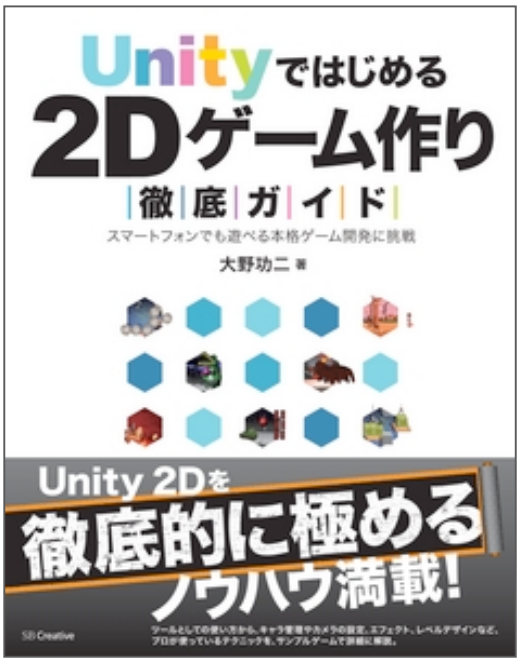
スマートフォンでも遊べる本格ゲーム開発に挑戦

Unity ではじめる
2D ゲーム作り徹底ガイド

入門＆徹底 UnityPRO ガイド

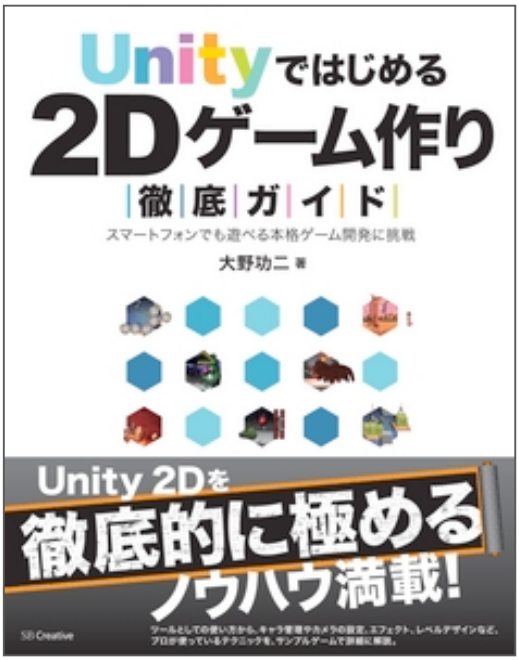


O Planning

2014/10/27 O-Planning

はじめに

「Unity ではじめる 2D ゲーム作り徹底ガイド」をお買い上げいただきました皆様。
本当にありがとうございます（また、これからご購入を予定されている方は、ぜひ、書店などでお手に取ってみてください）。



Amazon
「Unity ではじめる 2D ゲーム作り徹底ガイド 」
スマートフォンでも遊べる本格ゲーム開発に挑戦
http://www.amazon.co.jp/dp/4797376708/

「Unity ではじめる 2D ゲーム作り徹底ガイド」では、Unity2D を使用した本格 2D ゲーム制作を紹介しました。しかし、「ちょっと難しい」と思われる方もいれば、「もっと Unity2D の機能を知りたい」と思われる方もいらっしゃるでしょう。
そこで、付録として**「入門＆徹底 UnityPRO ガイド」**を収録しました。よろしければ、ぜひご一読ください。

※この書籍は、Unity4.5.4 を元に執筆しております。
※この書籍の無断で配布を禁止します。

©2014 Kouji Ohno

目次

第 1 章
Unity 入門（初心者向け）

1.1. Unity のインストール

Unity2D を使って、ゲーム開発に挑戦してみましょう！
まずは、Unity2D が実装された Unity4.x をインストールしましょう。

Unity のダウンロードとインストール

Unity は、Unity Japan のホームページからダウンロードできます。

Unity Japan
http://japan.unity3d.com/

Unity Japan のページを開いたら、サイト右上の「Download」リンクをクリックして、ダウンロードページから Unity をダウンロードします（図 A01_01_001）。

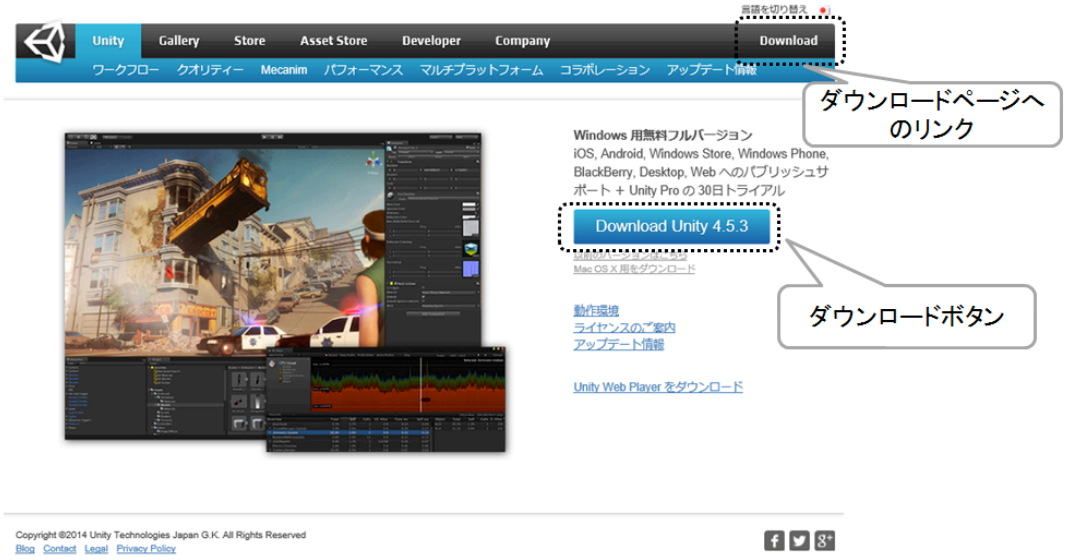


図 A01_01_001 Unity のダウンロードページ

ダウンロードされた 1GByte ほどのファイルは、実行形式のインストーラです。
インストーラを実行して、Unity をインストールします。
なお、Windows の場合は、ログインしているユーザー権限によっては、Unity が正常に動作しない場合があります。そのような場合は「管理者権限」のあるユーザーでインストールしてみてください。

また、古いバージョンの Unity がある場合は、上書きされてしまいます。

もし、古いバージョンの Unity を残したまま新しいバージョンの Unity インストールしたい場合は、古いバージョンの Unity フォルダ名を変更するか、新しいバージョンのインストーラで指定する「インストール先フォルダ」の場所を、旧 Unity とかぶらないように指定してください。これで、新旧バージョンの違う Unity を 1 つのパソコンで一緒に使うことができます。

ただし、新しいバージョンの Unity で、古いバージョンの Unity で作成したプロジェクトファイルを開くと、自動的にプロジェクトファイルのバージョンがアップデートされます。古いバージョンの Unity では読めなくなる可能性があるので注意が必要です。

Unity のユーザー登録

Unity がインストールでできたら、起動してみましょう。

Unity を起動すると、最初にアクティベート画面が表示されます（図 A01_02_002）。

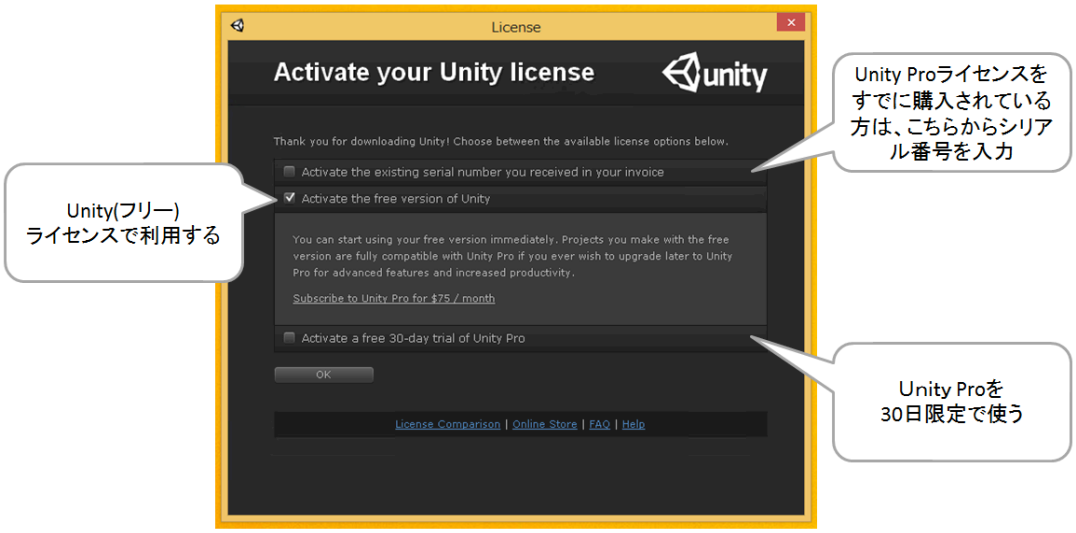


図 A01_02_001 Unity のアクティベート画面

Unity はユーザー登録しなくても、30 日間は Unity PRO ライセンスがそのまま使えます。ユーザー登録すると、PRO 版の機能は使えなくなりますが、30 日後も無料で Unity を利用することができます。

Unity のライセンス

Unity のライセンスには、「Unity」と「Unity PRO」の 2 つのライセンスがあります。

「Unity」ライセンスは、ユーザー登録をすることによって無料で Unity を使えるライセンスです。Unity の一部の機能は使えませんが、それでも、Unity で作成したゲームを商用

目的で配布することができます。ただし、このライセンスでは、利用者の年間の総収益や総予算が US$ 100,000 までとなっています（2014/10/05 時点での規約内容です）。それ以上の収益や予算がある場合は、Unity Pro ライセンスを購入しなければなりません。

Unity Pro ライセンスは、Unity のすべての機能が使えるライセンスです。Unity ホームページにある Store ページから Unity PRO ライセンスを購入できます（月々75$で利用できるサブスクリプションもあります）。

この 2 つのライセンスの詳細については、Unity のライセンスページをご覧ください。

Unity Licenses
http://japan.unity3d.com/unity/licenses

Unity ソフトウェアライセンス契約 バージョン 4.x
http://japan.unity3d.com/company/legal/eula

Unity Store
https://store-jp.unity3d.com/

また、この二つのライセンスで使える Unity の機能の違いを、簡単に次の表にまとめてみましたので参考にしてみてください（表 1.1.1.1）。

表 1.1.1.1 Unity と Unity PRO ライセンスの違い

項目	Unity PRO	Unity
費用	有料	無料
昨年度の売上が US$100,000 を超える企業や団体による使用およびライセンス取得	○	×
Unity の基本機能	○	○
3D グラフィックスの基本機能	○	○
Shuriken パーティクルシステム	○	○
リアルタイム Spot/Point シャドウやソフトシャドウなどの拡張グラフィック機能	○	×
静的バッチング、オクルージョンカリングなどの高速表示機能	○	×
HDR、トーンマッピング、フルスクリーンポストエフェクト、デデファードレンダリングなどのハイクオリティな画質を実現する表示機能	○	×
アニメーションの基本機能(Mecanim など)	○	○
Mecanim IK などのアニメーションの拡張機能	○	×
アプリケーションの作成	○	○
LOD(三次元オブジェクトの効率的な表示機能)	○	×

ナビメッシュやパス検索などの AI に関する基本機能	○	○
RakNet によるマルチプレイヤーネットワーク	○	○
オーディオの基本機能	○	○
フィルタ処理などのオーディオの拡張機能	○	×
MonoDevelop によるコードの編集機能	○	○
個々のプラットフォームで動作するネイティブプラグインの作成、実行（Unity ライセンスでも、iPhone や Android などのモバイル端末では利用可能です）	○	△
プログラムの処理負荷を測定するプロファイラ・GPU プロファイリングなどの機能	○	×
アプリケーション起動時に表示される Unity ロゴ・スプラッシュ画面のカスタマイズ	○	×
アセットバンドル（作成したプログラムが、ネットからゲーム中で使用するシーンやグラフィックなどのアセットをダウンロードできる機能）	○	×

分かりやすく解説すると、市販ゲームのように美しくリアルなグラフィック画像を表示するための機能や、高速化のための機能、高度な AI をサポートする機能などが有料の Unity PRO ライセンスで使えます。ただし、これらの機能がなくても、そこそこ市販ゲームに近いグラフィックを実現することは十分可能です。なお、様々な機能やデータを購入できる「アセットストア」を利用することで、無料の Unity ライセンスでは実現できない機能でも、一部分だけ限定的に実現できることもあります。

Unity では大きなバージョンアップの際に、新機能追加に伴い、よく使われる便利な機能が無料の Unity ライセンスでも使えるように提供されることが何度もありました。筆者の感覚ですが、無料の Unity ライセンスでも Unity の機能の 70%くらいは使えて、しかもゲーム開発のおもしろさは 120%体験できます。

本書で扱うサンプルゲームは、無料の Unity ライセンスだけで使える機能で作っています。Unity PRO の機能は、本格的なプロのゲーム開発者向けの機能がメインですので、まずは、無料の Unity ライセンスを使い、どうしても Unity PRO の機能が使いたくなったり、開発したゲームが高額の収益を上げたときに Unity PRO を購入すると良いでしょう。

作ったゲームが大ヒットしたら、ぜひ Unity PRO ライセンスを購入してくださいね。

コラム：Windows8/8.1 で Unity が起動しない・不安定な場合

Unity は英語用アプリケーションとして作成されました。そのため、一部の環境（特に Windows8/8.1）では Unity が不安定になることがあります。現在、Unity Japan によって日本語対応が薦められていますが、それまで、Unity が起動しなかったり不安定になった場合は、下記のことを試してみてください。

・DirectX11 モードで起動する

DirectX の動作の問題によって Unity が起動しない場合があります。このような時は、Unity を DirectX11 モードで起動すると改善されることがあります。まず、Unity をインストールしたフォルダから、”Unity.exe”を見つけて、適当な場所にショートカットを作成します。作成したショートカットを右クリックしてプロパティーダイアログを開き、ショートカットタグの「リンク先(T):」項目の最後に、”-force –d3d11”と起動オプションを追加します。次回からは、このショートカットで Unity を起動します。

・日本語アカウント名の場合は、英語名アカウントを作って起動する

Windows を日本語アカウント名にしていると、ユーザーフォルダや設定情報などに日本語が含まれるため、Unity や Unity に付随するプログラムが誤動作を起こすことがあります。このような場合は、「コントロールパネル」からユーザーアカウントを編集して、英語名アカウントを作成し、このアカウントで再ログインしてみてください。

・プロジェクトフォルダを日本語名から英語名に変更する

プロジェクトフォルダや、そのパスの一部に日本語が使われている場合、Unity や Unity に付随するプログラムが誤動作を起こすことがあります。プロジェクトフォルダや、そのパスがすべて英語名になるように変更してみてください。

・Unity をプロジェクトを読まないで起動する

Unity を起動しても、プロジェクトの読み込みで失敗する場合は、プロジェクトを読まない状態で Unity を起動してください。アイコンをダブルクリックした後、素早く ALT キーを入力すると、Unity はプロジェクトファイルを読まずに起動します。正常に起動できたら、プロジェクトファイルに問題がある可能性があります。プロジェクトフォルダ名、およびパス名などの変更を試してみてください。

・Asset Store にアクセスできない

一部のウィルス対策ソフトや、ネット設定では、Asset Store に接続できないことがあります。

下記のページにて、対策法が示されていますので、参考にしてみてください。

Windows 版 Unity でよくあるトラブルと対処方法

http://japan.unity3d.com/blog/?p=1346

1.2. Unity の基礎

Unity がインストールできたら、実際に使ってみて基本操作について慣れておきましょう。
（紹介するサンプルは、Sample1_1 フォルダにあります）

プロジェクトを作る

Unity でゲームを制作する場合、まずは「プロジェクト(Project)」を作ります。
Unity を起動したら、メニューの「File」の「New Project」を選択してください。
「Project Wizard」ダイアログが開きます（図 A02_01_001）。

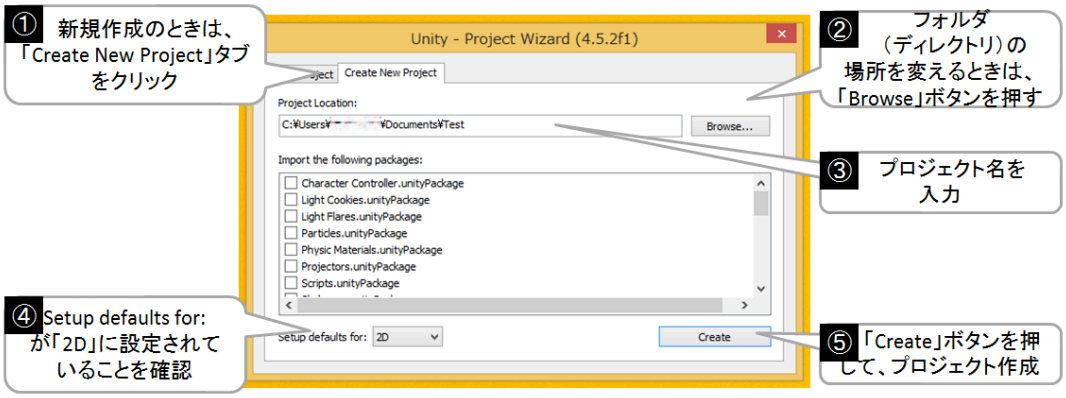


図 A02_01_001 「Project Wizard」ダイアログ

Project Wizard ダイアログの「Project Location」にプロジェクト名を入力して、「Create」ボタンを押します。プロジェクト名は適当で構いません。なお、「Setup defaults for」の項目では、3D と 2D が選択できます。本書では、Unity2D の機能を使ってゲームを作るため「2D」を選択します。

Unity の画面構成

プロジェクトが作成されると、Unity のエディタ画面が表示されます。以後、本書では、この画面を「Unity エディタ」と呼びます（図 A02_02_001）。

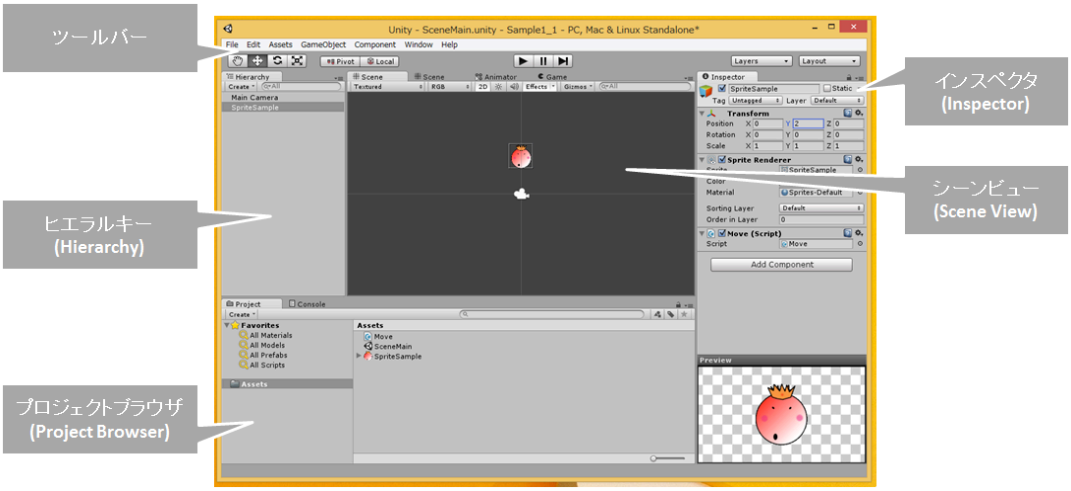


図 A02_02_001 Unity の画面構成

まずは、Unity エディタの各ウィンドウの機能について説明しましょう。

- **シーンビュー(Scene View)**
  プレイヤーキャラや敵、背景などの設置や位置調整をすることができるビューです。
  ゲーム実行中でも、シーンビューにはその状態がリアルタイムで反映されます。
  また、ゲーム実行中でも、シーンビューで編集が可能なため、ゲームをテストプレイしながら「この敵は、もうちょっと後ろに移動させたい」と思ったら、すぐに試すことができます。テストプレイ中の変更をシーンに反映させることはできませんが、ゲーム実行中のオブジェクトを CTRL+C でコピーして、ゲーム終了後に CTRL+V でシーンにペーストすることができます。

- **ゲームビュー(Game View)**
  ゲームの実行画面を表示するビューです。図 A02_02_001 のような Unity のデフォルトのレイアウトの場合は、ゲームを実行するとゲームビューに切り替わります。

- **プロジェクトブラウザ(Project Browser)**
  ゲーム開発に必要なシーン、グラフィック、サウンド、スクリプトなどを格納しているフォルダを閲覧できるブラウザです。作成したプロジェクトフォルダを Windows のエクスプローラや Mac のファインダーで開くと”Assets”というフォルダがあり、このフォルダの中身がそのままプロジェクトブラウザの実体となっています。
  ただし、この Assets フォルダは、プロジェクトブラウザ以外で、「ファイル・フォルダの移動」「ファイル・フォルダの削除」「ファイル・フォルダのリネーム」を行うと、Unity がプロジェクトを構成するために管理している「メタデータ」が破損する原因となります。ファイル・フォルダの移動・削除・リネームなどは、必ずこのプロジェクトフォルダで行ってください。
  なお、ファイルの追加に関しては、メニューの「Assets」から「Import New Asset」

で追加する他に、直接、プロジェクトブラウザにファイルをドラッグ＆ドロップすることでも可能です。ファイルの上書きについては、エクスプローラやファインダーからしか行えず、プロジェクトブラウザに同名のファイル名をドラッグ＆ドロップしても、名前が変更されて新規ファイルとして追加されます。

・ **ヒエラルキー(Hierarchy)**
Unity のゲームシーンは、キャラから背景まで、すべて**「ゲームオブジェクト(GameObject)」**によって作られ構成されます。このゲームオブジェクトを、階層構造で編集できるのが「ヒエラルキー」です。キャラや敵を追加したり編集したりするときは、このヒエラルキーで各ゲームオブジェクトを選択します。

・ **インスペクタ(Inspector)**
インスペクタは、その時点で選択されているゲームオブジェクトの詳細情報を表示・編集するビューです。ゲームオブジェクトの位置情報から、グラフィック・サウンドの設定、プログラムの仕方によっては体力のゲームパラメータなども、インスペクタで設定・編集することができます。また、シーンビューと同様にゲーム実行中でも、その情報がリアルタイムに変化するため、ゲームをテストプレイしながらデバッグやゲームの難易度調整などをすることが可能です。

この他にも、Unity のビューには、プログラムのエラー情報などが見られる「コンソール（Console）」、ゲームオブジェクトのアニメーションを編集できる「アニメーションビュー（Animation）」などがあります。これらのビューは、CTRL+（数値）のショートカットが割り当てられており、CTRL+1 ならシーンビューがアクティブになります（巻末のショートカットリストを参照）。

また、各ビューは、Unity4.3 以降であれば、SHIFT+SPACE キーで全画面表示にできます（Unity4.2 までのバージョンでは SPACE キーのみです）。

なお、Unity の画面構成は、「Window」メニューの「Layouts」から自分にあったものを選択できます。Unity エディタの画面右上の「レイアウトドロップダウン」からでも切り替え可能です（図 A02_02_002）。

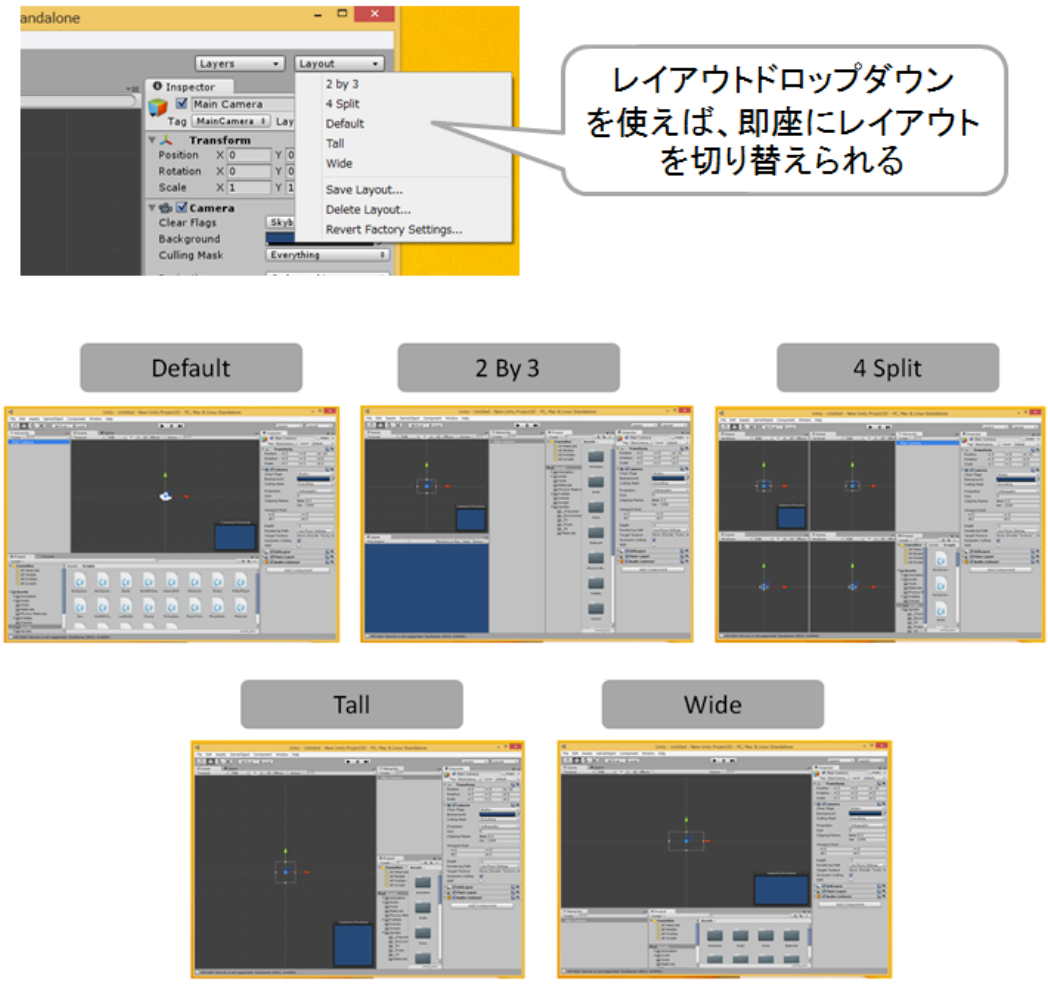


図 A02_02_002 Unity のレイアウト

Unity4.x では、図の「Default」のレイアウトが標準となりました。もし、過去の Unity バージョンのレイアウトで使いたい方は、「4 Split」を選択すると良いでしょう。

また、個々のビューは移動したり大きさを変えたり、ドッキングさせることが可能です（図 A02_02_003）。

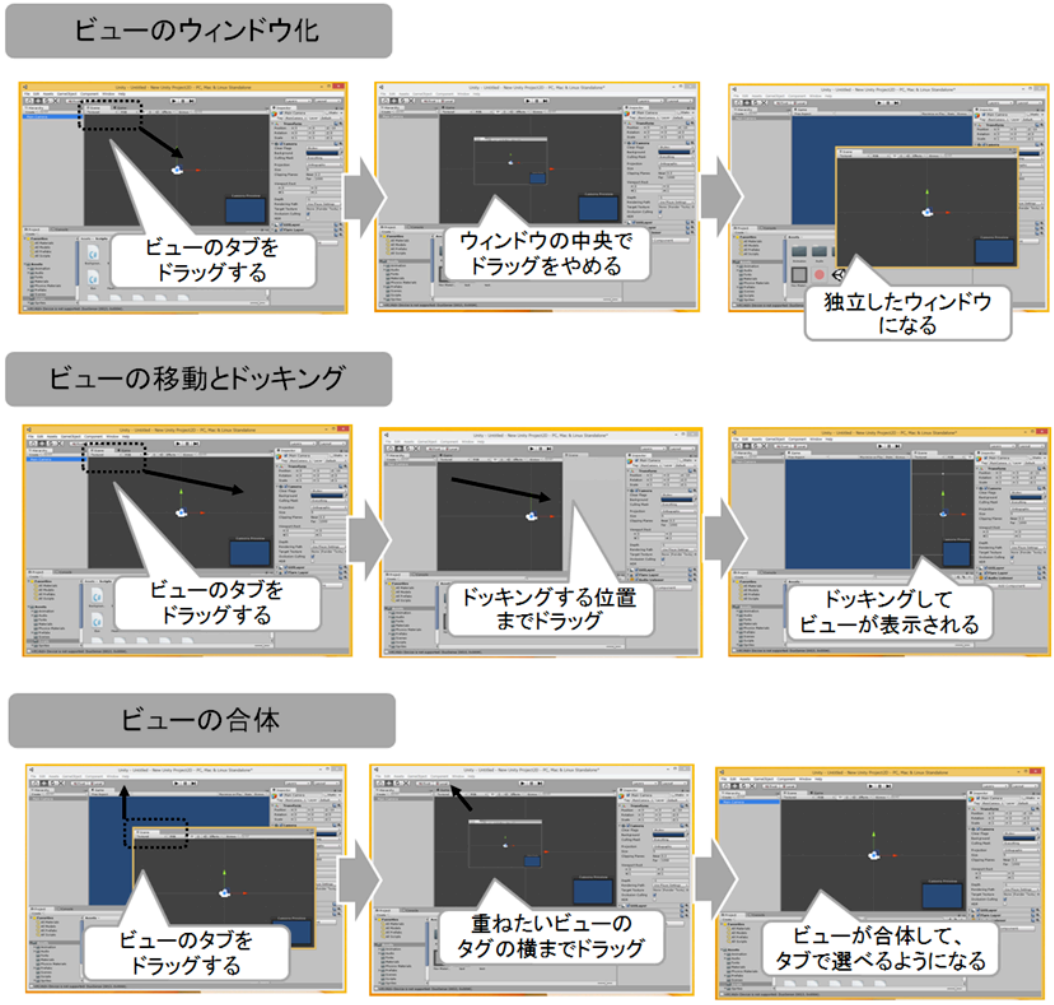


図 A02_02_003 ビューの操作

さらに、各ビューの右上端には、ビューのタグを編集できるメニューがあります。

「Maximize」で最大化、「Close Tab」でタブを閉じ、「Add Tab」で新しくタブをビューに追加できます（図 A02_02_004）。

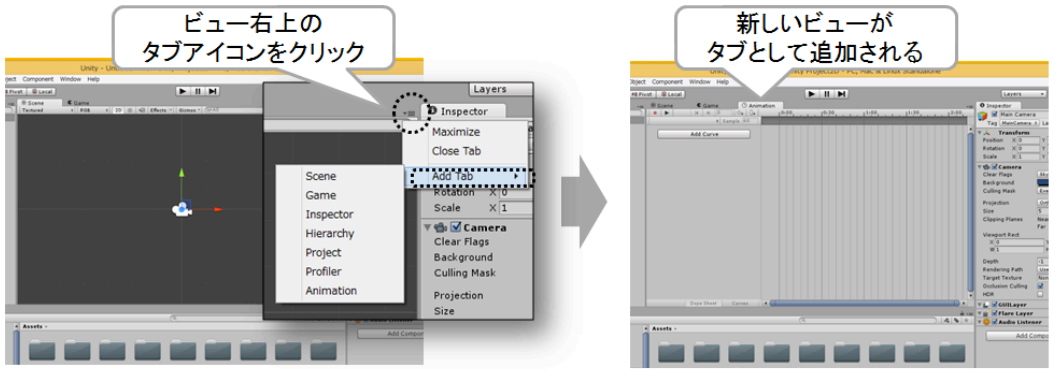


図 A02_02_004 ビューとタブメニュー（タブの追加）

ビューをカスタマイズした Unity の画面構成は、「Window」メニューの「Layouts」から「Save Layout...」で保存して、Layouts メニューに追加することが可能です。削除する場合は、同メニューの「Delete Layout...」を実行します。

まずは「プロジェクトブラウザ」にアセットを登録する

Unity で 2D ゲームを作る場合、最初に「プロジェクトブラウザ」の”Assets”フォルダ内に、スプライト画像やサウンドファイルなどをドラッグ＆ドロップして**「インポート（登録）」**します（図 A02_03_001）。

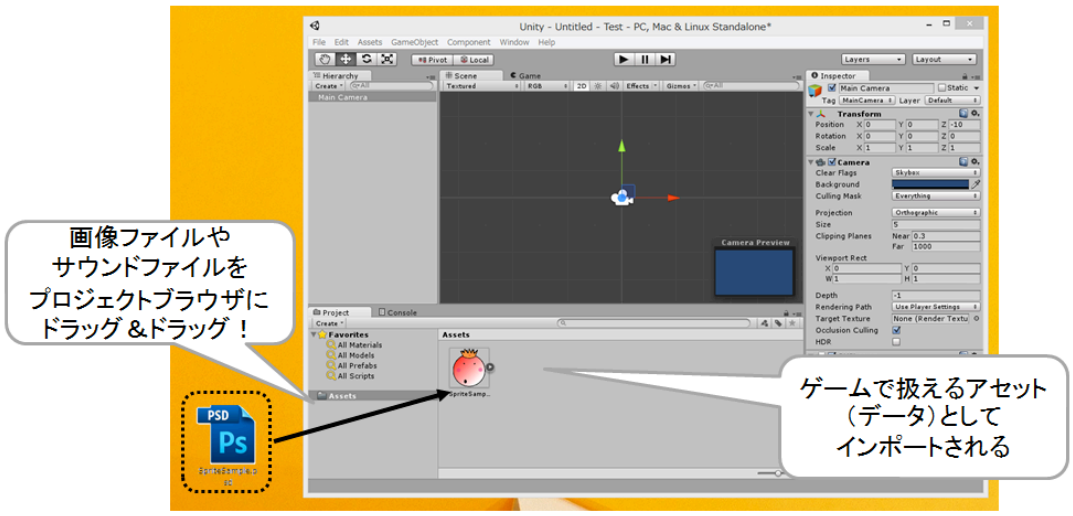


図 A02_03_001 「プロジェクトブラウザ」とアセットのインポート

このプロジェクトブラウザは、ゲームで使う画像データやサウンドデータ、そしてスクリプトなどの素材をまとめて管理するビューです。この中にゲームで扱うデータをインポートして、はじめてゲームプログラムからロードしたり参照したりすることが可能となります。

「シーンビュー」でゲーム画面を作る

次は「シーンビュー」について説明しましょう。

まずは、「プロジェクトブラウザ」に先ほど登録した画像を「シーンビュー」にドラッグ＆ドロップしてみてください。スプライト画像がゲームのスプライトとしてシーンに追加されます（図 A02_04_001）。

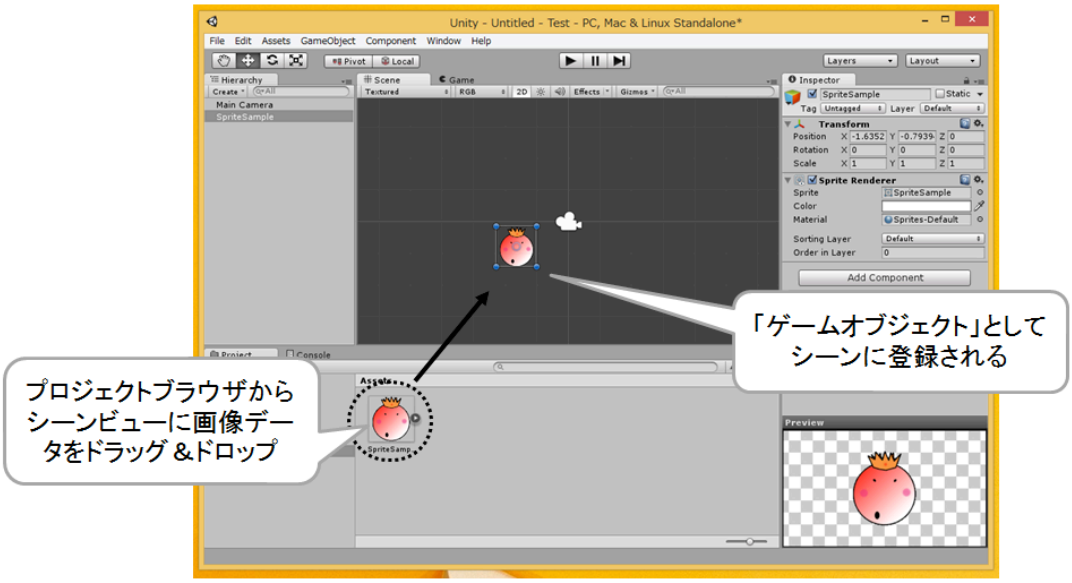


図 A02_04_001 「シーンビュー」とスプライト画像の登録

このように「シーンビュー」は、ゲーム画面を作るビューです。

追加したゲームのスプライトなどは**「ゲームオブジェクト（GameObject）」**と呼ばれるオブジェクトに変換されます。

このゲームオブジェクトは、シーン内では図 A02_04_001b のに「左手座標」で管理されます。

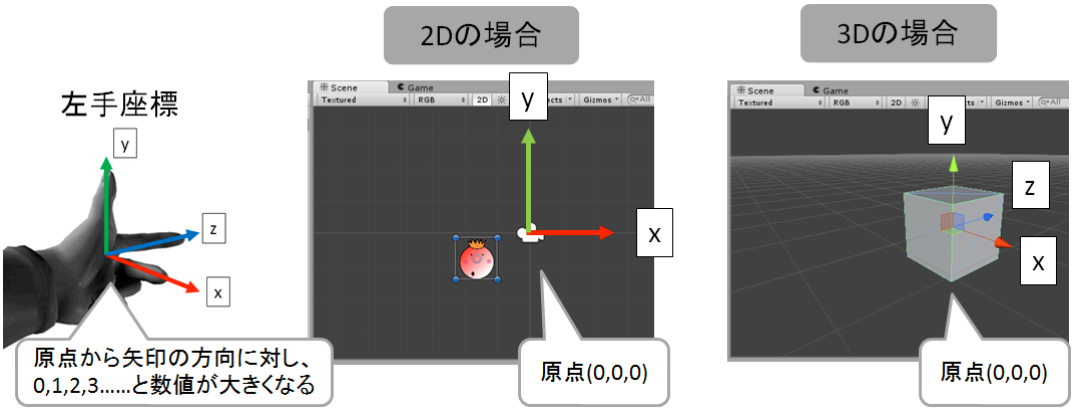


図 A02_04_001b Unity の座標系

ゲームオブジェクトは、移動・回転・拡大して編集することができます。ゲームオブジェクトのコピーも削除も可能です（図 A02_04_002）。

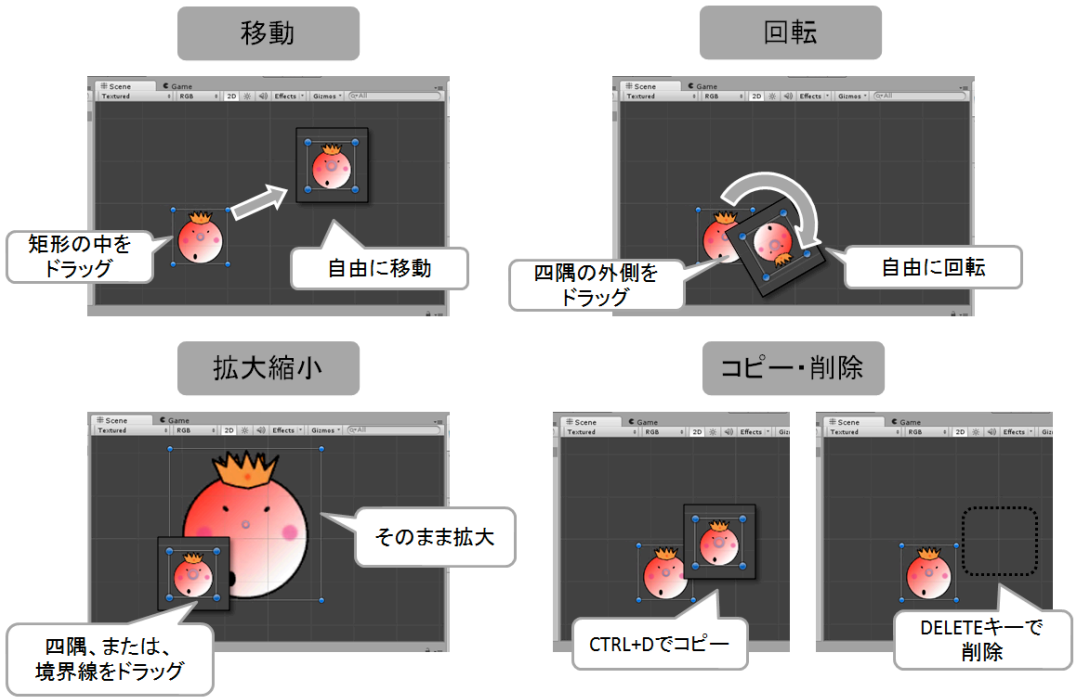


図 A02_04_002 シーンビューでのスプライトの編集

また、これらの操作はツールバーの「トランスフォームツール」を使って編集モードを切り替えることもできます（図 A02_04_003）。

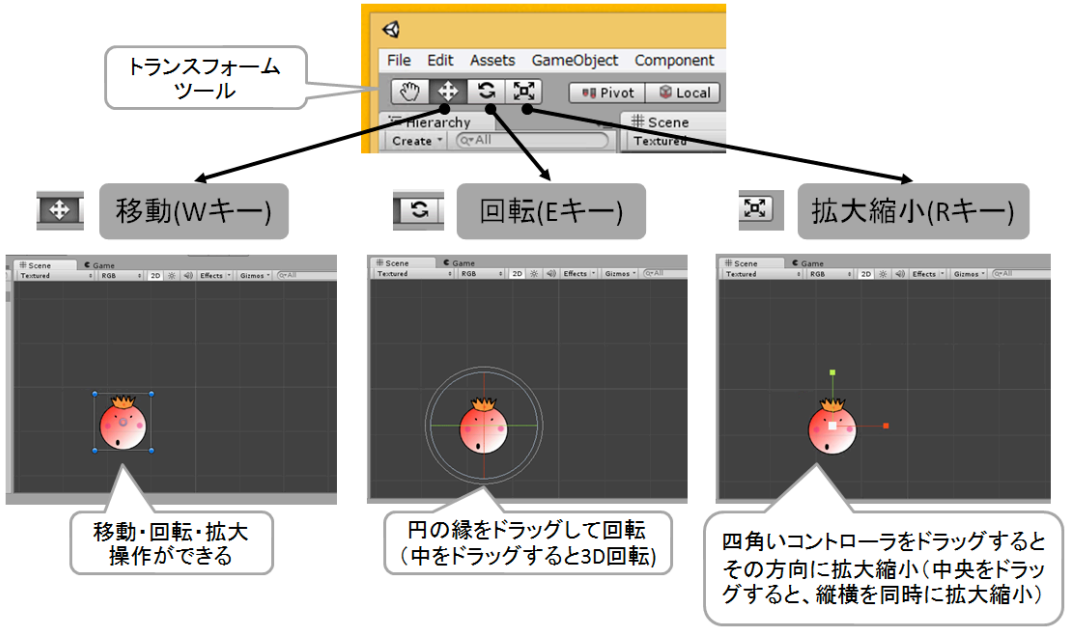
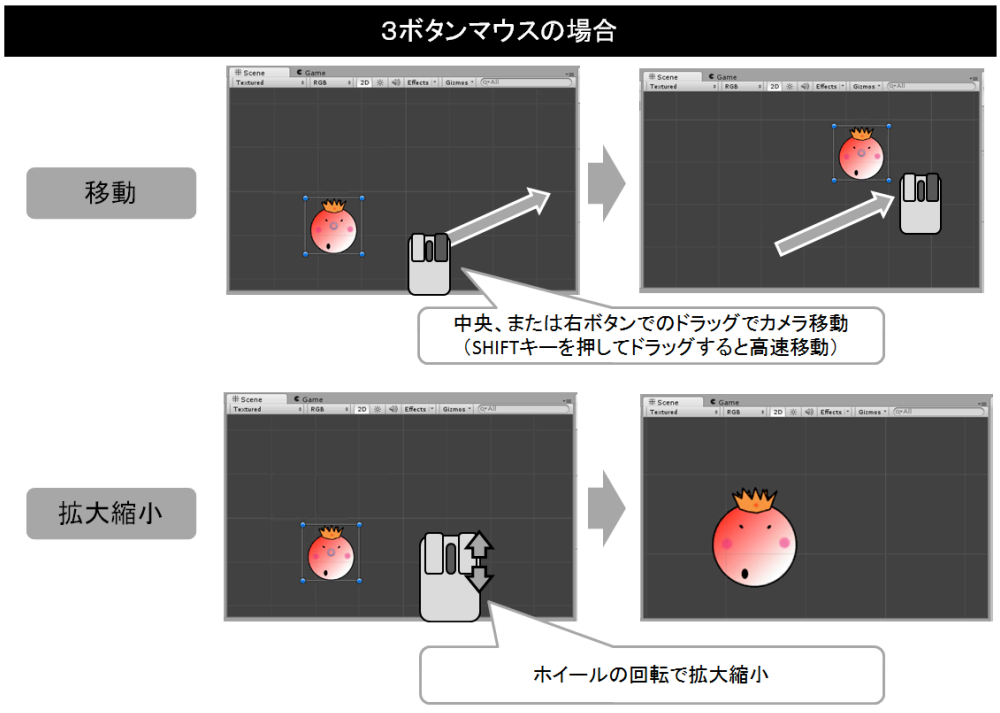


