

3D Platformer Tutorial

Building a 3D Platform Game in Unity 2.0

目次

1.はじめに	
学習出来ること	5
知つておくと良いこと	6
プロジェクト構成	6
ファイル	7
記述上の約束事	7
Unityでの約束事	8
プロジェクト	8
ゲームオブジェクト、コンポーネント、アセット、プレハブ	8
ゲームオブジェクト	8
コンポーネント	9
アセット	9
プレハブ	9
謝辞	10
2.はじめの一歩	
Lerpzを動かす	11
プロット	11
Lerpzを追加	12
Character ControllerとThird Person Controllerスクリプト	19
Lerpzを動かす	19
キャラクタアニメーション	20
アニメーションブレンディング	20
Third Person Player Animation スクリプト	20
ギズモ	21
ジェットパック	22
パーティクルシステムを追加	23
ライトの追加	26
プロブシャドウ	29
プロブシャドウの追加	30
新しいレイヤの作成	31
スクリプトのコンセプト	33
組織と構成	34
死と再生	36

Fallout Death スクリプト	37
再生ポイント	38
どのように動作するのか	40
3. シーンの設定	
はじめの一歩	42
アイテムの配置	43
体力アイテム	43
バリアフィールド	44
集得できるアイテムのスクリプト	45
ジャンプ台	48
4. GUI	
ユーザインターフェース	50
Unity2の新GUIシステム	50
詳しい情報	51
ゲーム内HUD	51
GUI Skinオブジェクト	52
スタートメニュー	56
シーンの設定	57
背景	58
ボタン	60
ゲームオーバー	64
5. 敵対者	
敵対者と衝突	69
レーザトラップ	69
レーザトラップの実装	70
Laser Trapスクリプト	73
概要	73
ロボットガード	75
探索と破壊	77
出現と最適化	79
どのように動くのか	80
6. オーディオと最終調整	
導入	83
オーディオ	83
サンプルノート	84
Lerpzの脱出にサウンドを追加！	84
環境サウンド	86
ジャンプ台	87
取得可能アイテム	88
バリアフェンス	90

プレイヤ	90
ロボットガード	94
シーン切り替え	96
バリアフェンスの解除	96
7. 最適化	
なぜ最適化するのか？	110
レンダリングのFPSをモニタする	110
状態ディスプレイの理解	111
レンダリングの最適化：2カメラシステム	112
8. 終わりに	
あまり探索されていない道	114
より良いゲームにするための提案	114
意図的なミスの修正	114
レベルの追加	115
敵の追加	115
得点の追加	115
ネットワークハイスクアシステムの追加	115
複数プレイヤサポート	115
より詳しく	115
9. スクリプト付録	
StartMenuGUI	116
GameOverGUI	117
GameOverScript	118
ThirdPersonStatus	118
LevelStatus	120
HandleSpaceshipCollision	122

はじめに

Unityは強力なゲーム開発ツールです。1人称視点シューティングからパズルまで様々なゲームに適用できます。



高さマップ地形、ネットワーキング、物理計算、スクリプトなど、数え切れないほどの機能により、Unityは、初めてのユーザを怖気づかせるかもしれません。しかし、これら多くのツールを使いこなすことには、無限の価値があります。

このチュートリアルは、3人称視点をもつ、完全な3Dプラットフォームゲームの開発プロセスを体験します。プレイヤ操作、衝突判定、スクリプト、プロブシェーダ、基本AI、ゲームHUD、シーン切り替え、スポットオーディオ効果が含まれます。

学習できること

このチュートリアルはUnityでのゲーム開発における技術的な面に注目しています。次のことが学べます：

- キャラクタの操作
- プロジェクタ
- オーディオリスナー、オーディオソース、オーディオクリップ
- 複数カメラ(を、どのように切り替えるか)
- UnityGUI スクリプトシステム
- 衝突物
- メッセージとイベント

- ライティング
- パーティクルシステム
- プロブシャドウ
- スクリプト (AI、ステートマシン、プレイヤ操作)

