにこれらのOrientationとIDの2つの設定は、iOS、Android共通でどちらかを変更すると、別のタブでも同じように反映されます（4）。

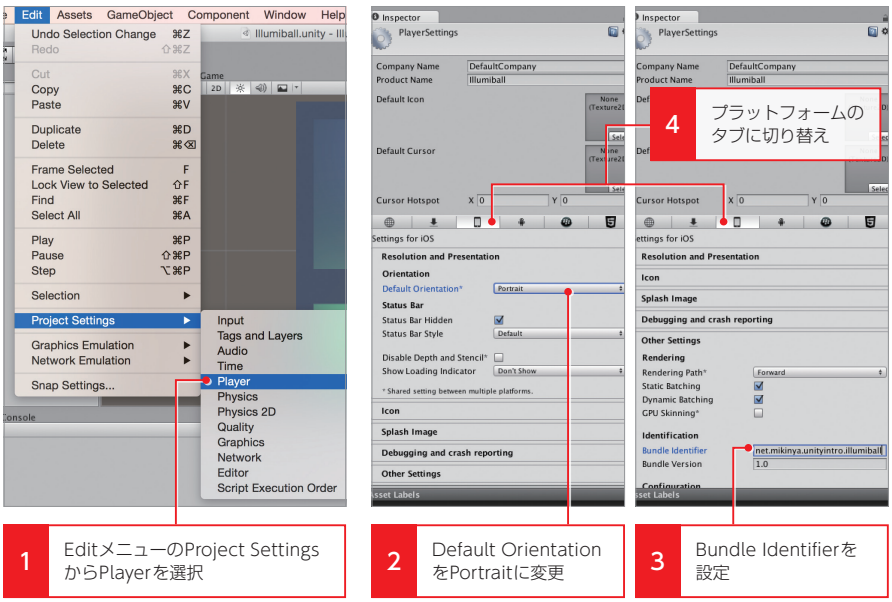

図3-3-7 ビルドの設定

プロジェクトの設定ができたら、ビルドしてみましょう。その前にどのシーンをビルドに含めるかの設定が必要です。これはFileメニューのBuild Settingsから行えます。今回はシーンが1つしかないため、Add Currentボタンを押すか、Projectビューのシーンファイルを Scene In Buildのパネル内にドラッグ＆ドロップで追加してください（1）。

その後、端末を繋いだ状態でBuild and Runをクリックすればビルドが実行され、実機での再生が始まるはずです（2）。起動後は、端末を傾けるとその方向にちゃんとボールが移動するか確かめてみましょう。

TIPS

Scene In Buildの無効化と削除

一度Scene In Buildに追加したシーンは、チェックボックスを外して無効化するか、「Command（Ctrl）+Delete」で削除することができます。

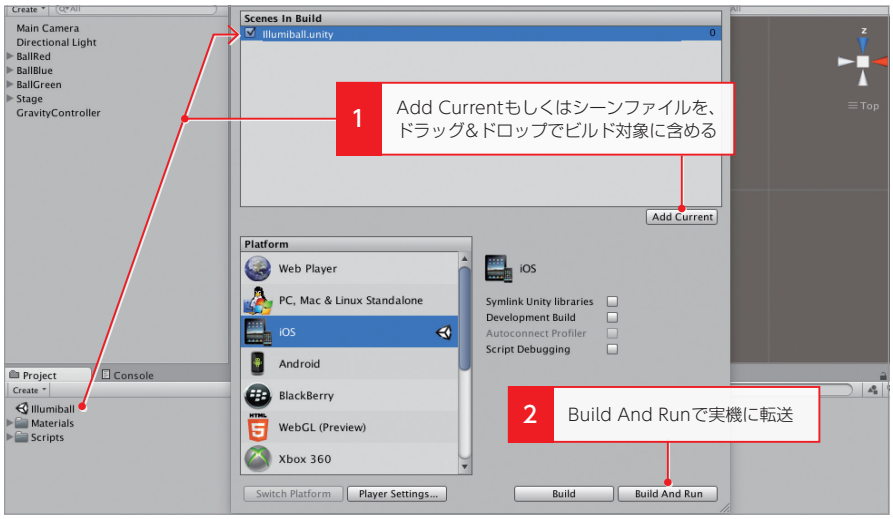

図3-3-8 ビルドシーンの追加とビルドの実行

COLUMN

プロジェクトフォルダの中身

Unityプロジェクトが入っているフォルダのなかには、以下のフォルダが存在しています。簡単に役割を紹介します。

Assetsフォルダ

UnityエディターのProjectビューと紐付いており、3Dモデルやマテリアル、スクリプトなどのアセットファイルが格納されます。

Libraryフォルダ

Switch Platformによりコンバートされた、アセットファイルのプラットフォーム用データが格納されます。また、オブジェクト間のリンク情報が格納されます。

ProjectSettingsフォルダ

Unityエディターやビルドに関する設定情報が格納されます。

Tempフォルダ

Unityが起動中に利用する一時的なデータが格納され、Unityが終了するときに自動的に削除されます。

また、MonoDevelopでスクリプトを開いた場合は、プロジェクトフォルダの直下にAssembly-CSharp.csprojや「プロジェクト名」.slnなどのファイルが自動生成されます。これらはMonoDevelop用のプロジェクトファイルや設定ファイルであり、万が一消してしまっても再生成されるため問題ありません。