図 A02_04_003 「トランスフォームツール」での操作

シーンビューのカメラを操作して、シーン内の移動や拡大縮小することもできます（図 A02_04_004）。



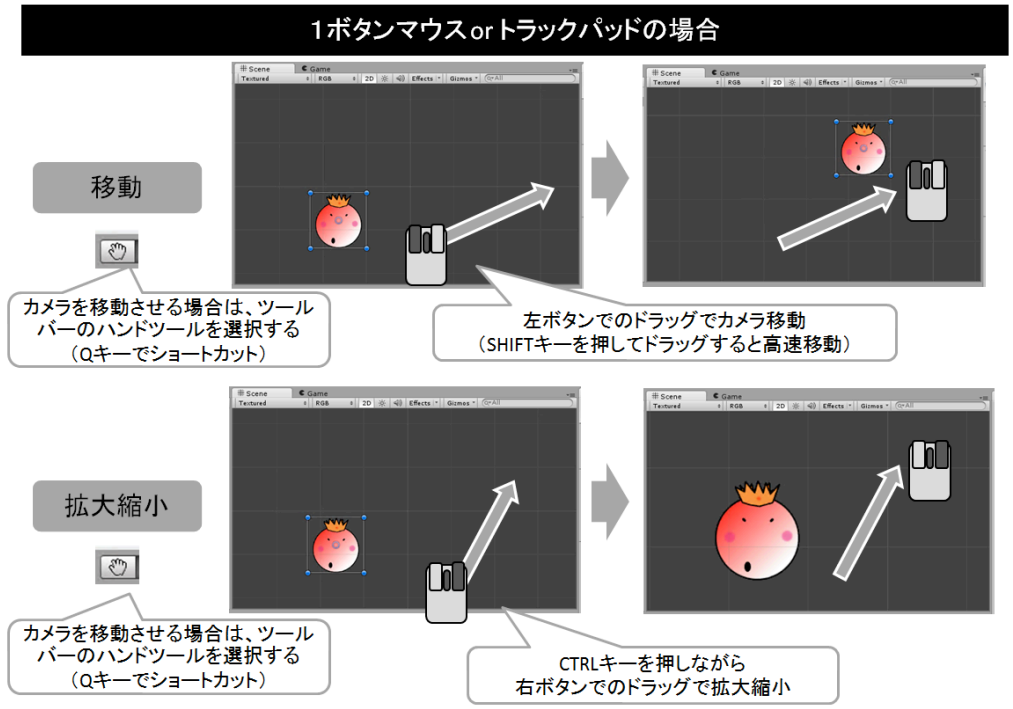
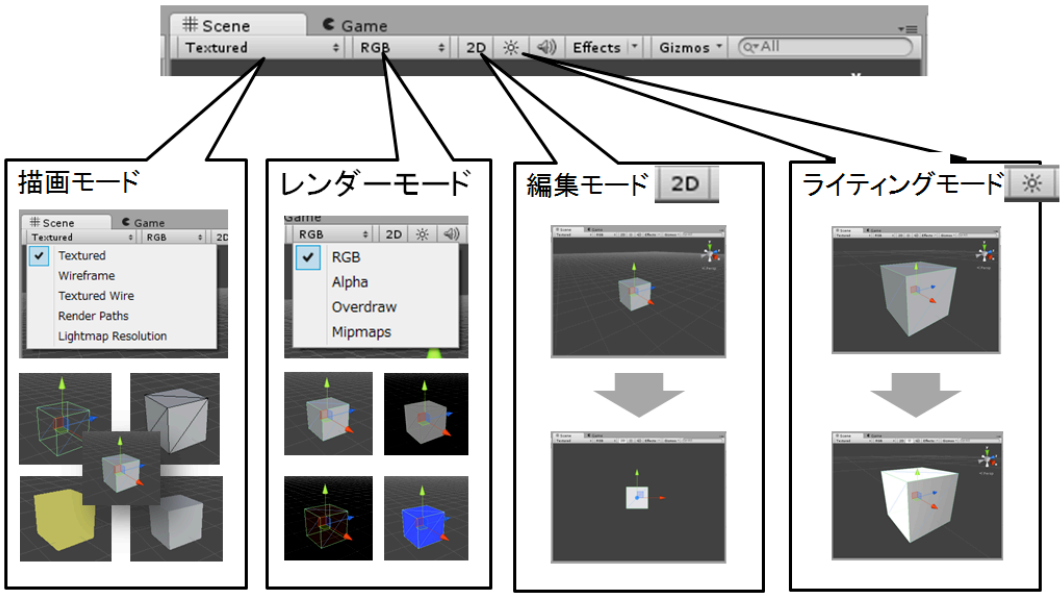


図 A02_04_004 シーンビューでのカメラ操作

さらに、シーンビューには、現在のシーンの情報表示をコントロールできる「シーンビューコントロールバー」があります（図 A02_04_005）。



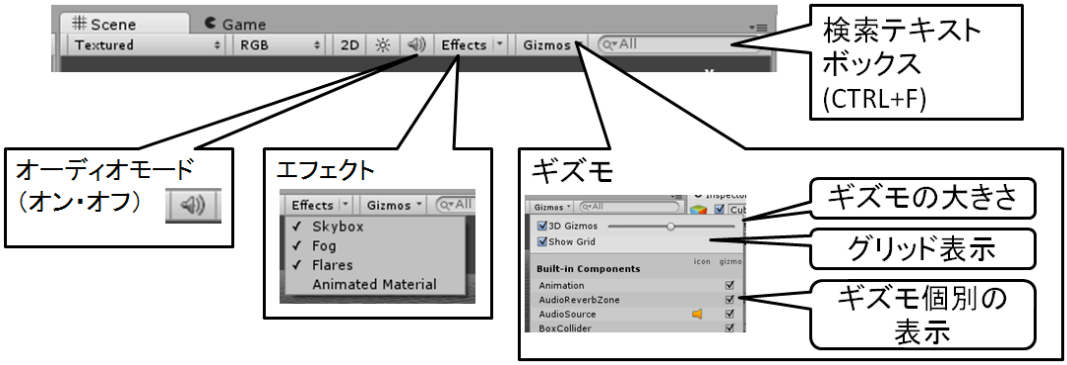


図 A02_04_005 「シーンビューコントロールバー」の機能

- **描画モード（Draw Mode）**
  シーンビュー内の表示方法を切り替えます。テクスチャ表示の他に、ワイヤーフレームなど、シーン編集に最適な表示方法が選択できます。

- **レンダーモード（Render Mode）**
  ゲーム画面をレンダリングする場合の中間情報を表示できます。画像のアルファやミップマップレベルなどを見ることができます。2D ゲーム制作ではあまり気にしなくてもよい機能です。

- **編集モード**
  2D/3D の編集モードの切り替えができます。2D ゲームの場合は「2D」、3D ゲームの場合は「3D」にすると編集しやすくなります（本書では Unity2D の機能だけを説明するため 3D の編集モードについては説明しません。3D の編集モードについても知りたい方は、Unity の公式サイトに解説がありますので、そちらを参考にしてください）。

- **ライティングモード**
  「シーンを編集しやすいライティング」と「ゲーム実行中の実際のライティング」を切り替えることができます。

- **オーディオモード**
  シーンに BGM や SE などのゲームオブジェクトを設定している場合、このサウンドをシーンビューでもチェックできるようにオンオフします。

- **エフェクト**
  Skybox や Flare など、Unity が持っているエフェクト機能をオンオフします。
  三角のマークを押すことで、ドロップダウンリストから個別にチェックすることも可能です。

- **ギズモ**
  「ギズモ」とは、ゲームオブジェクトを操作するための矢印やアイコンのことです。

各種ギズモの表示や操作のオンオフ、アイコンの大きさなどを設定できます。
シーンビューに表示されているグリッドのオンオフも可能です。

・ **検索テキストボックス**
シーンビュー内のゲームオブジェクトを検索します。検索したいゲームオブジェクト名を入力すると、そのゲームオブジェクトが表示されます。また、三角のマークをクリックするとゲームオブジェクト名の他に、コンポーネント名（Type）による検索も指定できます。なお、ショートカットの CTRL+F を押すと、すぐに検索テキストボックスに文字を入力することができます。

これらのシーンビューのカスタマイズは、2D ゲームを開発する場合は、最初から必要になることは、ほとんどありません。しかし、ステージが見づらかったり編集しづらいと思ったら、これらの機能を思い出してみてください。

「ヒエラルキー」でゲームオブジェクトを選択する

シーンビューにどのようなゲームオブジェクトが設置されているのか、ツリーリスト形式で分かりやすく見ることができるのが「ヒエラルキー」です。
シーンビューでゲームオブジェクトを選択すると、ヒエラルキーにも反映されます（図 A02_05_001）。

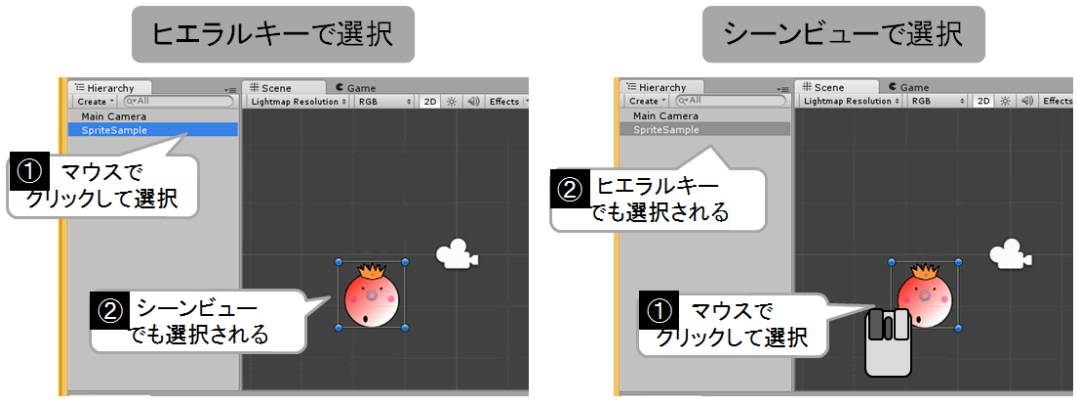


図 A02_05_001 「ヒエラルキー」でのゲームオブジェクト選択

逆に、ヒエラルキーからゲームオブジェクトを選択することも可能です。シーン内に映っていないゲームオブジェクトも、ヒエラルキーのゲームオブジェクトをダブルクリックするだけで、シーンビューのカメラが移動して表示してくれます（図 A02_05_002）。

図 A02_05_002 シーンビューでのカメラ操作

また、ヒエラルキーでは、ゲームオブジェクトをコピーしたり階層化することができます。

例えば、次のようにゲームオブジェクトをコピーして、コピー元のゲームオブジェクトにドラッグ＆ドロップすることで、ゲームオブジェクトが階層化され親子関係ができます（図 A02_05_003）。

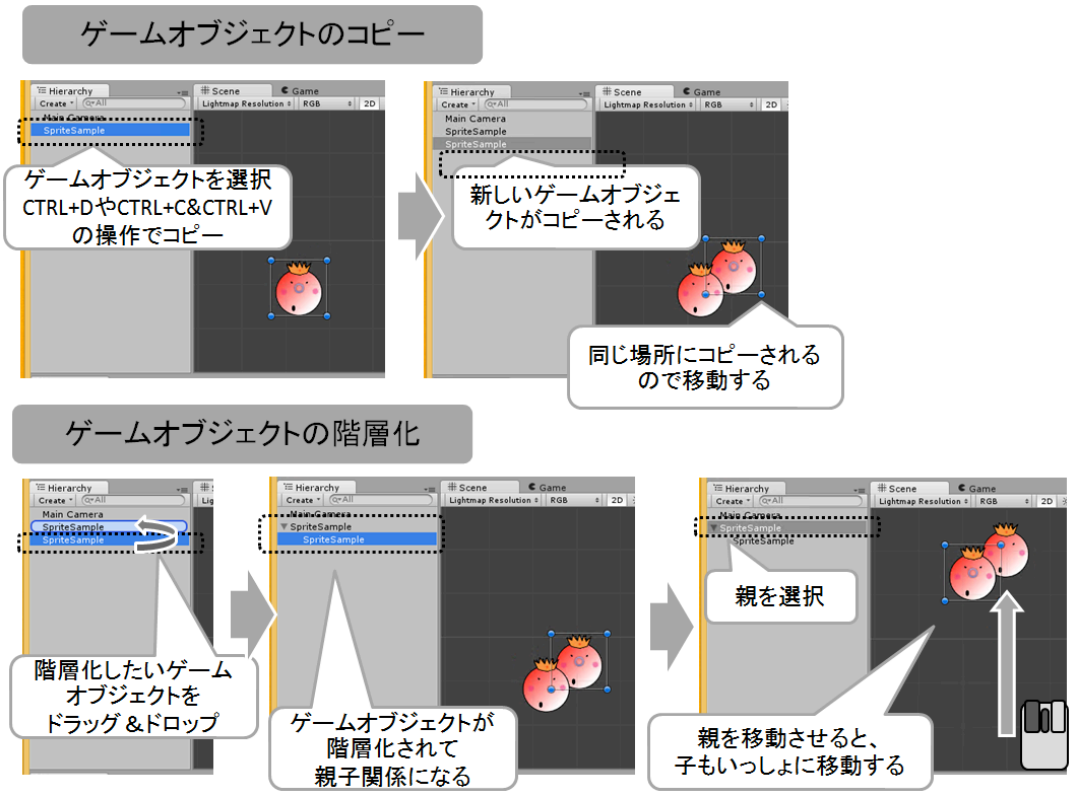


図 A02_05_003 シーンビューでのゲームオブジェクトのコピーと階層化

なお、不要なゲームオブジェクトは、ヒエラルキーで選択して DELETE キーで削除できます。

このようにヒエラルキーは、シーンにおけるゲームオブジェクトの構成や構造を編集することができます。

「インスペクタ」でプロパティを設定する

スプライトをはじめ、シーンビューで扱えるゲームオブジェクトには、座標などが設定できる様々なプロパティが存在します。これらのプロパティを直接編集できるのが「インスペクタ」です。シーンビューかヒエラルキーでゲームオブジェクトを選択すると、インスペクタにそのゲームオブジェクトのプロパティが表示されます（図 A02_06_001）。

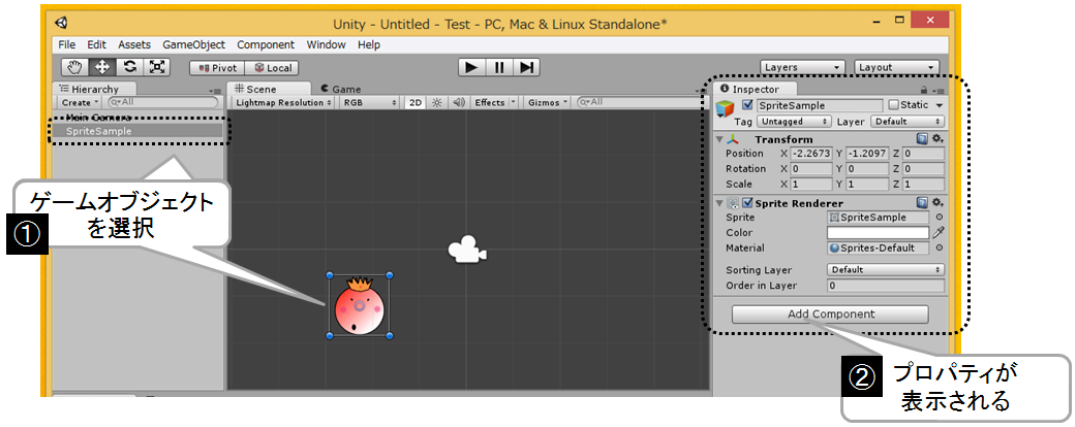


図 A02_06_001 「インスペクタ」にゲームオブジェクトのプロパティを表示する

試しに、Transform の x,y 値に直接数値を入力してみてください。シーンビューにその結果が即座に反映されます。また、各値の名前で、マウスをクリックして左右にドラッグすると数値入力しなくても数値が変わります（図 A02_06_002）。

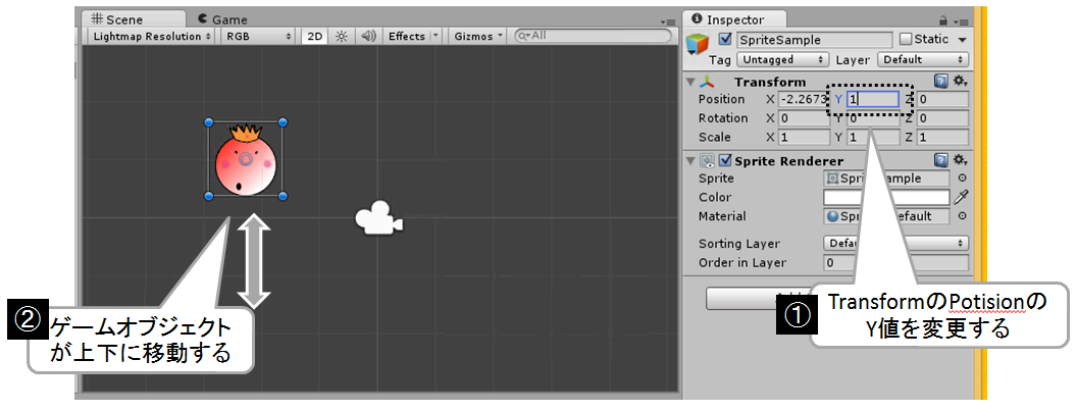


図 A02_06_002 インスペクタでのプロパティの操作

このインスペクタの能力は非常に強力で、ゲーム実行中でもインスペクタでプロパティを編集して実験することが可能です（ただし、編集した値はゲーム終了と同時に元に戻ります）。

MonoDevelop でスクリプトをプログラミングする

Unity でゲームをプログラムする場合は、「MonoDevelop」と呼ばれるツールを使ってスクリプトを作成します。その手順は、ちょっとだけ複雑です。

まず、プロジェクトブラウザの「Create」ボタンを押して「C# Script」を選択します。新しいスクリプトが作成されるので、名前を”Move”に変更しましょう（図 A02_07_001）。

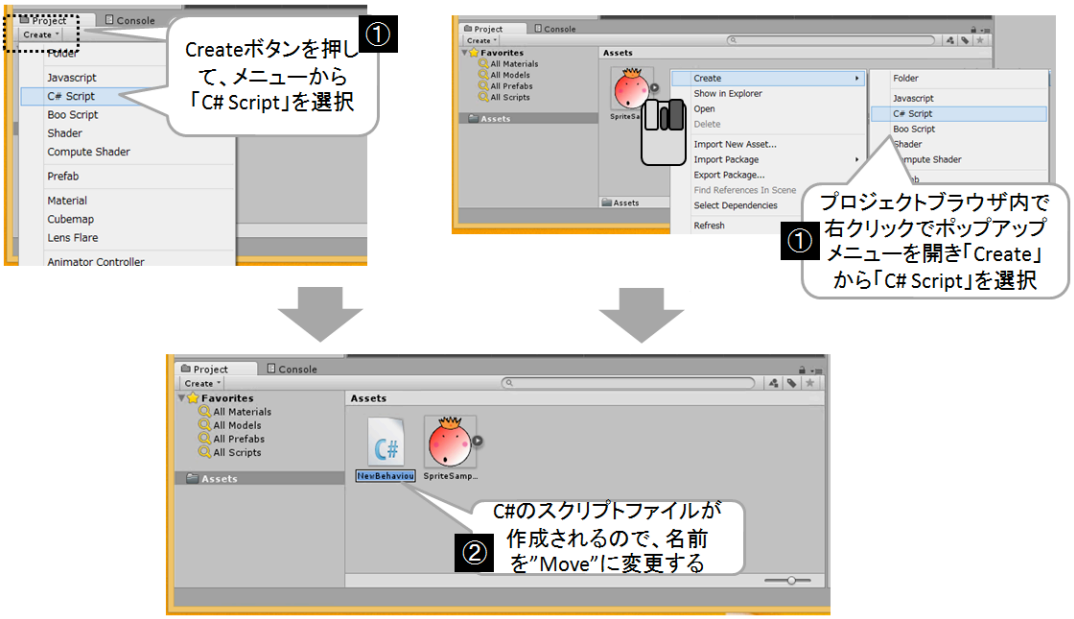


図 A02_07_001 スクリプトの作成

次に作成した Move スクリプトをヒエラルキーかシーンビュー、またはインスペクタにドラッグ＆ドロップして追加します（コンポーネントの数が多い場合は、インスペクタ以外のビューでアタッチする方が簡単です）。これで、Move スクリプトがどのゲームオブジェクトのプログラムなのかアタッチして関連付けされました（図 A02_07_002）。

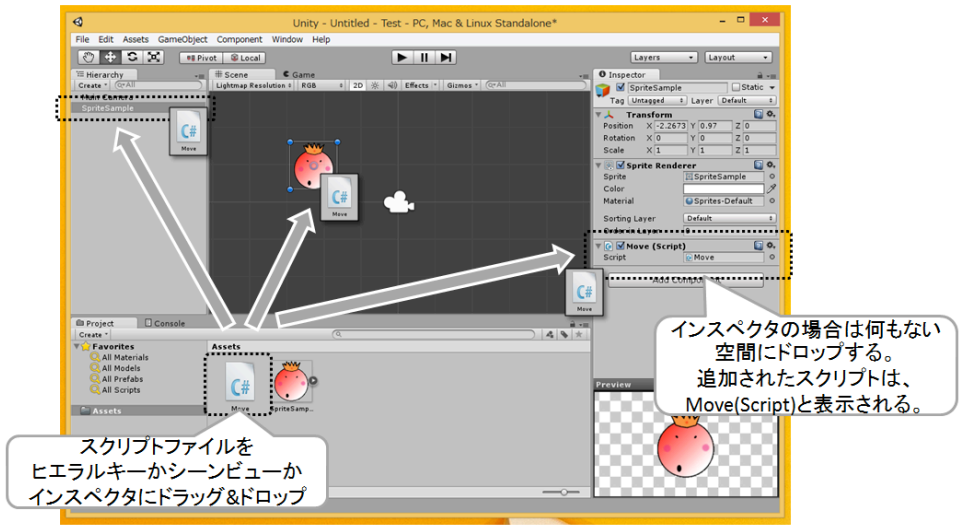


図 A02_07_002 スクリプトのアタッチ

最後に、スクリプトにプログラムを記述します。

アセットブラウザの Move.cs をダブルクリックして MonoDevelop を起動します（図 A02_07_003）。

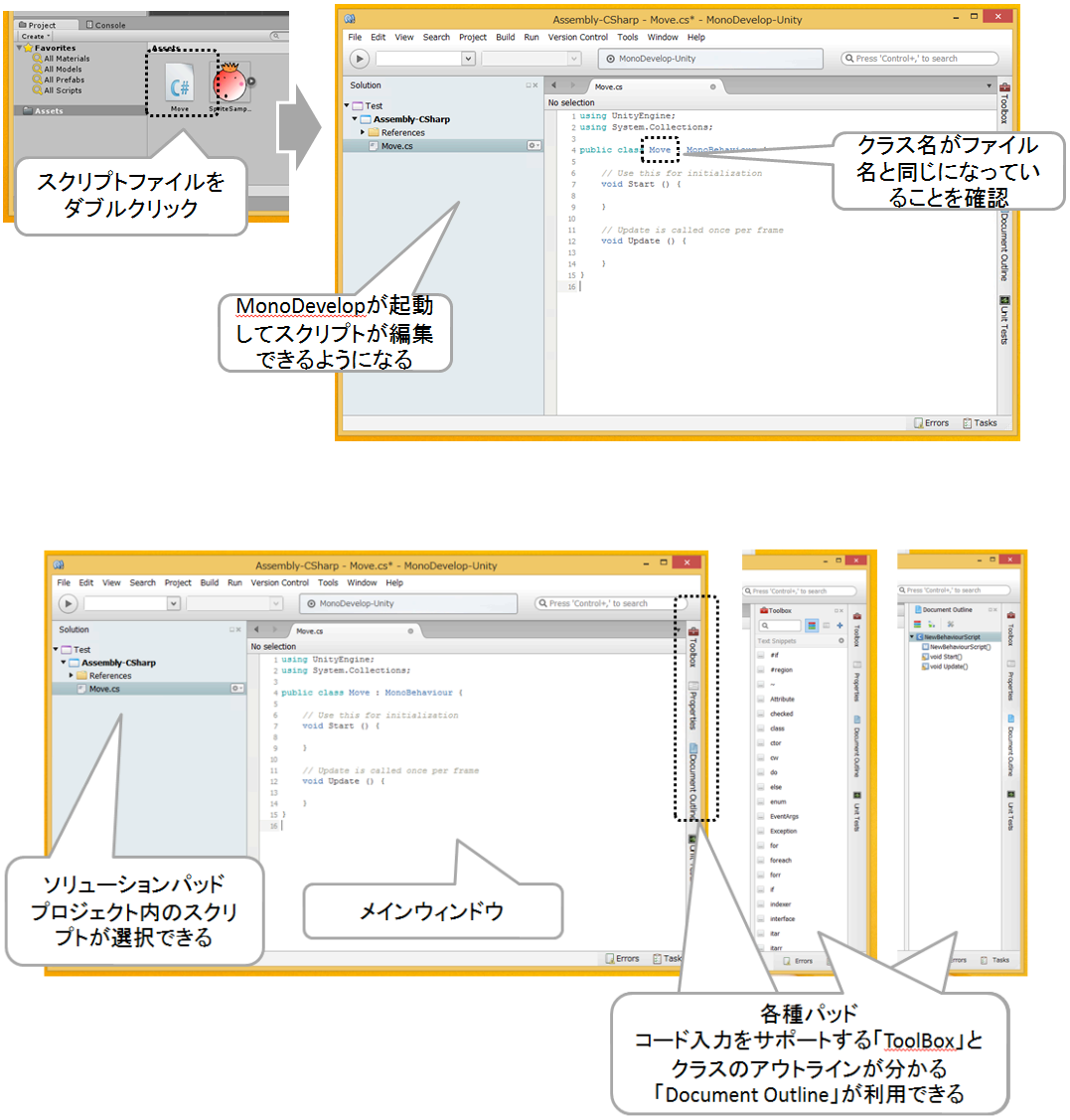


図 A02_07_003 スクリプトを MonoDevelop で編集する

Monodevelop が起動できたら、次のスクリプトを入力してください。

ソース 1.2.1.1 : Move.cs

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

```
public class Move：MonoBehaviour {

 // Use this for initialization
 void Start () {

 }

 // Update is called once per frame
 void Update () {
  // ここから……
  float vx = Input.GetAxis ("Horizontal");
  float vy = Input.GetAxis ("Vertical");
  transform.Translate(new Vector3 (vx, vy, 0.0f));
  // ここまでを追加する
 }
}
```

これでプログラムは完成しました。

※スクリプトの名前を変更する場合、一度、名前を確定した後に名前を変更してもクラス名は変更されません。そのため、例えば初期値として設定されるファイル名”NewBehaviourScript”で決定すると、その後にファイル名を変更しても、ファイル名は”Move”になりますが、Monodevelop でスクリプトファイルを見るとクラス名は”NewBehaviourScript”のままになります。Unity ではファイル名とクラス名が異なる場合はエラーとなるため、このような場合は MonoDevelop でクラス名を手動で”Move”に修正してください。

コラム：MonoDevelop と日本語の対応状況

Unity から起動される MonoDevelop ですが、Unity のゲームエディタと同じく日本語の対状況はあまりよくありません。特に Windows8/8.1 や Mac から MonoDevelop のエディタで日本語を入力をしようとしても、変換中の漢字が見えなかったり、時には入力できなかったりと、日本語変換の FEP が正常に動作しません。
また、Unity のバージョン（正しくは、その Unity といっしょにインストールされる MonoDevelop のバージョン）によっては、ソースコードにおける日本語データの扱いが異なっています。

・Unity 4.1.2 まで

Unity4.1.3 までのバージョンでは、プロジェクトブラウザで作成したスクリプトファイルが「BOM 無し UTF-8」という文字コードの形式で保存されます。しかし、この文字コードの場合、いっしょにインストールされた MonoDevelop や Mono が正しく日本語コードを解釈できません。そのため、日本語入力がおかしくなるだけでなく、特定の日本語文字では、コンパイルエラーになったり、次のステート（命令文）がコメントとして扱われ動作しないなどの問題が発生します。
これを回避するには、Unity で作成されたスクリプトファイルを「BOM 無し UTF-8」から「BOM 有り UTF-8」に変換する必要があります。Windows であれば、フリーソフトの「文字コード判定&変換ツール.NET」などで変

換可能です。

・Unity 4.1.3 以降

Unity4.1.3 以降では、プロジェクトブラウザで作成したスクリプトファイルが「BOM 有り UTF-8」で自動的に作成されるにようになりました。そのため、以前のバージョンにあったような日本語コードにおけるコンパイルエラーなどの問題が解消されました。

ただし、MonoDevelop の日本語入力の問題自体が解消されたわけではないため、FEP 入力は本来求められる正常な動作をしていません。そのため、変換中の漢字がエディタ上で確認できない他に、FEP での入力・編集中にソースの表示がおかしくなる（見えないおかしな文字コードが入力されるなど）が、たまに発生する場合があります。

また、MonoDevelop のリファクタリング機能で、複数のファイルに変更を加えるような操作を行うと、BOM 無し UTF-8 に戻ってしまうようです。

このように、MonoDevelop の日本語対応は不完全ですので、こまめにバックアップを取っておきましょう。また、スクリプトファイルの編集中におかしな動作をするようになったら、Windows であれば「秀丸」など他のエディタを使って改行コードなどを確かめて正しい内容に変換して見てください。

なお、現在、Unity Japan では、この問題に対応中とのことですので、将来的には MonoDevelop の日本語問題は解消されるでしょう。

ゲームを実行してゲームビューで遊ぶ

では、ゲームを実行してみましょう。

ツールバーの「実行」ボタンを押してゲームを実行します。「ゲームビュー」に切り替わり、ゲーム画面が表示されます（図 A02_08_001）。

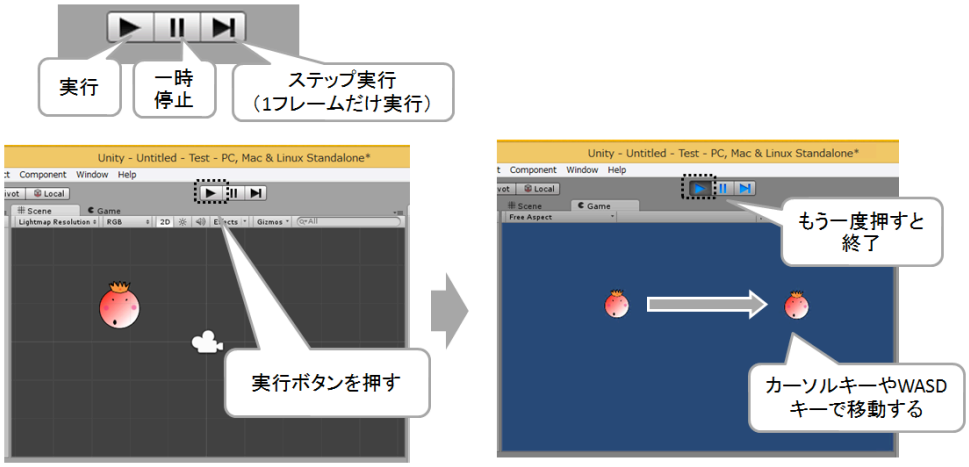


図 A02_08_001 ゲームの実行とツールバー

スクリプトにエラーがなければ、キーボードのカーソルキーか WASD でゲームオブジェクトが上下左右に移動します。

なお、ゲームビューのコントロールバーからは、画面の表示方法やゲーム情報の表示の設定ができます（図 A02_08_002）。

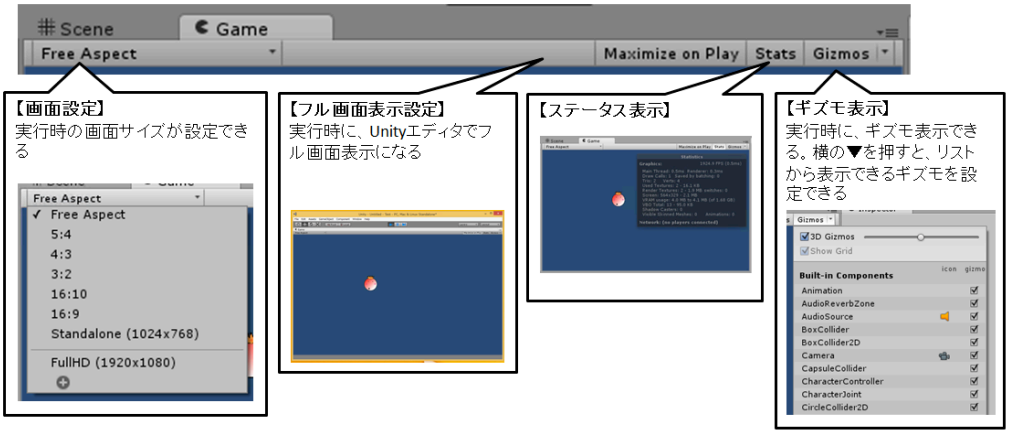


図 A02_08_002 「ゲームビューコントロールバー」の機能

フル画面で確認したい場合は、実行前に「Maximize On Play」ボタンを押してください。次回から、フル画面でゲームが実行されます。

「コンソール」でエラーを確認する

もし、うまくゲームが動作しない場合は、「Window」メニューから「Console Ctrl+Shift+C」を選択して、「コンソール」を開いてください。何かエラーがあれば、このコンソールにエラーが表示されます（図 A02_09_001）。

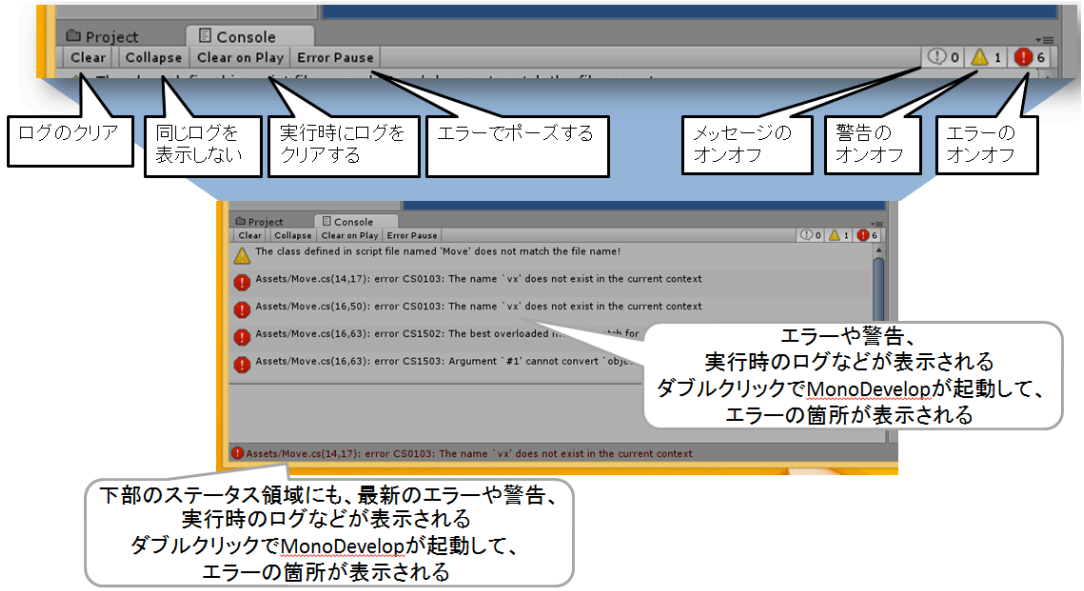


図 A02_09_001 コンソール

なお、Unity では、Monodevelop でスクリプトを編集した後、Unity エディタに切り替えた時点で自動的に C#のコンパイルを行っています。そのため、コンパイルエラーがコンソールに出るまでに、数秒～数十秒の時間差がありますので注意してください。

デバッグする

ゲームの開発中にバグが発生し、プログラムをデバッグしたいこともあるでしょう。

そこで、Unity のデバッガを使ったデバッグ方法をご説明します。

まず、デバッグを行うときは、「Assets」メニューから「Sync MonoDevelop Project」を選んで実行します。MonoDevelop が起動して Unity デバッグ情報をやり取りできるようになります（なお、これは Unity を初めて実行するときに必要な操作で、2 回目以降は不要です。もし、うまくデバッグできないときは、この操作をもう一度繰り返してください）。

デバッグの方法は簡単です。ブレークポイントでプログラムを止めたい場合は、そのソースコードの左側をマウスでクリックしてブレークポイントでマークします。実行ボタンを押すとターゲットの選択になるので、”Unity Editor”が選択されていることを確認して「Attach」ボタンを押してください。さらに、Unity エディタで実行ボタンを押すと、ブレークポイントで停止します（図 A02_10_001）。

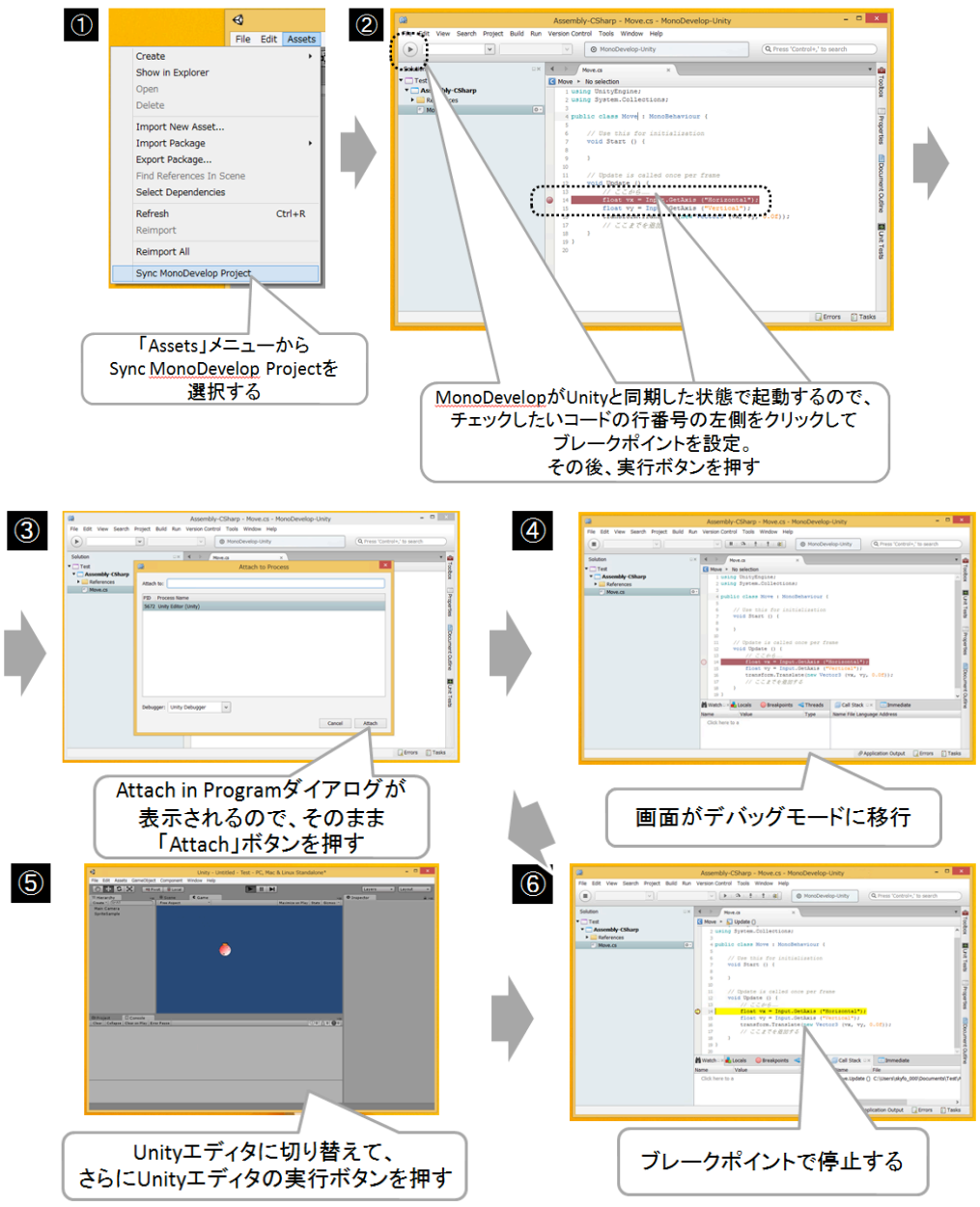


図 A02_10_001 デバッグの手順 その 1

これで MonoDevelop がデバッグモードに移行して、ツールバーからトレース、ステップ、ステップオーバーなどの実行が可能になります。また、「Watch」パッドや「Locals」パッドで変数を確認したり、「Call Stack」パッドでどのようにコードが実行されたのか確認することができます（図 A02_10_002）。

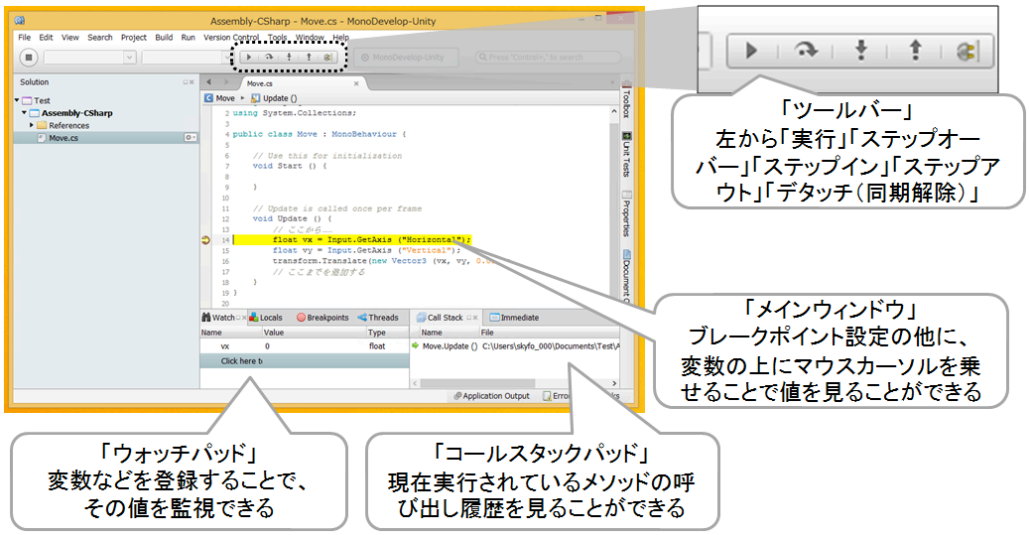


図 A02_10_002 デバッグの手順 その 2

Unity の開発マシンで実行している環境では、このデバッガが自由に使えますので、ぜひ活用してみてください。

なお、iPhone や Android などの他のプラットフォームでも、実機で動作させてデバッガでデバッグすることが可能です（なんと、無線 LAN で接続してデバッグも可能です）。また、Debug.Log などの API を使用して、コンソールやログモニタツールなどに出力して確認することもできます（各プラットフォームでログを確認できるツールを起動しておく必要があります）。

各プラットフォームでのデバッグについては、Google などで”Unity 実機 デバッグ”で検索して、解説サイトなどを参考にしてください。

シーンを保存する

最後に、これまで作成してきたシーンの内容を保存しましょう。

「File」メニューの「Save Scene」か CTRL+S キーで保存できます（図 A02_11_001）。

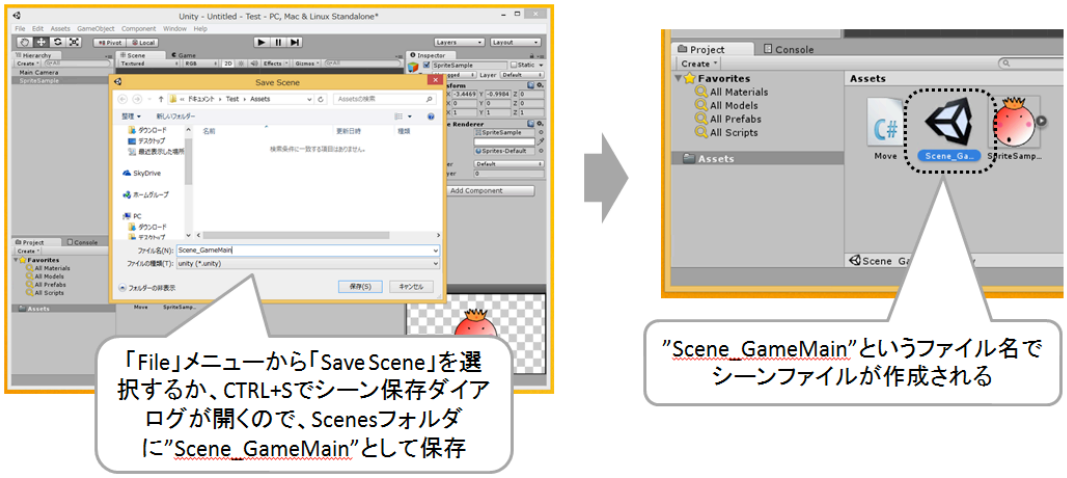


図 A02_11_001 シーンの保存

Unity は比較的安定したツールですが、それでもたまにフリーズしたりすることがあります。こまめに CTRL+S キーで保存するクセをつけておきましょう。

ビルドしよう！

作ったゲームは、他の人にも遊んでほしいですよね。
そこで、他のパソコンでも遊べるように、このゲーム（プロジェクト）をビルドします。
まずは、「File」メニューの「Build Settings...」を選んで、現在のシーンを登録します（図 A02_12_001）。

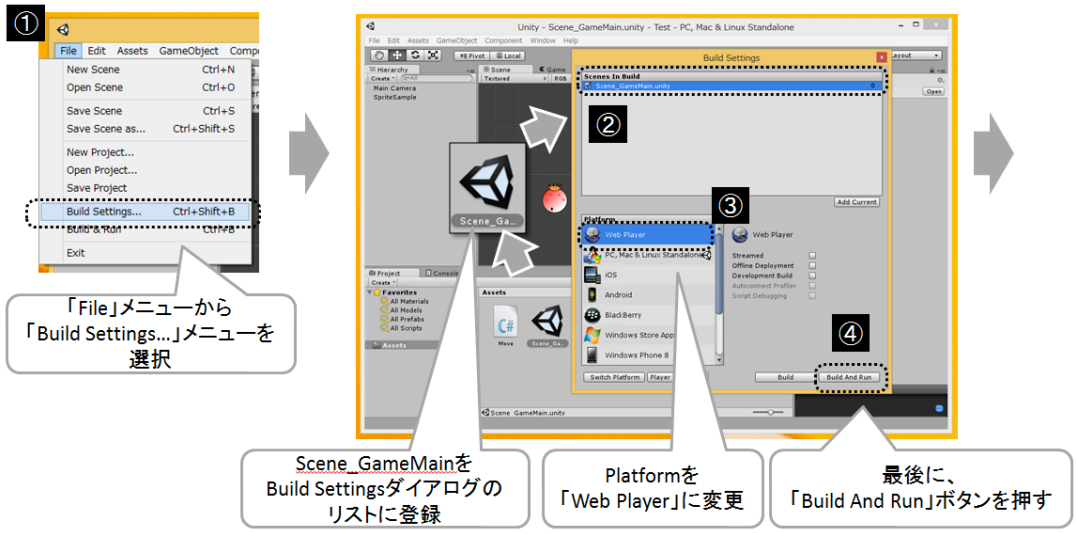


図 A02_12_001 ゲームをビルド その 1

この「Build Setting」ダイアログで先頭に登録されたシーンが、ゲームアプリを実行した最初のシーンとなります。また、現在デフォルトでプラットフォームが「PC,Mac & Linux Standalone」になっています。これは、Windows なら EXE ファイルといったように各プラットフォームの実行ファイルを作成します。より多くの人に遊んでもらうなら、プラットフォームはウェブブラウザの方が良いでしょう。「Platform」を「Web Player」に切り替えて、「Build And Run」ボタンを押してください。ファイル名入力のダイアログが表示されるので、適当な名前を入力して OK ボタンを押すとビルドが実行されます。ビルドに成功したら、ウェブブラウザが起動してゲームを遊べます（図 A02_12_002）。