このチュートリアルでは、これらの機能がゲーム開発においてどのように使われるかを解説します。

知っておくと良いこと

このチュートリアルではスクリプトを広範囲で利用します。サポートされている言語のうちひとつに慣れていることが重要です： JavaScript、C#、Boo. (チュートリアルではJavaScriptを用います)

また、Unityのインターフェースを用いて基本的な操作が行えることを前提としています。Assetのシーン中での移動、コンポーネントのゲームオブジェクトへの追加、インスペクタでの属性の編集などです。

プロジェクト構成

UnityはプロジェクトのAssetを特定の方法で構成することは強制しません。しかし、Assetを種類ごとにフォルダに分けて並べるのがよいでしょう。Textures, Models, Sound effects.などです。Unityの技術において、小さいプロジェクトではこの方法が有効です。さらに複雑なプロジェクトでは、Assetを機能ごとに分けるとよいでしょう。Player, Enemies, Propsh, Scenery.などです。

このチュートリアルプロジェクトは何人かのチームメンバーにより行われました。そして、彼らの異なるスタイルを反映しながら進みました。筆者にとって興味深いことに、プロジェクト構成は“小さな”プロジェクト構成を反映する物となりました。

ゲームオブジェクトとコンポーネントの抽象化(Abstract GameObjects & Components)

UnityはそれぞれのシーンAssetを開発プロセスの中心に起きます。これにより、ゲーム開発をグラフィカルに行うことができ、多くの操作がドラッグアンドドロップで行われます。これは、レベルデザイン作業の大部分で顕著ですが、すべてのAssetがこの方法で表示できるわけではありません。いくつかのAssetはビジュアルというより、抽象的です。そのため、抽象的なアイコンやワイヤフレームギズモにより表されます。たとえば、オーディオソース(Audio Sources)やライト(Light)です。中には、まったく表示されないものもあります。スクリプトがそのひとつです。

スクリプトはUnityのシーンにおいて、Assetとゲームオブジェクトが、どのように相互作用するかを決めるものです。この相互作用はすべてのゲームの核となります。そのため、スクリプトの中に、コメントを残しておくことは、よい方法です。

このチュートリアルでは、記述されているスクリプトを読み、理解できることを前提としています。コメントがあちこちに書かれています。しかし、いくつもの重要な特徴的なスクリプト記述のテクニックやコンセプトは、詳細を説明します。

スクリプトは多数のコメントを使って書かれています。また、なるべくわかりやすいように設計され、記述されています。チュートリアルを読みながらスクリプトを読み進めてください。また、いろいろ書き直して実験してみるのもよいでしょう。

ファイル

最新のプロジェクトファイルは下記のURLからダウンロードできます：
<http://unity3d.com/support/resources/files/3DPlatformTutorialStart.zip>

Scenes フォルダには最終結果である次のシーンが含まれています：メインメニューシーン、ゲームオーバーシーン、ゲームのレベルが含まれたシーン

このチュートリアルは、オブジェクトのシーン内への配置など、Unityの基本的操作は理解していることを前提としています。そのため、シーンを開始すると、すでに、景観やものが配置され他状態になっています。

記述上の約束事

これは、多くの情報を含んだ、とても長いチュートリアルです。読みやすくするために、いくつかの約束事があります：

背景色と内容

このようなボックスに書かれたテキストには、メインのテキストを補助する、追加の情報が記述されています。

スクリプトコードは次のように書かれます：

```
// これはサンプルです
Function Update()
{
    DoSomething();
}
```

NOTE チュートリアルに含まれるスクリプトには、豊富なコメントが書かれており、読みやすいように設計されています。テキスト内では、スペースの都合により、コード断片からは除かれている場合もあります。

Unity上で何かのアクションを行わなければならない場合は、次のように書かれます：

-  クリックしてください
-  次はこれです。
-  次にPlayをクリックしてください

スクリプトの名前、Asset、メニューアイテム、インスペクタ属性は太字のテキストで書かれます。上に示された例のUpdate()のように、スクリプト関数とイベント名の記述には等幅フォントが使われます。