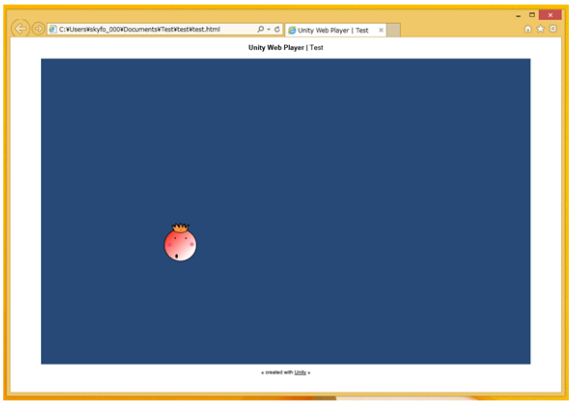


図 A02_12_002 ゲームをビルド その 2

実際に、ゲームを他の人にも遊んでもらうには、作成したファイルをインターネットにアップロードする必要があります。レンタルサーバーサービスなどを利用していない方は、オンラインストレージサービスの Dropbox などでも URL 付きで公開可能ですので、利用してみてください。

Unity には、ゲーム開発をサポートする強力な仕組があります。

それが「アセットストア（Asset Store）」です。アセットストアは、ゲームのキャラモデルやステージモデル、便利なプログラムや開発ツールなど、ゲームに必要な様々なアセットがダウンロードできるサービスです。

Asset Store は、「Window」メニューの「Asset Store」を選んで実行し、Asset Store ビューを開いて利用します（図 A02_13_001）。

図 A02_13_001 Asset Store

無料のアセットはそのまま利用できますが、有料のアセットを購入する場合は、ユーザー登録やクレジットカード登録などが必要です。

本書では基本的にアセットストアは利用しませんが、ゲーム開発で悩んだときは、このアセットストアを訪れて見るのも良いでしょう。問題を解決してくれるアセットがあるかもしれません。

困ったら……

ここまで、Unity の初歩の初歩を解説してきました。
もっと入門用の知識が欲しい方は次のページを参考にしてみてください。

・Unity Japan デベロッパーページ

http://japan.unity3d.com/developer/

開発者向けのページです。ドキュメントから勉強会のイベント情報まで、様々な情報がこのページに集約されています。

・Unity Japan チュートリアル

http://japan.unity3d.com/developer/document/tutorial/

日本語で読める Unity のチュートリアルページです。

・Unity Learn Tutorials

http://unity3d.com/learn/tutorials/modules

本家 Unity の学習ページです。英語ですが、一部ビデオは Unity Japan によって日本語字幕が付けられています。ビデオ視聴の際に、字幕を「日本語」に選択してみてください。

・Unity Japan ドキュメント

http://docs-jp.unity3d.com/

Unity のドキュメントが一覧できるページです。Unity の使い方から、スクリプトリファレンスまで、日本語で読むことができます。

また、本書の巻末にも、Unity でゲームを作る上で参考になるサイトを紹介していますので、ぜひチェックしてみてください。

1.3. Unity の仕組

さて、Unity の基本的な使い方は分かりました。次は、Unity の仕組と構造に説明しましょう。

Unity の開発スタイル

最初に、ゲーム開発における「プログラミングスタイルの違い」について説明します。

現在、ゲーム開発におけるプログラミングスタイルは、筆者の考えでは大きく分けて 3 つあります。

1 つ目は、ゲームの仕様や動作に基づいて、ゲームの流れをプログラムで記述し、それに合わせてデータを用意して動作させる「フロー駆動型プログラミング」です。メインループと呼ばれるプログラムの大きな流れがあり、その流れの中にキャラ移動などの各種処理を記述します。昔のゲームは、基本的にフロー駆動型プログラミングが主流でした（図 A03_01_001）。

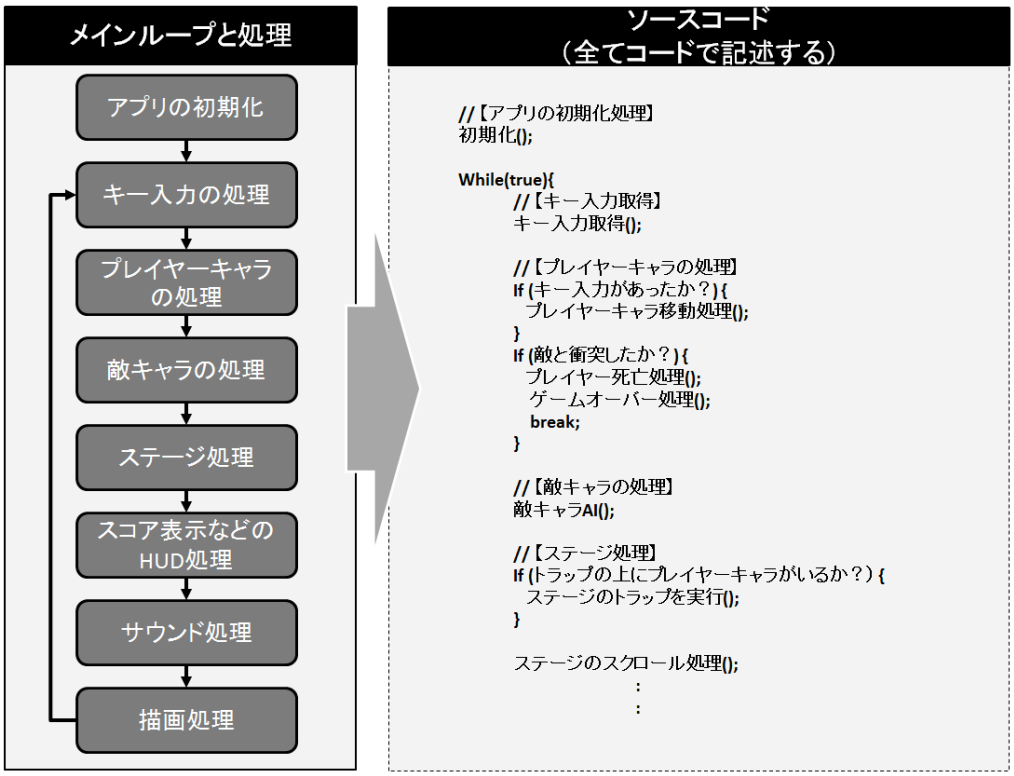


図 A03_01_001 フロー駆動型プログラミング

2 つ目は、プログラムの大きな流れはフレームワークやゲームエンジンが処理し、個々の対応したイベントのみをプログラムで記述する「イベント駆動型プログラミング」です。GUI を使用した OS のアプリケーション開発などでは一般的なスタイルです（図 A03_01_002）。

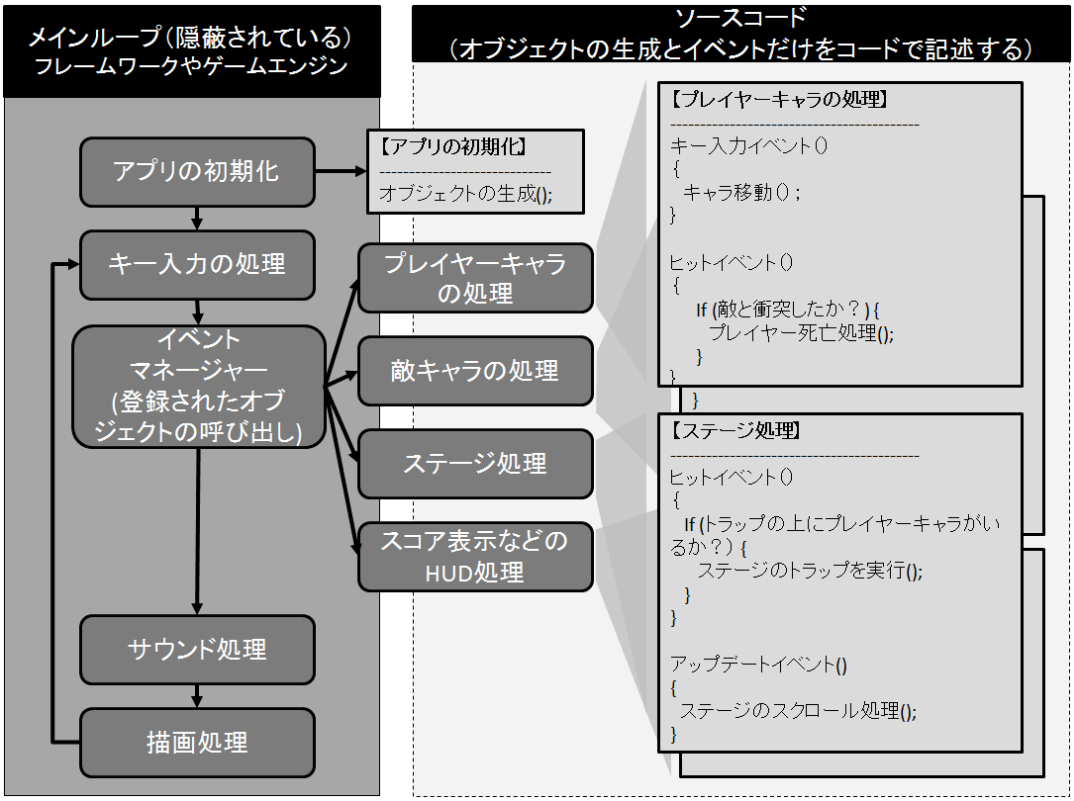


図 A03_01_002 イベント駆動型プログラミング

イベント駆動型プログラミングでは、メインループは隠蔽されています。ユーザーはフレームワークのオブジェクトを継承して作成し、必要に応じたイベントに対してプログラムを記述します。オブジェクト指向プログラムが浸透し、フレームワークやゲームエンジンの再利用が活発に行われるようになると、このイベント駆動型プログラミングがゲームでも使われるようになりました。ただし、ゲームでは特別な処理が多いので、フロー駆動型プログラミングとイベント駆動型プログラミングが混在することも少なくありません。

3 つ目は「データ駆動型プログラミング」です。これは、ユーザーによって作られたデータ（オブジェクト）が主導となって駆動するプログラムです。データドリブンなプログラムとも言います。Unity の場合は、このデータ駆動型プログラミングに近い開発スタイルです。Unity ユーザーは、まず、シーンの中にキャラや背景などのゲームオブジェクトを作成し、それらのデータがどのような振る舞いをするのかプログラミングしていきます。
（図 A03_01_003）。

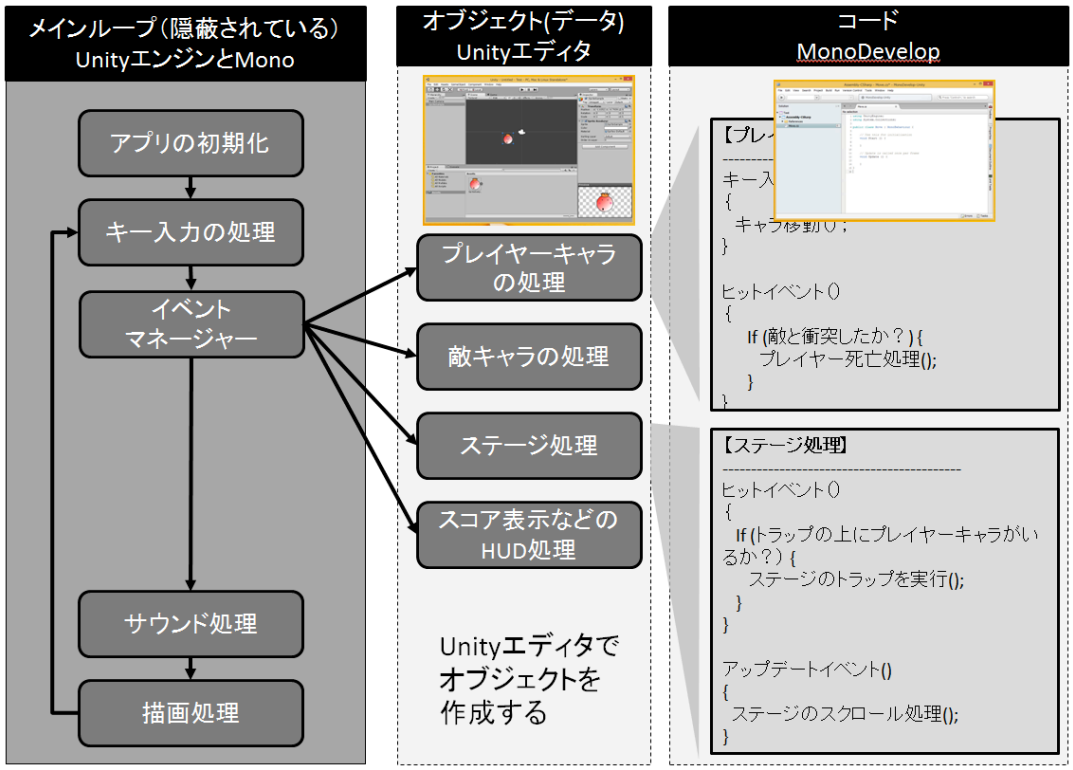


図 A03_01_003 データ駆動型プログラミング

さらに、個々のゲームオブジェクトがイベント駆動型プログラミングによって動作します。メインループは隠蔽され、オブジェクト（ゲームオブジェクト）は Unity エディタの編集で作成されます。オブジェクトのパラメータ（プロパティ）も Unity エディタから入力することができるため、イベントの記述が不必要な処理であれば、コードさえも書かなくて済むのです。

今回紹介した 3 つのゲーム開発スタイルのうち、「フロー駆動型プログラミング」の経験しかない方は、Unity の「データ駆動型プログラミング」に戸惑うかもしれません。最初に、「メインループはどこで記述すればいいんだ？」と思われることでしょう。Unity ではメインループは Unity エンジンの中にあるため、メインループをプログラムで記述する必要はありません。また、プレイヤーの移動スクリプトを作成するには、まずプレイヤーのデータ(オブジェクト)を作成してから記述する必要があります。このように、コードが主体のフロー駆動型プログラミングに慣れている方であればあるほど、Unity の場合は最初のとっかかりで悩むことになるかもしれません。

※なお、ここで紹介した「フロー駆動型プログラミング」「イベント駆動型プログラミング」「データ駆動型プログラミング」の用語の使い方は、専門の方から見ると相応しくないかもしれません。ただ、分かりやすく理解を速めるために、あえて大ざっぱな括りで今回はこのような説明をしています。

シーン

Unity では、ゲームを作る場合、まず「シーン」と呼ばれるゲーム空間を作ることからスタートします。1 つのプロジェクトに複数のシーンを作ることで、ゲームのタイトルやメニューやステージなどを分けて作ることができます（図 A03_02_001）。

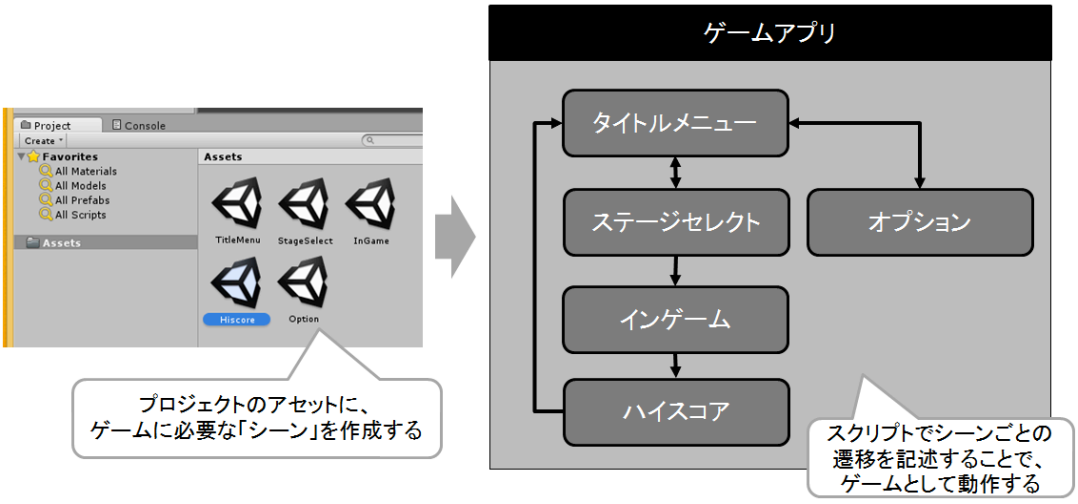


図 A03_02_001 プロジェクトとシーンの関係

新しいシーンには、デフォルトで”Main Camera”と呼ばれるカメラだけが設置されています。3D ゲームであれば、必要に応じてライトや 3D オブジェクトなどを追加で作成して設置します。2D ゲームの場合は、ライトは不要で、「スプライト」と呼ばれる 2D 表示のオブジェクトだけで画面を作ることができます（図 A03_02_002）。

図 A03_02_002 シーンとゲームオブジェクトの関係

また、これらシーンに設置されるものは、すべて「ゲームオブジェクト(GameObject)」から作られています。

ゲームオブジェクト

Unity でゲームを作る場合、シーンに「ゲームオブジェクト」を作成して配置していきます。

ゲームオブジェクトは、「GameObject」メニューから作成できます。2D スプライトであっても 3D のキューブであっても同じです。

試しに 3D ゲームオブジェクトをを一つ作って、シーンに配置してみましょう。

まずは、シーンビューの「2D」ボタンを押して、「3D」に切り替えてください。続けて、「GameObject」メニューから「Create Other」を選び、「Cube」を実行します。すると、画面に箱（Cube）が表示されます（図 A03_03_001）。

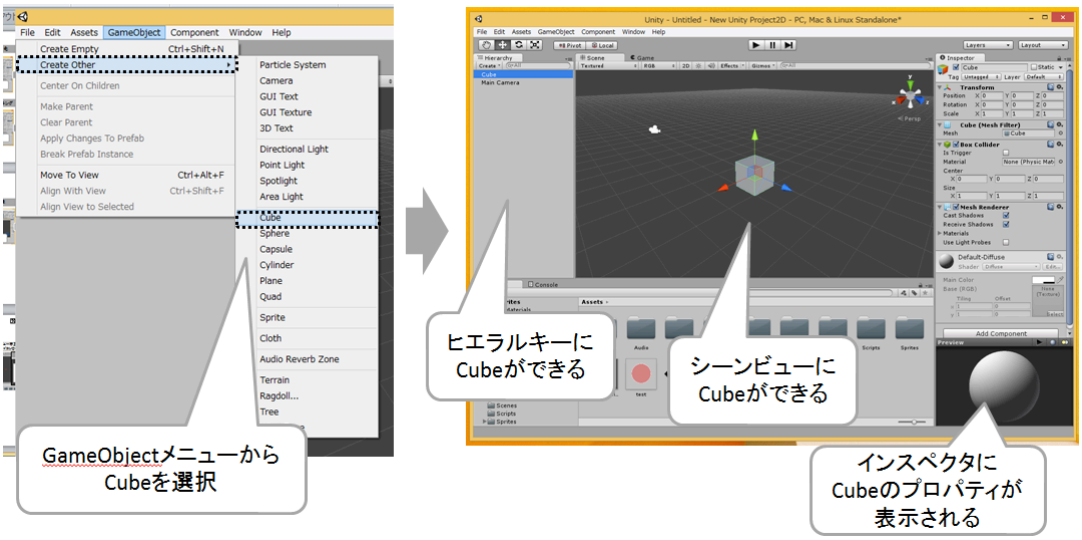


図 A03_03_001 ゲームオブジェクトの作成

もし、作成した Cube が見えない場合は、シーンビューのカメラの位置が離れています。ヒエラルキーに「Cube」が作られているので、これをダブルクリックしてください。自動的にカメラが移動して、シーンビューの中央に Cube が表示されます。

ゲームオブジェクトから作成できるものには、図 A03_03_002 のようなものがあります。

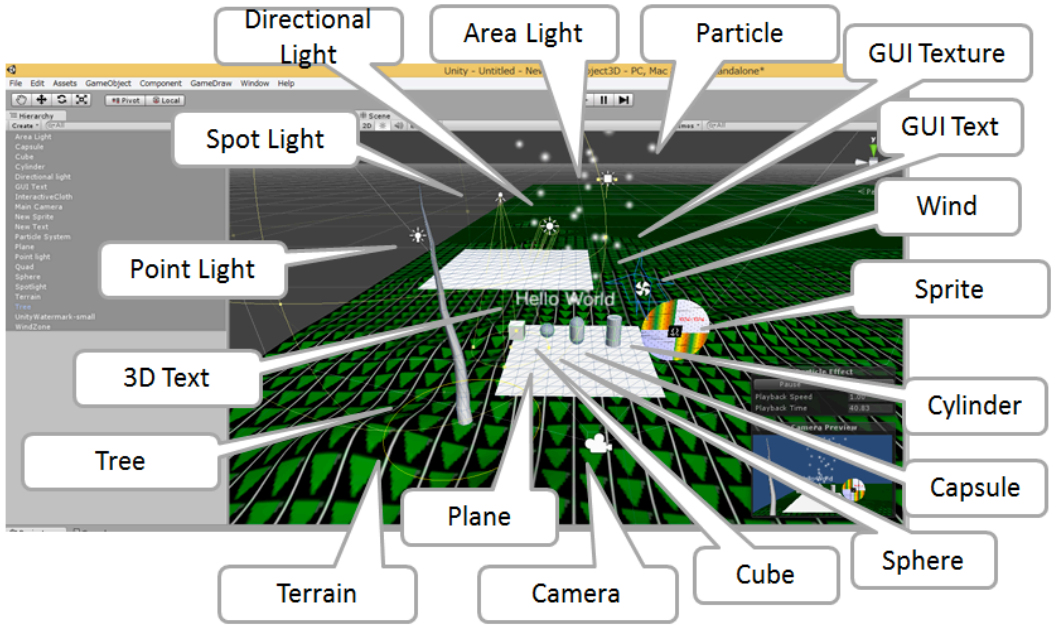


図 A03_03_002 ゲームオブジェクトの種類

表 1.3.3.1 メニューから作れる GameObject の種類	
オブジェクト名	**説明**
Particle System	エフェクトを作るためのゲームオブジェクト。Unity3.5 からは新パーティクルシステムの「Shuriken」が利用される。
Camera	ゲーム中にシーンを撮影するカメラ。複数設置して切り替えることも可能。
GUI Text	GUI 用のテキストを表示するためのゲームオブジェクト。 常に 2D として表示される。
GUI Texture	GUI 用の背景などのグラフィックを表示するためのゲームオブジェクト。常に 2D として表示される。
3D Text	テキストをシーン内の 3D オブジェクトとして表示するゲームオブジェクト。
Directional Light	太陽のようにシーンを一方向から無限に光を照らすライト（無限光、無限遠光源）。
Point Light	豆電球のように一点から光を放つライト（点光源）。
Spotlight	その名の通り一定範囲を照らす「スポットライト」。
Area Light	一定の四角形の範囲を柔らかリアルに表現するライト。ただし、このライトはライトマップへの焼き込み情報用で、リアルタイムに光源計算はされない。また、Unity PRO ライセンスのみ利用可能。
Cube	キューブ形状のゲームオブジェクト。

Sphere	球体のゲームオブジェクト。
Capsule	薬のカプセルのような形状のゲームオブジェクト。
Cylinder	円柱のゲームオブジェクト。
Plane	厚みのない「板」のゲームオブジェクト。複数の格子状のポリゴンで作らている。
Quad	一枚の板ポリゴンのゲームオブジェクト。
Sprite	2D 画像を高速に表示できるスプライトのゲームオブジェクト。
Cloth	布のように柔らかく揺らめくクロス物体を表現できるゲームオブジェクト。
Audio Reverb Zone	サウンドが反響するゾーンを指定するゲームオブジェクト。
Terrain	自然の地形を作成し表示するためのゲームオブジェクト。
Ragdoll	物理演算によって計算される人型の形状を扱うゲームオブジェクト。
Tree	「樹木」を自動的に生成し表示するゲームオブジェクト。
Wind Zone	物理エンジンを使って「風」が吹いているゾーンを設定するゲームオブジェクト。RigidBody コンポーネントのゲームオブジェクトのみ影響を受ける。

このように、Unity のゲームオブジェクトには、様々な機能があります。

ただし、2D ゲームを作るだけであれば、使用するゲームオブジェクトは「Sprite」と「Camera」のみです（エフェクトも使うなら「Paricle System」も）。なので、種類が多いからといって臆することはありません。

なお、前に説明したスプライト作成方法の手順では、プロジェクトブラウザからスプライト画像を直接シーンビューへドラッグ＆ドロップする方法でした。実は、この操作を行うと、自動的に Unity が Sprite ゲームオブジェクトを作成してシーンビューに配置してくれていたのです。便利機能が最初から実装されているわけですね。

コンポーネント

さて、この「ゲームオブジェクト」ですが、Sprite や Camera といった種類ごとに、各ゲームオブジェクトの種類が存在しているわけではありません。ゲームオブジェクトは、シーンの中でオブジェクトを扱うためのもっとも基本的な「器」です。それに「機能」が実装された「コンポーネント」を追加することで、「Sprite」や「Camera」などのゲームオブジェクトになるのです（図 A03_04_001）。

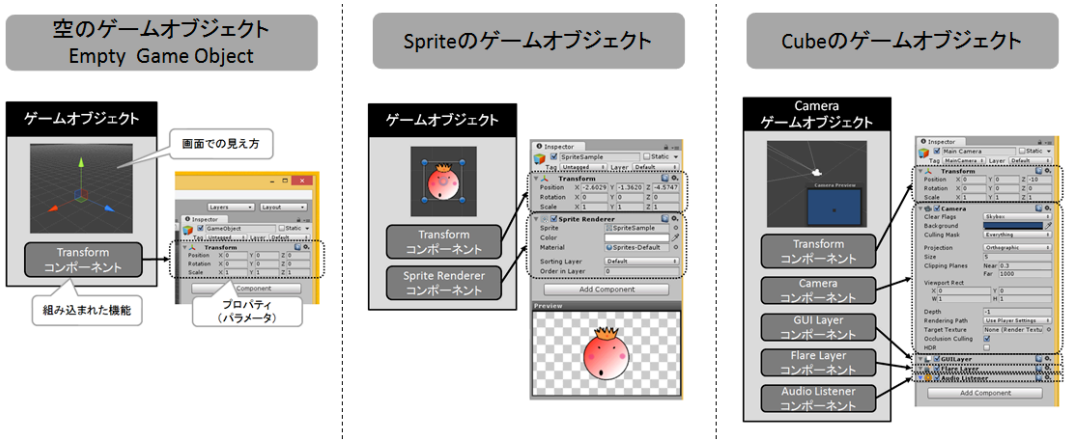


図 A03_04_001 ゲームオブジェクトとコンポーネント

このような仕組から、機能（コンポーネント）を持たない「空のゲームオブジェクト」を作ることもできます。

「Game Object」メニューから「Create Empty」を選択すると、何も機能を持たない「空のゲームオブジェクト」が作成されます。この「空のゲームオブジェクト」は、座標とスケール情報を持つ「Transform Component」だけしかありません。この「空のゲームオブジェクト」にコンポーネントを追加することで、既存のゲームオブジェクトを作ることもできます。

例えば、Sprite オブジェクトであれば、次の手順で作成できます。

1. 「Game Object」メニューから「Create Empty」を選択して、「空のゲームオブジェクト」を作成する（図 A02_04_002）。

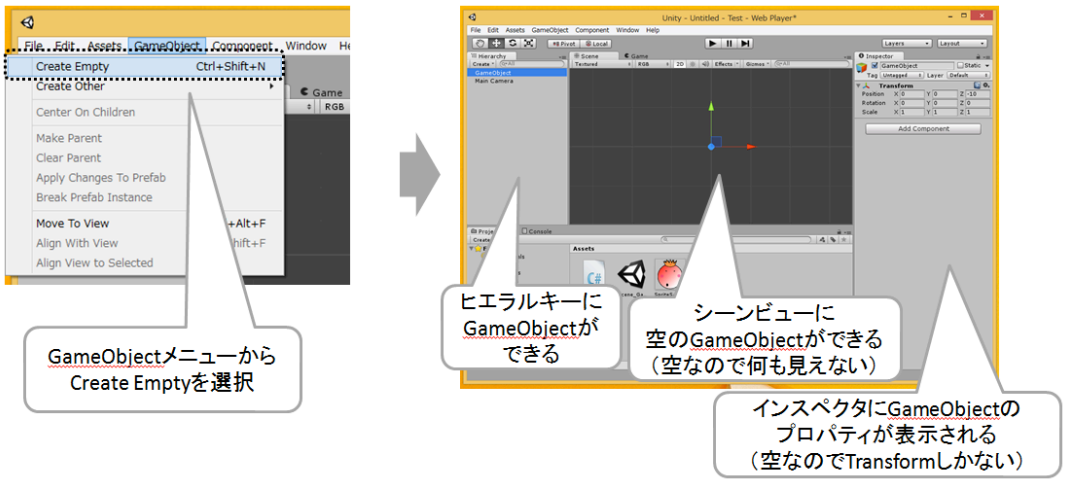


図 A03_04_002 空のゲームオブジェクトの作成

２．ヒエラルキーで「空のゲームオブジェクト」を選択して、インスペクタから「Add Component」ボタンを押す。そして、「Spirte Rendarer」を選択する（図 A02_04_003）。

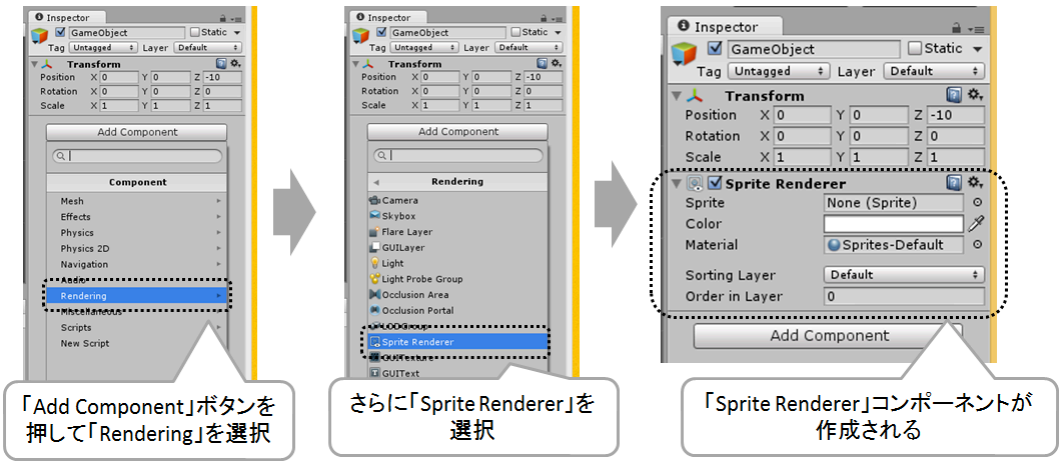


図 A03_04_003 コンポーネントの追加

３．インスペクタの「Spirte Rendarer」の中にある「Sprite」に、スプライト画像を指定する（図 A03_04_004）。

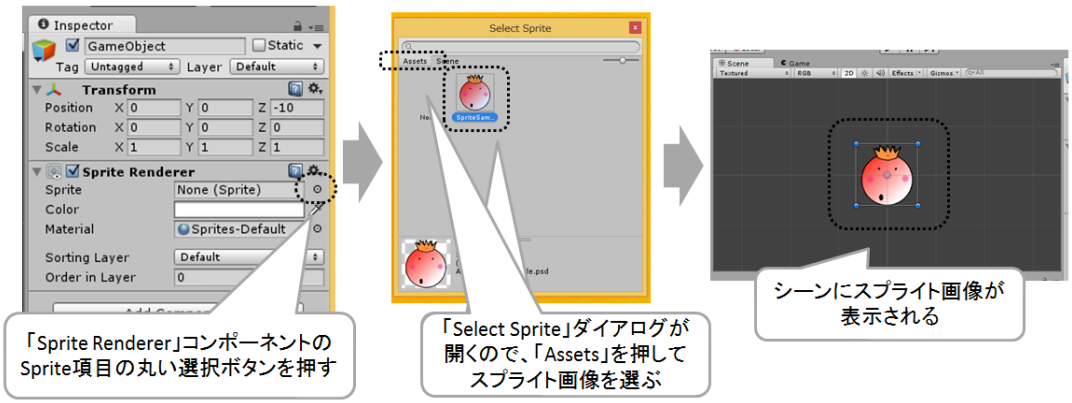


図 A03_04_004 「Sprite Randare」にスプライト画像を設定 r

これで、Sprite ゲームオブジェクトが作成できました。
このように、様々な機能をゲームオブジェクトに持たせることができるのが「コンポー

ネント」なのです[1]。

[1] コンポーネントは、コンポーネント名の横の▼アイコンをクリックすることで、コンポーネントのプロパティリストを表示・非表示を切り替えられます。ただし、コンポーネントを非表示（閉じた）場合、シーンビューのギズモも非表示になります。そのため、Sprite Renderer を非表示にした場合、スプライトをシーンビューで移動させることができなくなります。もう一度、Sprite Renderer の▼アイコンをクリックして表示すれば編集可能になります。

Unity のおもしろい点は、スクリプトさえもコンポーネントであるということです。

そのため、最初に説明したスクリプトファイルをドラッグ＆ドロップする方法以外にも、インスペクタで直接スクリプトを指定して追加することもできます。

インスペクタの「Add Component」ボタンを押して「Script」を選択すると、コンポーネントとして追加できるスクリプトの一覧が表示されます。使いたいスクリプトを選択すると、そのスクリプトがコンポーネントとして追加されます（図 A03_05_001）。

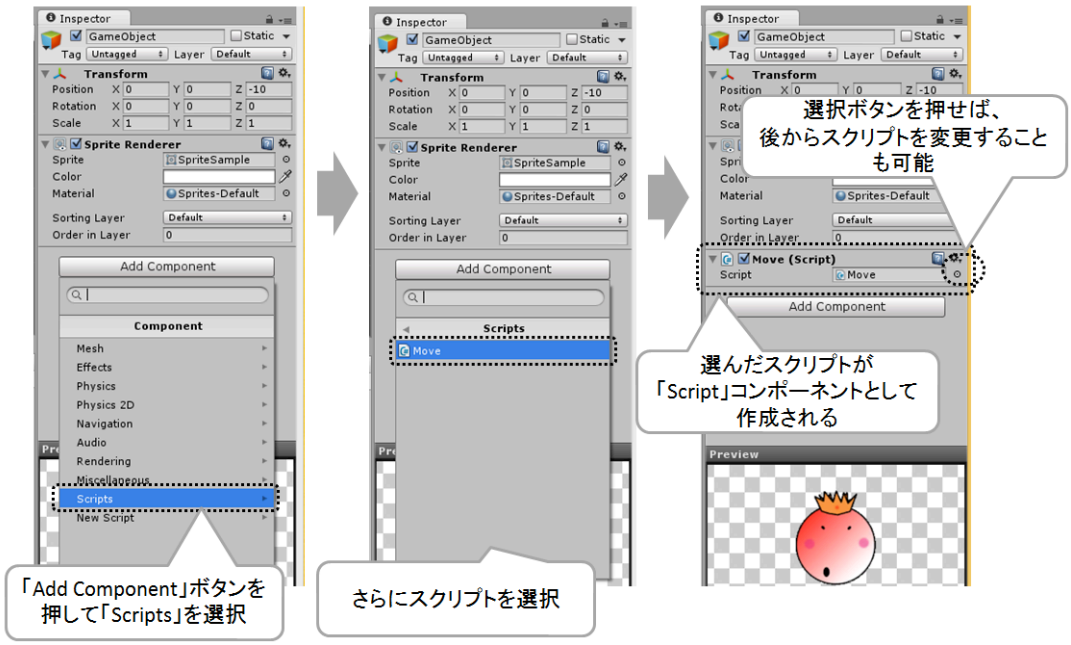


図 A03_05_001 スクリプトコンポーネント

なお、コンポーネントに追加できるスクリプトは、基本的に MonoBehaviuor クラスを継承したクラスだけです。その代り、スクリプトコンポーネントは、1 つのゲームオブジェクトに何個でも追加することができます。

Unity でゲームを作るコツは、すでに Unity が持っている機能についてはコンポーネントを追加して実装することです。逆に、各ゲーム特有のゲームオブジェクトの動かし方や機能については、スクリプトをプログラムして実装します。

汎用性の高いスクリプトを作ってしまえば、既存のコンポーネントと同じく、様々な場面でゲームオブジェクトに利用できるため、プラモデルを作るように簡単にゲームを作ることができます（図 A03_05_002）。

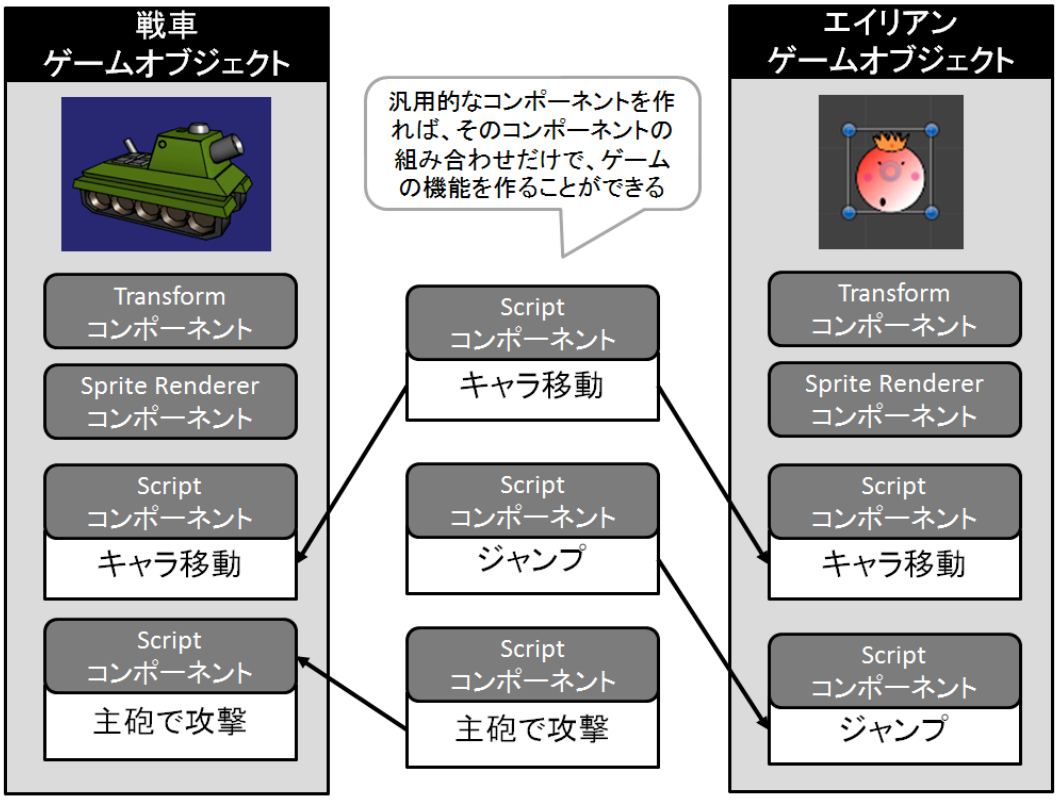


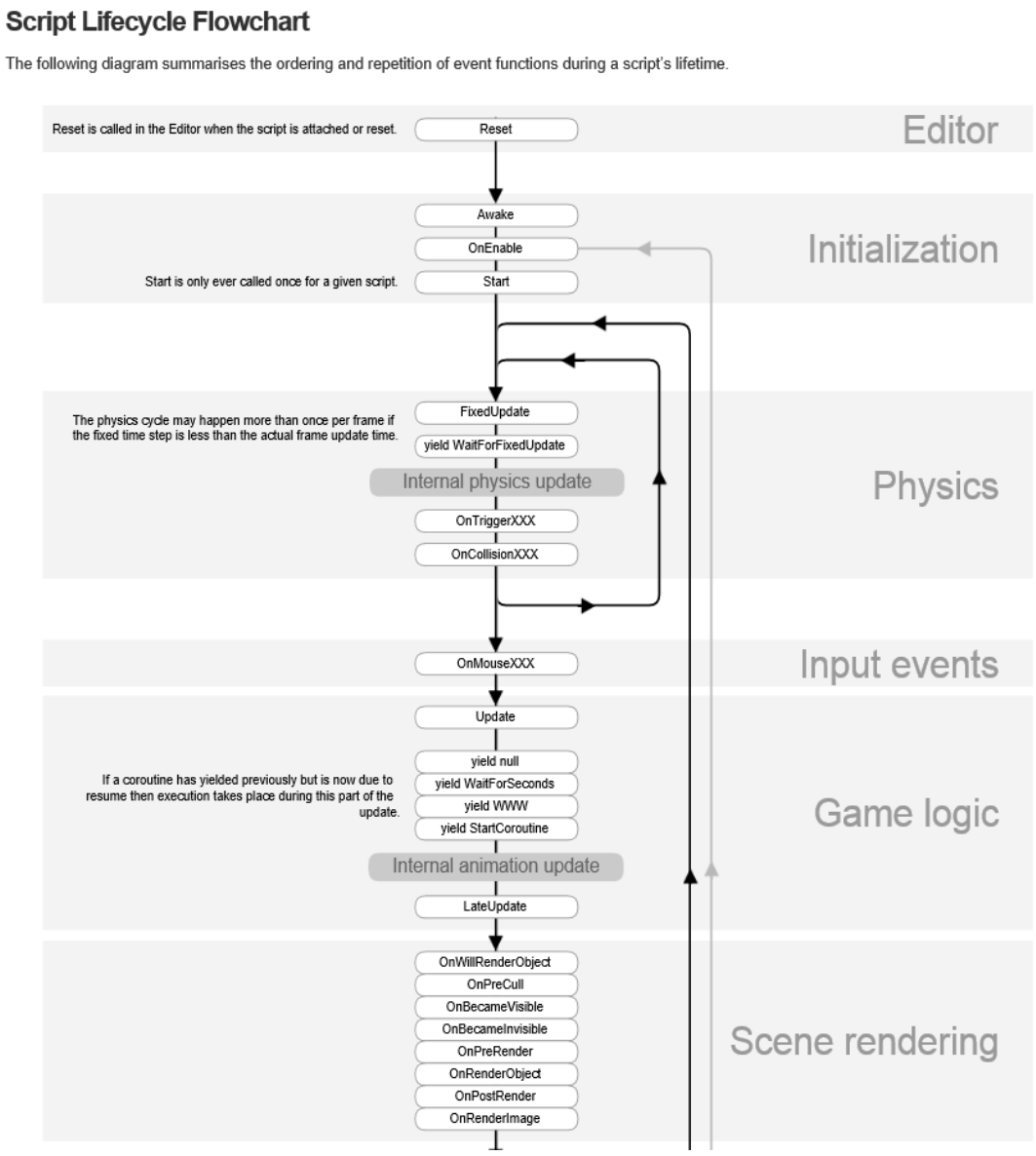
図 A03_05_002 汎用性の高いスクリプトの組み合わせ

これは、Unity の素晴らしい仕組みです。

スクリプトコンポーネントを使いこなすことで、Unity でのゲーム開発が効率的に、かつ楽しくなるのです。

MonoBehaviour のメッセージ呼び出しの流れ

ゲームオブジェクト（MonoBehaviour）では、状態にあわせて Unity エンジンからメッセージが送信されます。この各メッセージの呼び出し順番ですが、Unity Manual では、図 A03_06_001 のように紹介されています。



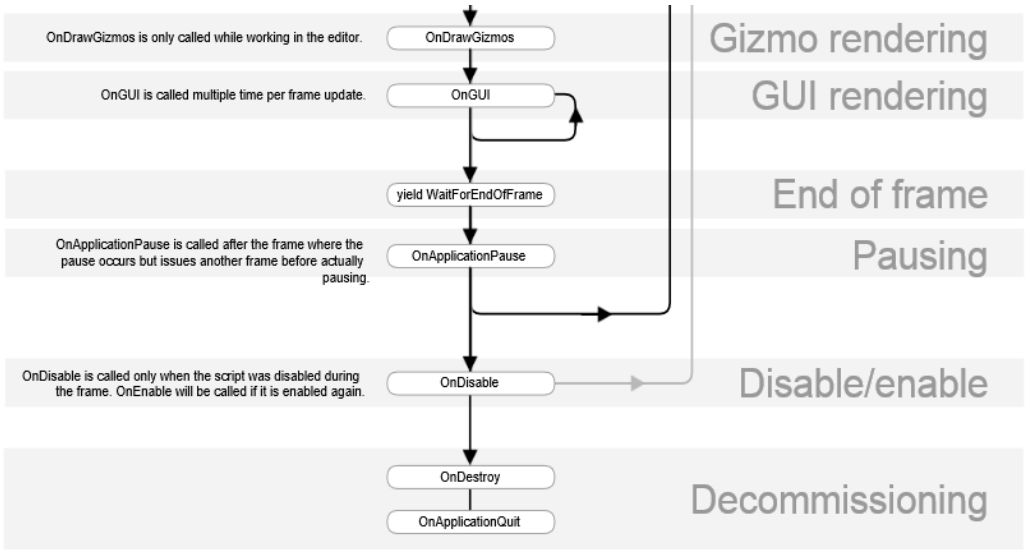


図 A03_06_001 スクリプトライフサイクルフローチャート

※Unity Manual：Execution Order of Event Functions より（http://docs.unity3d.com/Manual/ExecutionOrder.html）

スクリプトの実行順番

1 つのゲームオブジェクトに複数のスクリプトが設定できることを紹介しました。ところで、これらのスクリプトは、どのような順番で実行されるのでしょうか？　実は、Unity ではこれらのスクリプトの順番は決まっていません。通常は、ゲームオブジェクトを参考にロードしたスクリプトの順番で実行されるのですが、シーンが再ロードされるごとに、キャッシュなどの機能が働いて、ゲームオブジェクトのロード順番が毎回同じになりません。ほぼランダムだと言って良いでしょう。

そのため、ゲームオブジェクトの参照などは、すべてのゲームオブジェクトやコンポーネントがロードされた Start メッセージで行うのが安全です。また、一つのゲームオブジェクトに複数のスクリプトを設定した場合は、どちらのスクリプトが先に実行されても問題ないように作っておく必要があります。しかし、どうしてもスクリプトの実行順番を決めたい場合もあるでしょう。

そこで、Unity には、スクリプトを優先的にどの順番で実行させるかという機能があります。Unity エディタのメニューから「Edit」の「Project Settings」を選択して、「Script Excution Order」を選びます。インスペクタに「MonoManager」が表示されるので、優先して実行したいスクリプトをプロジェクトブラウザからドラッグ＆ドロップしてください。スクリプトを登録すると、上から下に向かって優先的にスクリプトが実行されます。スクリプト名をドラッグして順番を変えることができます。「Default Time」よりも上に配置すれば、一般的なスクリプトよりも早く実行されます。下に配置すれば遅く実行され

ます。また、表示されているスクリプト名の横の数値を変えることで、スクリプト優先度を変えたり、実行時間のタイミングを変えることができます（図 A03_07_001）。

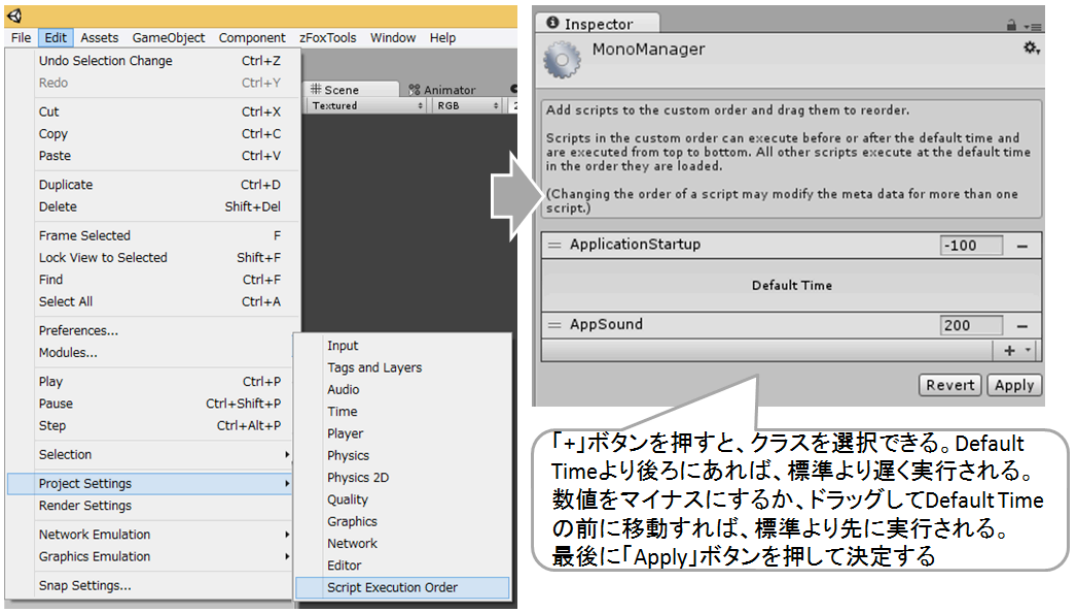


図 A03_07_001 スクリプトの実行優先度を変更する

なお、このインスペクタに表示されていないスクリプトは、ランダムに実行されると考えてください。

この機能は、あくまで最終手段です。基本的にはスクリプトの実行順番に影響されない方法でプログラミングしましょう。

Unity がゲームを動作させる仕組み

さて、Unity でゲームを作る場合の、ざっくりとした流れが理解できたかと思います。

では、実際に Unity では、これらのゲームオブジェクトやコンポーネント、そして、スクリプトをどのように動作させているのでしょうか？

まず、Unity が持つ 3D や 2D 機能についてですが、プラットフォームごとにネイティブに動作する Unity エンジンが用意されており、ゲームオブジェクトやコンポーネントなどを実行しています。

次にスクリプトですが、こちらは Mono と呼ばれるマルチプラットフォーム向けの言語エンジンが処理しています（図 A03_06_001）。

図 A03_06_001 Unity と Mono の関係

Unity が扱う Mono では AOT コンパイラというものが搭載されており、MonoDevelop で開発した C#や Java スクリプトは、共通中間コードの CLI に変換された後、さらに各プラットフォームで高速に動作するネイティブコードに変換されてビルドされます。そのため、スクリプトでゲームを開発しても、高速に動作させることが可能なのです。

とは言っても、プログラムの仕方によっては、C++の 10 倍以上遅くなることもあります（図 A03_06_002）。

そこで、ポイントとなるのが、Unity エンジンから提供される機能やコンポーネント、そして、API などの活用です。これらの機能は、ネイティブな環境で動作する Unity エンジンの一部であるため、高速に動作します。例えば、アタリ判定や物理シミュレーションなどは、C#でオリジナルのプログラムを作るよりも Unity の物理シミュレーションを利用するほうが圧倒的に高速です。C#で高速なアルゴリズムを開発するよりも、手っ取り早く高速化の近道となります。

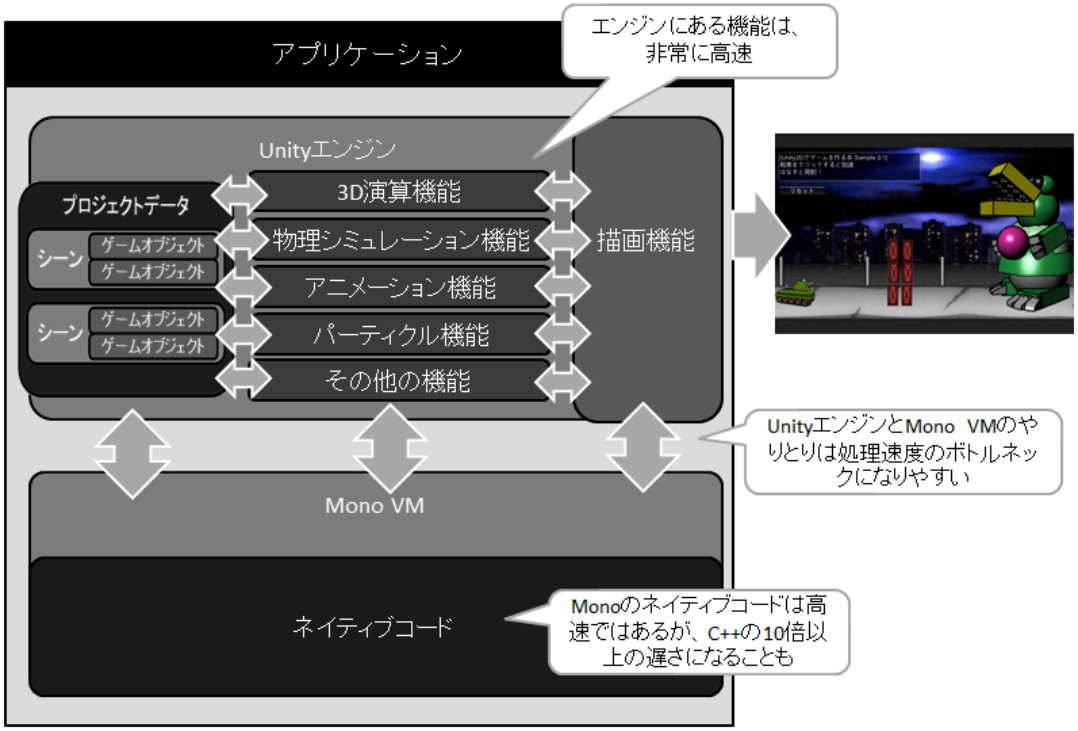


図 A03_06_002 Unity を効果的に高速に動作させる方法

Unity で短時間に、かつ、レスポンスの良い高速なゲームを作るなら、Unity の機能を隅々まで知り尽くして使いこなすことがポイントとなりますので、覚えておいてください。

第２章
Unity2D 入門（初心者向け）

2.1. Unity2D とスプライト画像

Unity の基本を理解したら、いよいよ Unity2D の機能についてご紹介しましょう。

プロジェクトを作成する

まずは、プロジェクトを作りましょう。作り方は覚えていますか？

「File」メニューから「New Project...」を選択して、Project Wizard を開きます。続けて、プロジェクト名を入力したら、「Setup defaults for:」のリストボックスに表示されている「3D」を「2D」に変更して、「Create」ボタンを押します。Unity エディタが Unity2D に特化したモードで再起動し、新しいプロジェクトが開きます（図 A03_01_001）。

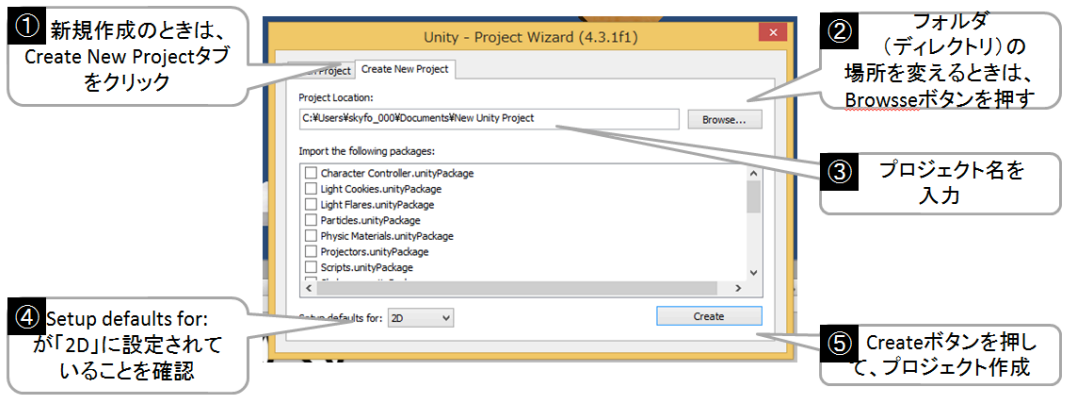


図 B01_01_001 Unity2D 用のプロジェクトを作成

これで準備ができました。

Assets にフォルダを作る

Unity でプロジェクトを作成したら、最初にプロジェクトブラウザで assets フォルダの下に作業用の「フォルダ」を作成しましょう。

Unity では、どのようにフォルダを作っても良いのですが、一部「Plugin」フォルダや「Resources」フォルダは、その名前で「機能」が決定します。そのため、最初に Unity で一般的に「お作法」として使われているフォルダ名でフォルダを作成しておくことで、安全にプロジェクトを作成・編集することができます。

Unity でフォルダを作成する場合、まずは、プロジェクトブラウザでフォルダを作成したい親フォルダを選択して、「Create」のドロップダウンリストから「Folder」を選択します。親フォルダを右クリックして、ポップアップメニューの「Create」から「Folder」を選択することも可能です（図 A02_03_001）。

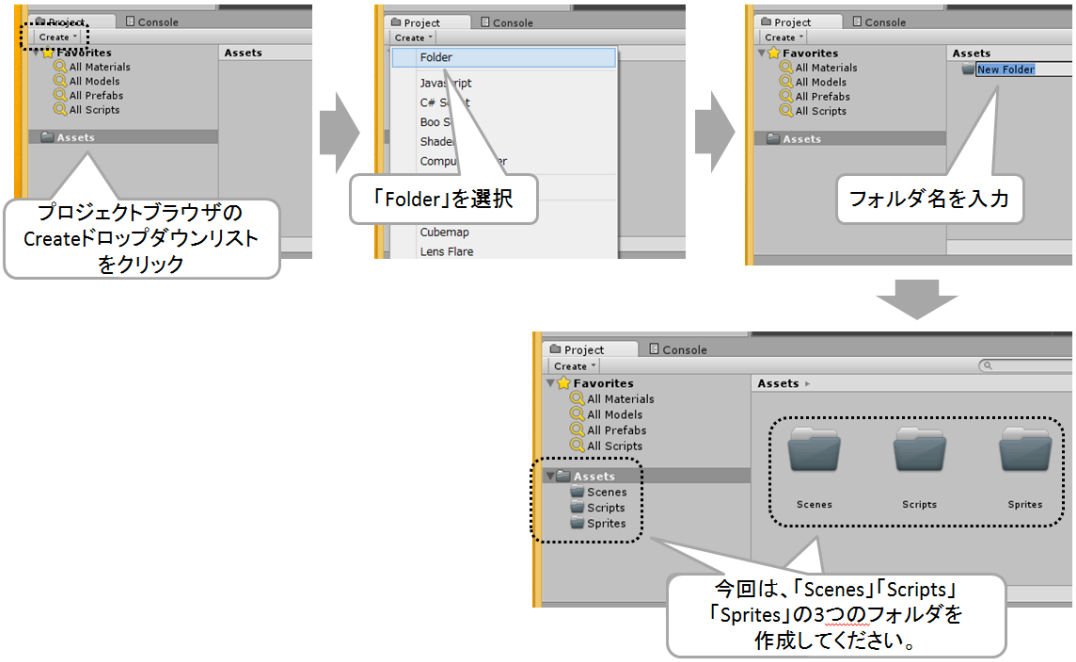


図 B01_02_001 プロジェクトブラウザで Assets にフォルダを作る

新規フォルダを作成した後は、フォルダ名を付けます。
フォルダ名ですが、Unity では次のような基本ルール（お作法）で付けます（表 2.1.2.1）。

表 2.1.2.1 Unity のフォルダ名

フォルダ名	説明
Scenes	ゲームの一画面を構成するシーン(*.scn)を保存するフォルダ。
Prefabs	ゲームで何度も使われるオブジェクトを「プレハブ」として保存するフォルダ。
Scripts	ゲームで使うスクリプトを保存するフォルダ。
Sprites	ゲームで使う 2D グラフィックのスプライトデータを保存するフォルダ。このフォルダ名のフォルダに画像を追加すると、自動的にテクスチャタイプが Sprite として登録される。
Animation	ゲームで使うアニメーションデータを保存するフォルダ。
Materials	ゲームの 3D モデルデータに設定するマテリアルデータ（テクスチャデータ）を保存するフォルダ。
Physics Materi	ゲームの物理エンジンで使用するフィジカルマテリアルを保存する

als	フォルダ。
Fonts	ゲームで使用するフォント画像データを保存するフォルダ。
Audio	ゲームで使用する BGM や SE などのサウンドデータを保存するフォルダ。
Resources	ゲームプログラム内ではなく、ゲームプログラムから別ファイルとしてデータをロードして扱うフォルダ。Resources.Load という API でロードすることができる。
Editor	Unity のエディタ機能を拡張するためのスクリプトを保存するフォルダ。なお、ゲームを作るだけであれば、このフォルダを作成する必要はありません。
Plugins	Unity で作成したゲームを、iPhone や Android など個々のプラットフォームで動作させるためのネイティブプラグインを保存するフォルダ。このフォルダでは、さらに iPhone なら”iOS”、Android なら”Android”というように個々の名前が決まっている。なお、ゲームを作るだけであれば、このフォルダを作成する必要はありません。

ちょっと量が多いですが、先に必要なフォルダを作って置けば、ゲーム開発中にファイルが増えて収集が付かなくなるといったことが未然に防げます。今回のサンプルプログラムであれば、”Scenes”,”Scripts”,Sprites”の 3 つのフォルダを作成しておきましょう。

なお、機能が決まっている「**Resources**」「**Editor**」「**Plugins**」の 3 つ以外のフォルダは、自由にフォルダ名を付けられますが、Unity のサンプルプログラムや、多くの開発者がこの「お作法」に乗っ取って作られています。他の資料やプログラムを参考にするときにも便利ですので、このガイドラインに沿った名前を付けて置いた無難です。

一方、「**Resources**」「**Editor**」「**Plugins**」の 3 つのフォルダ名は Unity で機能が予約されており、別の意味のフォルダとして扱うことはできません。特に「**Editor**」「**Plugins**」は、Unity の動作に影響を及ぼすフォルダです。基本的にゲームだけを作る場合は、編集の必要のないフォルダなので、同名のフォルダを違う目的で作ったり、スクリプトを保存しないようにしましょう（これらのフォルダは、”Scenes/Editor”といったように孫フォルダの名前で使ったとしても、Editor 機能が働くので注意が必要です）。なお、「**Editor**」「**Plugins**」のフォルダは、Asset Store でプラグインを購入すると自動で作成される場合があります。このように自動で作成された「**Editor**」「**Plugins**」は削除しないように注意してください。

また、フォルダ名やファイル名には日本語を付けないほうが無難です。開発している環境によっては正常に動作しなくなる場合があります。

また重要なことなので、もう一度言いますが、プロジェクトブラウザで管理するこの Assets フォルダは、プロジェクトブラウザ以外で、ファイルの移動・削除・リネームなどを行うと、Unity がプロジェクトを構成するために管理している「メタデータ」が破損する原因となります。ファイル・フォルダの移動・削除・リネームなどは、必ずこのプロジェクトフォルダで行ってください。

ただし、ファイルの上書きは、エクスプローラから行う必要があります。プロジェクトブラウザで同名のファイルをドラッグ＆ドロップしても上書きにはならず、ファイル名の

末尾に番号を付けられてアセットが新規追加されてしまいます。

なお、メタデータが壊れると、設定していたゲームオブジェクトのプロパティがリセットされてたり、プロジェクトブラウザの挙動がおかしくなることがあります。このような場合は、「Assets」メニューから「Reimport All」を実行することで回復することがあります。が、逆に、プロジェクトがさらに破損することもあります。とにかく慣れるまで、プロジェクトブラウザ以外で Assets フォルダの中を編集しないように気を付け、もし、おかしくなったら、プロジェクトフォルダのバックアップを取ってから、「Reimport All」することを覚えておきましょう。

スプライト画像の作成方法

さて、Assets フォルダの準備ができたら、次はスプライト画像を用意しましょう。

スプライト画像は、簡単に分けると「抜きのないベタ画像」「抜き（透過色）のある画像」「アルファブレンドする画像」の 3 種類に分けることができます（図 A03_03_001）。

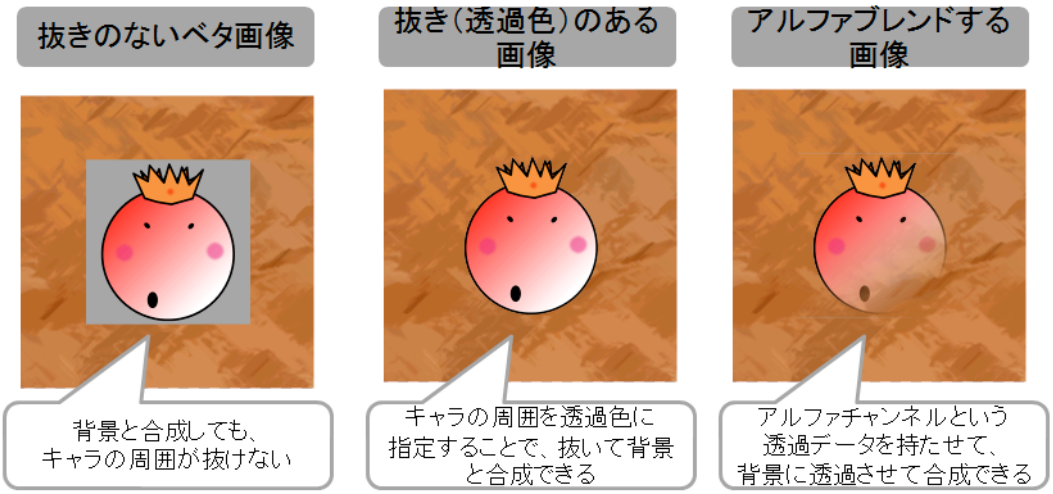


図 B01_03_001 スプライト画像の種類

- **抜きのないベタ画像**
  背景と合成してもキャラの周囲が抜けないベタな画像です。一番高速に描画できる画像です。また、BMP から PSD ファイルまで、どのファイルフォーマットでも保存できます。

- **抜き（透過色）のある画像**
  透過色（抜き色）を指定して、画像の不要な部分を抜いて背景と合成できる画像です。単純な合成なので、高速に描画できます。ただし、保存できるファイルフォーマットは、PNG や PSD ファイルなど、透過色を保存可能なファイル形式のみとなります。

・ **アルファブレンドする画像**

透過色を「抜き」だけでなく、段階を持った「透明度」として情報を持っている画像です。このような画像を背景と合成することを「アルファブレンド（またはアルファブレンディング）」と言います。幽霊のような半透明なキャラを表現できる他に、画像の輪郭を少し透過させることで綺麗な合成を実現することができます。ただし、アルファブレンドできる画像には、「アルファチャンネル」という画像情報を付加できるファイルフォーマットでなければいけません。PSD ファイル,PNG ファイルで保存できます（筆者のお勧めは、編集がしやすい PSD ファイルです）。

「抜きのないベタ画像」は、どんな画像編集ソフトでも作成できます[2]。

しかし、「抜き（透過色）のある画像」「アルファブレンドする画像」については、Photoshop か GIMP などの透過色やアルファチャンネルが編集できるソフトでなければ作成できません。

まず、「抜き（透過色）のある画像」の場合ですが、Photoshop や GIMP では、透明なレイヤーを作成し、この上に画像のレイヤーを作成して、透過する部分を抜いて画像を PSD ファイルで保存します（図 A03_03_002）。

[2] Photoshop で「抜きのないベタ画像」を確実に作成する場合は、「レイヤー(F)」メニューの「画像を統合(F)」を実行します。このレイヤーの画像統合をしなかった場合、例えレイヤーが 1 枚であっても、目に見えないアルファ情報が残ってしまうことがあります。Unity で「抜きのないベタ画像」をインポートしたときに圧縮画像として認識されないときは、画像統合をしてみてください。

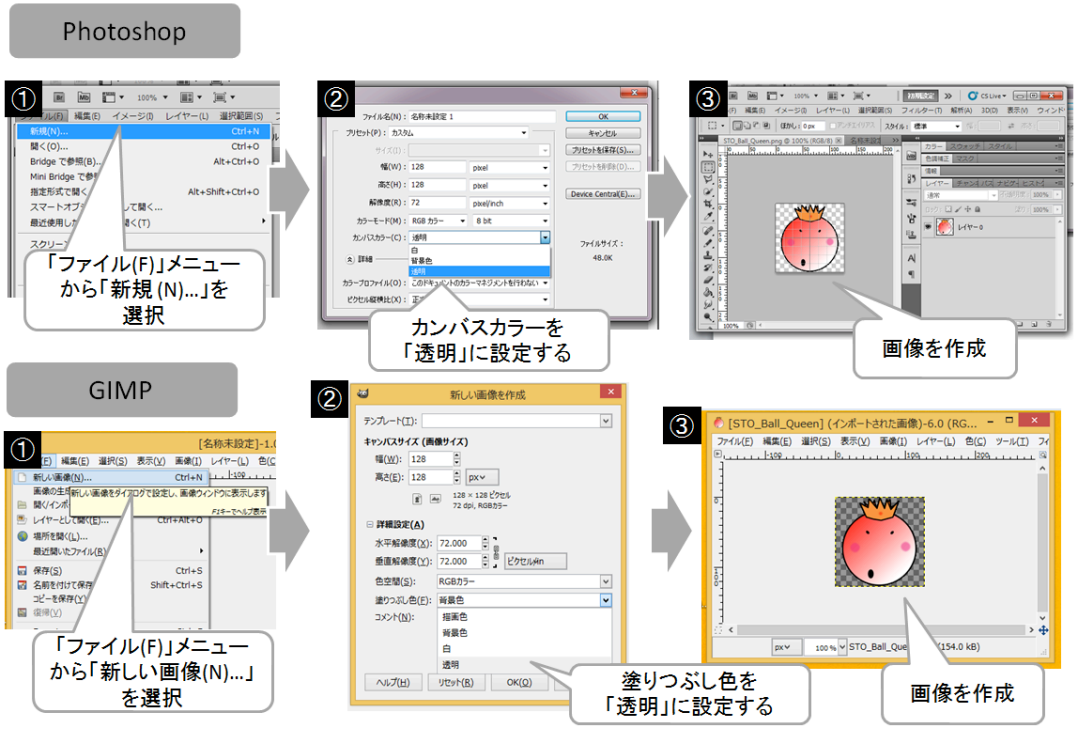
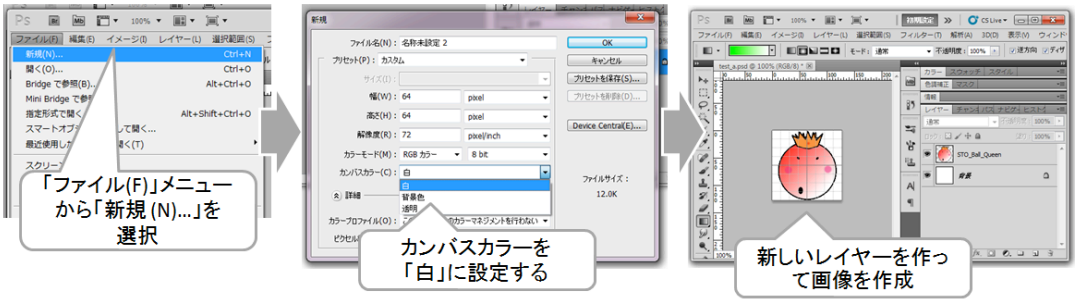


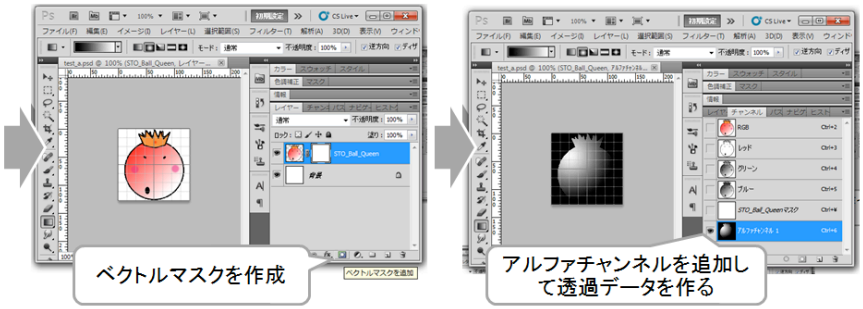
図 B01_03_002 「抜き（透過色）のある画像」の作成方法

一方、「アルファブレンドする画像」の場合は、もう少し複雑になります。

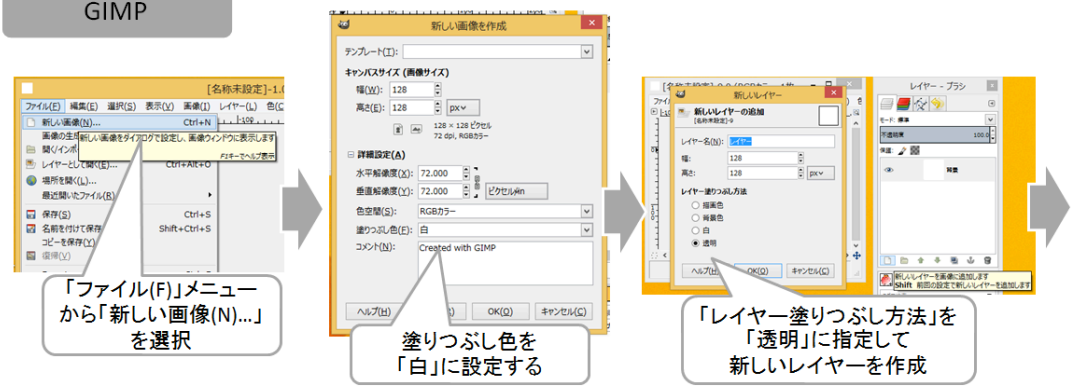
Photoshop や GIMP で、まず透明なレイヤーを作成し、その後べた塗りのマスクを作成します（Unity では、このベタ塗のマスクがないと正しくアルファブレンディングできないので注意してください）。できたら、このレイヤーの上に画像を作成します。さらに、ここから、どれくらい画像を透過させるのか「チャンネル」に「アルファチャンネル」を追加して編集します。「アルファチャンネル」が作成できたら、PSD ファイルで保存します。GIMP の場合は、「File」の「Export as...」メニューから PSD ファイルでエクスポートしてください（図 A03_03_003）。

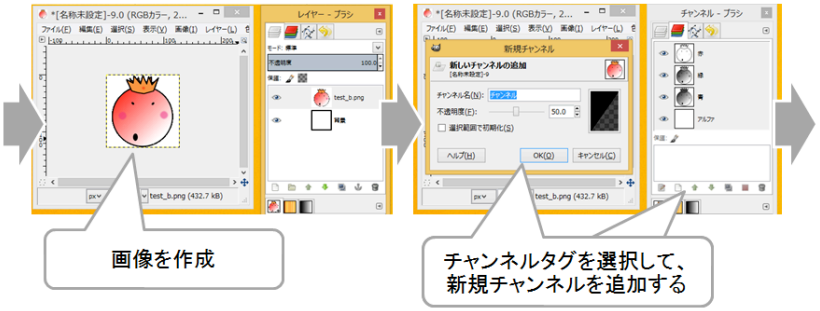
Photoshop





GIMP





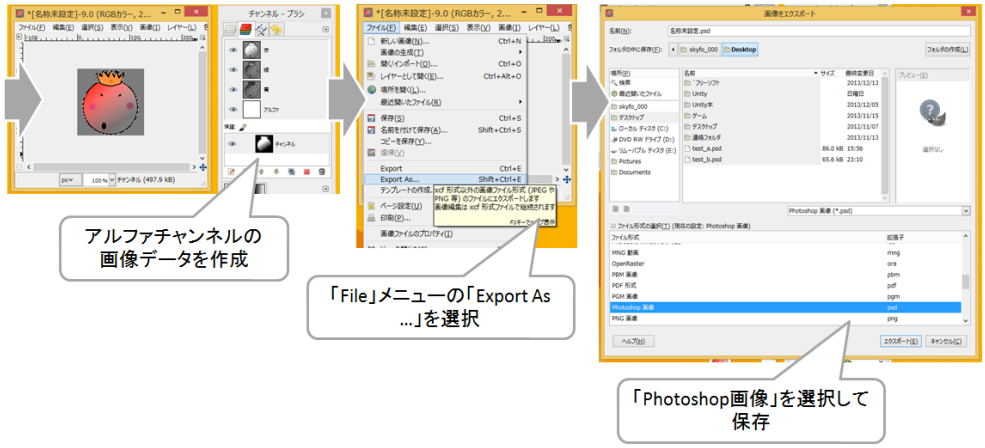


図 B01_03_003 「アルファブレンドする画像」の作成方法

この「アルファブレンドする画像」の場合、単純にキャラを幽霊のように透過させることができる他に、図 B01_03_004 のように輪郭線のアンチエイリアス部分を透過させて、キャラと背景の境界線を綺麗にすることもできます。

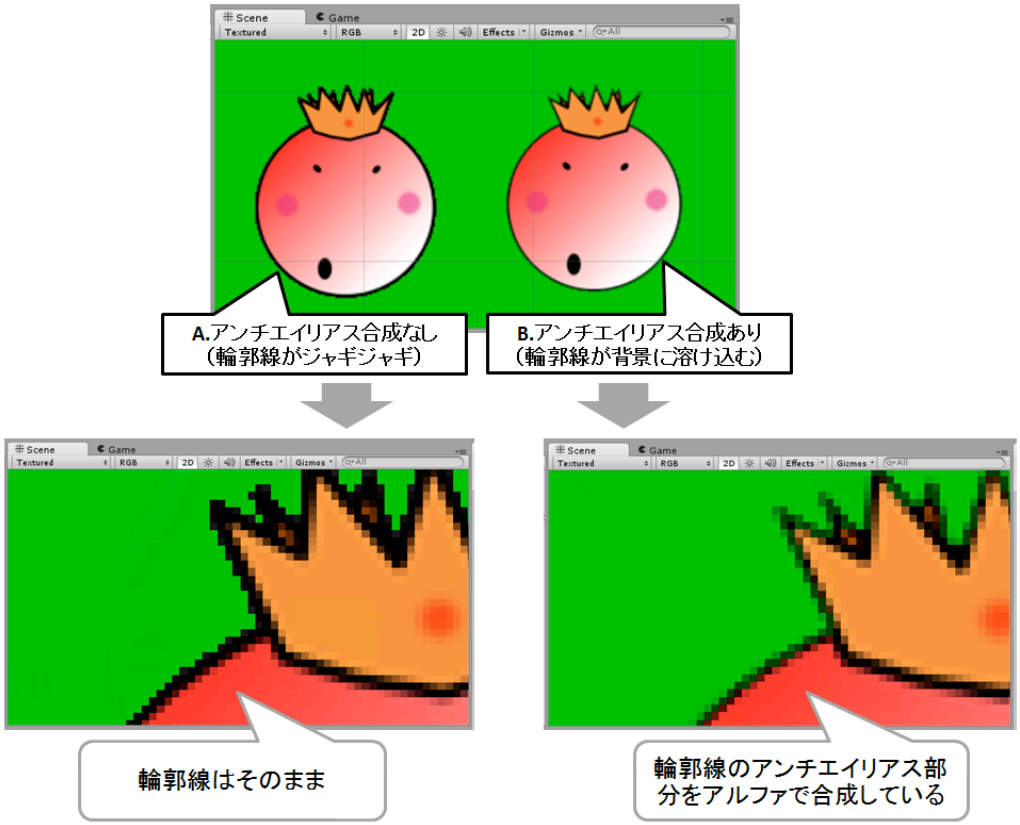


図 B01_03_004 アンチエイリアス合成

特にアドベンチャーゲームのように、アニメ絵の美しいキャラと背景を合成する場合は、この処理が必須となります。このようなアルファチャンネルのデータの作り方には、様々な方法がありますが、筆者は Photoshop を使って図 B01_03_005 の方法でやっています。

図 B01_03_005 Photoshop による背景とアルファブレンドするアンチエイリアスの作り方

この方法は、GIMP でも可能です。操作は変わりますが、データ作成の流れはだいたい同じなので、GIMP を使っている方は挑戦してみてください。

なお、アニメ絵のような「線」と「塗り」で描ける絵については、Photoshop のようなビットマップツール(ドットを編集することがメインのツール)ではなく、Illustrator やフリーで公開されている Ink Space などのベクトルツール（ドットではなく線や塗を編集することがメインのツール。ドローツールとも言う）を使えば、PNG ファイルでセーブするだけで、自動的に輪郭線のアンチエイリアスがアルファチャンネルで透過色として設定されます。ベクトルツールで絵を描くほうが得意な方は、ぜひ試してみてください。

2.2. Unity2D とは？

次は、Unity2D について、もう少し大きな視点から説明していきましょう。

Unity2D の構成

Unity2D は、大きく分けて「スプライト」「スプライトインポータ（Texture インポータ）」「Physics2D」「Mecanim」の 4 つの機能として考えることができます。

まず 1 つ目は、画面に 2D 画像を表示する「スプライト」です。

このスプライト機能は SpriteRendarer コンポーネントによって実現されており、スプライトの高速表示だけでなく、ソーティングレイヤーやオーダー機能によって、スプライトの前後関係を個別に指定することができます。また、3D ゲームオブジェクトと同様に、Transform コンポーネントの z 値を指定することでも、前後関係を付けることも可能です（図 B02_01_001）。

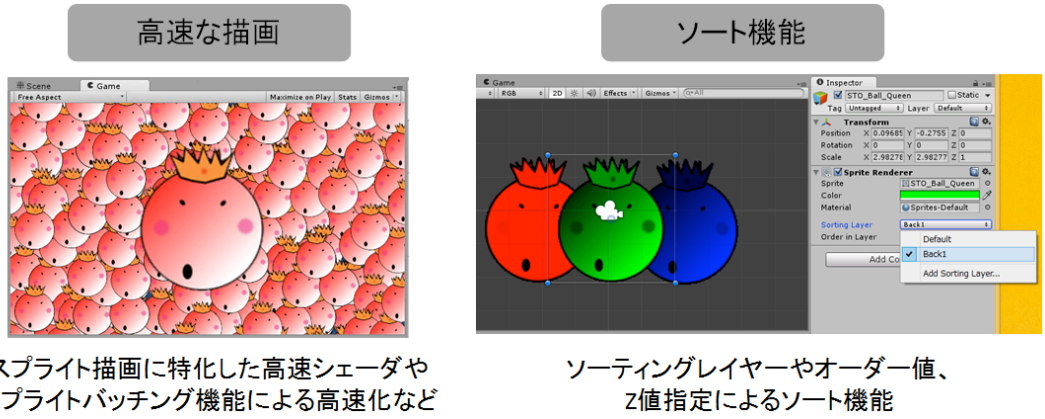


図 B02_01_001 Unity2D の「スプライト」

2 つ目は、このスプライトを表示するための「スプライトインポータ（従来の Texture インポータを拡張したスプライト画像取り込み機能）」です。Unity2D のスプライトインポータでは、1 枚の画像から 1 枚のスプライトを設定できる他に、1 枚の画像から複数のスプライト画像を設定することもできます、逆に、複数のスプライト画像を結合して 1 枚のアトラス（スプライトが集まった 1 つの画像データ。スプライトシートとも言います）として扱うこともできます。ただし、この機能は Unity PRO ライセンスのみで使えます（図 B02_01_002）。

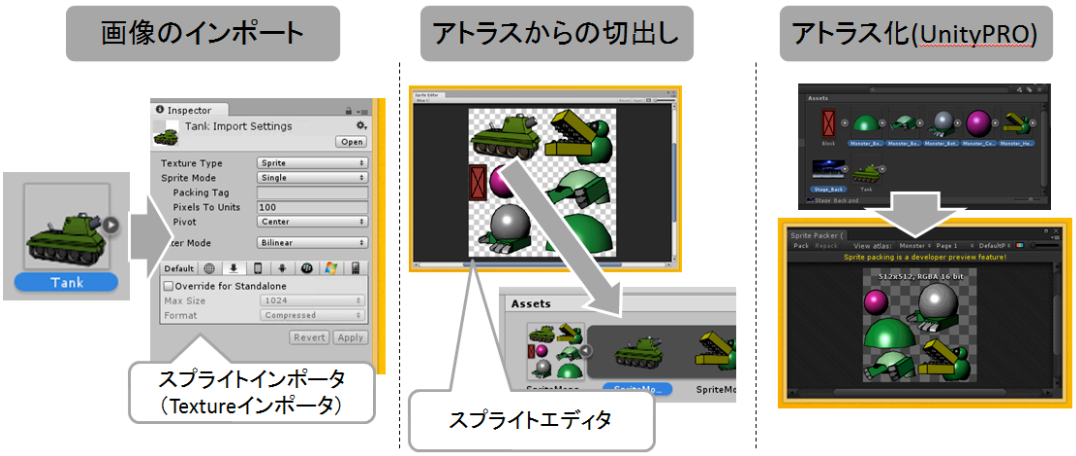


図 B02_01_002 Unity2D の「スプライトインポータ」

3 つ目は、スプライト画像を 2 次元の物理シミュレーションで動かせる「Physics2D」です（図 B02_01_003）。

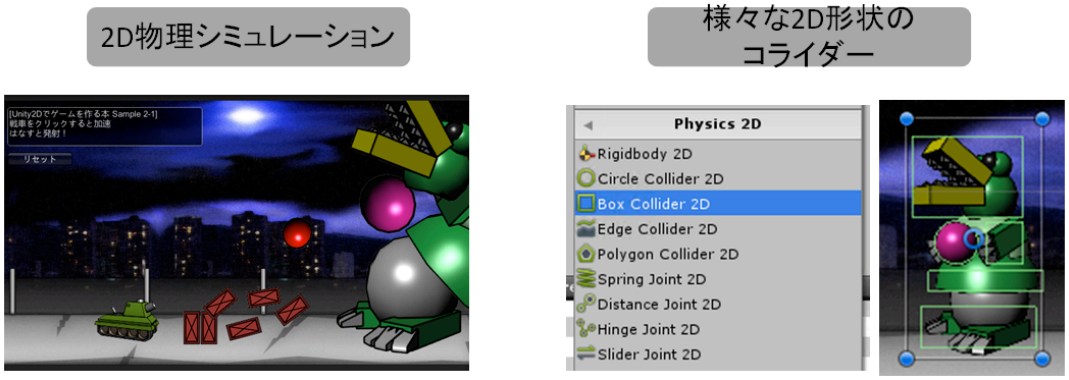


図 B02_01_003 Unity2D の「Physics2D」

Unity には、元々3D ゲーム用の「Physics」と呼ばれる 3D 物理シミュレートを実現するコンポーネントが実装されています（本書では、以後 3D 物理シミュレーションを「Physics3D」と表記します。ただし、3D 物理シミュレーションを実現する Physics コンポーネントはそのまま「Physics」と表記します）。これを使って 2D の物理シミュレートを行うこともできるのですが、Unity2D では、さらに高速に処理できる Physics2D が搭載されました。

最後に 4 つ目ですが、「Mecanim」のスプライト対応です。

Mecanim とは、元々は Unity4.0 から追加された 3D キャラクタのアニメーション機能です。3D キャラクタモデルや 3D オブジェクトを、アニメーター（Animator ビュー）とアニメーションビュー(Animation ビュー)で自由自在に編集できるのできます。この強力な Mecanim が Unity2D にも対応しました（図 B02_01_004）。

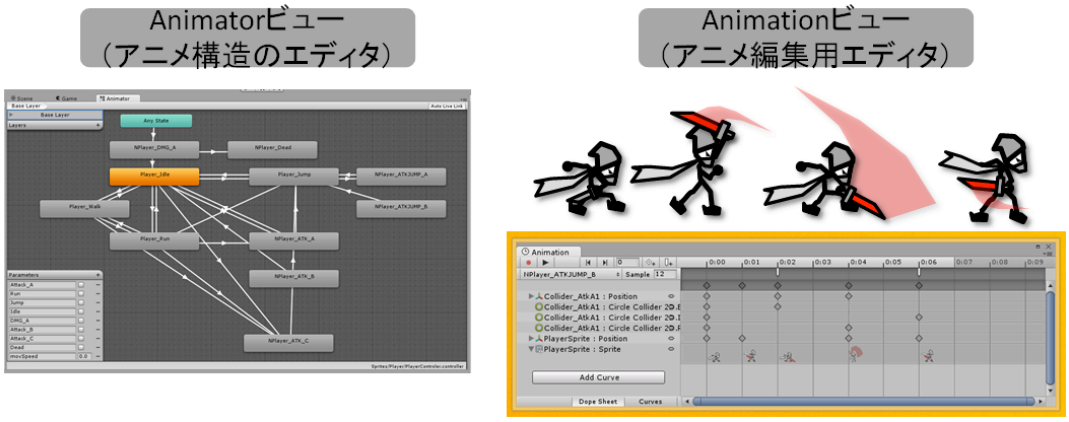


図 B02_01_004 Unity2D の「Mecanim」

これにより、Unity でも簡単に 2D アニメーションを作れるようになっただけでなく、多関節キャラなどの複数のパーツを用いた 2D キャラのアニメーションも Unity だけで可能となったのです。

Unity3D と Unity2D の関係

さて、Unity2D の基本的な構造が分かったところで、次はこの Unity2D が本来の 3D ゲームを作る Unity とどのように関係しているのか説明しましょう。

Unity2D は、Unity の機能の一部であり、ゲームオブジェクトに Unity2D の機能を実装したコンポーネントを追加することで、スプライト機能などを実現しています。おもしろいことに、Unity の 3D ゲームを実現する機能（以後、本書ではこれを「Unity3D」と表記します）と Unity2D は共存可能です（図 B02_02_001）。

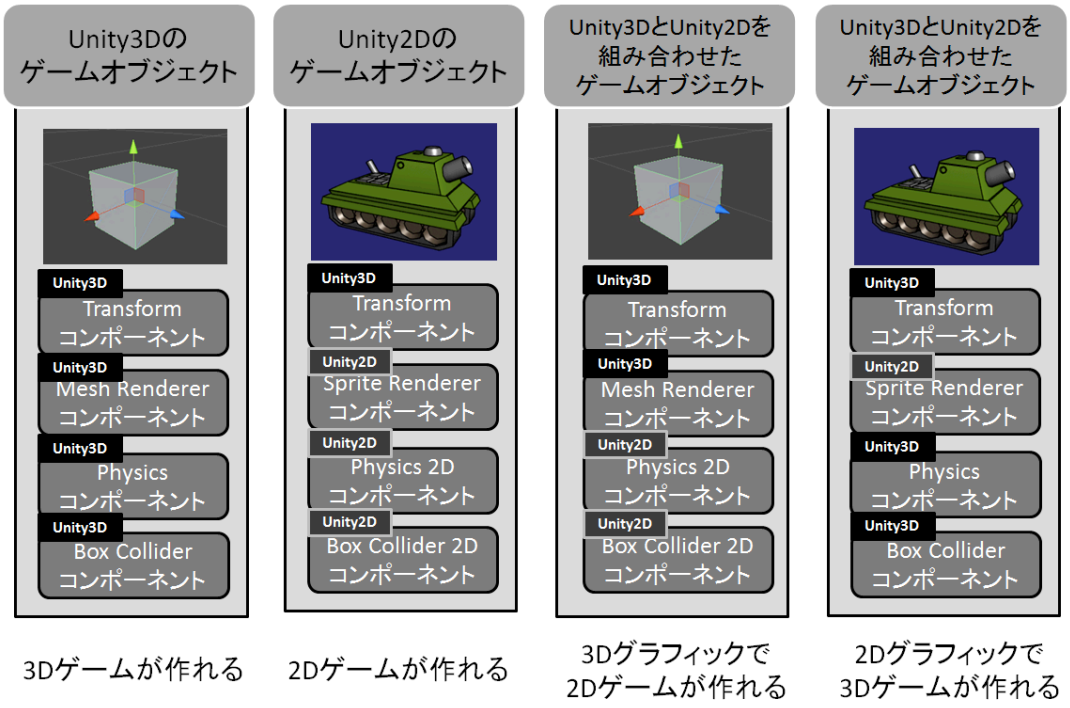


図 B02_02_001 Unity2D と Unity3D の組み合わせ

そのため、Unity2D で表示しているスプライトを Unity3D の 3D 物理シミュレーションで動作させることもできれば、Unity3D で球体などの 3 次元のゲームオブジェクトを表示しながら、2D 物理シミュレーションで動作させることも可能です。

つまり、Unity3D と Unity2D は、その組み合わせで様々なことが行えるのです。

簡単に Unity3D と Unity2D の機能の対応ついて表 2.2.1.1 にまとめてみました。

表 2.2.1.1 Camera コンポーネントのパラメータ

Unity3D	Unity2D	備考
Quad	Sprite	スプライトは Unity2D の機能。ただし、Unity3D でも Quad ゲームオブジェクトで 1 枚のマテリアル画像をスプライトのように表示することは可能
Texture インポータ	Sprite インポータ	1 枚のスプライト画像から複数のスプライトを指定するといった Sprite インポータは Unity2D のみの機能
Physics	Physics2D	3 次元物理シミュレーションは Physics コンポーネントで、2 次元物理シミュレーションは Physics2D コンポーネントで実現可能。1 つのシーンで同時に使用することもできるが、1 つのゲームオブジェクトで共存はできない。

Collider	Collider2D	アタリ判定用のコライダー。3 次元のアタリ判定は Collider コンポーネントで、2 次元のアタリ判定は Collider2D コンポーネントで判定可能。1 つのシーンで同時に使用することもできるが、1 つのゲームオブジェクトで共存はできない。
Mecanim	なし	Unity2D にはアニメーション機能はない。 Unity3D の Mecanim が Unity2D に対応しているので、これを利用する。
Camera	なし	Unity2D 専用のカメラはない。 Unity3D のカメラコンポーネントが、3D も 2D も対応しているので、これを利用する。
Light	なし	Unity2D 専用のライトはない。 Sprite は基本的にライトの影響を受けない。ただし、シェーダを切り替えることで、ライトの影響を反映することが可能
Sound	なし	Unity2D 用のサウンド機能はない。 ただし、Unity3D のサウンド機能が、3D も 2D も対応しているので、これを利用する。

特に覚えておいてほしいのが、Unity3D の Physics3D と Unity2D の Physics2D の関係です。

3 次元物理シミュレーションは Physics コンポーネントで、2 次元物理シミュレーションは Physics2D コンポーネントで実現可能です。また、1 つのシーンで同時に Physics と Physics2D は共存して使用することができます。ただし、Physics の 3 次元物理シミュレーション(Rigidbody コンポーネント)と Physics2D の 2 次元物理シミュレーション(Ridibody2D コンポーネント)は、お互いに影響することはありません（図 B02_02_002）。