Unityでの約束事

Unityはユニークな開発システムです。他のシステムを使う多くの開発者は、コードエディタを使い、Assetをつかったりロードしたりするために、作業時間の90%をコードの編集に費やします。Unityは違います：コードではなくAssetを中心においているため、他の3Dモデリングツールと同様、Assetにフォーカスしています。そのため、Unityでの開発で、キーとなる約束事と専門用語を理解することは非常に重要です。

プロジェクト

Unityで作られるゲームはプロジェクトからなります。プロジェクトは、モデル、スクリプト、レベル、メニューなどをすべて含みます。通常ひとつのプロジェクトファイルがすべてのゲーム要素を含みます。Unity 2を使い始めて最初にすることは、プロジェクトファイルを開くことです。(インストールが終わったばかりであれば、Island Demoプロジェクトが開かれます)

シーン(Scenes)

プロジェクトはシーンと呼ばれるドキュメントを1つ、または、複数含みます。シーンは1つのゲームレベル、または、重要なユーザインタフェース要素を含みます。例えば、ゲームメニュー、ゲームオーバー画面、シーン切り替えなどです。複雑なゲームでは、シーン全体を初期化の用途のみに用いる場合もあるでしょう。つまり、ゲームの全てのレベルはシーンです。しかし、全てのシーンがゲームのレベルである必要はありません。

ゲームオブジェクト、コンポーネント、アセット、プレハブ (GameObjects, Components, Assets & Prefabs)

Unityを理解するには、ゲームオブジェクト(GameObject)とコンポーネント(Component)の関係を理解する必要があります。

ゲームオブジェクト(GameObjects)

ゲームオブジェクトはUnityの基本ブロックです。ゲームオブジェクトはコンポーネントと呼ばれる様々な機能のコンテナになります。コンポーネント以外を含む場合もあります。全てのゲームオブジェクトは、位置と向きを指定する、トランスフォームコンポーネントを持ちます。

ゲームオブジェクト階層(GameObject Hierarchies)

ゲームオブジェクトの真の力は他のゲームオブジェクトを含むことができ、OSのファインダ(エクスプローラ)で扱う、フォルダのように働くことです。これにより、ゲームオブジェクトを階層化することができます。複雑なモデルや完全なライティングがひとつのゲームオブジェクトとして定義できます。(事実、Unityに登場する多くのモデルは、モデリングパッケージで定義されたとおりの、ゲームオブジェクトの階層です)

内部のゲームオブジェクトは、子ゲームオブジェクトと呼ばれます。

コンポーネント(Components)

コンポーネントは、ゲームオブジェクトの構成要素です。これなしには、興味深いことは何もできません。

コンポーネントはメッシュ、マテリアル、地形データやパーティクルシステムなど、描画要素を表すものもあります。他にも、カメラ、ライトなどもっと抽象的な、物理的に表示されないものもあります。それらは、アイコンとワイヤフレームのガイドラインで表示されます。

コンポーネントは、単体では存在せず、常にゲームオブジェクトに付加されます。複数のコンポーネントをひとつのゲームオブジェクトに付加できます。ゲームオブジェクトは、たとえば、複数のスクリプトを格納できるなど、ある種類のコンポーネントは同じ種類であっても複数保持できます。しかし、パーティクルシステムを定義するようなコンポーネントは1つのゲームオブジェクトに1つだけです。もし、複数のパーティクルシステムを使用したい場合は、通常ゲームオブジェクトを階層化し、複数がパーティクルシステムを保持するようにします。.