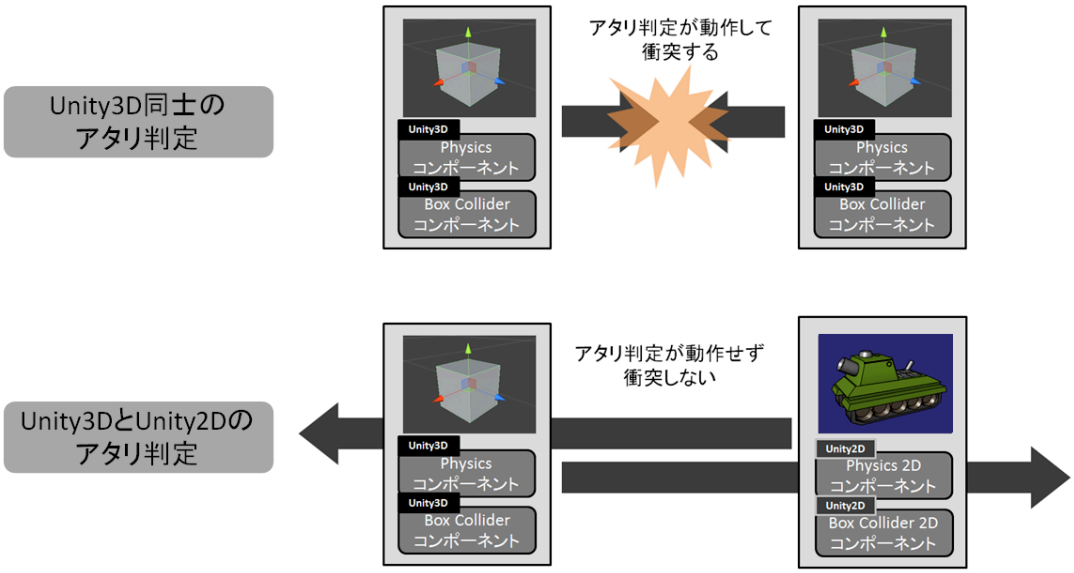


図 B02_02_002 Unity3D と Unity2D のアタリ判定の組み合わせと、その結果

また、1 つのゲームオブジェクトで Physics3D と Physics2D 共存はできません。インスペクタでコンポーネントを追加しようとすると、エラーがでます。Collider コンポーネントも同様です（図 B02_02_003）。

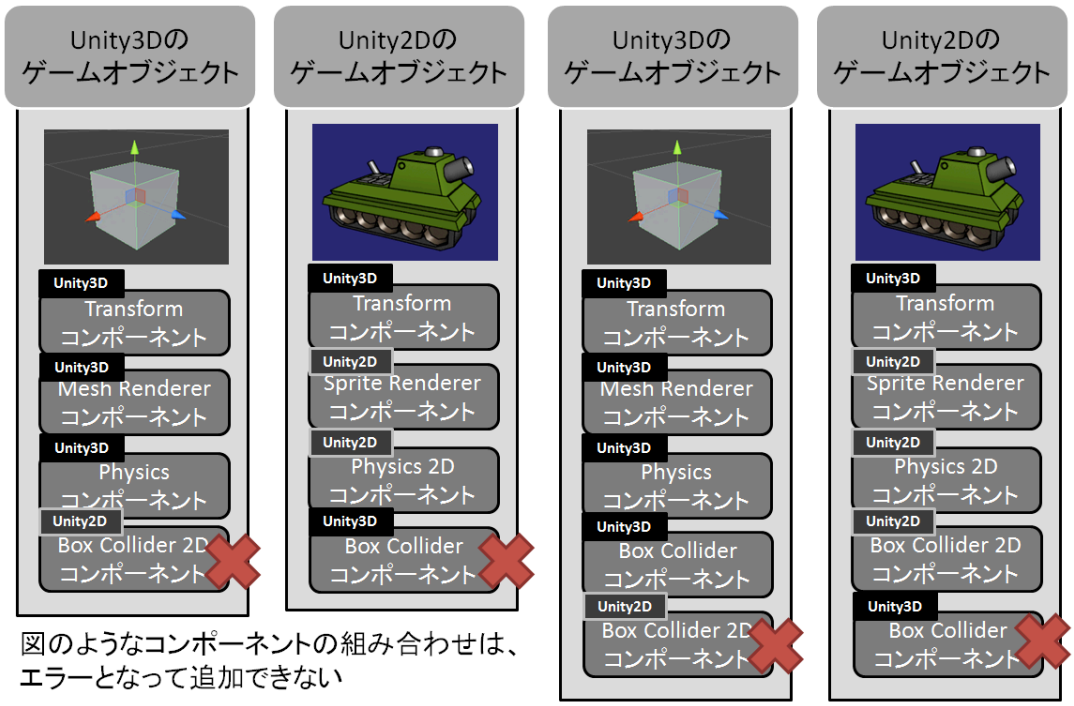


図 B02_02_003 共存できない Unity3D と Unity2D の組み合わせ

このような仕組であるため、Physics3D の 3 次元物理シミュレーションと Physics2D の 2 次元物理シミュレーションにおいては、ゲームを作る前にどちらを利用するか、最初に決めて置く必要があります。

Physics3D と Physics2D の特性の違い

Unity に実装されている 2 つの物理エンジンの Physics3D と Physics2D は、Unity 社が独自に開発したエンジンではありません。Physics2D は「Box2D」というオープンソースの 2D 物理エンジンを採用しています。一方、Physics3D はビデオカードメーカーで有名な NVIDIA 社の 3D 物理エンジン「PhysX」を採用しています。

つまり、同じ「物理エンジン」というプログラムではあるものの、そのバックボーンが大きく違うわけです。そのため、物理エンジンを使用した結果においても、特性の違いが如実に現れます。

例えば、図 B02_02_003 のように物体を急激に拡大した場合、Physics2D と Physics3D では、その結果がことなります。Physics3D(PhysX)では、ゲームオブジェクトを瞬間的に拡大させるとそのちからによって、大きく弾けます。Physics2D(Box2D)では、ゲームオブジェクトを急激に拡大するとヌルっと他のゲームオブジェクトが移動します。はじけた力で他のゲームオブジェクトが飛んでくれないのです。

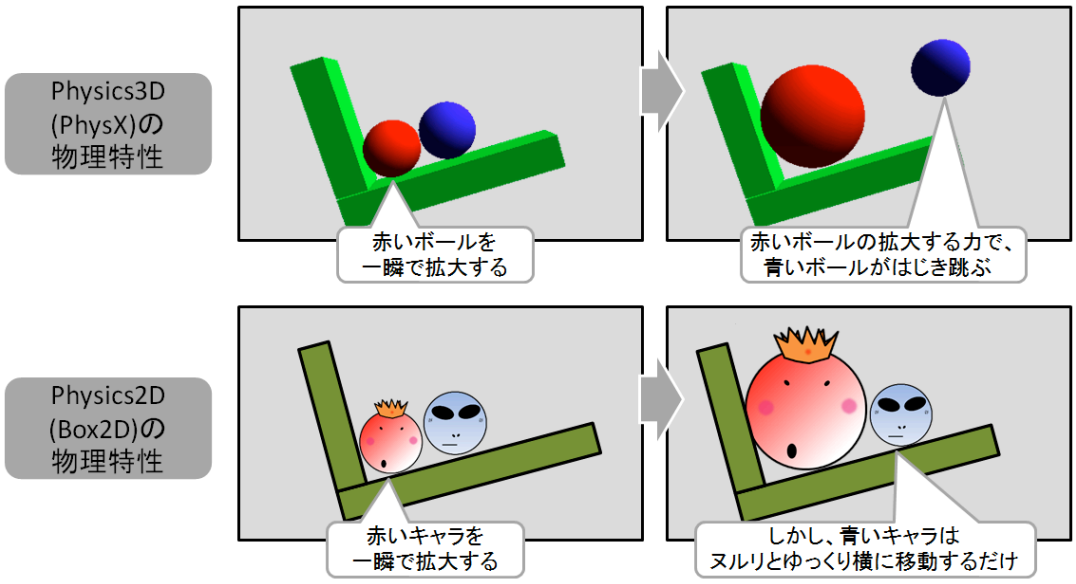


図 B02_03_001 Physics3D(PhysX)と Physics2D(Box2D)の物理特性の違いの一例

Physics2D でも、拡大するときにスクリプトで周囲に力を加えて弾き飛ばすことができますが、簡単によりリアルに実現するなら Physics3D の方が最適でしょう。

この他にも、物体が落下したり、移動して他の物体と衝突した場合の「物体の接触状態」

でも、Physics2D と Physics3D で特性が変わります。

Physics3D では、物体が他の物体と接触した場合は、コライダー同士がちょっとだけめり込みます。しかし、Physics2D では、接触後（めり込んだ後）にコライダー同士に隙間ができます（図 B02_02_003）。

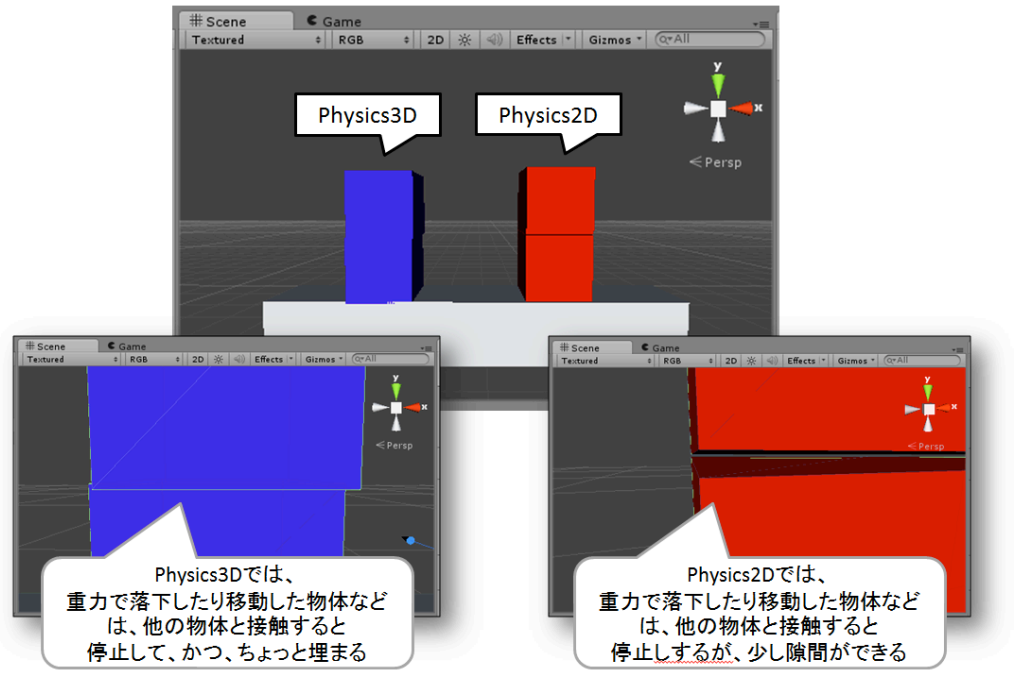


図 B02_03_001 Physics3D(PhysX)と Physics2D(Box2D)の物理特性の違いの一例

そのため、Box2D でコライダーの形状で接触点を自前で計算して取得しようとしても、隙間があって離れているため接触点は求まりません。

どんな状況でも必ずこのような結果になるわけではありませんが、基本的にはこのような動作になるので、Physics2D と Physics3D で特性としてぜひ覚えておいてください。

Physics3D と Physics2D の機能の違い

Physics2D は Physics3D の Physics のクラス構造をベースに作られているため、Physics2D か Physics のどちらか一方が分かっていれば、どちらに移行してもすぐに扱うことができます。

ただし、実装されている機能には大きな違いがあります。例えば、Physics には爆発をシミュレートする Physics.Explotion というメソッドがありますが、Physics2D にはありません。Physics2D の元となった Box2D は 2D ゲーム用に最小限の物理シミュレートを高速に実現するためのエンジンです。一方、Physics3D の PhysX は、リッチなプラットフォームで実行されることを前提とした強力な物理エンジンです。用意されているパラメータも多

く、様々な調整が可能ですが、逆に Box2D に比べると処理が重たいという問題があります。そのため、Physics3D と Physics2D の「どちらが優れているか？」という視点で物理エンジンを選択するのではなく、「作ろうとしているゲームに最適な物理エンジンはどちらか？」という視点から考えなければなりません。

そこで、物理エンジンの機能の違いを表 2.4.1.1 のリストにしましたので、参考にしてください。

表 2.4.1.1 物理エンジンの機能の違い

機能	Physics3D	Physics2D
速度	普通	高速
Gravity Scale によるオブジェクトごとの重力変更	×	○
物体をスケーリングしたときの反発	○	×
ジョイントコライダーのジョイントに一定の力が加わった場合の自動破壊	○	×
移動軸のコンストレイン（固定）	○	×
回転軸のコンストレイン（固定）	○	△
Slider Joint（スライドするジョイント）	△（作れる）	○
オリジナルジョイントの作成	○	×
物理特性マテリアルにおける一定方向からの摩擦係数設定	○	×

Unity2D で扱う Camera コンポーネントの設定

Unity2D でゲームを作るために、Project Wizard の「Setup defaults for:」を「2D」で設定して作成したプロジェクトでは、シーンの”Main Camera”が自動的に 2D ゲームに最適な状態で設定されます。そのため、Unity で 2D ゲームを作る場合は、特にカメラコンポーネントの細かい設定を知らなくても問題ありません。しかし、その内容を知って置けば、何か困った時に役立つこともあります。そこで、Camera コンポーネントの機能を設定するプロパティについて説明しましょう（図 B02_05_001）。

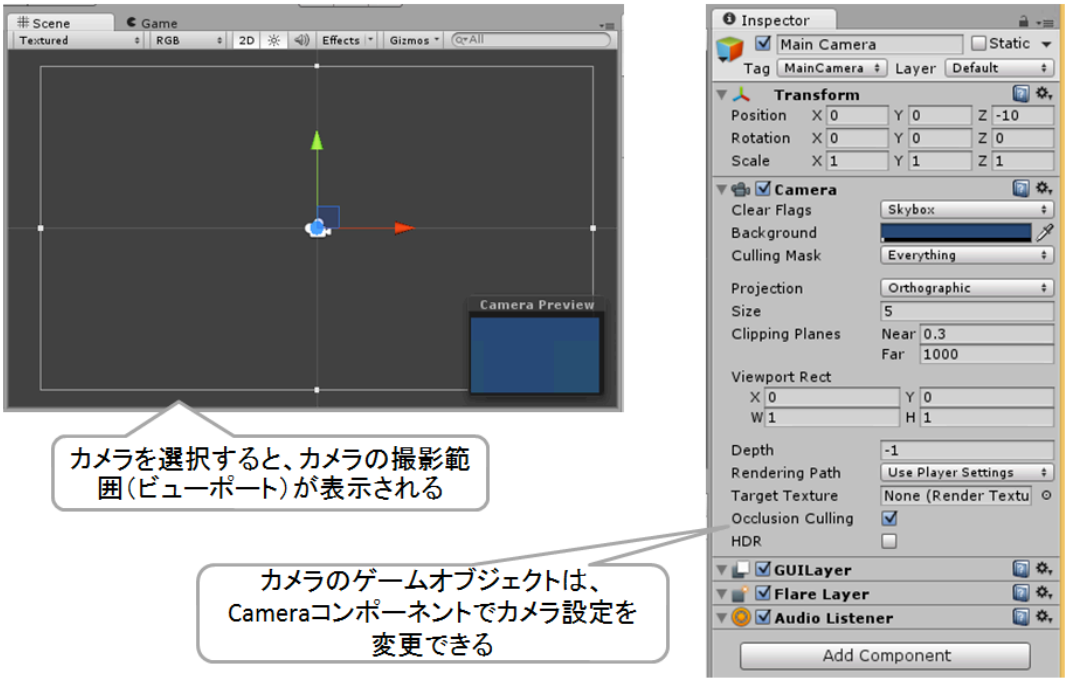


図 B02_05_001 カメラコンポーネントとプロパティ

Camera コンポーネントのプロパティは表 2.2.5.1 の通りです。

表 2.2.5.1 Camera コンポーネントのプロパティ

プロパティ名	説明
Clear Flags	描画ごとに画面をどのようにクリアするか指示するフラグです。複数カメラがある場合に効率的に画面をクリアように設定もできます。デフォルトは”Skybox”になっており、Skybox コンポーネントがカメラにない場合は、バックグラウンドカラーでクリアされます。
Background	画面をクリアするときのカラーです。クリックするとカラーとアルファ値（透明度）を設定できます。
Culling Mask	カメラがどのレイヤーをレンダリングするか設定します。デフォルトは”Everything”（すべて）になっています。
Projection	プロジェクションは、どのように 3D オブジェクトを画面に投影するが設定します。デフォルトの”Perspective”（パースペクティブ）は、3D オブジェクトを立体感のある透視投影で表示します。一方、” Orthographic”（オルソグラフィック）は、距離感を感じさせない正投影（平行投影）で表示します。
Field of view	カメラの視野角です。0～180 度の範囲で指定します。また、視野が狭くなればズームイン、視野が広ければズームアウトのように感じられるでしょう。ただし、視野角が広くなるほど表示されるものが多く

	なるため、処理が遅くなります。
Clipping Planes	カメラがオブジェクトを撮影（レンダリング）できる距離です。Near がカメラ手前方向の限界距離で、Far が奥行方向の限界距離です。Near が近く Far が遠い距離になるほど表示されるもが多くなり、処理が遅くなります。
Viewport Rect	投影面の原点座標と大きさです。デフォルトは X:0 Y:0 W:1 H:1 と、中央の原点が(0,0)で大きさは縦横ともに 1 になっています。なお、実際に表示される画面の大きさは、本書で後程説明する Build Setteing で設定します。
Depth	複数のカメラを設定した場合に、カメラの描画順を決めるための位置の深さです。
Rendering Path	このカメラが使用するレンダリングパス（レンダリング方法）を指定します。デフォルトは、”Use Player Settings”になっています。
Target Texture	カメラで撮影した画像を、指定したターゲットテクスチャにのみレンダリングする機能です。Unity Pro ライセンスのみ使えます。
Occlusion Culling	カメラから見て隠れているオブジェクトをレンダリングしないことで、高速化する機能です。Unity Pro ライセンスのみ使えます。
HDR	HDR(High Dynamic Range)機能を使って、ブルームやグローといった光のエフェクトを実現します。

分かりやすく 3D カメラの設定と比較してみましょう。

表 2.2.5.2 Camera コンポーネントの 2D カメラと 3D カメラの比較

パラメータ名	2D カメラ	3D カメラ
Clear Flags	Skybox	Skybox
Background	青(RGBA=49,77,121,5)	青(RGBA=49,77,121,5)
Culling Mask	Everything	Everything
Projection	**Orthographic**	**Perspective**
Size	**5**	**なし**
Field of view	**なし**	**180** **(ただし 2D プロジェクトの場合は 60)**
Viewport Rect	X:0 Y:0 W:1 H:1	X:0 Y:0 W:1 H:1
Clipping Planes	Near:0.3 Far:1000	Near:0.3 Far:1000
Depth	-1	-1
Rendering Path	Use Player Settings	Use Player Settings
Target Texture	None	None
Occlusion Culling	オン	オン
HDR	オフ	オフ

この通り、2D カメラと 3D カメラの違いは、Projection のタイプとそれに付随する Size, Field of View パラメータだけです。2D カメラでも Projection のタイプを「**Perspective**」に設定すれば、3D カメラになります。なお、2D カメラの描画領域の大きさは「Size」プロパティで設定できます。Size プロパティの値が小さいほど、表示できる 2D 空間の大きさは小さくなり、画面には大きく表示されます。逆に、Size プロパティの値が大きくなれば表示できる 2D 空間の大きくなるので、スプライトは小さく表示されます。また、プロジェクトを Unity2D で作成しても、座標系は中央を原点(0,0)とした左手座標系のままです。座標の単位も、画面を正方形にした場合、カメラの端がビューポートの w,h に size を掛けた大きさ（デフォルトなら 1×5=5.0）になります（図 B02_05_002）。

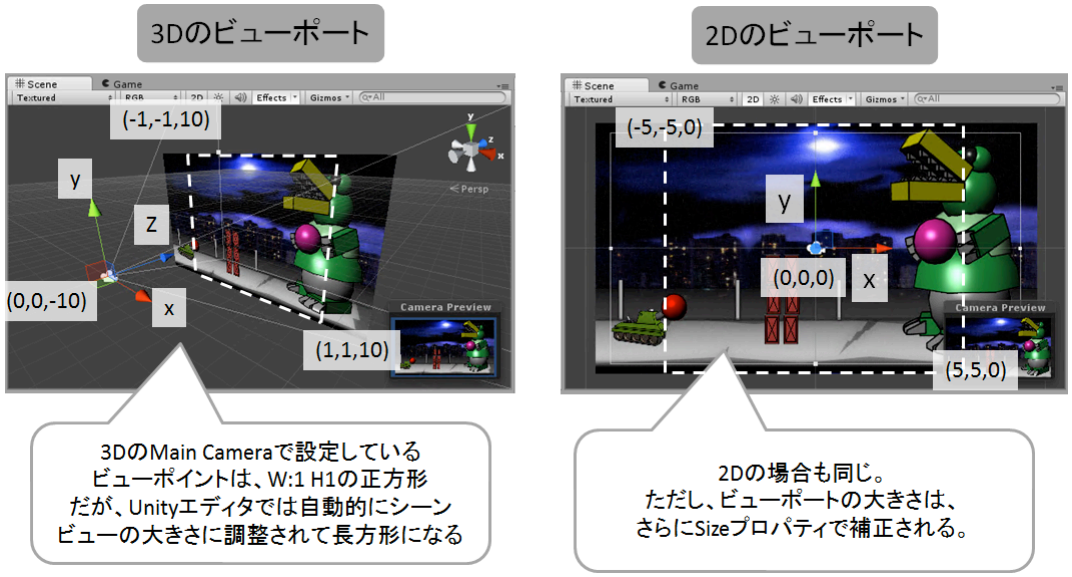


図 B02_05_002 ビューポートのサイズ

Vireport Rect の X,Y,W,H を実寸の画面サイズ（640x480 など）で指定すれば、2D 画面の解像度で座標を指定できるようになりますが、Unity では左手座標を変更できない他、物理エンジンが座標の大きな値で調整されていないため、思わぬバグの原因になる可能性があります。

また、ライトに関しては、Unity2D の Sprite ゲームオブジェクトを使う場合は、ライトの影響を受けませんので、作成して設置する必要はありません。

特にこだわりがなければ、カメラとライトの設定は、プロジェクト作成時のままで問題ないでしょう。

2.3. Unity2D をもっと使いこなす

次は、これまでに紹介しきれなかった Unity2D の機能について解説しましょう。

Multiple モードを使う

本書では、データ構造が分かりやすいように 1 画像に付き 1 スプライトでデータを扱っています。

しかし、この方法では、ドローコールと呼ばれる処理が増えてゲームが遅くなる可能性があります。また、デザイナーによっては、1 つの画像に複数のスプライト画像を持たせた方が編集しやすいという方もいるでしょう。

このような場合は、スプライト画像を Assets フォルダにインポートした後に、スプライトインポータの Sprite Mode から「Multiple」を選択することで、1 つのスプライト画像から複数のスプライトを切出すことができます。スプライトの切出しをするには、「Open Sprite Editor」ボタンを押して「スプライトエディタ（Spirte Editor）」を開きます（図 B03_01_001）。

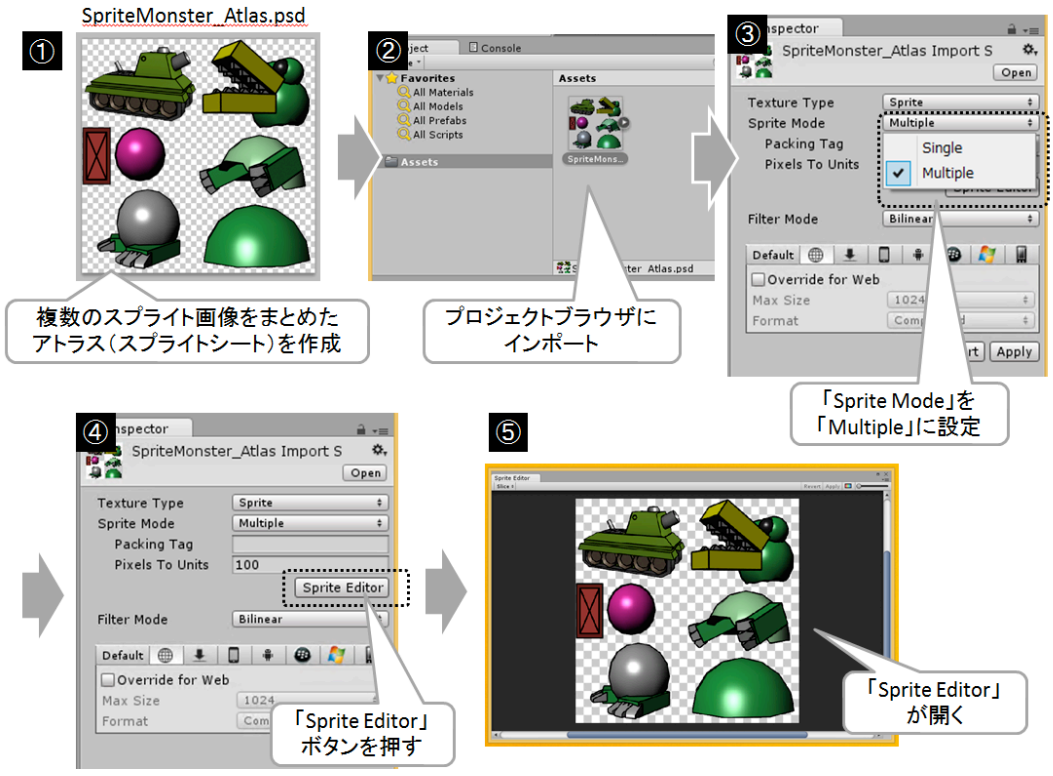


図 B03_01_001 Sprite Editor を開く

スプライトエディタは、1 枚のスプライト画像から複数のスプライトを切出すためのツールビューです（図 B03_01_002）。

図 B03_01_002 Sprite Editor の画面

スプライトを切出すには、マウスで切出したい画像の範囲をドラッグして指定するだけです。すると Sprite ダイアログが開いて、スプライトの名前やサイズなどのスプライト情報が入力できるようになります。初期設定のままでよければ、そのまま画像の範囲の設定を繰り返します。最後に、「Apply」ボタンを押せば、プロジェクトブラウザの選択したスプライト画像の子オブジェクトとして、スプライトが登録されます（図 B03_01_002）。

図 B03_01_003 スプライトの切出し

切出す画像や設定を間違った場合、「Revert」ボタンを押すと、編集前のアセット情報に設定が戻ります。

なお、途中で表示される Sprite ダイアログの各プロパティは、表 2.3.1.1 の通りです。

表 2.3.1.1 Sprite ダイアログのプロパティ

プロパティ名	説明
Name	スプライトの名前。ここで変更しなくても、プロジェクトブラウザからも変更可能。
Position	スプライトの切出し位置(X,Y)とサイズ(W,H)。 下にある「Trim」ボタンを押すと、画像の透明領域を最適化することが可能。
Pivot	スプライトを表示するときの基準点を設定。デフォルトは Center。 8 方向指定できる他に、Custom を選べば自由な位置に基準点を設定することができる。
Custom Pivot	Pivot で「Custom」を選択した場合に指定できる基準点の指定方法。 他の Pivot 設定とは違い、X,Y 座標で指定できる。

これらの設定は作成した後も、スプライトの領域を選択することで、再度編集することができます。

さて、切り出した複数のスプライトはプロジェクトブラウザで階層構造になっており、プロジェクトブラウザで確認することができます（図 B03_01_002）。

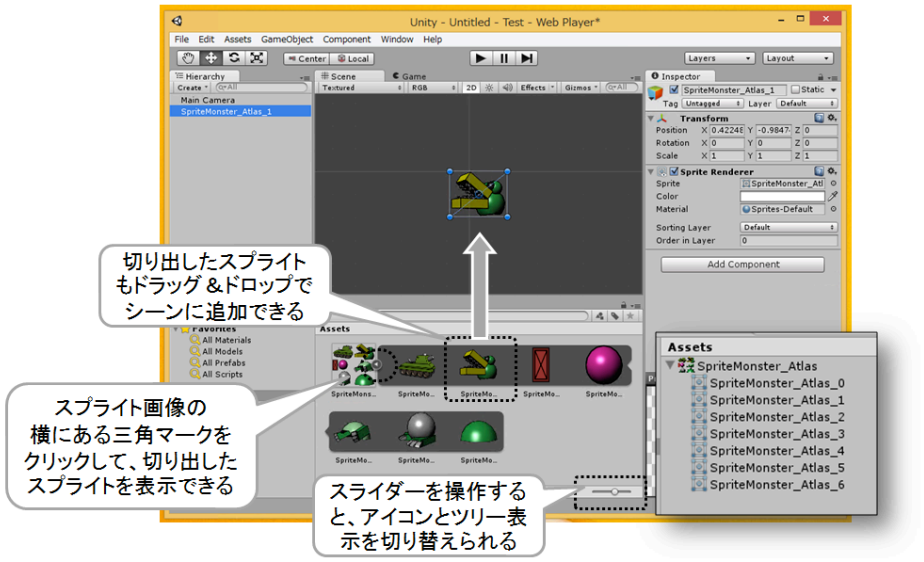


図 B03_01_004 スプライト画像の階層構造

ゲームで高速な処理が必用なのであれば、スプライトを Multiple モードで作成するのは必須になります。ぜひ、覚えておいてください。

Multiple モードでスプライトを作成する場合、もう一つ便利な機能があります。

それが「Slice」ボタンです。このボタンを押して Automatic モードで切出しを行うと、画像を解析して、自動でスプライトをスライス（切出し）してくれます（図 B03_02_001）。

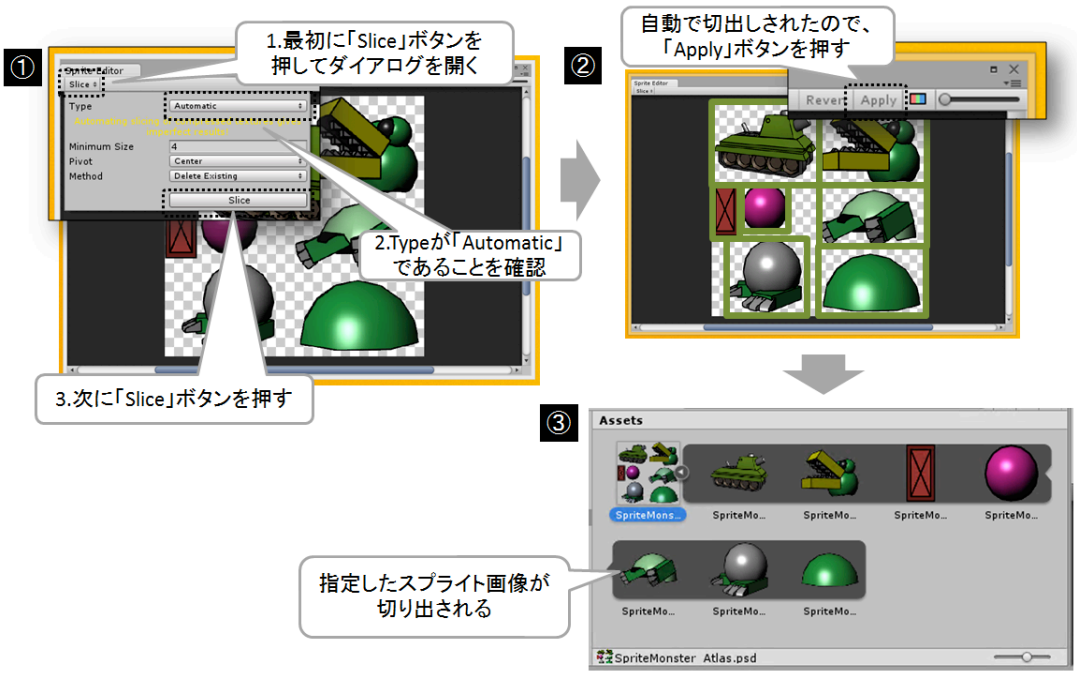


図 B03_02_001 Slice 機能の使い方

ダイアログに表示されるプロパティの内容は表 2.3.2.1 の通りです。

表 2.3.2.1 Slice ダイアログのプロパティ(Automatic の場合)

プロパティ名	説明
Type	切出し方法。「Automatic（自動）」と「Grid（サイズ指定）」の 2 つの方法を選べる。
Minimum Size	切出しするスプライトの最小サイズ。うまく使えば、「泡」などの小さく分断している複数のパーツで構成された画像でも 1 つのパーツとして自動で切出せる。
Pivot	スプライトを表示するときの基準点を設定。デフォルトは Center。8 方向指定できる他に、Custom を選べば自由な位置に基準点を設定することができる。自動で切出したスプライトはすべてこの Pivot 属性になる。
Method	自動で切出すときに、すでに設定されているスプライトについての処理方法。「Delete existing」はすでに設定されているスプライト

もリセットして置き変える。「Smart」の場合は、すでに設定したスプライトはそのまま保持するか、または画像の矩形指定を調整しつつ、未選択の画像を追加で設定する。画像を更新した場合に便利。「Safe」は、すでに設定したスプライトはそのまま保持して変更を加えず、未選択の画像を追加で設定する。

なお、Type を「Grid」にすれば、同じサイズのキャラや 2D RPG などで使うマップパーツ（マップチップ）を自動で確実にスライスすることもできます（図 B03_02_002）。

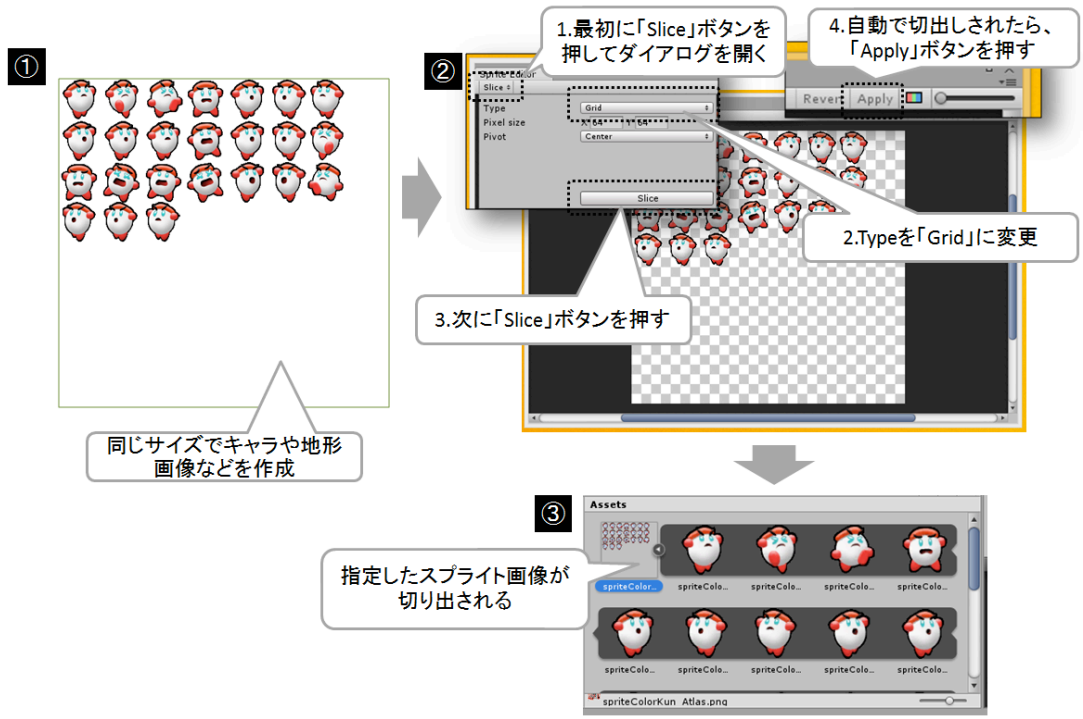


図 B03_02_002 Grid 機能の使い方

なお、Grid では、切出した画像が透明なスプライト（何も表示すべきものがない画像）については自動で除外されます。

特に RPG のようなマップパーツを使うゲームでは、この「Grid」が重宝するでしょう。

最適なアトラスのサイズは？

さて、これまで画像ファイルから、スプライトを切り抜く方法について説明してきました。このように複数のスプライトが 1 つの画像ファイルに保存されているファイルのことを「アトラス（またはスプライトシート）」と呼びます。このアトラス画像は、ターゲットとなるプラットフォームのメモリ容量（または最大のアプリサイズ）まで複数枚持たせることができます。

さて、このアトラスですが、プラットフォームごとにメモリに読み込める 1 枚の適切な

アトラスサイズ(最大テクスチャーサイズ:Max Texture Size)が決まっています。例えば、現在の iPhone や Android の最新機種では、2048x2048 まで問題ありません。少し古い機種だと 1024x1024 になります。これらの画像データの大きさは、GPU のチップ仕様で決まっており、このサイズをオーバーすると画像が VRAM に転送できません（図 B03_02_002）。

図 B03_03_001 アトラスサイズの問題

そのため、ターゲットのプラットフォームを決めたら、アトラスの最大サイズを最初に決めて、テクスチャーインポータで設定しておくのが無難です（図 B03_02_002）。

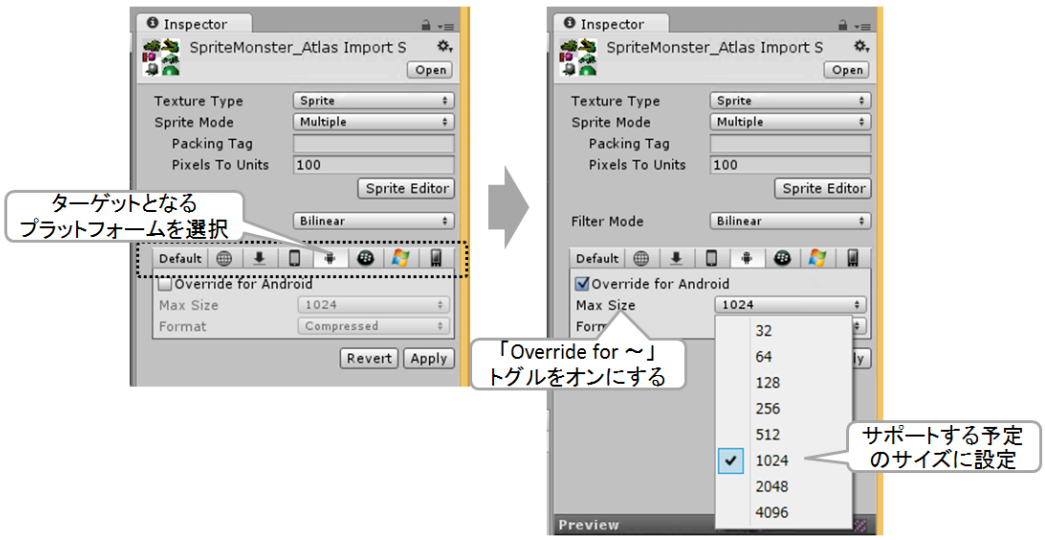


図 B03_03_002 テクスチャーインポータでのサイズ設定

参考までに、iPhone や Android の主要な機種のアトラスサイズを表 2.3.1.1 に抜き出してみたので、ゲーム作成前にチェックしてみてください。

表 2.3.1.1 iPhone と Android のアトラスサイズの目安

iPhone	Android
iPhone～iPhone3，iPodTouch 第 3 世代まで 1024x1024	**Android2.0～3.0** 1024x1024
iPhone3GS,iPhone4,iPad,iPad 2,iPodTouch 第 4 世代 2048x2048	**Android3.0～4.0** 2048x2048
iPhone5,iPhone4S,iPad 2(iOS5.1),iPad 3，iPodTouch 第 5 世代 4096x4096	**Android4.0～** 4096x4096

上記の表では Android の OS バージョンごとにアトラスサイズを記載していますが、あくまで筆者が調べて作成した目安のサイズです。実際には Android は OS に関係なく様々な GPU が搭載されているため、Android2.3 で 2048x2048 が問題ない機種もあれば、Android3.1 で 1024x1024 しか受け付けないできない機種もあります。これは最新の Android4.0 以降でも同じで、Android4.0 だからといって 4096x4096 のアトラスが使えるとは限りません。低価格の Android4.0 機種では、2048x2048 であることも十分ありえるのです。

機種ごとの GPU のスペックが知りたい方は、下記のサイトで検索することができます。

GFXBench
http://www.glbenchmark.com/

なお、スプライト画像、およびアトラスのサイズは、縦横ともに 2 のべき乗（2,4,8,16,32,64,128,256,512,1024,2048～）がお勧めです。ゲームを表示するためのグラフィックチップである GPU の多くは、スプライト画像サイズが 2 のべき乗で高速に表示されるようになっています。また、画像データを圧縮してをメモリに保存する場合は、画像サイズが 2 のべき乗でないと、プラットフォームによっては画像データが圧縮されません。スプライト画像を表示したときに、画像が崩れる場合は、切り出したスプライトサイズやアトラス画像のサイズが 2 のべき乗になっているか確認してみてください（ちなみに、スプライト画像をインスペクタで表示して「NPOT」と表示される場合は、Not Power Of Two、つまり 2 のべき乗になっていません）。どうしてもアトラス画像のサイズを 2 のべき乗にできないときは、最低でも 16 の倍数にしておくのが安全です（16 の倍数でないない場合、GPU によっては画像にゴミやノイズが出る場合があります）。

このアトラスサイズの問題は、スマートフォンなどでは特にゲームが動作しない主な原因になりがちなので、ぜひ覚えておいてください。

iPhone,Android の減色問題

スプライト画像をインポートする場合、もう 1 つ大きな問題があります。

それは、画像の色数フォーマットです。

例えば、次のようなグラデーション画像を Assets フォルダにインポートすると、ターゲットにしているプラットフォームが Windows の場合は何も問題ありませんが、iPhone や Android に切り替えると、グラデーションの諧調がおかしくなります。このような画像の諧調がおかしくなる現象を「マッハバンド」と呼びます（図 B03_04_001）。

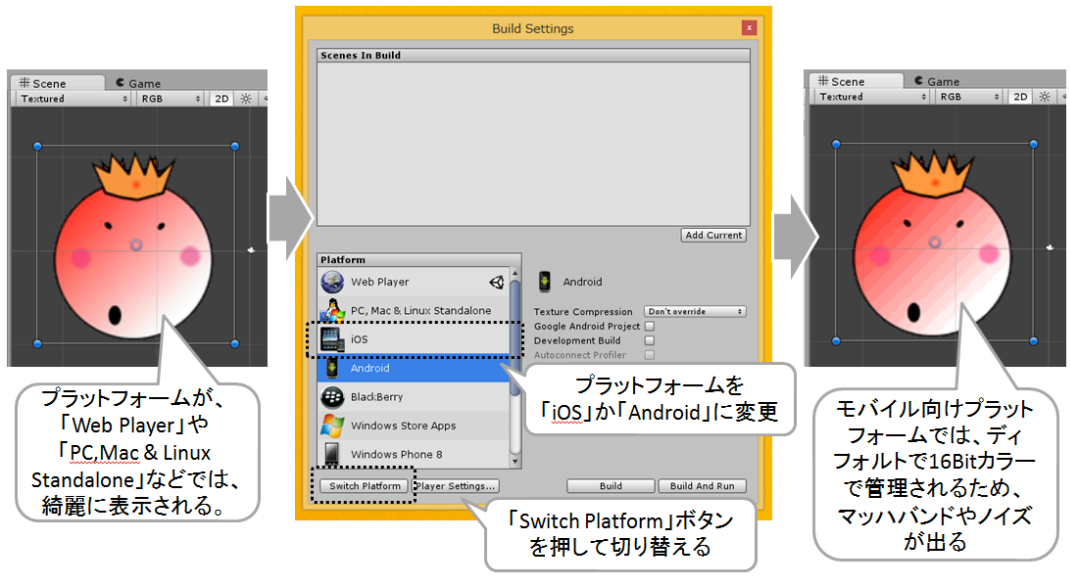


図 B03_04_001 プラットフォームにおけるマッハバンドの問題

Unity でマッハバンドが発生する理由は、各プラットフォームのインポータで指定している画像の色数フォーマットが原因です。

プラットフォームが Windows の場合、Unity はインポータでテクスチャをフルカラーとして扱います。しかし、iPhone や Android では、16 ビットカラーとして設定されており、色数の現象に合わせて画像も自動で減色されるため、マッハバンドが出てしまうのです。とくに、キャラのイラストでは致命的な問題となります。

これを解決するには、3 つの方法があります。

1 つは、インポータの「Format」プロパティを Truecolor にすることです。減色されないので、マッハバンドがでなくなります。しかし、当然ながらビルドされる画像データは大きくなるため、メモリを圧迫します（図 B03_04_002）。

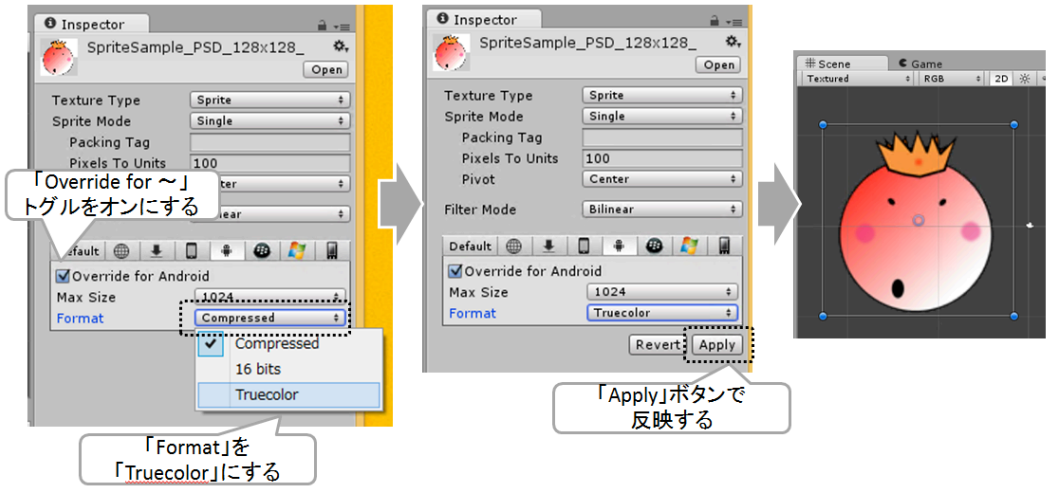


図 B03_04_002 インポータから各プラットフォームにあわせて設定を変える

圧縮可能な画像データであれば、Format を「Compressed」に設定することで解決できます。ただし、iPhone の場合であれば、アルファチャンネルのある画像だと、透明な領域と色のある領域の境目にノイズがでたりボケたりすることがあります。Android の場合では、アルファチャンネルのある画像は圧縮できません（Format プロパティの表示は Compressed ですが、実際には圧縮されません）。

指定した画像フォーマットの状態は、インスペクタの Preview で確認できます（図 B03_04_003）。

図 B03_04_003 指定した Format のタイプを Preview で確認

なお、この Format プロパティで指摘できるフォーマットは表 2.3.4.1 の通りです。

表 2.3.4.1 Format のプロパティ

プロパティ名	説明
Compressed	画像を 1 ピクセルあたり 4bit ほどで表現する圧縮データ iPhone では「PVRTC」、Android では「ETC」と呼ばれる形式が採用される。また、PVRTC の場合は、2 の二乗でかつ正方形サイズ、ETC の場合は 2 の二乗でアルファチャンネルなしでなければ圧縮できない。なお、圧縮された画像はボケたりノイズが入ることがある。圧縮可能な画像では「Compression Quality」プロパティで「Fast」「Normal」「Best」から画像品質を選択できる。
16bits（デフォルト）	画像を 1 ピクセルあたり 16bit で表現するデータ RGBA16(RGBA4444)フォーマットと呼ばれる形式
Truecolor	画像を 1 ピクセルあたり 32bit で表現するデータ RGBA32(RGBA8888)フォーマットと呼ばれる形式

もう 1 つの方法は、Unity の 16bit カラー(RGBA16)で読めるファイルとして画像ファイルを保存して Unity にインポートする方法です。

ただ残念なことに、Photshop や GIMP でサポートしている Unity で扱える有力な画像フォーマットは表 2.3.4.2 に限られており、16bit カラー(RGBA16)で、かつ、透過色とアルファチャンネルに対応したフォーマットがありません。

表 2.3.4.2 Unity で扱える主な画像フォーマット

画像フォーマット	説明
Photoshop,GIMP: PSD	Photoshop や GIMP で保存できるマルチレイヤー形式の画像フォーマット 透過色・アルファチャンネルを指定可能
Photoshop,GIMP: PSD インデックスカラー	Photoshop や GIMP で保存できるマルチレイヤー形式の画像フォーマット 指定した色をカラーインデックスとして保存できる。 透過色・アルファチャンネルは指定できない
PNG-8	色数 256 色までのインデックスカラー形式の画像フォーマット。透過色は指定可能、アルファチャンネルは指定できない
PNG-24	RGB を 24Bit で表現できる画像フォーマット 透過色を指定できるが、アルファチャンネルは指定できない なお、Photoshop の「ファイル」メニューから選べる「web およびデバイス用に保存」で保存できる PNG-24 は、アルファチャンネルがある場合、PNG-24 ではなく PNG-32 で保存されるので注意が必要
PNG-32	RGBA を 32Bit で表現できる画像フォーマット

TGA	PNG-24 にアルファチャンネルの 8bit を追加している。 透過色・アルファチャンネルを指定可能 Photoshop や GIMP で保存できる画像フォーマット 16bit での保存が可能なため、あらかじめディザリングした画像などはそのままデータを Unity へ持って行ける。ただし、TGA の 16bit は透過色・アルファチャンネルに対応していないため指定できない

そこで、クオリティの高い自動減色アセットを使って、プラットフォームのビルド時にマッハバンドが出ないように圧縮することが可能です。この方法の場合は、自動での減色に失敗しない限り、高いクオリティで色数を減らせるので、マッハバンドが綺麗に消えてノイズもでません。有料ツールであれば、高機能な総合スプライトツール「OPTPiX imesta 7」や、安価な「Texture Packer」などがあります。

最後の方法は、最初から 16bit カラー(RGBA16)としてイラストレータに絵を描いてもらいインポートする方法です。グラデーションについては、「ディザリング」と呼ばれる近い色を複数使用して表現する方法で描いてもらい色数を 16bit カラーに収まるようにします。ただし、手動で 16bit カラーに対応したディザリングを行うのは大変な労力です。幸い Photoshop には、無料で使える「12bits Filter」というプラグインがネットからダウンロードできます。

Telegraphics: 12bits Filter
http://www.telegraphics.com.au/sw/product/12bits

「12bits Filter」は、Photoshop の 32bit 版のプラグインです。ファイルをダウンロードして、ZIP ファイルの中の dist フォルダを、Photoshop をインストールしたプラグインフォルダ（Windows7 なら"C:¥Program Files (x86)¥Adobe¥Adobe Photoshop CS5.1¥Plug-ins”など）にコピーするだけです。

後は、Photoshop を起動して、「フィルタ(T)」メニューの「Telegraphics」から「12bits(dither)」を選択して実行するだけです。後は、PSD か PNG-24 で保存すれば Unity でマッハバンドが出なくなります（図 B03_02_004）。

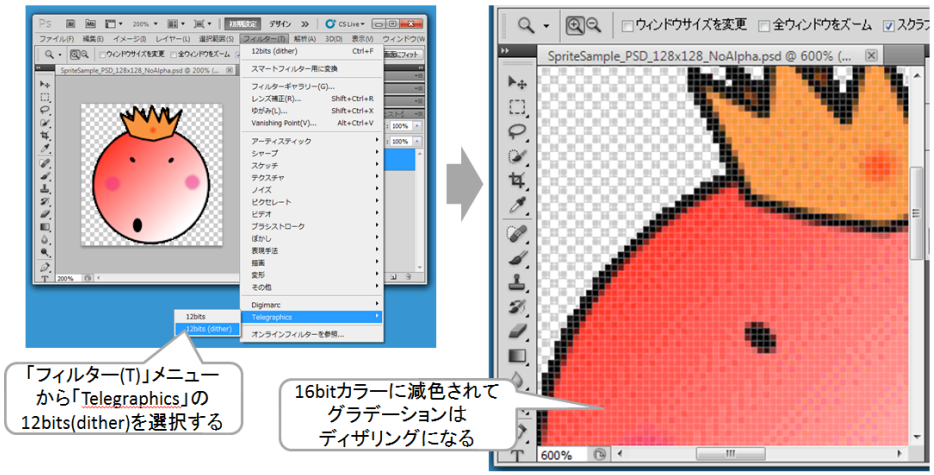


図 B03_04_004 12bits Filter を使ったフィルタリング処理

ただし、ディザリングで減色しているため、スプライトを拡大すると画像のディザリングが見えてしまいます。これればかりはどうしようもないので、頻繁に拡大が必用なスプライト画像などは、サイズが大きくなるのを覚悟でインポータの Format プロパティを True color にしましょう。

なお、ここで紹介した他にも Unity 用画像の減色に対応したアセットや様々ツールがありますので興味のある方は調べてみてください。

iPhone,Android のその他の問題

スプライト画像のインポートについては、その他にも様々な問題が発生します。

本書ですべて扱うにはページ数が足りないため、症状別にその問題と解決策を解説したサイとをご紹介します。

OPTPiX Labs Blog：
Unity で、もっとキレイな 16bit カラーテクスチャを使おう！
http://www.webtech.co.jp/blog/game-develop/2562/
（Unity における 16bit カラー問題を解説）
CEDEC 2014『工程の手戻りを最低限に 2D エンジン活用における傾向と対策』発表資料
http://www.webtech.co.jp/blog/game-develop/7179/
（Unity をはじめ様々な 2D エンジンやプラットフォームの画像の扱いを解説）

KAYAC DESIGNER'S BLOG：
Unity やるには必須！RGBA 画像減色の基礎をまじめに書いてみた
http://design.kayac.com/topics/2014/02/unity-RGBA4444.php
（Unity における 16bit カラー問題を解説、また Photoshop のプラグイン 12bits Filter:

Telegraphics を使っての減色方法を解説）

Unity Japan の高橋氏が開発したインポータなどが公開されているサイト：
Unity iOS/Android におけるテクスチャ画質の改善
http://keijiro.github.io/posts/mobile_texture_compression/
（アルファチャンネル付きだと、境界線にノイズが入る問題などを解説。また Unity だけでなく減色できるプラグインも公開されているが、Unity2D で使うには改造が必用。）

テラシュールウェア
[Unity3D]ios アプリサイズを節約する
http://terasur.blog.fc2.com/blog-entry-140.html
（Unity における減色方法などを解説）
[Unity3D]フルカラーなテクスチャを 16 ビットに原色
http://terasur.blog.fc2.com/blog-entry-204.html
（Unity における減色方法などを解説）
[Unity]アプリサイズを減らす 2
http://terasur.blog.fc2.com/blog-entry-384.html
（Unity における PNG 圧縮ツール TinyPNG を使った方法を解説）

mieki256's diary
#4 [unity][cg_tools] RGBA4444 に変換できるツール
http://blawat2015.no-ip.com/~mieki256/diary/201402154.html
（Unity における減色方法などを解説。GIMP 専用の 16-bit（was ARGB4444）Dither Script を紹介）

2.4. Unity 2D のちょっとしたテクニック

もう一つ、Unity2D のちょっとしたテクニックについてご紹介しましょう。

ピクセルパーフェクト

本書で紹介してきた 2D ゲームを作る方法では、ターゲットのプラットフォームにあわせて表示する画面サイズを調整する方法でした。しかし、この表示方法では、スプライト画像が拡大縮小されるため、ドット絵を綺麗に出したい場合は、思ったように表示されないこともあります（ただし、このような状況によるクオリティの低下は、ほとんど目立ちません）。ゲームによっては、スプライト画像の 1 ドットをターゲットプラットフォームのディスプレイの 1 ドットで表示したい場合もあります。このような表示方法を「ピクセルパーフェクト」と言います。

Unity2D でピクセルパーフェクトを実現するには、Camera コンポーネントの Projection プロパティを「Orthographic」に設定します。そして、下記のコードを実装します。

ソース 2.4.1.1： PixelPerfectCamera

```
[ExcutableInEditMode]
public class PixelPerfectCamera : MonoBehaviour {

  Public float pixelsToUnits = 100;

  void Update() {
    camera.orthographicsSize = Screen.height / pixelsToUnits / 2;
  }
}
```

このコードを適当なカメラゲームオブジェクトに追加すれば、ピクセルパーフェクトが実現できます。

なお、表示するスプライト画像は、必ずインポータの PixelsToUnits プロパティが 100 でなければピクセルパーフェクトにならないので注意してください。また、スプライト画像のシェーダーを Spirte/Default,Sprite/Diffuse など手動で設定した場合は、PixelSnap のチェックをオンにする必要があります。

この他、2D に関するカメラ設定などは、Unite2014 の講演ビデオなどを参考にしてみてください。

Unite 2014：Unity2D のよくある問題と解決方法
http://japan.unity3d.com/unite/unite2014/schedule

2.5. Unity PRO の機能

最後に、Unity PRO で使える Unity2D の機能についてご紹介しましょう。

自動でアトラス化

これまで手動でアトラスを作成する方法について説明してきました。

しかし、実際にゲームを開発していくと、何度も仕様を変更したくなるため、最後までアトラスが綺麗に決まらないということも少なくありません。

そこで、Unity PRO には、アトラスを自動で作成してくれる機能があります。

このアトラスの自動化ですが、仕組みは簡単です。1 つのアトラスにまとめたい複数のスプライトに、テクスチャーインポータからタグを付けるだけです。同じタグ名のスプライトは、ビルド時に 1 つのアトラス画像として結合されます（図 B04_01_001）。

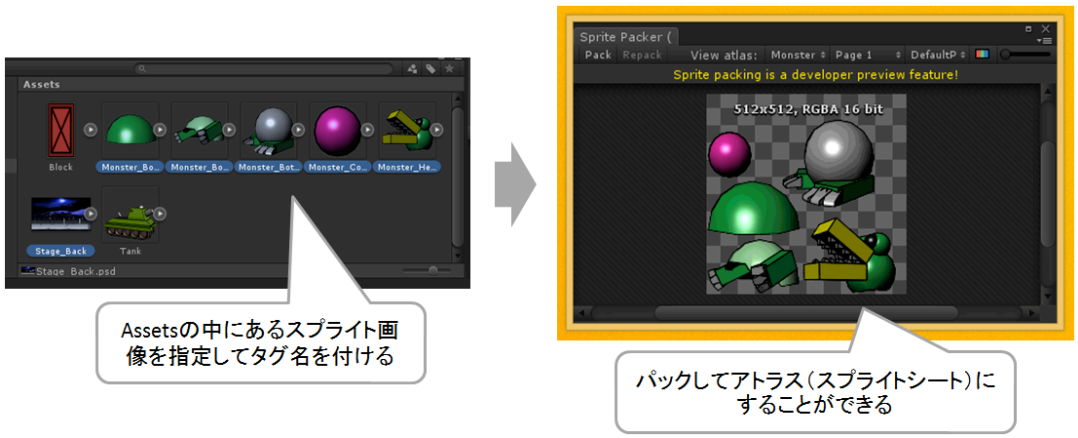


図 B04_01_001 スプライト画像からアトラスを作成する

では、実際にやってみましょう。

まず、Unity PRO の Sprite Packer 機能をオンにします。「Edit」メニューから「Project Settings」の「Editor」を選んでください。インスペクタに「Editor Settings」が表示されるので、Sprite Packer の項目の Mode プロパティを「Enabled For Builds」か「Always Enabled」に設定します（図 B04_01_002）。「Enabled For Builds」では、ビルド時にスプライト画像をパックします。そのため Unity エディタで再生ボタンを押してゲームを実行しているときは、パックしたアトラス画像は使用されません。何かバグがあった場合は、実機で確認することになります。一方、「Always Enabled」では、Unity エディタでゲームを実行するときも、パックしたアトラス画像が使用されます。パックに関して何か問題があれば、すぐに見つけることができるでしょう。

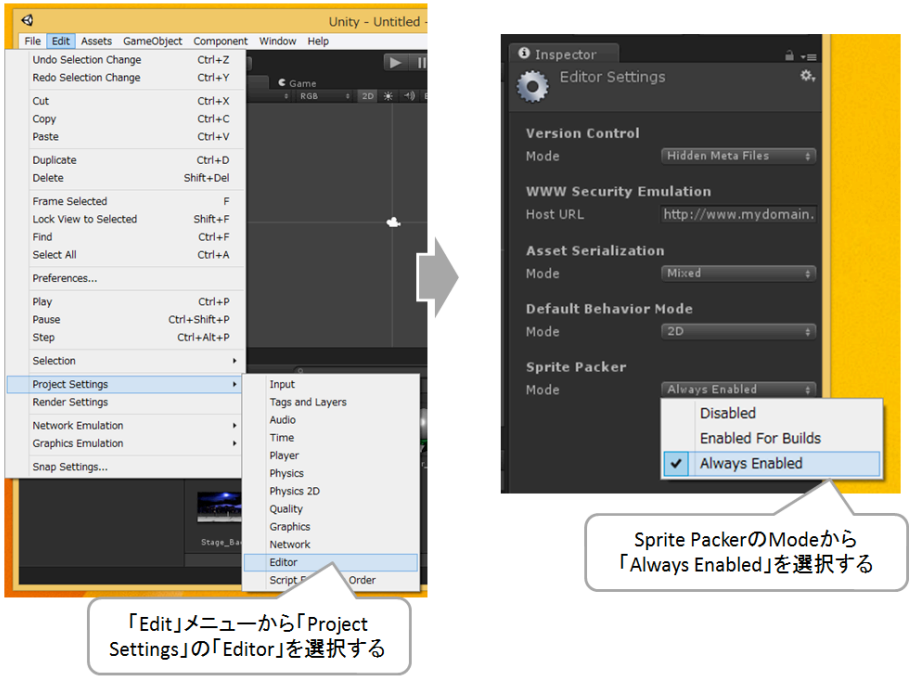


図 B04_01_002 Sprite Packer を有効にする

次は、スプライト画像のパック手順です。まず、プロジェクトブラウザから、パックしたいスプライト画像を複数選びます。SHIFT キーを押しながら選択すると、一括選択。CTRL キーを押しながら選択すると、個々のスプライト画像の選択をオンオフできます。選択できたら、インスペクタの「Packing Tag」にタグ名を入力して、Apply ボタンを押します（図 B04_01_003）

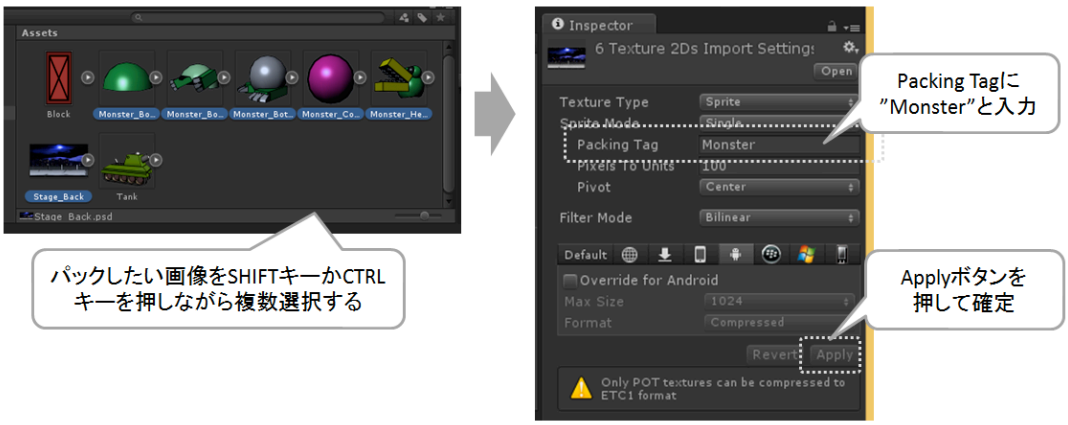


図 B04_01_003 スプライト画像にタグ名を付ける

タグ名を付けることができたら、次は「Window」メニューの「Sprite Packer」を選んで Sprite Packer ビューを開きます。そして、左上にある「Pack」ボタンを押してパックを実行します。成功すればビューにパックしたアトラス画像が作成されます（図 B04_01_004）。

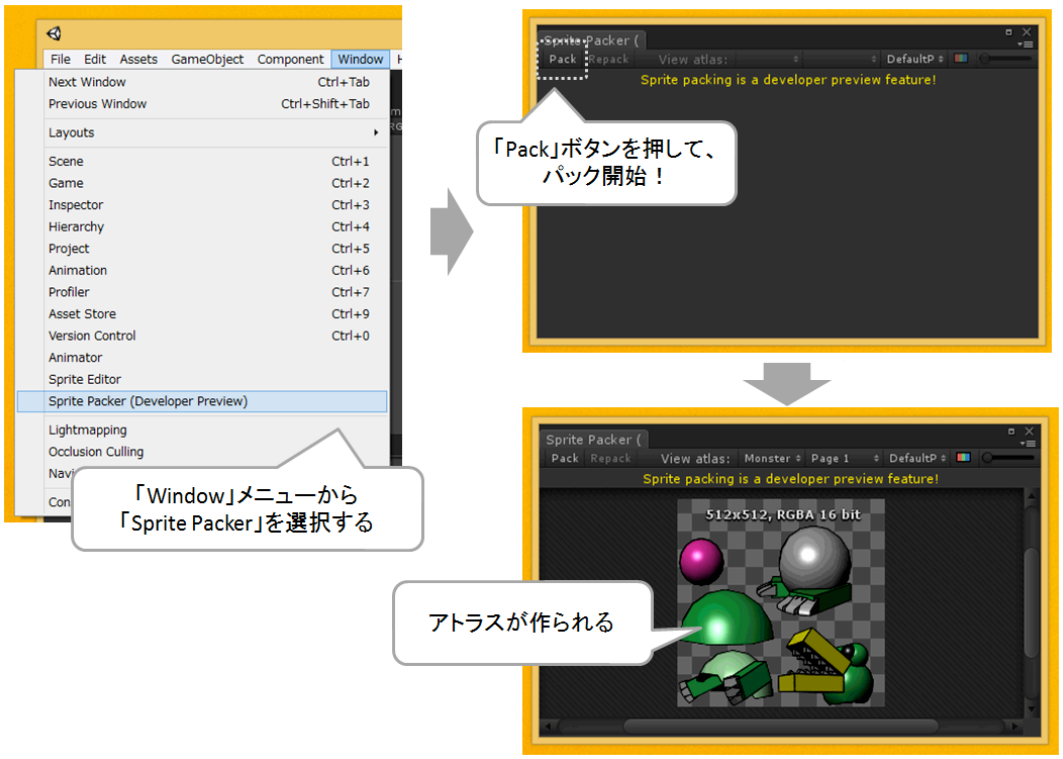


図 B04_01_004 スプライト画像をパックする

Sprite Packer ビューの機能は図 B04_01_005 の通りです。

図 B04_01_005 Sprite Packer の画面

まず、スプライト画像のパックは「Pack」ボタンで行います。

パックされたアトラス画像はタグ名から「パックグループ」に分かれます。”Monster”というタグ名を付けたら”Monster”というグループにアトラスが作られます。

なお、スプライト画像をパックする場合は、基本的に同じテクスチャーフォーマット（インスペクタでスプライト画像を表示したときの Format プロパティのタイプ）でまとめられます。そのため、同じ Monster タグの画像であっても、「足」だけが Truecolor で、それ以外が 16bits であれば、”Monster(Group0)”,”Monster(Group1)”というようにグループが分けられ、アトラスも分割されます。また、アルファチャンネルの有無でもグループが分かれます。パックグループは図 B04_01_005 のドロップリストボタンで確認できます。

さらに、パック処理中にデフォルトで指定されている最大アトラスサイズ(2048x2048)を超えると、今度は「パックページ」で分けられます。こちらも図 B04_01_005 のドロップリストボタンで確認できます。

さて、このようなパックの「パッキングポリシー（ルール）」ですが、通常はデフォルトの「Default Packer Policy」が適用されています。アトラスの最大サイズが 2048x2048 などを変えた場合は、「カスタムポリシー」を作成する必要があります。カスタムポリシーは Unity エディタで設定するのではなく、Assets フォルダの Editor フォルダにスクリプトを作成して指定します。詳しくは、下記の URL のマニュアルをご覧ください。

Unity 公式マニュアル:Sprite Packer
http://docs-jp.unity3d.com/Documentation/Manual/SpritePacker.html

なお、Sprite Packer は Resources フォルダの中にある Sprite アセットはパックできません。また、パックしたアトラス画像は、”Project¥Library¥AtlassCache.”に保存されています。

さて、この便利なアトラス化の機能ですが、アセットストアで販売されている 2D Toolkit などでも利用することが可能です。Unity のライセンスは、パッケージとして購入すると結構高額なので、Unity PRO の機能が必用ない方は、2D Toolkit などのアセットの利用を検討されても良いでしょう。逆に、会社などで中規模から大規模のゲームを開発される方は、「アセットバンドル」と呼ばれるネットを介してアセットデータのロード機能が必須となるので、Unity PRO のパッケージ版か、月ごとにお手軽な値段で利用できるサブスクリプションライセンスの購入を検討してみてください。

コラム：便利な SpritePacker アセット

Unity PRO の機能である Sprite Packer ですが、Unity ユーザーによって独自の Sprite Packer アセットもいくつか公開されています。
まず、単純に Assets フォルダの中のスプライト画像をパックしたいのであれば、「ケットシーウェア」さんの「SpritePacker」がお手軽で便利です。パックしたい Sprite い画像を選択して Sprite Packer の Create ボタンを押すだけでアトラス画像を作成してくれます。

ケットシーウェア
【Unity】スプライトを 1 枚にまとめる簡易 SpritePacker
http://caitsithware.com/wordpress/?p=263

有料のアセットであれば、アセットストアから購入できる「2D Toolkit」があります。
Unity2D に対応したスプライトパック機能の他に、フォント表示などにも対応しています。

Unikron SOFTWARE：2D Toolkit
http://www.unikronsoftware.com/2dtoolkit/

また、様々なゲームエンジンで利用できる「TexturePacker」というスプライトツールもあります。
シンプル操作ながらも非常に強力なスプライトパックツールで、プロも多く利用しています。
TexturePacker 作者の公式インポートアセットがアセットストアで提供されているので、これを利用することで、Unity に TexturePacker で作成したスプライトシート（アトラス）をインポートすることが可能です。
有料ですが、無料で試せる体験版もあります。

CodeAndWeb：TexturePacker
http://www.codeandweb.com/texturepacker

Sprite Packer 機能を使いたいけど、Unity PRO や 2D Toolkit などの購入を躊躇われている方は、これらのアセットの利用も検討してみると良いでしょう。

２.6.　罠・罠・罠

Unity と C#でゲームを作る上で、ハマりやすい罠についてご紹介します。

Unity の物理エンジンと単位の問題

Unity の物理エンジンを利用してゲームを作る場合､ぜひとも注意しなければならいないことがあります。それは「単位」です。

Unity はシーン中の大きさを「Unit」という単位で表現しています。Tranform で指定した座標やスケールが 1.0 であれば 1.0Unit になります。また、物理エンジンは 1Unit を 1 m として扱います。一方、重さは Mass1.0 で 1kg です。

この「単位」を念頭においてプログラムしないと、物理エンジンを利用して思ったようなゲームを作ることができなくなります。その理由は簡単です。物理エンジンを利用したゲーム開発では、移動などにおいて「エネルギー」の概念が必用だからです。

例えば、1kg の重さのものを 1m 移動させるのと、1g のものを 1m 移動させるには、必要なエネルギーが違います。特に摩擦がある場合では、物体が動き出すまでに必要なエネルギーに差がでます。そのため、Rigidbody2D.AddForce で力を加える場合は、同じエネルギーでも、物体の質量によって移動できる物体と移動できない物体ができてしまいます。例えば､勝手に Mass1.0 を 1g と考えてゲームを作った場合、1000g のものを動かすつもりでも､物理エンジン内では 1000kg の物体であるため､移動させるために必要なエネルギー量はとてつもなく大きな数値となります。

また、重力による落下は、物理法則では質量には比例せず、どんな物質でも同じ速度で落下します。この落下スピードは、当然 1Unit を 1m として計算しているので、勝手に 1 Unit を 1cm と考えてゲームを作ってしまうと､異常に落下速度が速いゲームになってしまいます。このような場合は、紹介した GravityScale 値を変更したり、Physics2D の Gravity 値を Unity エディタか Physics2D のプロパティ値から変更しなければなりません。

なお、角度については、インスペクタで表示されている角度はオイラー角（物体の角度を x,y,z をそれぞれ 360 度で表したもの）で表現されています。スクリプトからは、弧度法による角度からラジアン、ベクトル表現からクォータニオンなどの表現が可能です。

これまでの話をまとめると、表 3.4.5.1 のようになります。

表 3.4.5.1 Unity における単位

項目	説明
座標(Transform.position)	1.0 で 1Unit(1m) また、Unity のシーンにおける座標限界は x,y,z 軸ともに±100,000 が限界（この範囲外の値にした場合は、そのゲー

	ムゲームオブジェクトは表示されません）
角度(Transform.rotation)	インスペクタで表示されるときは、ローカル座標でのオイラー角。スクリプトの場合はクォータニオン(Quaternion)オイラー角で指定する場合は、下記の通り。 Transform.rotation = Quaternion.Euler(45,90,0); Transform.localEuler = new Vector3(45,90,0); なお、Transform.Rotate(45,90,0)と記述することで指定した角度を相対的に回転させることができる
スケール(Transform.scale)	1.0 で 1Unit(1m)
重さ(Rigidbody2D.mass)	1.0 で 1kg
重力加速度(Rigidbody2D.gravity)	X=0,Y=-9.81 「Edit」メニューの「Project Settings」から「Physics 2D」を選択して、インスペクタに「Project2DSettings」を表示して、「Gravity」値を変更することが可能

これはら、忘れずに覚えておいてください。

Unity のシーンにおけるスケールの問題

「Unity の単位」について説明しましたが、関連してぜひ覚えておいてほしいことがあります。それは、「シーンにおけるスケールの問題」です。

まず、Unity のシーンにおける座標限界は x,y,z 軸ともに±100,000 が限界です（この範囲外の値にした場合は、そのゲームゲームオブジェクトは表示されません）。そのため、最初からこの広さ以上のゲームを作りたい場合は、ゲームオブジェクトのスケールを 1/10 に縮小するか、ステージを分けるかなどの工夫が必用となります。

例え、ゲームで利用するシーンの空間の大きさが±100,000 に収まっても、物理エンジンを使用する場合は注意が必用です。

Unity の物理エンジンは、ゲームオブジェクトの座標や速度、そしてエネルギーなどを float 型（32 ビット単精度浮動小数点型）で処理しています。float は、-3.4×10^{38} ～$+3.4 \times 10^{38}$ までの範囲を数値として表現できます。一見、無限にも思える数値ですが、小数点以下の値を持つ値の掛け算が複数回行われると、あっという間に計算の誤差が大きくなります。試しに C#で、”float a = 1000000.0f + 0.1f”を実行してみてください。結果は、1000000.1 ではなく 1000000.0 になります。0.1 どっかにいっちゃいました。

これは、前にも説明したように float が数値をそのまま数字のデータとして持っているわけではなく、大きな数を表現できる計算式のパラメータとして保存しているためです。そのため、極端に大きな数値同士の演算や、極端に差の大きい数値同士を演算すると、簡単に誤差が生じます。このような float による浮動小数点演算の特性から、Unity の物理エンジンでもシーンの原点(0,0)に近いほど誤差はなくなり、原点から遠くなるほど誤差が大き

くなります。float の 32 ビット単精度浮動小数点型なら誤差を許容できる有効桁数は 7 桁です。7 桁もあれば十分に思えるかもしれませんが、実際には、座標に掛け算することも多いため、3 桁同士の掛け算が誤差を許容できる範囲となります。つまりシーンのワールド座標(1000.0f, 1000.0f)であれば誤差を気にせず Physics2D の物理エンジンを動かせますが、ワールド座標(10000.0f, 10000.0f)であれば確実に誤差がでます。最悪の場合、ゲームオブジェクトがガクガク震えたり、アタリ判定（コライダー）を突き抜けたりします。これは座標だけでなく、速度でも同じです。ワールド座標(100.0f, 100.0f)にあるゲームオブジェクトでも、速度（rigidbody2D.velocity）が(10000.0f, 10000.0f)だとアタリ判定がおかしくなるでしょう（実際には、速度は内部でリミッターが掛かっており、極端に座標が大きくない限りおかしな挙動にはならないようになっています）。なお、これらの問題は、数値が極端に小さい場合でも同様です。

そのため、物理エンジンを利用したゲームを作る場合は、最初にシーン内で扱うエリアの大きさ、キャラクタの大きさなどを決めて置く必要があります。特にランナー系ゲームで、1 つのステージで走る距離が 10000unit(m)を超えるような場合は、事前に物理エンジンのテストをしておいた方が無難です。物理エンジンに指定したパラメータによっては、ゴール付近でおかしな物理挙動をしてしまう可能性があります（図 C04_07_001）。

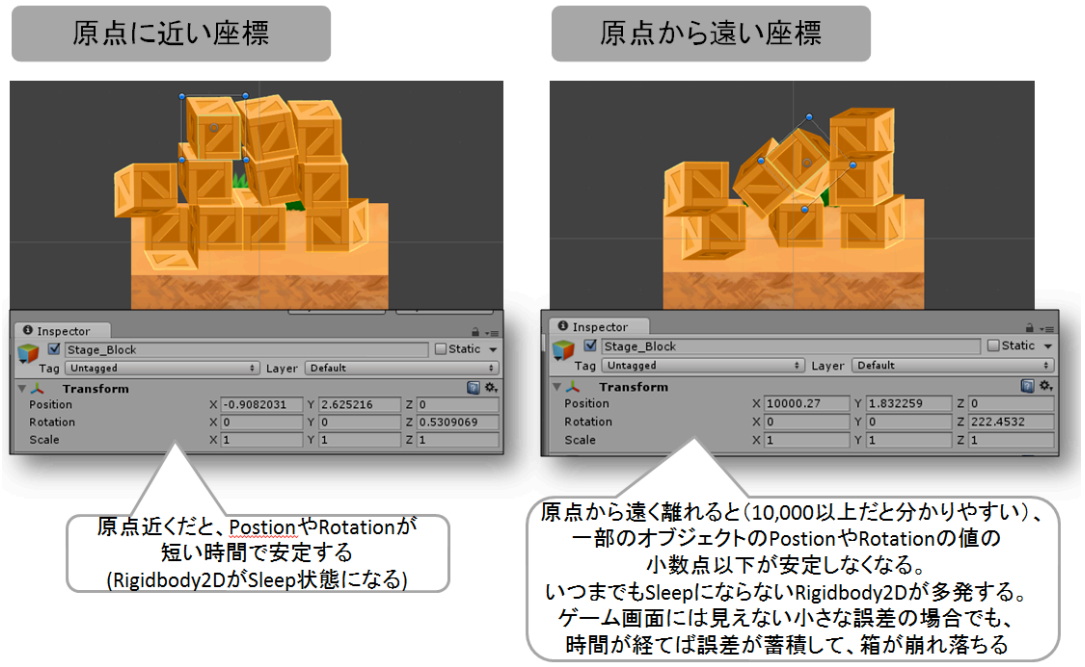


図 C04_07_001 スケールの問題

もし、このような問題が発生した場合は、ゲームオブジェクトのスケールを 1/10 に縮小するなどの工夫が必用となります。

なお、スケールを縮小しすぎた場合も同様の問題が発生しますので、極端に広いエリアのゲームを作る場合や、逆に、極端に小さい数値で座標を管理するゲームを作る場合にも、

事前に物理エンジンの実験をしておくことをお勧めします。

浮動小数点の罠

Unity で扱う C#では、小数を扱う場合、「float 型（32 ビット単精度浮動小数点型）」と「double 型（64 ビット倍精度浮動小数点型）」を使用することができます。

float は-3.4×10^{38} ～$+3.4 \times 10^{38}$（有効桁数 7 桁）までの範囲を数値として表現できます。 double では$\pm 1.5 \times 10^{-324}$ ～ $\pm 1.7 \times 10^{308}$（有効桁数 15～16 桁）までの範囲を数値として表現できます。この範囲は float.MinValue, float.MaxValue, double.MinValue, double.MaxValue として定義されています。

また、C#のソースで”0.1”という小数を表現する場合、そのまま”0.1”と記述すると double として扱われます。float として表現したい場合は、”0.1f”のように数値の末尾に”f”または”F”を付けます（unsigned の場合は”uf”,”UF”）。double の場合は、”0.1m”のように”m”または”M”を付けます（unsigned の場合は”um”,”UM”）。

Unity では処理速度を優先するために、高速に処理できる float 型を使うのが一般的です。

float にしろ double にしろ、非常に大きな（または小さな）数字を扱えるので、とても便利に思えます。しかし、この浮動小数点型には、簡単にバグを引き起こしてしまう「比較の罠」「演算の罠」「エラーの罠」3 つの罠が存在します。

これは Unity でゲームを作る場合でも避けられない、ひっかかりやすい罠です。

- **比較の罠**
  浮動小数点型では、等号の比較を行う場合、整数と同じように 2 つの数値をそのまま”==”記号でチェックするのは危険です。試しに C#で、”0.3f + 0.732f == 1.032f”という条件式を実行してみてください。結果は、恐るべきことに true ではなく false が返ってきます。double の場合は、”0.1+0.2==0.3”が false になります。これは、float や doble が小数の数値をそのままデータとして持っているわけではなく、大きな数を表現できる計算式のパラメータとして保存していることで、わずかな誤差がでるためです。ソースからは見えない小数点以下の要素が存在するのです。
  そこで、確実な精度で浮動小数点型の等号比較を行う場合は、C#では”Math.Abs((0.3f + 0.732f) - 1.032f) < Single.Epsilon”と記述します。2 つの数値の差分を絶対値で計算した後に、C#に定義されている「float.Epsilon（0 より大きい最小値。dobule の場合は Double.Epsilon）」で比較して限りなく等しいか判断しているわけです。
  なお、float や double には、簡単に正確な比較ができるように float.CompareTo や float.Equals 関数が用意されています。また、Unity でも float の比較できる **Mathf.Approximately** 関数が用意されています。

- **演算の罠**
  「比較の罠」で、例として小数点同士を加算すると誤差が生じることを説明しました。これが「演算の罠」です。

この罠は、もっと簡単な方法でも発生します。例えば、0.001f を 1000 回足し算すると、1.0f ではなく 0.9999907f になります。加算するごとに誤差が蓄積されるのです。1.0f を期待してプログラムすると、大きなバグの要因となります。もし、1.0 を期待するのであれば、Mathf.Lerp 関数などを使って 0～1.0f の値に対して、i を 1000 が 1.0 となる割合として指定して計算する方が安全です。

・ **エラーの罠**

プログラムで演算を行うと実行時エラーが出る場合があります。int 型などの整数であれば、数値を 0 で割ってしまう「ゼロ割演算エラー（Zero Divide Error）」は皆さんも一度は経験あるのではないでしょうか。実行してみないとバグが発覚しない実に嫌なバグです。

しかし、float や double の浮動小数点型では、ゼロ割演算（ゼロ除算）の結果は、エラーにならず**「無限大（Infinity）」**になります。ゲームプログラムが落ちないという意味では嬉しい仕様かもしれません。が、想定していない計算バグによりゼロ割演算が発生している場合は、Unity のコンソールにもエラーとして表示されないためバグを見逃すことになります。

このようなエラーにならない仕様は他にもあります。int 型などの整数型では表現できる数値の範囲を超えるとオーバーフローが発生します。プログラムは停止しませんが、オーバーフロー後の変数は、正しい計算をできません。そこで、C#では、Checked,Uncheck 文を使うことで、オーバーフローを例外として発生させて OverflowException でキャッチできます。が……float や double の浮動小数点型では、そもそもオーバーフローという概念がありません。無限大として表現されます。この無限大は、float なら float.NegativeInfinity（負の無限大）, float.PositiveInfinity（正の無限大）として定義されています。もし、数値が無限大かチェックしたい場合は、float.IsInfinity ,float.IsNegativeInfinity,float.IsPositiveInfinity 関数で調べることが可能です。

最後に**「NaN（Not a Number）」**について説明します。

float や double の浮動小数点型では、特殊な状況下で誤った計算が行われると NaN という非数（数字ではない値）を返します。C#で簡単に float の値を NaN にしたければ”a = 0.0f / 0.0f”を実行します。a の値は非数となり、float.NaN の定義と同じ値になります。非数の状態からでも継続して演算は可能です。先ほどの a の値に 0.1f を加算すると結果は 0.1f になります（NaN ではなくなります）。この NaN の状態はエラーではありません。ですが、プログラム的にはとても大きなバグをはらんでいる可能性があります。この値を Unity や OS の API に渡してしまうことで、実行時エラーになる可能性もあるのです。NaN で問題が起こることは稀ですが、もしバグが発生して気になることがあれば、float.IsNaN 関数を使って数値が NaN でないかチェックすると思わぬ解決に繋がるかもしれません。

どうでしょうか？　軽く float や double の浮動小数点が嫌いになったかもしれません。

この他にも、誤差を少なく計算するに、近い数値同士から計算を始めるなどのテクニックがあります。それこそ詳細に解説しようとすれば 1 冊の本になるでしょう。ただし、もっとも危険な事項については、この「比較の罠」「演算の罠」「エラーの罠」の 3 つの罠だけですので安心してください。大丈夫です（たぶん）。

なお、さらに小数計算の精度がほしい場合は、C#には「Decimal 型（128 ビット 10 進数浮動小数点型・有効桁数 28～29 桁）」が用意されています。金融機関などでお金を計算する場合などに使われるます。計算誤差が少ないのが特徴ですが、処理速度が遅く、また、極端に大きな数値や小さな数値では誤差が発生するため、ゲームではほとんど使用されません。最後の手段と思っておきましょう。

第３章
Unity2D で作ったゲームの高速化とメモリ管理（中級者向け）

3.1. Unity 高速化のお約束

Unity でゲームが作れるようになると、「処理落ち」をはじめとする処理速度に関する問題に突き当たります。

そこで、Unity を高速で動作させるための様々なノウハウをご紹介しましょう。

高速化の基本

それでは、まず Unity を使う上で、高速化の基本となるポイントを箇条書きで紹介していきます。

- Unity にある機能は、まず、その機能を使って速度を確かめる
  それでも遅い場合に C#でコーディングして早くなる可能性があれば実装する
- ゲームオブジェクト、タグなどの検索は、動的に変化しないものであれば、Awake や Start で検索して保存（キャッシュ）しておく。Update や FixedUpdate で検索しない
- ハッシュでの検索や比較が可能なクラスやプロパティは、文字列でなくハッシュで操作する

高速化のメモリ管理と高速化

次に、Unity でメモリ操作をする場合の注意点を箇条書きで紹介します。

Unity では、C++などのネイティブ言語とは違い、メモリの確保や解放を自動で行ってくれます。ただし、複数のメモリ解放し分断された空きメモリ空間をまとめるガベージコレクション（通称ガベコレ）が発生した場合は、この処理のために極度に処理速度が低下します。そのため、いかにガベコレを発生させないかが高速化へのポイントとなります。

- Unity では、ガベコレを最小限に抑えることが高速化に繋がる
- 特に Update や FixedUpdate でメモリの確保や解放が頻繁に発生するような処理をしない
- 文字列クラスの文字列操作のメソッドは、意識してなくても大量にメモリを確保・解放する場合があるので注意する。スコアなどは、スコアが変化しない場合は、文字列の変更を行わない。

・ どうしても、任意のタイミングに集中してガベコレが行われる場合は、大量のガベコレが行われる前に、System.GC.Collect メソッドを使って要求し、分散して行っておく

スクリプトの書き方でゲームを高速化する

本書では、C#を使ってスクリプトをコーディングしています。
C#は、高速な部類に入る言語ですが、それでもネイティブな C++に比べれば 10 倍近く処理速度が遅いのが現状です。そこで、C#で Mono や.Net、そして Unity のクラスを呼び出す場合に、高速化につながる方法をお教えします。

・ ループ内でメモリ確保・解放を極力避ける

・ ループ内での文字列操作は、メモリ確保・解放に繋がる可能性があるので、事前に処理できるものはループ内に入れない。

・ GameObject を取得時のキャストは as の方が高速

・ クラス内で参照するゲームオブジェクトやコンポーネントは、Awake や Start でキャッシュしておく

・ this.transform など、MonoBheivour が持っているコンポーネントも、参照時に Get Component を行っているので、高速化するなら Awake や Start でキャッシュしておく

ゲームオブジェクトの高速化

ゲームオブジェクトも、管理の仕方や作り方次第で、処理速度が大きく変わります。

・ ゲームオブジェクトの階層を必要以上に深くしない

・ 不要になったコンポーネントは削除する

・ カメラ外で動作の必要がなくなったゲームオブジェクトはアクティベーション(SetActive)を false にする
ただし、transform などの内部のプロパティの情報がほしい場合は、ゲームオブジェクトではなく、必要のないコンポーネントの enalbled を false にする

- 必用のないゲームオブジェクトに Rigidbody や Rigidbody2D、コライダーなどを付けない
- 不要なコンポーネント内の機能をプロパティからオフにする（レンダラー系コンポーネントの Cast Shadows や Recive Shadows プロパティなど）
- ゲームオブジェクトの確保・解放を少なくする
  また、頻繁に確保・解放をする必要があるゲームオブジェクトは、解放するのではなく休止させたり、パラメータを上書きしたりして再利用を行う
- Instantiate によるゲームオブジェクトの生成は、プレハブよりシーン内からのコピーの方が高速

スプライトの高速化（アルファチャンネル編）

次は、Unity2D のスプライトの高速化について説明しましょう。

Unity2D では、スプライトを高速に表示するために、ちょっとおもしろい仕組みを採用しています。それは、スプライト画像の透過色以外の部分をポリゴン形状として持たせて描画するという仕組みです。これにより、スプライトを描画するときに 1 ドットごとに行われるアルファテスト（透過色の判定処理）が不必要となるため、多くのプラットフォームで高速にスプライトを描画することができます。アルファ付きの画像であっても、最低限の面積でアルファテストが行われるようになるため、高速化が期待できます（図 Q001_05_001）。

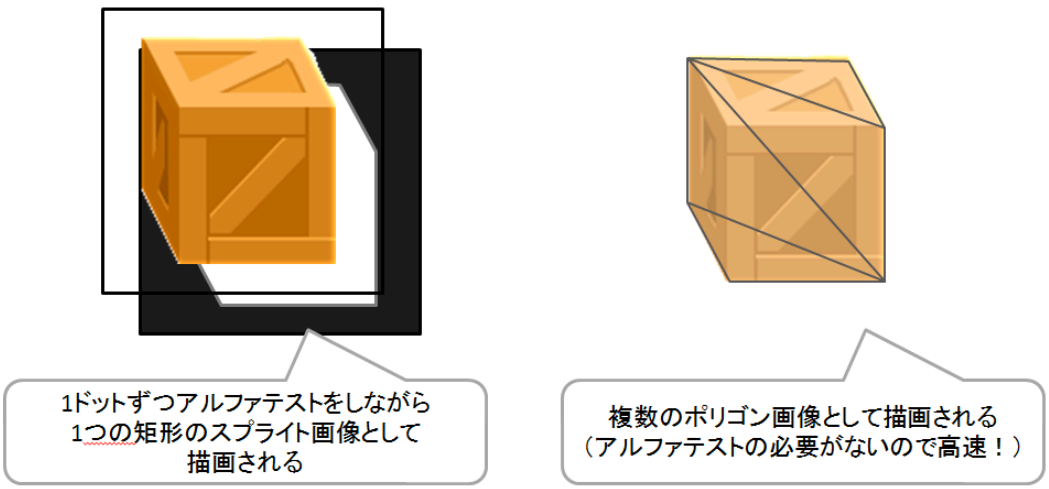


図 Q001_05_001 スプライトの高速表示

当然、ポリゴン数が少ないか描画面積が少ないほど高速に描画できるため、形状によって速度も変化します。また、この状態でも SpriteRendarer の Color プロパティのアルファ値を変えることで、半透明にすることもできます。

ただし、この機能は Unity PRO のみの機能になります。

この機能を利用するには、Spirte インポータの「Texture Type」プロパティを Advanced に変更し、「Mesh Type」プロパティを「Full Rect（矩形）」から「Tight（図形の形状に合わせてポリゴンで分割）」に変更します。また、「Extrude Edges」の数値で、図形の境界線からどれだけ外側に余白をとってメッシュ（分割されたポリゴンの集合）を作成するか指定することが可能です（図 Q001_05_002）。

図 Q001_05_002 Unity PRO の機能を使ったスプライト画像の高速化のための設定

実際にスプライトが分割されて描画されるかについては、シーンビューで表示リストから「Texture Wire」を選択してください。スプライトの分割されたメッシュを確認することができます（図 Q001_05_003）。

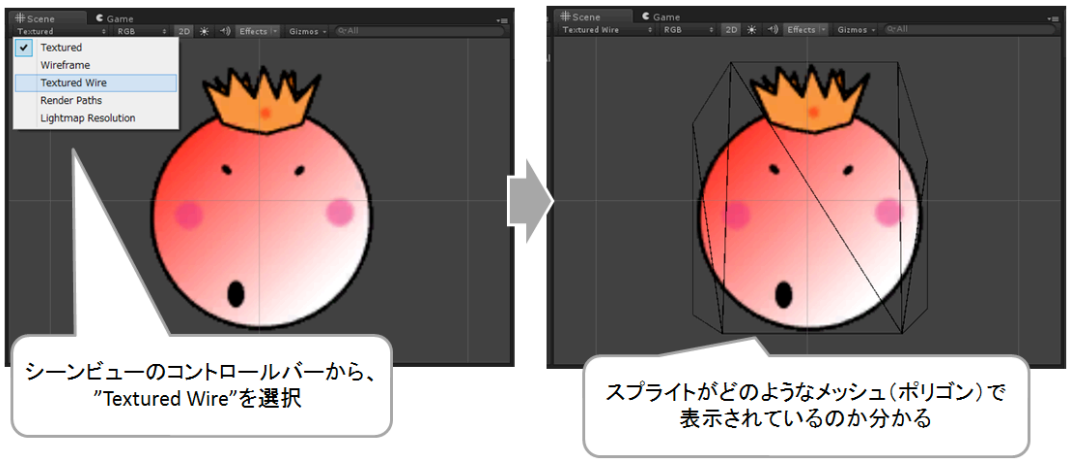


図 Q001_05_003 スプライトのメッシュ表示を確認する

スプライト画像のフォーマットの設定における高速な表示方法を処理速度番に並べると、次のようになります。

1. アルファチャンネルやアルファ値を持たないスプライト画像の矩形表示
2. アルファチャンネルやアルファ値を持たないスプライト画像を、UnityPro 機能の「Tight」で表示
3. アルファチャンネルやアルファ値を持つスプライト画像を、UnityPro 機能の「Tight」で表示
4. アルファチャンネルやアルファ値を持つスプライト画像の矩形表示

Unity Pro を利用させている方は、ぜひ「Tight」設定を試してみください。

スプライトの高速化（ドローコール編）

Unity では、グラフィックを描画（レンダリング）する場合、描画を実行するための命令の呼び出しを「ドローコール」と呼んでいます。このドローコールは、1 つのゲームオブジェクトで 1 ドローコールになるわけではありません。ドローコールの正体はグラフィックボードにアクセスするためのグラフィックライブラリの呼び出しの命令です。そのため、グラフィックボードにとって都合の良い手順でグラフィックの描画を指示してやると、ドローコールは格段に数が減ります。逆に、グラフィックボードの苦手な手順で描画を指示すると、ドローコールが増えます。

ドローコールは、Game ビューの「Status」をクリックすると表示されます。なお、UnityPro では、「Window」メニューから「Prfomance」を選択するだけで、ドローコールの変化をグラフで詳細に教えてくれます（図 Q001_06_001）。

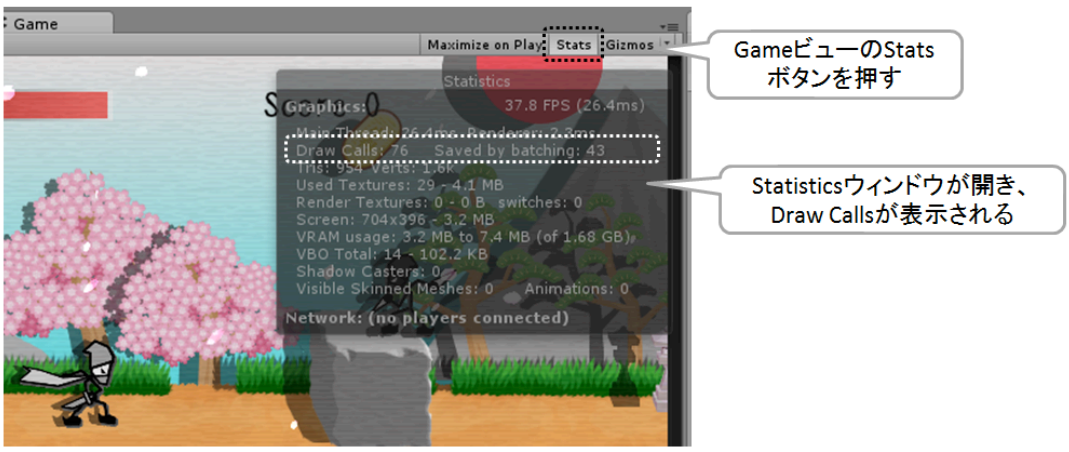


図 Q001_06_001 ステータスを表示してドローコールを確認する

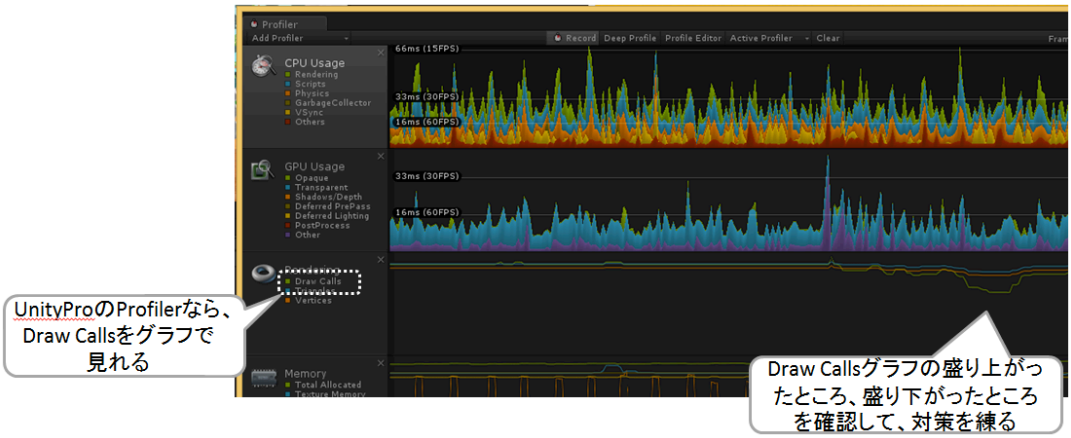


図 Q001_06_002 Unity PRO の Profiler ビュー

では、ドローコールを、もっとも分かりやすい例で説明しましょう。

図 Q001_06_003 のようなスプライトを表示する場合、1 つ 1 つのゲームオブジェクトから描画命令を実行すると、ゲームオブジェクトの数だけドローコールが発生することになります。しかし、同じスプライト画像のデータをバラバラのメッシュ座標（頂点座標）として描画することで、ドローコールの数が 2 つになります。ドローコールは少なければ少ないほど処理速度が速くなります。このような高速化の処理を「バッチング（または、ドローコールバッチング）」と呼びます。

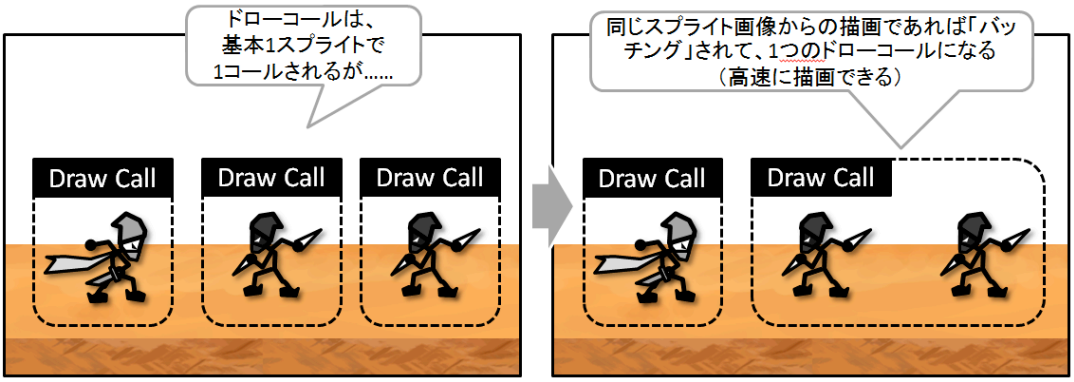


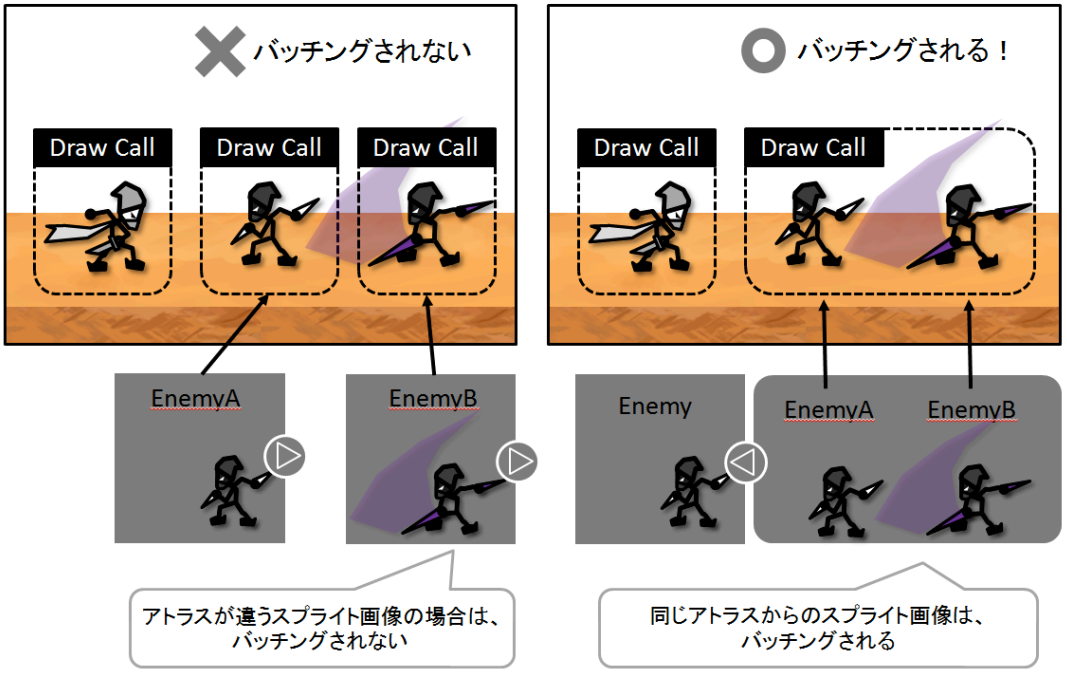
図 Q001_06_003 ドローコールとバッチング

Unity では、Unity2D のスプライトにおいて、このバッチングを自動で行っています。

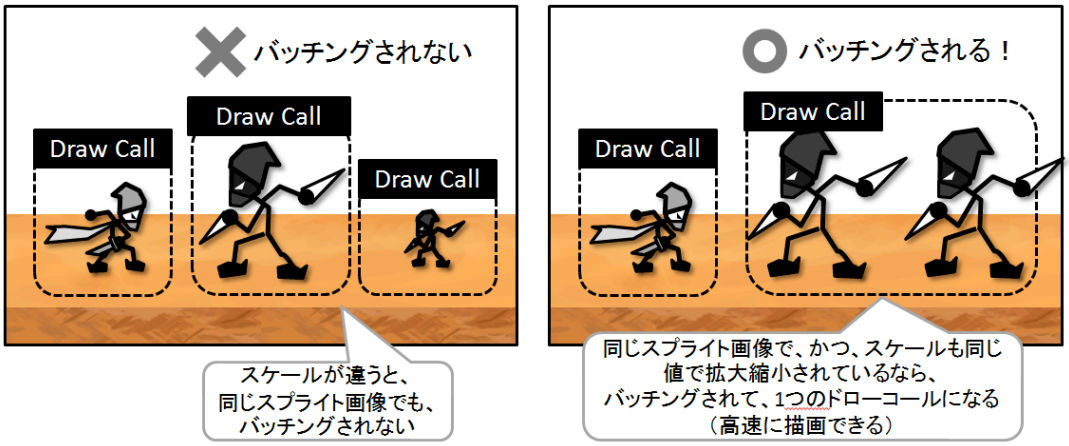
ただし、バッチングが行えるのは、同じスプライト画像で、かつ、Rotate や Scale などが同じ場合のみです。また、これらが同じ場合でも、SpriteRenderer コンポーネントの Color プロパティで Color 値やアルファ値を変更しているときは、バッチングされないことがあります。

ドローコールが最適化されるケースは、次の通りです（図 Q001_06_004）

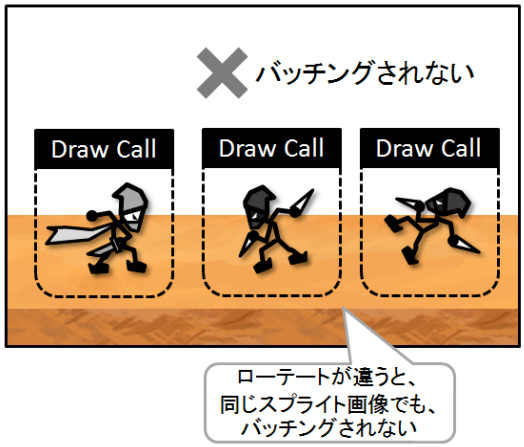
アトラスの違い

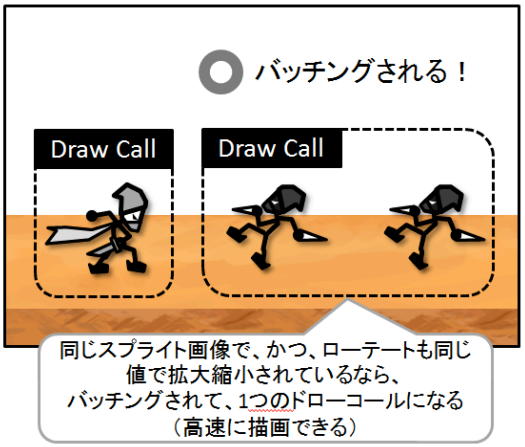


スケールの違い

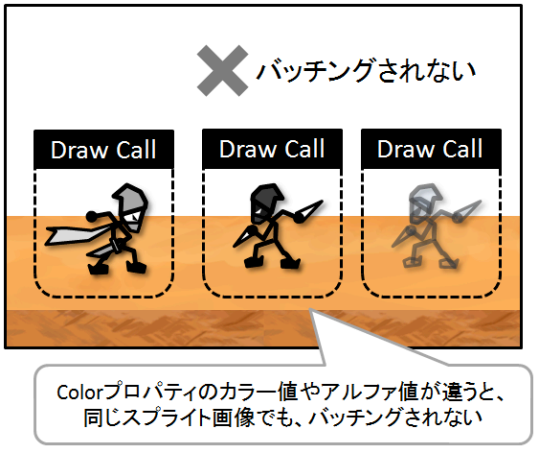


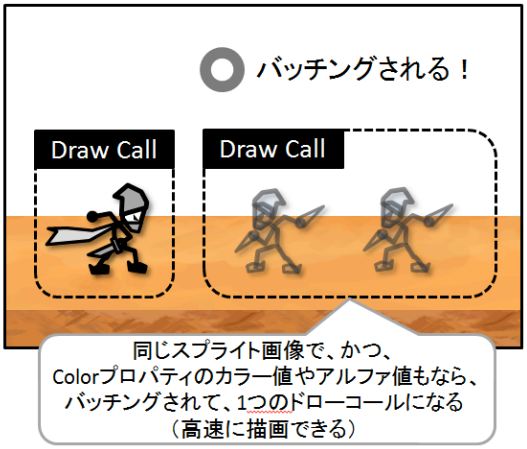
ローテートの違い





アルファの違い





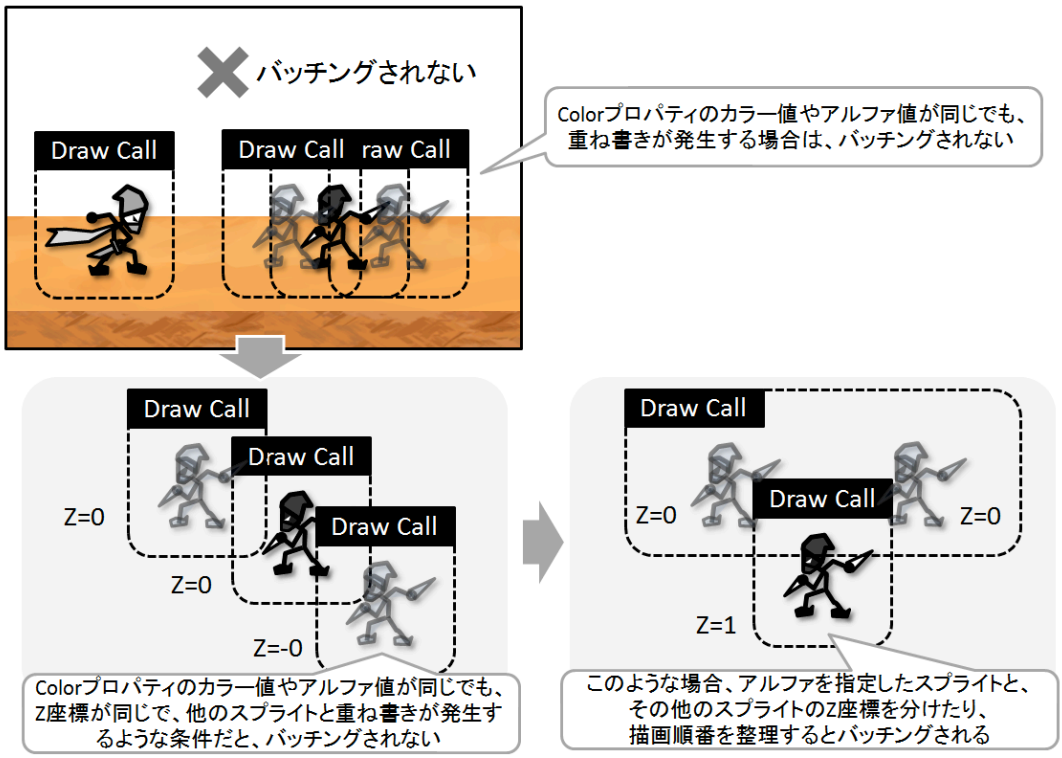


図 Q001_06_004 ドローコールがバッチングされる条件・されない条件

このように、キャラのように動いているゲームオブジェクトのドローコールをバッチングすることを「Dynamic Batching（ダイナミックバッチング）」と言います。

この他に Unity には、背景のように動いていないものをバッチングする「Static Batching（スタティックバッチング）」の機能があります。ゲームオブジェクトをインスペクタで見たときに表示される「Static」のチェックボックスが、それです。このチェックボックスをオンにすることで、動いていないゲームオブジェクトの描画をスタティックバッチングとして指定できます。これにより、同じアトラスとのスプライト画像であれば、スケールやローテートが違っていてもバッチングされます。ただし、この機能は Unity PRO のみの機能となります（図 Q001_06_005）。

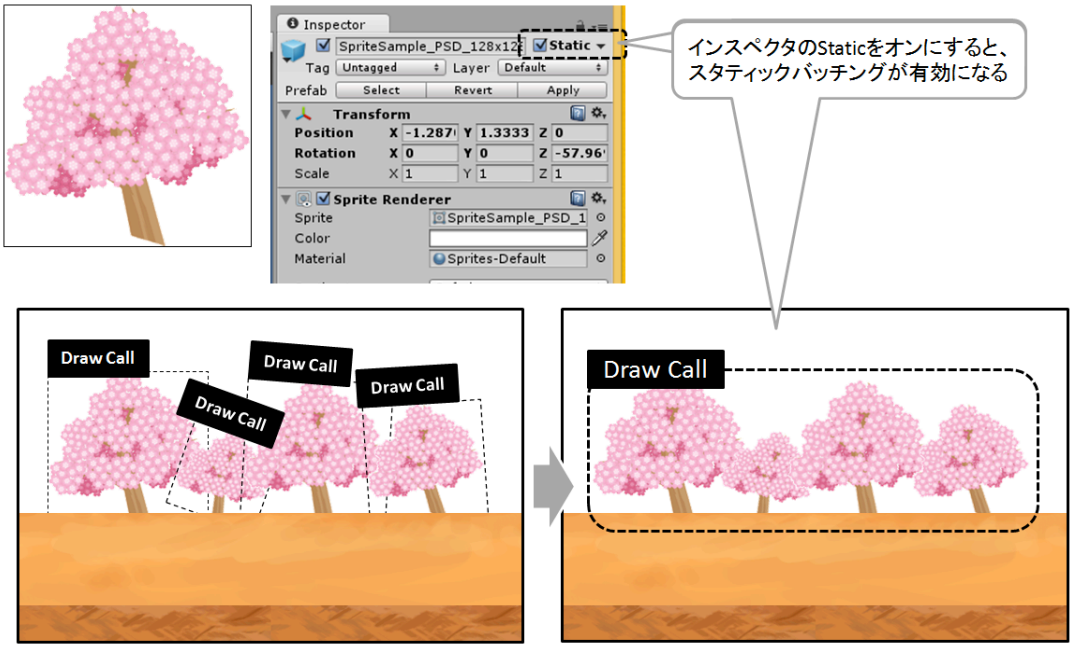


図 Q001_06_005 スタティックバッチング

これらのバッチングは、PlayerSettings の Other Settings グループにある「Static Batching」「Dynamic Batching」でオンオフ設定が可能です（図 Q001_06_006）。

図 Q001_06_006 「Static Batching」「Dynamic Batching」の設定

では、スプライトの高速化についてまとめましょう。

- 表示を高速化するには、ドローコールを減らす必要がある
- ドローコールはバッチング機能によって、同じ Rotation,Scale のスプライトなら節約することができる
- アトラスを作ることで、ドローコールを減らせる
- 半透明で表示する必要のない画像は、アルファチャンネルを作らずに、ポリゴンのスプライトとしてインポートする
- 大量のスプライトを表示したい場合は、パーティクルシステムを使う手段も検討する（ParticleSystem ゲームオブジェクトはバッチング効率が良いため）
- 前景や背景に 3D のゲームオブジェクトがある場合は、Unity Pro ならオクルージョンカリングの機能で、カリング（カメラに映らないものをレンダリングしない）して高速化できます（詳しくは http://docs-jp.unity3d.com/Documentation/Manual/OcclusionCulling.html）。

プラットフォームごとの高速化

Unity エディタ上で、高速に動作するようになっても、いざ、iPhone や Android などで動作させると思ったように速度がでないことがあります。プラットフォームごとの特徴を掴んで、高速化させる必要があるのです。

- メモリの消費は可能な限り抑える
- 数十メガ単位の極端に巨大な配列やオブジェクトなどは、ゲーム開始時に確保を行い、ゲームプレイ中に確保したり解放したりしない（OS レベルでの不要アプリのメモリ解放処理などが動作する可能性があります）
- スプライトの画像サイズや色数、アルファなどを、各プラットフォームが高速に処理できる形にインポータで設定する

また、GPU によっては、アルファを利用した半透明が極端に不得意（処理速度が遅い）な場合があります。マルチプラットフォームで高速に動作するゲームを作る場合は、アルファチャンネルを使用したスプライト画像や、SpriteRenderer のアルファ値による半透明描画を極力抑えるようにしましょう。

遅くなっている原因を調べる

最後に、どうしても高速化できない場合の対応方法について説明します。

まず、高速化する場合は、やみくもに高速化のための工夫を凝らしても上手くいくとは限りません。一番、処理が重くなっている「ボトルネック」を探す必要があります。

そのためには、作成したプログラムの実行情報が欠かせません。もっとも簡単に見ることができる実行情報は、Game ビューの Statstics ウィンドウです（図 Q001_08_001）。

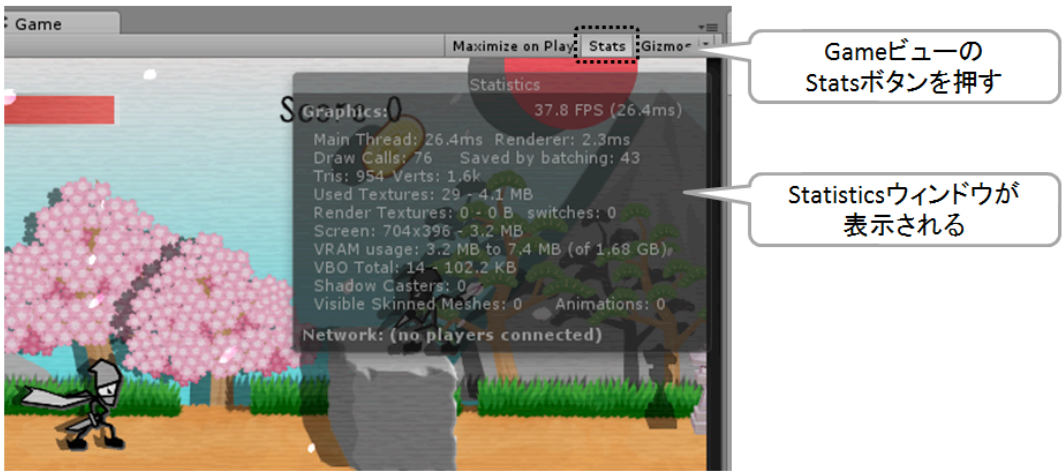


図 Q001_08_001 Statisics を利用したパフォーマンスの確認

フレームレートやドローコール数を確認することができます。

Statstics ウィンドウの各項目の内容は次の通りです（表 3.1.8.1）。

表 3.1.8.1 Statstics の項目説明

項目	説明
Graphics	FPS
Main Thread	1 フレームあたりのメイスレッドの処理時間
Renderer	1 フレームあたりの描画処理に掛かった時間
Draw Calls	1 フレームあたりのドローコール数（バッチング後の数）
Saved by batching	1 フレームあたりのバッチングでまとめたドローコール数
Tris,Verts	1 フレームあたりの Tris（三角ポリゴン数）と Verts（頂点数）
Used Textures	テクスチャの最小・最大メモリ使用量
Render Tectures	レンダーテクスチャの最小・最大メモリ使用量
Switches	レンダーテクスチャの切り替え回数
Screen	描画スクリーンサイズと、その画像容量

VRAM usage	VRAM の最小・最大の使用サイズ （of の後の数値は実 VRAM サイズ）
VBO Total	Vertex Buffer Object（頂点情報オブジェクト）の最小・最大サイズ
Shadow Casters	影の描画数
Visible Skinned Meshes	レンダリングされたスキンメッシュ数
Animations	再生アニメーション数

表示されているスプライトの数に対して、異様に重いと思ったらフレームレートとドローコール数をチェックして原因を探ると良いでしょう。

また、次のようなスクリプトで、ごくごく簡単ですが処理負荷を調べることもできます（ソース 3.1.8.1）。

ソース 3.1.8.1 :

```
float st = Time.time

// 計測したい処理

// 処理に掛かった時間
Debug.Log(srting.Format("time = {0}",Time.time - st);
```

高速化を行うなら、とにかく一番負荷の重い場所を調べて高速化するのが鉄則です。

例えば、Prfomance ビューを確認して、スクリプトエンジンの負荷が重いことが分かったら、どんなに描画処理を高速化しても早くなりません。スクリプト処理が重いのですから、その負荷の高いスクリプトを見つけて、高速化できるか検討する方が近道です。

まずは、計測するクセを付けるようにしましょう。

Unity Pro のプロファイラ機能

ボトルネックを探すのに、もっとも活躍するのが、Unity Pro のプロファイラ機能「Prfomance Analyzer」です。Unity Pro であれば、「Window」メニューから「Prfomance」を選び、Profiler ウィンドウを開くだけで、描画処理やスクリプトエンジンなどの動作負荷の状態を詳細に教えてくれます（図 Q001_09_001）。

図 Q001_09_001 Profiler ウィンドウを開く

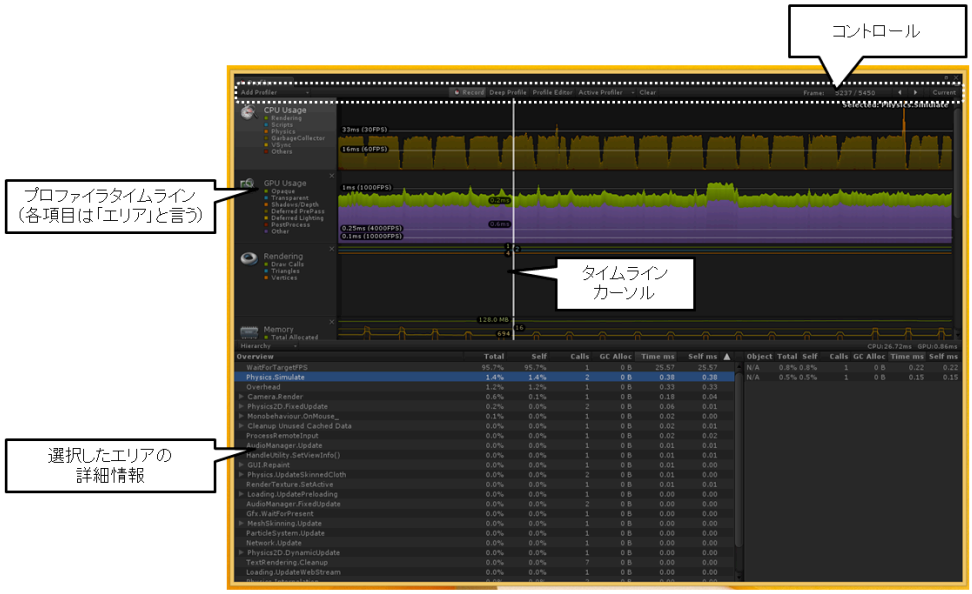


図 Q001_09_002 Profiler ウィンドウの機能

この Prfomance ウィンドウのコントローラから操作を行うことで、プロファイル機能を利用することができます（図 Q001_01_001）。

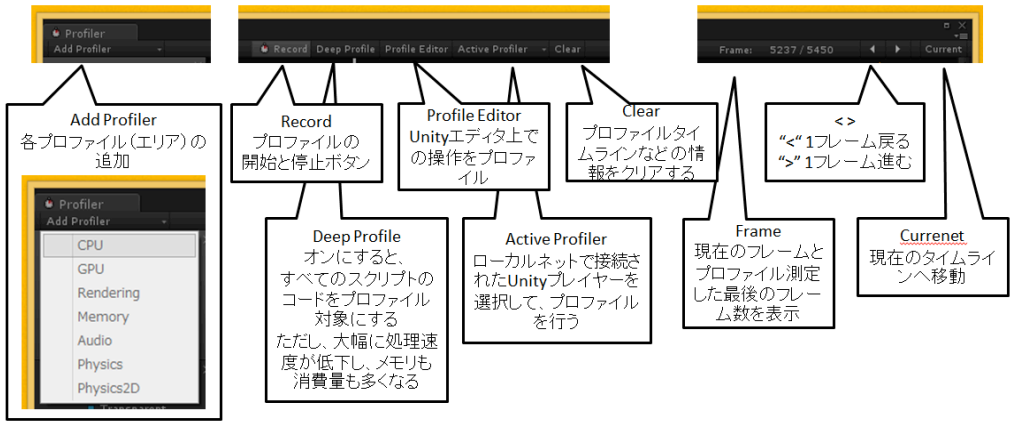


図 Q001_09_003 Profiler ウィンドウのコントローラ

Prfomance ウィンドウの下部は、上部のプロファイラタイムラインのエリアで選択したプロファイラの種類によって、表示される情報が変わります。例えば、プロファイラタイムラインで「CPU Usage（CPU 使用率）」を選択すると、下部には各コードの CPU 使用率がリストで表示されます（図 Q001_09_004）。また、「Rendering」を選択すると、下部にはレンダリングに関する詳細情報が表示されます（図 Q001_09_005）。

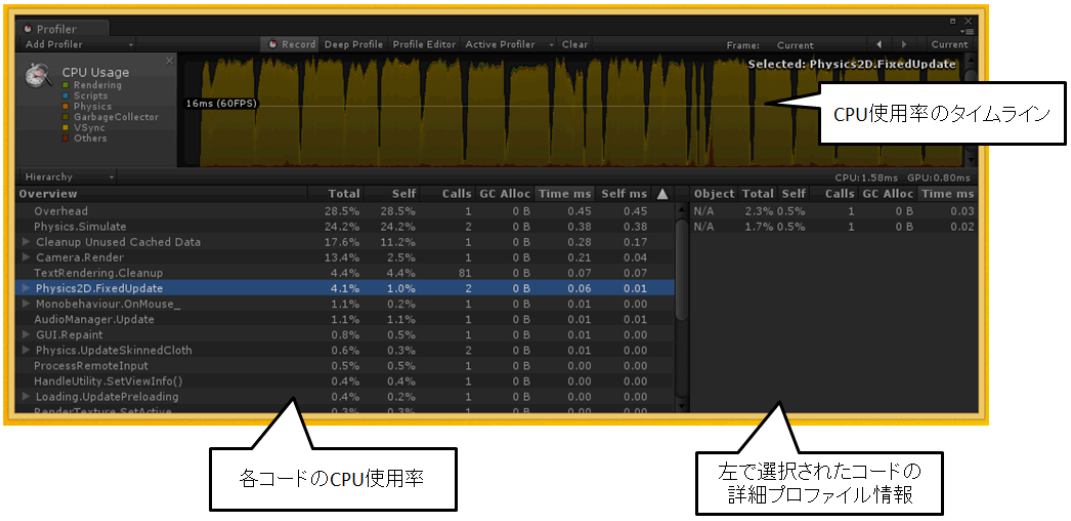


図 Q001_09_004 「CPU Usage（CPU 使用率）」

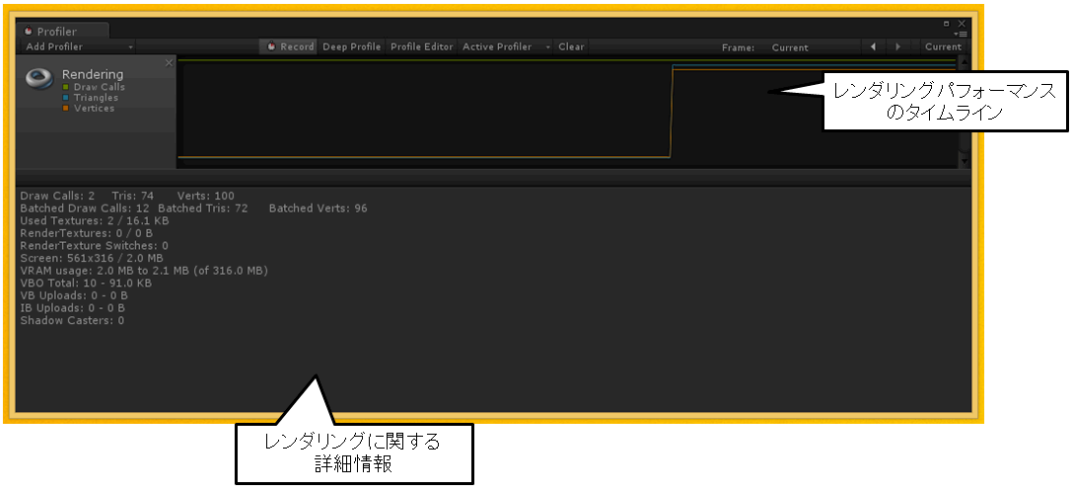


図 Q001_09_005 「Rendering」

なお、「CPU Usage」か「GPU Usage」を選択している場合は、下部にドロップリストが表示され、「Hierachy（階層表示）」か「Raw Hierachy（グループ化して表示）」が選べます（図 Q001_09_006）

図 Q001_09_006 階層表示の切り替え

このように、高度な機能を持つ Unity のプロファイル機能ですが、はじめて使う場合は、まず通常のモードでプロファイルを行います。このとき、「CPU Usage」を見て、描画処理などではなく、CPU によるプログラム処理が重たい場合は、コントローラの「Deep Profile」をオンにして、全スクリプトのコードのプロファイル情報を取得し、遅い処理を絞り込みます。

なお、ターゲットのプラットフォームが、iPhone や Android の場合は、Active Profiler から、それらのターゲットデバイスを選択してください。実機からのプロファイル情報を取得することができます（ただし、現在の Unity4.x 以降のバージョンでは iPhone プロファイルはできないようです）。

３.２.　処理速度を固定する

Unity で物理エンジンを使ったパズルゲームを作った場合、実装方法を間違ってしまうとクリアできないゲームができてしまいます。その原因は、物理エンジン内で使われている「時間」です。今度は、そのような問題が置きないようにするためのノウハウについて説明します。

Unity で扱う 2 つの時間

Unity では、TimeManager クラスで時間を計測することができますが、この中で扱っている時間スケールは「実時間」と「ゲーム内時間」の 2 種類に分かれます。

「実時間」は、現実と同じ時を刻んでいます。Time クラスの Time.unscaledTime,Time.unscaledDeltaTime,Time.realtimeSinceStartup,Time.timeSinceLevelLoad は、実時間を返します。

もう 1 つは、「ゲーム内時間（物理エンジンが使用している時間）」です。この時間は、進み方こそ現実の時間と同じ速度で進みますが、ゲームのプログラムが停止している間は、時間が止まってしまいます。Time.time,Time.deltaTime,Time.smoothDeltaTime,Time.fixedTime, Time.fixedDeltaTime で取得できます。

iPhone や Android などのスマートフォンであれば、ゲームを一旦やめてブラウザに切り替えたりすると、その間、現実の「実時間」は進みますが、「ゲーム内時間」は停止します。ブラウザの例は分かりやすいですが、実は、こっそりと OS が OS の仕事をするためにゲームを瞬間的に止める場合もあります。この時間は、大変短いためにプレイヤーは気が付かないでしょう。しかし、実際には、ゲーム内時間は止まっているため、ブラウザを開いたといった操作を行わなくても、「実時間」と「ゲーム内時間」に誤差が生じるのです（図 Q002_01_001）。

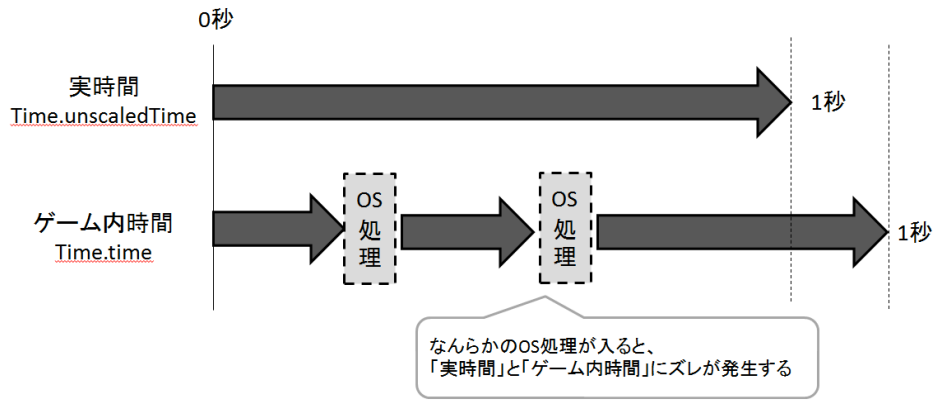


図 Q002_01_001　「実時間」と「ゲーム内時間」で発生するズレ

そのため、「10 秒後に開く箱の中に、11 秒後に爆発する爆弾を爆発前に入れろ！」といったパズルでは、箱の開く時間を Time.time のゲーム内時間で実装し、爆弾を Time.unscaledTime の実時間で実装してしまうと、実行しているプラットフォームによっては、爆発と箱の開く時間がずれて、どんなにがんばっても爆弾が先に爆発してクリアできないパズルとなります（図 Q002_01_002）。

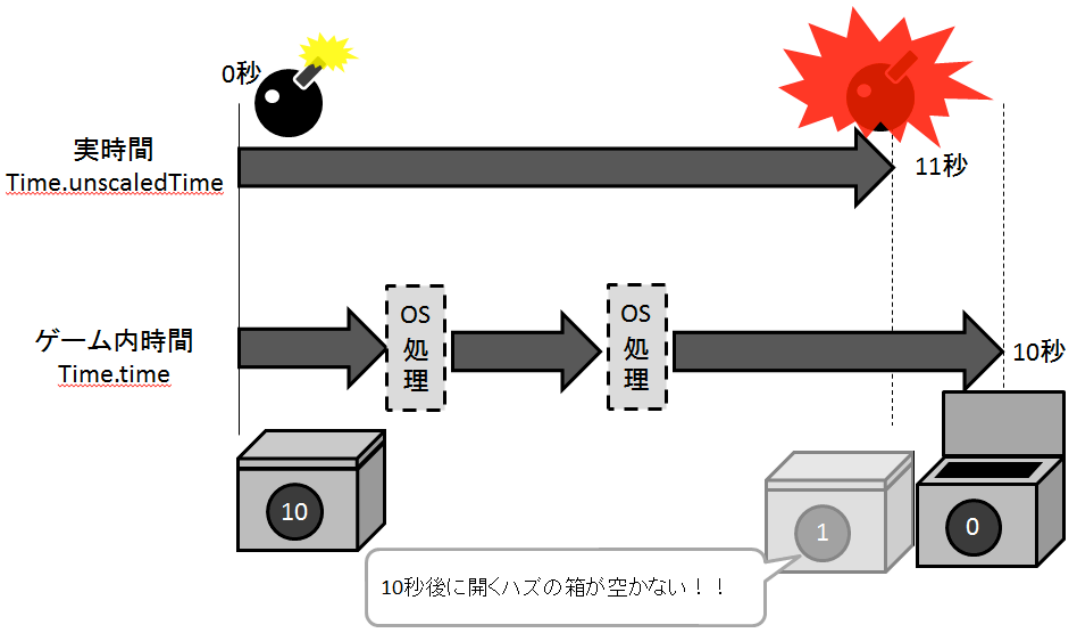


図 Q002_01_002 「実時間」と「ゲーム内時間」で発生するズレによるクリアできないパズル

基本的に Unity でゲームプログラムをする場合は、「ゲーム内時間」で計測し処理する必要があるということを忘れないでください。

Time.time と Time.fixedTime の違い

さて、Unity では「実時間」と「ゲーム内時間」がズレることについて説明しました。
この「ゲーム内時間」は、Time.time と Time.fixedTime で取得できます。
まず、Time.time ですが、ゲームを起動してからのゲーム内時間を測定しています。また、前の Update フレームからの経過時間は、Time.deltaTime で計測できます。そのため、Update 内で、ゲームオブジェクトを Transform.position を操作して移動する場合は、速度 * Time.deltaTime（経過時間）で計算します（図 Q002_02_001）。

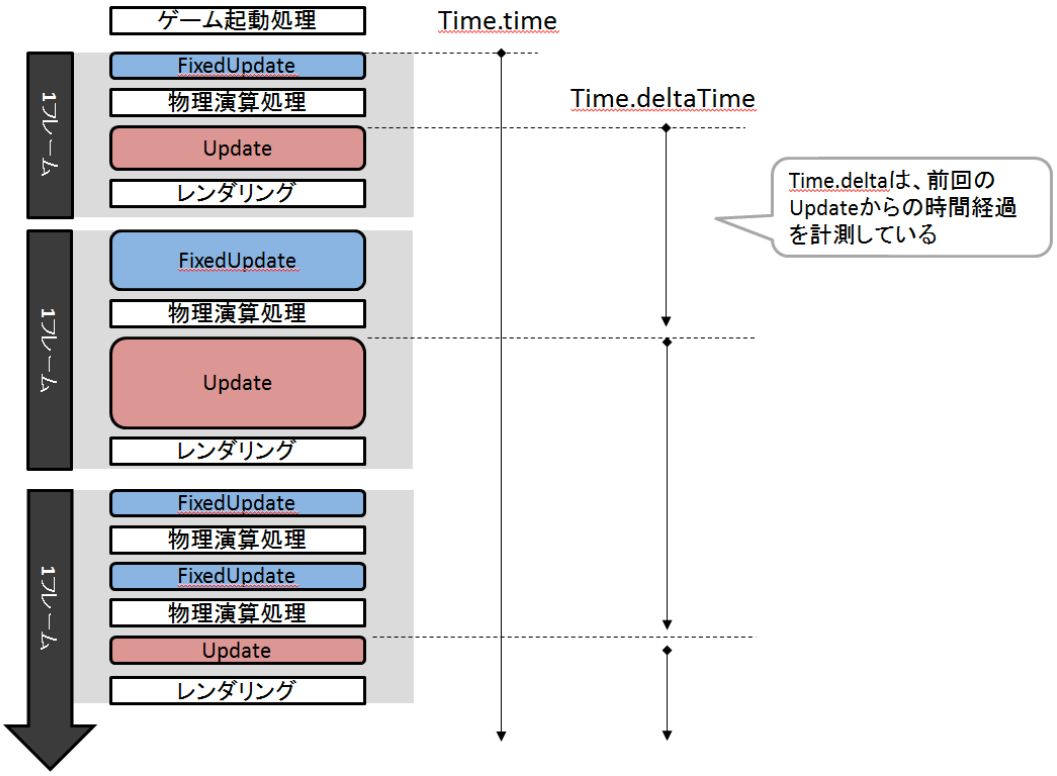


図 Q002_02_001 Time.time と Time.fixedTime の関係

一方、Time.fixedTime は、物理演算処理をした時間の合計を表現しています。

物理演算の処理タイミングは、「Edit」メニューの「Project Settings」から「Time」を選択して表示される TimeManeger の「Fixed Timesetep」プロパティで設定できます（図 Q002_02_002）。

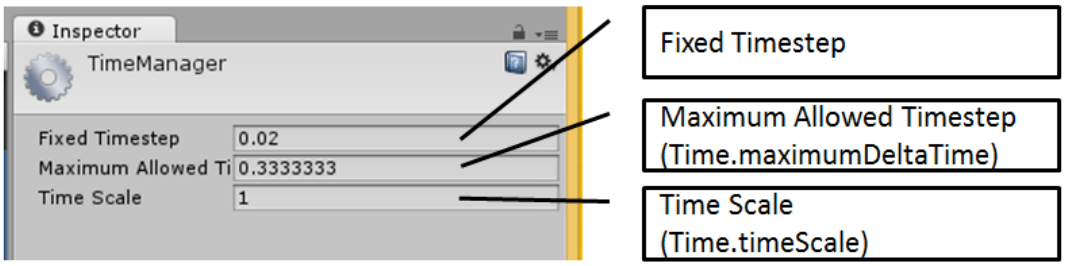


図 Q002_02_002 TimeManeger の設定

Unity の初期設定では、描画のフレームレートは可変フレーム、物理エンジンの動作タイミング（Fixed Timesetep）は 0.02 秒(1/50 フレーム)で設定されています。なお、この Fixed Timesetep は、Time クラスの Time.fixedDeltaTime をインスペクタに表示したものです。

物理演算は、この Time.fixedDeltaTime で指定された時間で処理が行われるため、1 フレーム内の処理が Time.fixedDeltaTime で設定したの時間内に収まったら、物理演算処理はスキップされます。基本的に、ゲーム実行開始から、処理落ちが一切発生しない場合は、Time.time と Time.fixedTime の値は同じになります（図 Q002_02_003）。

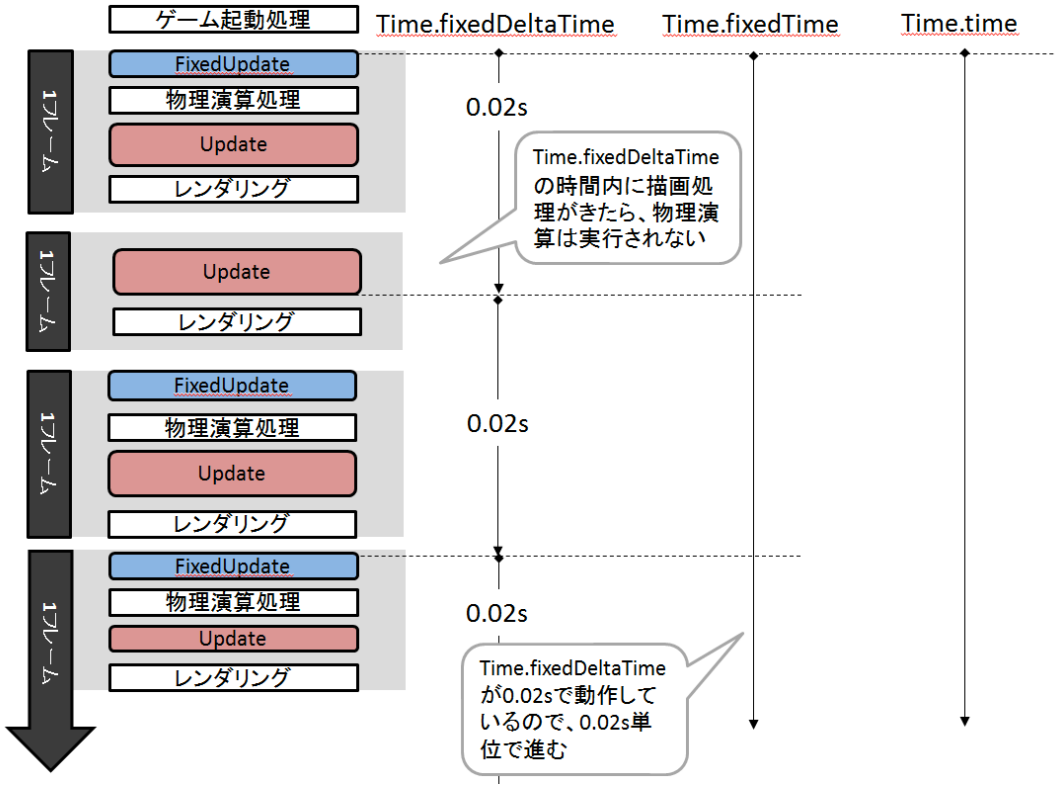


図 Q002_02_003 Time.fixedDeltaTime と Time.fixedTime と Time.time の関係

逆に、1 フレーム内の処理が Time.fixedDeltaTime で設定した時間を超えた場合（処理落ちした場合）は、遅延した時間分の処理を取り戻そうと、送れた分だけ繰り返し演算処理が実行されます。このとき、Time.time は処理落ちした時間も含めて経過時間が計算されるのですが、Time.fixedTime は Time.fixedDeltaTime を加算して計算を行っているため、Time.time と Time.fixedTime で時間のズレが発生していしまいます（図 Q002_02_004）。

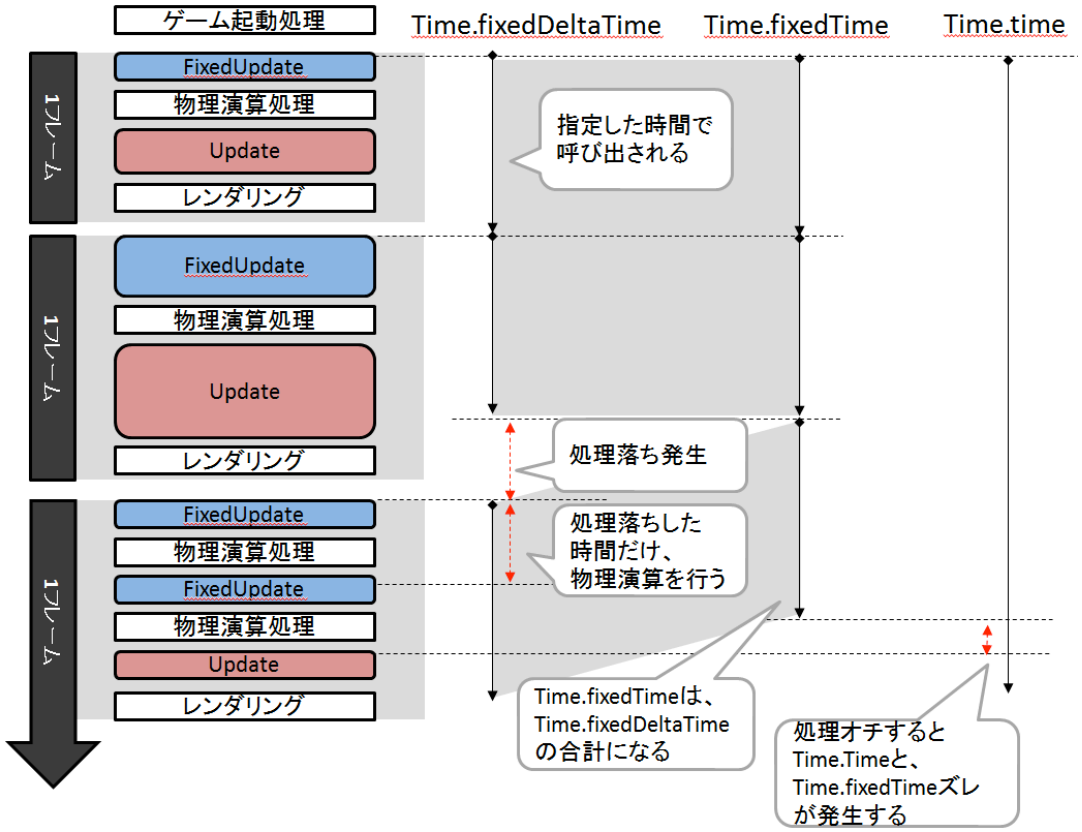


図 Q002_02_004 処理落ちによる Time.time と Time.fixedTime のズレ

これは、Physics3D（PhysX）、Physics2D（Box2D）に関係なく、処理落ちに合わせて自動でフレームレートを最適化するように作られているためです。ただし、長時間に渡り処理落ちが発生すると、それを次フレームで回復しようとすると、「処理落ち」「処理落ちの回復で、さらに処理落ち」「その処理落ちの回復で、さらに処理落ち」という負の連鎖が繰り返されることになります。そこで、Unity では「Maxmium Allowed Timestep」プロパティで、処理落ちを許容できる時間を設定します。これは、Time クラスの Time.maximumDeltaTime をインスペクタに表示したものです。

このプロパティは、指定された時間内に物理演算が終了しなかった場合、そこで処理を中断して、物理演算処理が遅れた時間の結果を TimeManager に反映させません。つまり、物理演算のゲーム内時間を遅らせることで、次のフレームで発生する遅延解消処理を無効にして、リセットしているわけです。これにより、物理エンジン側が長時間遅延した場合でも、処理落ちが回復可能な時間で次のフレームで物理演算が行われます。

なお、Time.maximumDeltaTime の初期値は 0.3333333 です。また、推奨値は 1/10（0.6）～1/3（0.2）秒です（図 Q002_02_005）。

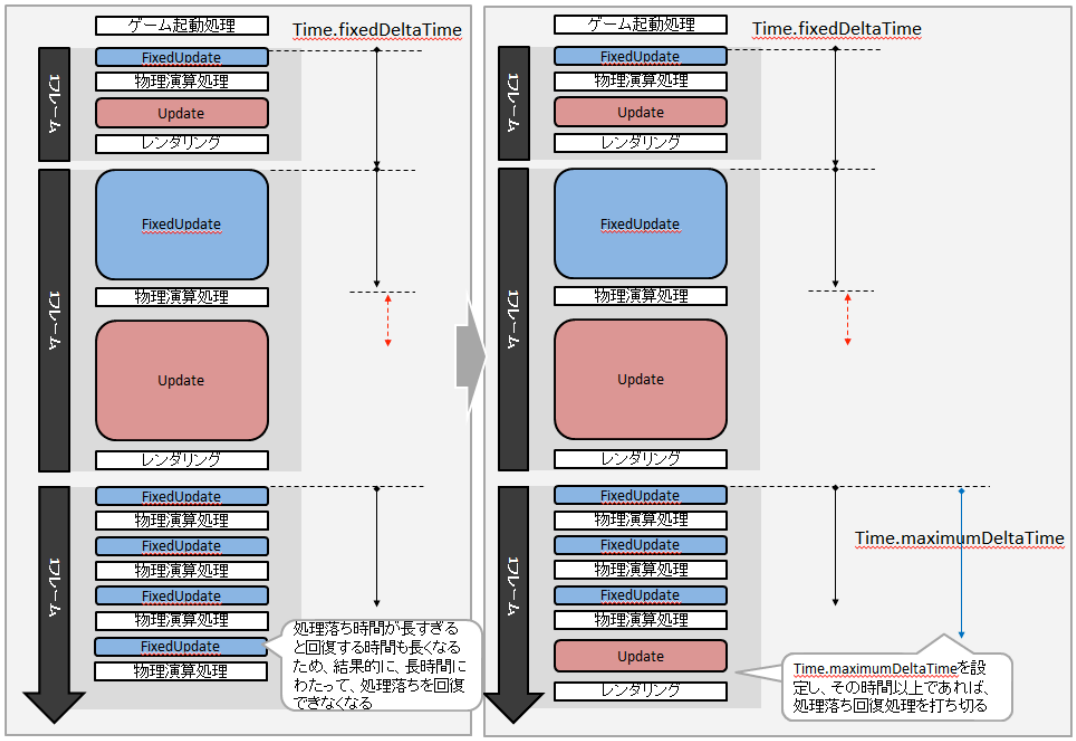


図 Q002_02_005 Time.maximumDeltaTime の仕組み

当然ながら、このような長時間の処理オチが発生した場合も、Time.time と Time.fixedTime の時間がズレます。

このような Unity の物理演算の仕組みのため、「ゲーム内時間」を表す Time.time と Time.fixedTime は、同じ値にはならないことを知って置いてください。そして、移動計算などを行う場合は、Update 内では Time.deltaTime を、FixedUpdate 内では Time.deltaFixedTime を利用することを守ってください。

処理落ちによって発生する物理演算処理の計算誤差の問題

さて、Unity では「実時間」と「ゲーム内時間」、そして、ゲーム内時間でも Time.time と Time.fixed がズレることについて説明しました。これらの問題を把握していないと、特に物理エンジンを使ったパズルゲームを作る場合に、解けないパズルができてしまうなど根深い問題を抱えることになります。

実は、物理エンジンには、もう一つ大きな問題があります。それは、物理エンジンが処理を行う間隔や処理時間などの負荷の違いによって、物理演算の結果が異なってしまうという問題です。例えば、NinjaSlaherX では、PC とスマートフォンで落石ステージの岩の転がり方の結果が変わります。これは、PC とスマートフォンの処理速度が違うためです。物理エンジン内で使われている float 計算の誤差をはじめ、いろいろ原因が考えられます(Unity では、あまりにも物理エンジンの処理負荷が大きいと、物理計算を途中で打ち切る仕

様ではないかという話もあります）。

著者の経験では、この問題を回避するには、描画のフレームレートと、物理エンジンの処理タイミングを安定（固定）させることで、プラットフォームごとの結果の誤差を最小にすることができます（100%ではありません）。

フレームレートを固定する

では、Unity で描画のフレームレートを固定する方法から紹介しましょう。

Unity では、初期設定の状態では、描画方法は可変フレームになっており、描画のタイミングは一定ではありません。そのため、Update が呼び出されるタイミングも処理負荷によって変化します。シューティングゲームのような場合、この可変フレームで作ってしまうと、処理速度が落ちた場合に、最悪、弾がワープしてしまいます（図 Q002_04_001）。

図 Q002_04_001 処理落ちとフレーム跳び

そこで、Unity のフレームレートを固定します。

Unity でフレームレートを固定する場合は、「Vsync 同期をオフにする」「Application.targetFrameRate にフレーム数を設定する」の 2 つの設定が必用です。

まず「Vsync 同期をオフにする」ですが、「Edit」メニューの「Project Settings」から「Quality Settings」を選択して、インスペクタに Quality Settings のプロパティを表示します。クオリティの一覧と、設定できるプロパティが表示されるので、全部のクオリティから「VSync Count」が「Don't Sync」になるように設定してください（図 Q002_04_002）。

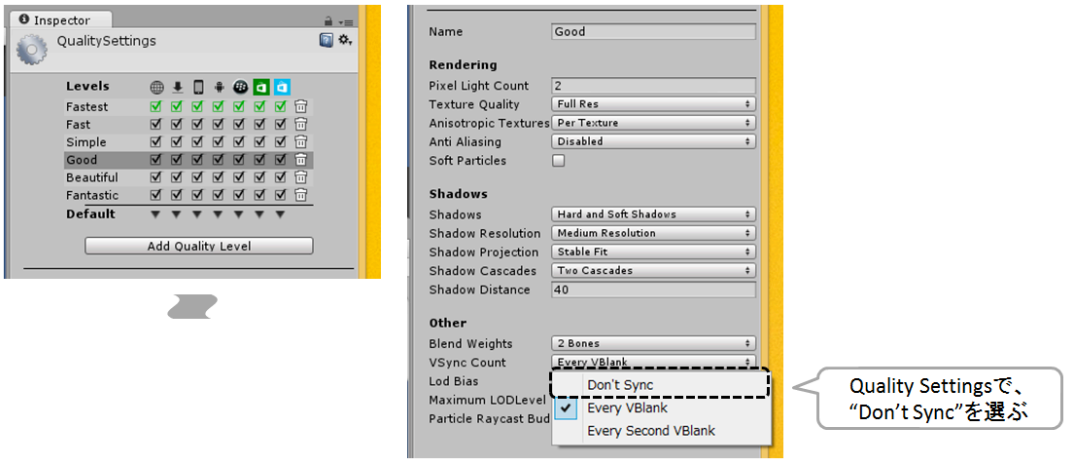


図 Q002_04_002 Vsync 同期設定

次に、Unity のフレームレートは、Application クラスの targetFrame で設定できます。初期値は-1（可変フレーム）です。これを 60 と指定することで描画が 60 フレームに固定されます。30 に設定すれば、30 フレームです。

なお、Vsync 設定はスクリプトから QualitySettings.vSyncCount で設定することも可能です。なので、ゲーム起動時に次のように指定すれば、Unity エディタから設定をする必要はありません（ソース 3.2.4.2）。

ソース 3.2.4.2：

```
QualitySettings.vSyncCount = 0;  // Quality Settings->VsyncCount->Don't Sync
Application.targetFrameRate = 60;// FrameRate
```

なお、VSync の同期をオフにすると、ジッターが発生するという問題があります。

VSync とは、ディスプレイが 1 回画面を描画するごとに出される信号です。古くはブラウン管テレビにおいて、表示がおかしくならないように出されていた信号なのですが、現在の液晶ディスプレイなどでも利用されています。

そのため、VSync の同期をオフにすると、描画開始タイミングを同期させられないため、画面の書き換えが見えるために発生するノイズ「ジッター（ティアリング）」が見えてしまいます（図 Q002_04_003）。

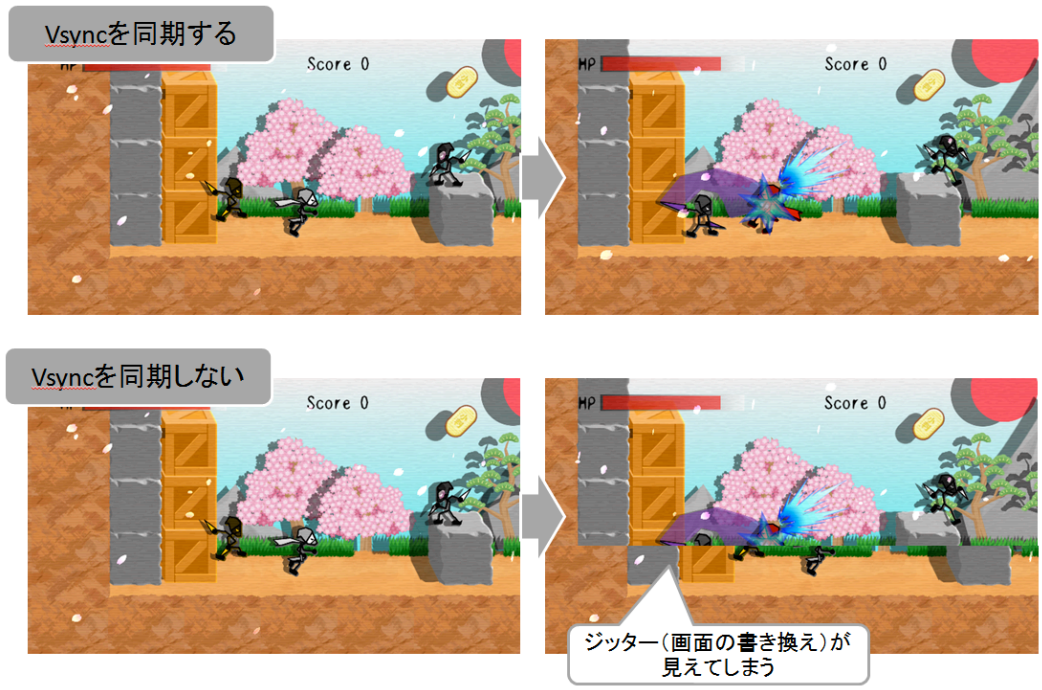


図 Q002_04_003 Vsync 非同期によるジッター発生の問題

VSync はディスプレイごとに異なり、60fps のものもあれば 120fps のものもあります。また、60 の倍数ではない 50fps や 75fps の周期で VSync を発行するディスプレイもあります（最近話題の Oculus Rift DK2 は 75fps です）。

このように、固定フレームにした場合は、表示するディスプレイが設定したフレームレートに対応していないと、ジッターが発生する可能性があることを覚えておいてください。

物理エンジンの処理タイミングを固定する

次に、実際に移動処理などを行う物理エンジンの実行間隔を決定します。FixedUpdate が呼ばれる間隔ですね。

こちらは、「Edit」メニューの「Project Settings」から「Time」を選択して表示される TimeManeger から設定できます（図 Q002_05_001）。

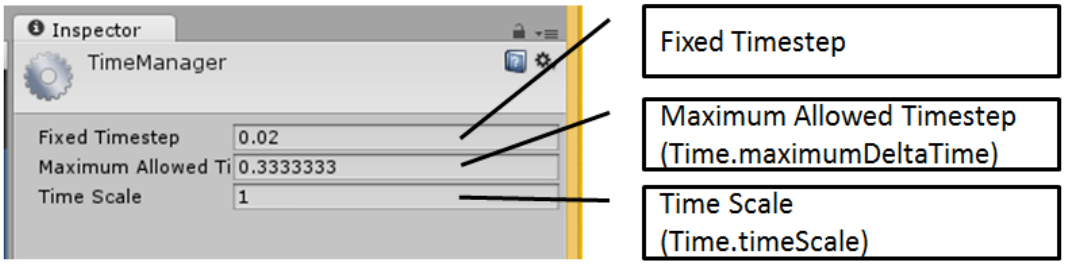


図 Q002_05_001 TimeManeger の設定

「Fixed Timestep」は、物理演算および FixedUpdate が実行される間隔です。

1.0 で 1 秒をとして設定します。初期値は 0.02 なので、初期状態だと 50 フレームで動作していることになります。60 フレームで動かす場合は、0.0166666 を設定します。これで、Update メッセージ（描画処理）と FixedUpdate メッセージ（物理演算処理）の処理が理論上は一致します。ただし、間隔を短く設定すればするほど、当然ながら 1 フレームあたりの処理負荷は重たくなります。固定フレームで処理落ちした場合、1 フレームを超えることが 1 秒続けば、1/30 フレームになります。

次に、前に説明した「Maxmium Allowed Timestep」を設定して、物理演算の間隔を固定します。例えば、物理演算の処理タイミングを 50 フレームで固定化したい場合は、「Maxmium Allowed Timestep」を「Fixed Timestep」と同じ 0.02 にします。この設定は、初期設定としてスクリプトに記述することも可能です（ソース 3.2.5.1）。

ソース 3.2.5.1：

```
Time.fixedDeltaTime          = 0.02f;  // FixedUpdate Interval Time
Time.maximumDeltaTime        = 0.02f;  // Physics And FixedUpdate Maximum Delta Time
```

描画も物理演算処理も、すべて 60 フレームの固定フレームで動作させるなら、ソース 3.2.5.2 のようになります（もちろん処理が重たければ処理落ちします）。

ソース 3.2.5.2：

```
QualitySettings.vSyncCount = 0;        // Quality Settings->VsyncCount->Don't Sync
Application.targetFrameRate = 60;      // FrameRate

Time.fixedDeltaTime          = 0.01666667f;     // FixedUpdate Interval Time
Time.maximumDeltaTime        = 0.01666667f;     // Physics And FixedUpdate Maximum Delta Time
Time.timeScale = 1.0f;
```

これで、可変フレームよりも物理演算の結果の違いが少なくなります。

ただし、可変フレームと違い処理が間に合わないプラットフォームで動作させた場合は、

極端に処理オチが目立つので注意してください。どんな遅いプラットフォームでも、動作させたいのであれば、60 フレーム固定ではなく、30 フレーム固定で作ることなども検討してみてください。

Time.captureFramerate で処理タイミングを固定する

もう一つ、Time.captureFramerate を利用した処理タイミング固定する方法を紹介しましょう。Time.captureFramerate は、Application.CaptureScreenshot 関数を利用した画面キャプチャ用に作られたプロパティの機能です。簡単に、固定フレームレートのゲームを作りたい場合は、この Time.captureFramerate を利用することもできます。

ゲームを 60FPS 固定で動作させたい場合は、Time.captureFramerate に 60 を設定します。これにより、Time.deltaTime が常に 0.1666667 に固定され、実時間に関係なく 60FPS で描画と物理演算などの処理が動作します（当然、処理が間に合わなければ処理落ちします）。

なお、Time.captureFramerate の値を変更しても、Time.fixedDeltaTime の値は変わりません。Time.fixedDeltaTime を初期値の 0.02 で設定し、Time.captureFramerate を 60 に設定しても、Time.fixedDeltaTime の値は 0.1666667 ではなく 0.02 になります。

また、Time.captureFramerate は、Awake メッセージのタイミングで実行しても設定がゲームに反映されません。Start メッセージ以降の処理で設定する必要があります。

Time クラスのプロパティ

これまで説明してきた Time クラスのプロパティをまとめて紹介します（表 3.2.7.1）。

表 3.2.7.1 Time クラスのプロパティ

プロパティ名	説明
maximumDeltaTime	物理エンジンが遅延した場合に許容する最大遅延時間。 推奨値は 1/10（0.6）～1/3（0.2）秒。 Unity エディタの TimeManager に表示される「Maxmium Allowed Timestep」と同じ
timeScale	Unity エンジンの TimeManager が管理しているゲーム内時間のタイムスケール値。1.0 で 1 倍、2.0 にすれば 2 倍になる。 Unity エディタの TimeManager に表示される「Time Sclae」と同じ
time	ゲームを起動してからの実時間
deltaTime	前フレームの Update メッセージ実行からの経過時間（実時間） ただし、どんなに処理が遅延しても maximumDeltaTime の値以上にはならない。
smoothDeltaTime	deltaTime をスムース化した値

fixedTime	ゲームを起動してからのゲーム内時間
fixedDeltaTime	FixedUpdate メッセージ実行のインターバル時間。 1.0 で 1 秒。0.02 を指定すれば 50 フレームで動作することになる。値が小さいほど物理エンジンの計算分解能が上がり物理シミュレーションも精度が上がるが、処理速度は遅くなる。 Unity エディタの TimeManager に表示される「Maxmium Allowed Timestep」と同じ
unscaledTime	ゲームを起動してからの実時間 timeScale で変更したタイムスケールの影響を受けない。
unscaledDeltaTime	前フレームの Update メッセージ実行からの経過時間（実時間） timeScale で変更したタイムスケールの影響を受けない。
realtimeSinceStartup	ゲームを起動してからの実時間。ゲームアプリを休止した場合などの時間経過も含まれる（逆に、このプロパティ以外の時間プロパティはゲームアプリ休止中の経過時間は含まれない）。 timeScale で変更したタイムスケールの影響を受けない。
timeSinceLevelLoad	最後のシーンが読み込まれてからの経過時間（実時間）
frameCount	ゲームを起動してからのフレーム数
captureFramerate	0 以上の値を設定すると、指定したフレームレートで強制的に実行される。例えば 60 を設定した場合、Time.deltaTime が常に 0.1666667 に固定され、60FPS で描画と物理演算などの処理が動作する。 Application.CaptureScreenshot 関数を使って画面キャプチャを行う場合に利用する。ただし、Awake メッセージのタイミングで実行しても設定されない。Start メッセージ以降の処理で設定する必要がある。

「処理速度」の固定の罠

さて、処理速度を固定したい場合、いくつか注意しなければならないことがあります。

一つは、フレームレートや物理エンジンの処理タイミングを固定化しても、物理演算の誤差を完全になくすのは難しいということです。そのため、物理エンジンのゲームを作る場合は、少々の誤差でもクリアできるステージを作ることが重要となります。

もう一つの注意点として、利用しているクラスやアセットで使われている「時間スケール」が、実時間なのかゲーム内時間なのかということです。例えば、便利な Tween アセットとして多くの人達に愛用されている iTween は、そのスクリプト内で、時間待ち処理にコルーチンの WaitForSeconds を使用しています。さて、WaitForSeconds は、実時間とゲーム内時間のどちらで計測しているのでしょうか？　筆者が調べた結果では、WaitForSeconds は Time.time で計測されているようです。つまり、何か Time.fixedTime で計算させている処理と連動させようとすると、時間的にズレが発生しても良い処理にしなければなりません。

このように、自分が作成したプログラム以外でも、そのプログラムがどのような時間処理で動作しているか注意が必要です。

３.３.　メモリとデータロード

最後にメモリとデータロードについて解説します。

ゲームを１つのパッケージとして作成する場合、特にスマートフォンでは、多くの機種で動作するようにメモリサイズに気を配る必要があります。また、メモリに常に置くことができないデータについては、リソースファイルとして外部からロードする必要もあるでしょう。このようなメモリ管理のちょっとしたテクニックついてご紹介します。

メモリ情報の表示

まず、ゲーム実行中のメモリ情報についてですが、Unity エディタの場合、Game ビューの Statstics ウィンドウで確認することができます。また、Unity Pro 版であれば、Profiler ビューで確認可能です（図 Q003_03_001）。

図 Q003_03_001 Profiler ビューでのメモリ情報表示

スクリプトからもメモリ情報は取得できます（ソース 3.3.1.1）。

ソース 3.3.1.1 :

```
Debug.Log(string.Format("SystemInfo.systemMemorySize : {0} MByte",SystemInfo.systemMemorySize));
Debug.Log(string.Format("Profiler.usedHeapSize : {0} MByte",Profiler.usedHeapSi
```

```
ze / (1024 * 1024)));
   Debug.Log(string.Format("System.GC.GetTotalMemory(false) : {0} MByte",System.
GC.GetTotalMemory(false) / (1024 * 1024)));
```

実際に各プラットフォームでゲームを作る場合は、これらの機能を使ってゲーム中のメモリ使用状況を把握しておくと良いでしょう。特に、デバッグ用にリアルタイムにゲーム画面にメモリ情報を表示できるモードを作っておくと便利です。

Unity のシーン遷移におけるメモリ管理

Unity ではシーンで読み込むアセットを自動的に管理しています。ありがたいことに、シーンデータから必要なデータのみを検索して、メモリ使用量が最小限になるように実装されています（図 Q003_02_001）。

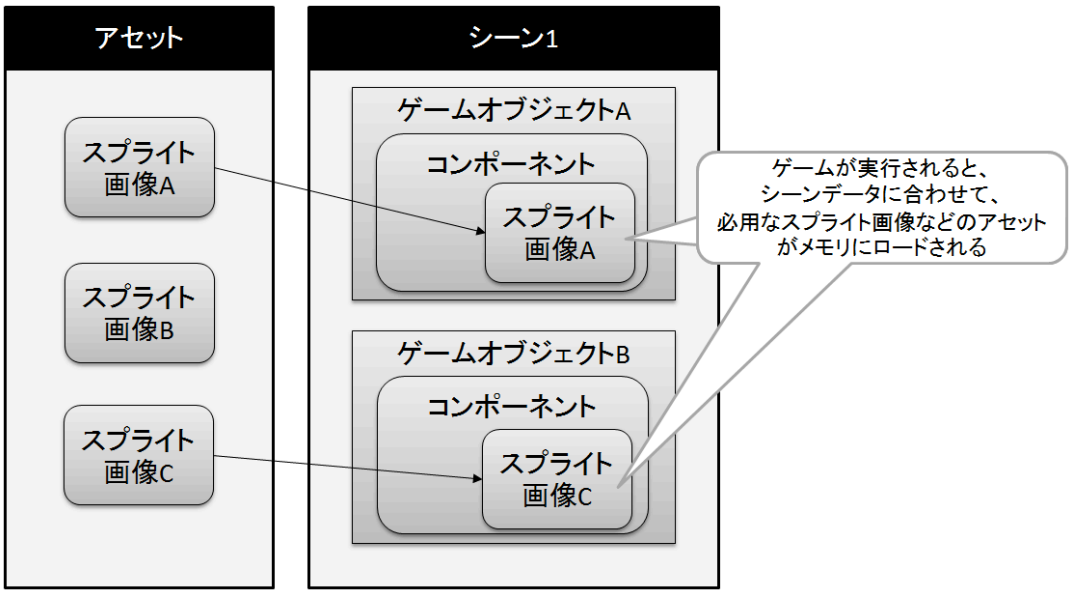


図 Q003_02_001 シーンロード時のアセットの読み込み

また、シーンを遷移した場合、必要なアセットデータのみがメモリ内に残るようになっています（図 Q003_02_002）。

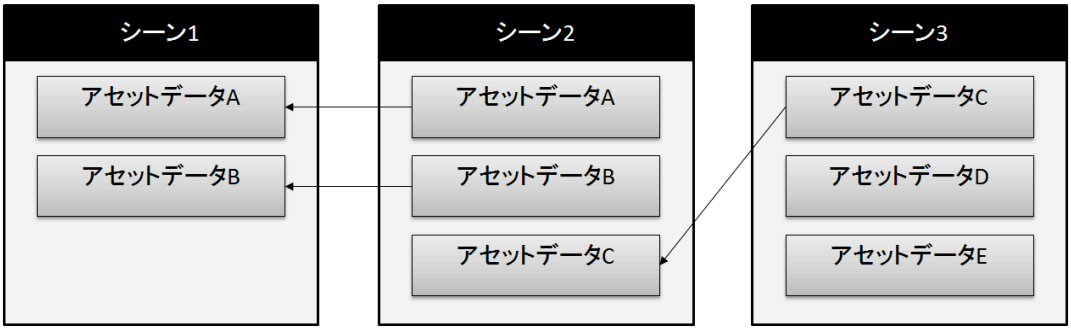


図 Q003_02_002 シーン遷移に伴うアセットデータのロードの仕組み

ただし、シーン遷移で不要になったアセットがメモリから解放されるのは、次のシーンをロードしてからです。そのため、シーン 2 からシーン 3 へと遷移する場合、瞬間的にメモリはアセット 2＋3 分だけ必要になります（図 Q003_02_003）。

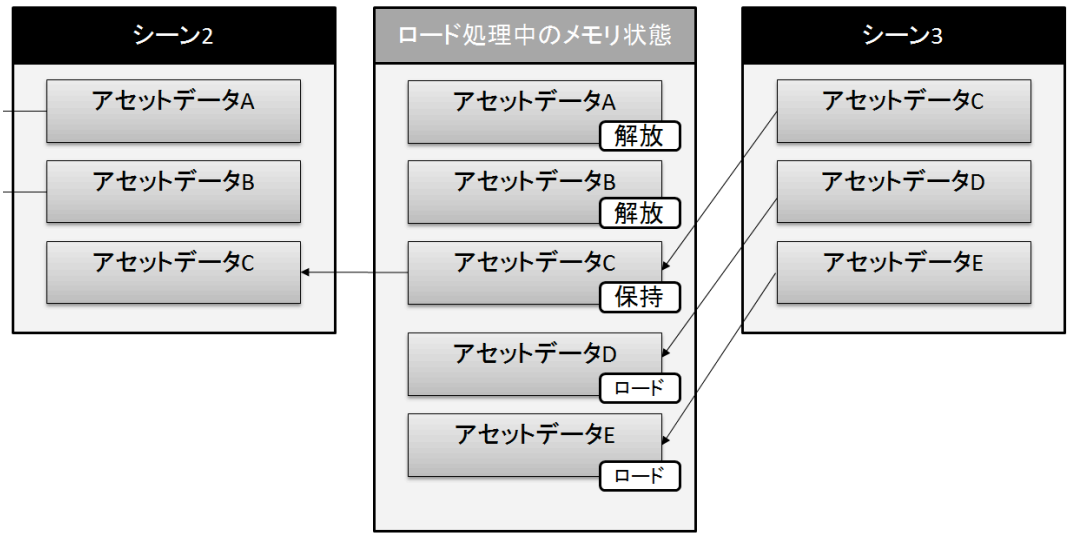


図 Q003_02_003 シーン遷移時に発生する最大メモリ消費の状態

この問題は、Unity ではじめてゲームを作る場合に、よく引っかかる問題ですので、覚えておいてください。なお、詳しくは、下記の Unite Japan 2013 のビデオで解説されていますので、興味のある方はぜひみてください。

[Unite Japan 2013] シーン／メモリ／アセットバンドル
http://vimeopro.com/unity3djp/unite-japan-2013-for-the-public/video/64375290

リソースファイルからのデータロード

さて、実際にゲームを開発して、メモリが足りなくなってきたら、データの管理方法について再検討する必要があります。解決方法の一つとして、Resourece フォルダを使ったリソースファイル機能の利用があります。

この機能では、各データを Resourece フォルダにインポートすることで、任意のタイミングでデータをロード、そして破棄することができます。場面によっては必要のないデータをメモリから削除できるのです。

Resourece クラスの関数は次の通りです（表 3.3.3.1）。

表 3.3.3.1 Resourece コンポーネントのプロパティ

関数名	説明
FindObjectsOfTypeAll	指定したタイプのオブジェクトを Resourece フォルダから検索して Object の配列で返す
Load	アセットのファイル名を指定してシーンにロードする
LoadAll	指定したフォルダ名の中のアセットをすべてロードする Resourece.LoadAll.<Texture2D>(“Textures”);のように、オブジェクトの型指定をしてロードすることもできる
LoadAsync	非同期で指定したファイル名のアセットを読み込む関数。ResourceRequest を返す。データがロードできたら、ResourceRequest の asset にオブジェクトが設定される
UnloadAsset	指定した Resourece フォルダからロードしたオブジェクトを解放する
UnloadUnusedAssets	Resourece フォルダからロードしたオブジェクトで、使用されていないものを解放する
LoadAssetsAtPath	エディタ拡張用の関数。フルパス名で指定して、アセットをロードする

必要がなくなれば解放メソッドを呼び出して、このアセットデータを解放することができます。

なお、このリソースファイルからのデータロードは、レガシーな機能となりつつあります。公式の Unity では、次に紹介するアセットバンドルの利用を薦めていますが、Unity PRO 版の機能であるため、UnityFree 版を利用している方はリソースファイルの仕組を使うのが良いでしょう。

さて、iPhone や Android の場合、通信量の制限からアプリの大きさに制限があります。

この制限を超えた場合、Wifi のみでしたアプリをダウンロードできなくなります。ユーザーが「ほしい！」と思ったときに、Wifi 環境のある家に帰らないとダウンロードして遊ぶことができないわけです。

そこで、アプリのデータファイルなどを分割して、アプリサイズを小さくする必要があります。resourece フォルダを活用すれば良いのでは？と思われるかもしれませんが、resourece フォルダはローカルで読み書きするための機能で、ビルド時にはパックされてアプリの中に結合されます。

これを解決してくれるのが、Unity PRO の機能である「アセットバンドル」です。

アセットバンドルとは、ローカルからもネットにあるサーバーからもロード可能なアセットファイルの機能です。アセットバンドルでは、ゲーム実行後に読み込むアセットファイルを、任意のスクリプトとして記述します。そして、ビルド後に、アセットバンドルをサーバーに設置して、ゲームアプリからダウンロードして使用します。

アセットバンドルは Unity PRO の機能なので、ちょっとハードルが高いですが、アプリサイズが巨大になった場合は、Unity PRO の購入と合わせてアセットバンドルの使用も検討してみてください。

３.4. ターゲットプラットフォームごとの問題

最後に、ターゲットプラットフォームの違いによって引き起こされる問題について紹介します。

iPhone（iOS）

POLYGONCOLLIDER2D が動的作成できない

iOS では、PolygonCollider2D を、描画領域を参照しながら動的に作成できません（スプライトの大きさの矩形領域に簡易的に変換させているようです）。
これを回避するためには、表示するスプライト画像のインポータ設定を下記のようにします。

1.Texture Type プロパティを「Sprite(2D / uGUI)」から「Advanced」に変更
2.Read/Write Enabled のチェックをオンにする

ただし、処理速度は低下しますので注意してください。

Windows,Mac

パソコンの場合、パソコンごとの GPU の違いによって、必ず同じ表示ができるとは限りません。3D ゲームならシャドーマップが表示されないことなどもあります。また、同じ GPU でも、Windows と Mac ではドライバの作りが違うため、思ったように同じ表示結果が得られないことがあります。2D の場合は、3D ほど互換性に問題があるとは思われませんが、オリジナルシェーダーなどを作成して 2D エフェクトなどを作成する場合は、Windows,Mac 以外にも、メジャーな GPU ごとにチェックができると、多くのマシンで同一の表示結果が得られるゲームを作ることができるでしょう。

第４章　巻末

4.1. Unity のその他の情報

アップデート

Unity のアップデートは基本的に、Download ページから取得できます。なお、もっとも最初に公開されるのは、Unity の英語ページです。

Unity ダウンロードページ
http://unity3d.com/unity/download
Unity Japan ダウンロードページ
http://japan.unity3d.com/unity/download/

また、現在では、週単位で配信されている「パッチリリース」と呼ばれるアップデートが行われています。パッチリリースは、海外の Unity フォーラムで公開されています。

Unity Patch Releases
http://unity3d.com/unity/qa/patch-releases

安定度を優先したい場合は、ダウンロードページで配布されているバージョンがお勧めですが、特定のバグをフィックスしたバージョンがすぐに必用であれば、パッチリリースを利用することをお勧めします。

4.2. Unity のショートカットリスト

Unity はショートカットを使うと効率よく作業が行えます（表 3.1.1.1）。

表 3.1.1.1 Unity エディタのショートカット

ジャンル	プロパティ名	説明
File	**CTRL+N**	New
	CTRL+O	Open
	CTRL+S	Save
	CTRL+SHIFT+S	Save Scene as
	CTRL+SHIFT+B	Build
	CTRL+B	Build and run
Edit	**CTRL+Z**	Undo
	CTRL+Y	Redo
	CTRL+X	Cut
	CTRL+C	Copy
	CTRL+V	Paste
	CTRL+D	Duplicate
	SHIFT+Del	Delete
	F	Frame (Center) selection
	CTRL+F	Find
	CTRL+A	Select All
	CTRL+P	Play
	CTRL+SHIFT+P	Pause
	CTRL+ALT+P	Step
Assets	**CTRL+R**	Refresh
Game Object	**CTRL+SHIFT+N**	New game object
	CTRL+ALT+F	Move to view
	CTRL+SHIFT+F	Align with view
Window	**CTRL+1**	Scene
	CTRL+2	Game
	CTRL+3	Inspector
	CTRL+4	Hierarchy
	CTRL+5	Project

	CTRL+6	Animation
	CTRL+7	Profiler
	CTRL+9	Asset store
	CTRL+0	Asset server
	CTRL+SHIFT+C	Console
	CTRL+TAB	Next Window
	CTRL+SHIFT+T AB	Previous Window
	ALT+F4	Quit
Tools	**Q**	Pan
	W	Move
	E	Rotate
	R	Scale
	Z	Pivot Mode toggle
	X	Pivot Rotation toggle
	CTRL+ Left Mouse Butt on	Snap
	V	Vertex Snap
Selection	**CTRL+SHIFT+1**	Load Selection 1
	CTRL+SHIFT+2	Load Selection 2
	CTRL+SHIFT+3	Load Selection 3
	CTRL+SHIFT+4	Load Selection 4
	CTRL+SHIFT+5	Load Selection 5
	CTRL+SHIFT+6	Load Selection 6
	CTRL+SHIFT+7	Load Selection 7
	CTRL+SHIFT+8	Load Selection 8
	CTRL+SHIFT+9	Load Selection 9
	CTRL+ALT+1	Save Selection 1
	CTRL+ALT +2	Save Selection 2
	CTRL+ALT +3	Save Selection 3
	CTRL+ALT +4	Save Selection 4
	CTRL+ALT +5	Save Selection 5
	CTRL+ALT +6	Save Selection 6
	CTRL+ALT +7	Save Selection 7
	CTRL+ALT +8	Save Selection 8
	CTRL+ALT +9	Save Selection 9
Unity(Mac の	**CMD+ ,**	Preference

み)

CMD+H	Hide Unity
CMD+H	Hide others
CMD+Q	Quit Unity

※ Mac の場合は、CTRL を CMD に置き換えてください。

４.３.　もっと凄いゲームを作りたい方は……

これで本書の Unity2D の紹介は終わりです。
今後、さらにプロも顔負けのゲームを作りたいと思ったら、実際に市販のゲームをプレイしてゲームの作り方を学んでみたり、Unity に関するネット情報などを集めてさらにレベルアップしましょう。
どこから手を付けて良いのか分からないと言う方のために、参考資料を用意しましたので、ご活用ください。

参考資料（ネット情報）

Unity について何か分からないことがあったとき、または、さらにレベルアップしたいと思ったときは、下記のサイトが手掛かりになるでしょう。

※【公式ページ】
Unity
http://unity3d.com/

Unity Japan
http://japan.unity3d.com/

※【ダウンロード】
Unity Download Page
（最新バージョンの Unity は Unity Japan よりもこちらの方が早く公開されます）
http://unity3d.com/unity/download

Unity Japan Download Page
http://japan.unity3d.com/unity/download/

UNITY-CHAN!
http://unity-chan.com/

※【リファレンス＆チュートリアル】
Unity Japan ドキュメントページ
http://japan.unity3d.com/developer/document/

Unity チュートリアル
http://japan.unity3d.com/developer/document/tutorial/

2D シューティングゲーム制作チュートリアル
http://japan.unity3d.com/developer/document/tutorial/2d-shooting-game/

※【相談ページ】
Unity Forums（Unity 公式のフォームラムです。英語でのやりとりがメインです）
http://forum.unity3d.com/forum.php

Facebook Unity ユーザー助け合い所
https://www.facebook.com/groups/unityuserj/

Unity 道しるべ
http://unity-michi.com/
Facebook Unity ユーザー助け合い所のまとめサイト

Stack Overflow
（海外のコンピュータ技術に関する情報掲示板。下記の URL は Unity で検索しています）
http://stackoverflow.com/search?q=Unity

※【Unity 情報が掲載されているブログ】
Unity Japan Officeal Blog
http://japan.unity3d.com/blog/

Unity Officeal Blog（本社のブログ。英語です）
http://blogs.unity3d.com/

テラシュールウェア
http://terasur.blog.fc2.com/

強火で進め
http://d.hatena.ne.jp/nakamura001/

新 masafumi's Diary
http://masafumi.cocolog-nifty.com/

Unity 学習帳

http://unitylab.wiki.fc2.com/

万年素人から Geek への道
http://d.hatena.ne.jp/shinriyo/

※【レベルアップの参考になるページ】
Qitta ： Unity 初心者がハマる 11 の「罠」～考察編
http://qiita.com/gamesonytablet/items/20b25ad9729e4a353c96

Unity 開発に関する 50 の Tips　～ベストプラクティス～（翻訳）
http://warapuri.tumblr.com/post/28972633000/unity-50-tips

※【アセット】
Unity Asset Store
https://www.assetstore.unity3d.com/

CRI ADX2 LE
http://www.adx2le.com/

Sprite Studio
http://www.webtech.co.jp/spritestudio/

※【その他】
Facebook Unity 職業安定所
https://www.facebook.com/groups/170236739727288/

参考資料（書籍）

さらに Unity やゲーム開発について知りたい場合に役に立つ書籍をご紹介します。

※【Unity 入門】
Unity4 入門 最新開発環境による簡単 3D ゲーム製作
浅野 祐一（著）荒川 巧也（著）森 信虎（著）SBC

ゲームの作り方 Unity で覚える遊びのアルゴリズム
加藤 政樹（著）SBC

※【リファレンス】
Unity ライブラリ辞典 ランタイム編
安藤 圭吾（著）カットシステム

Unity 4 ライブラリ辞典 エディタ編
安藤 圭吾（著）カットシステム

※【ゲームデザイン】
「レベルアップ」のゲームデザイン ―実戦で使えるゲーム作りのテクニック
Scott Rogers(著), 塩川 洋介(監訳)（翻訳), 佐藤 理絵子(翻訳) オライリージャパン

「タッチパネル」のゲームデザイン ―アプリやゲームをおもしろくするテクニック
Scott Rogers(著), 塩川 洋介(監訳)（翻訳), 佐藤 理絵子(翻訳) オライリージャパン

レベルデザイナーになる本 ー夢中にさせるゲームシーンを作成するー
Phil Co（著), 加藤 諒（編集), B スプラウト（翻訳） ボーンデジタル

参考資料（イベント）

ゲーム開発には、ネットや書籍だけでは得られない情報があります。Unity の最新の開発情報や、どんなゲームがヒットしやすいかなどです。

これらの情報は、イベントなどで手に入れることができます。

※【カンファレンス】

Unite Japan（Unity 主催による Unity の最新情報が公開されるカンファレンス）
http://japan.unity3d.com/unite/

Unite Japan 2013 の公開動画
http://vimeopro.com/unity3djp/unite-japan-2013-for-the-public

CEDEC 2014（国内最大のゲームカンファレンス）
http://cedec.cesa.or.jp/

Game Tools & Middleware Forum 2013（ゲームツールとミドルウェアのカンファレンス）
http://www.info-event.jp/gtmf2013/

※【インディーズイベント】

IGDA Japan （ゲーム開発者の勉強会を開催している NPO 団体）
http://www.igda.jp/

東京ロケテゲームショウ（IGDA 主催のインディーズ向けロケテイベント）
https://sites.google.com/site/locategameshow/home

BITSUMMIT 2014 -京都インディーゲームフェスティバル-
（インディーズを対象としたゲームフェスティバル）
http://www.pref.kyoto.jp/sangyo-sien/bitsummit.html

センス・オブ・ワンダーナイト 2013
（東京ゲームショーで行われているインディーズゲームの祭典）
http://expo.nikkeibp.co.jp/tgs/2013/business/sown/

ニコニコ自作ゲームフェス
（ニコニコ動画で行われているインディーズゲームフェスティバル）
http://ch.nicovideo.jp/indies-game

※【その他】

Global Game Jam Japn

（プロ・アマ問わず、ゲーム開発者が全世界のハッカソン会場で 48 時間でゲームを作るイベントです）

http://ggj.igda.jp/