アセット(Assets)

インポートした全てのアセットは、プロジェクトペインに表示されます。ほぼ全てのものが使えます：単純なマテリアルやテクスチャ、オーディオファイル、完全な、プレハブ化された(プレハブと呼ばれる)ゲームオブジェクトなどです。

たとえば、複数のモデルとアニメーションを持つ、プレイヤーキャラクタのプレハブは、ひとつのアセットとして定義できます。また、機能として必要な、スクリプトコンポーネント、オーディオクリップや、他のコンポーネントを含むことができます。そのため、アセットをシーンにドラッグすれば、即、操作可能なアバタが出来上がるのです。

カスタムアイコンとギズモ(Custom Icons & Gizmos)

Unityに、アセットのカスタムアイコンや、その他の視覚的情報を表示させられます。[次のチャプタ](#)にサンプルがあるので、参照してください。

プロジェクトのアセットは、プロジェクトペインに表示されます。シーンに追加すると、階層ペインに現れ、シーンに含まれたことを示します。(シーンは劇場のステージのようなものです。レベル、メニュー、マルチプレイヤーゲームのロビーなどになります)プロジェクトペインはプロジェクト中の全てのシーンで同一です。

プレハブ(Prefabs)

プレハブはテンプレート定義されたアセットです。ワープロでのテンプレートドキュメントのようなものです。プレハブをシーンに追加すると、Unityは、コピーではなく、プレハブへのリンクを階層ペインに追加します。これを、インスタンス化と呼びます。それぞれのリンクは、プレハブのインスタンスとなります。

プロジェクトペインでプレハブをクリックし、設定を変更すると、シーン中の全てのインスタンスに即座に羽委されます。これにより、プレハブは、弾丸や、敵など再利用可能な要素となります。もし、敵が正しく

動作していないときは、シーン中のそれぞれの敵を変更するのではなく、オリジナルのプレハブに設定されたスクリプトを変更すればよいのです。

しかし、特定のインスタンスいくつかだけに、設定を行いたい場合は、それも可能です：変更は、特定のインスタンスのみに影響します。

プレハブはプロジェクト及び階層ペインに青地で表示されます。

NOTE プレハブのインスタンスは追加のコンポーネントを追加すると、オリジナルのプレハブへのリンクが破壊されます。Unityは事前に警告を表示しますが。ただし、変更を行いリンクが破壊された後に、オリジナルのプレハブを交信することも可能です。

謝辞(Acknowledgments)

このチュートリアル作成に御尽力いただきました、次の方々に感謝します：

David Helgason, Joachim Ante, Tom Higgins, Sam Kalman, Keli Hlodversson, Nicholas Francis, Aras Pranckevirius, Forest Johnson And, of course, Ethan Vosburgh who produced the beautiful assets for this tutorial.

初めの一歩(First Steps)

どんなゲームにも、プレイヤがコントロールして、スターとなるキャラクタがいるものです。私たちのスターは**Lerpz**です。



Lerpzを動かす(Animating Lerpz)

このチャプタでは

- 第3者プレイヤとカメラコントロールの実装
- アニメーションのコントロールと合成
- パーティクルシステムを使ったジェットパックの噴射を実装
- プレイヤへのプロブシェーダの追加
- プレイヤ状態の管理
- プレイヤの体力、死と復活の管理

始める前に、このゲームはどんなものかを簡単に知っておいたほうが良いでしょう。私たちに必要なものは、、、

プロット(The Plot)

私たちのヒーローLerpz：彼はロボットワールドバージョン2を訪れたエイリアン。ロボットワールドバージョン1は、重大な境界を越えてしまい、あっけなく太陽に衝突して、消滅してしまいました。

不運なことに、Lerpzは：その地域の警官に宇宙船ごと捕らえられてしまいました。上やら下やらを見回しているうちに、Lerpzは彼の宇宙船を発見しました。しかし、Mr.Bigの、もっとも陥落で、異常なまで

に凶悪な弟、Mr. Even Biggerからどうしたら宇宙船を取り返すことができるでしょうか？

Mr.Biggerは、空中庭園に芸術的に並べられた、オイル缶を、心から愛しています。ホバー・パッドに置いたときの、輝く様子に、いつも見とれています。(もちろん、普通の庭用のライトを置くより安いですし)

しかし、Mr.Biggerは、まだ、気が付いていません！彼の節約家っぷりに感謝しましょう。Lerpzがオイル缶を全て集めれば、空中庭園のセキュリティシステムはパワーがオーバーロードしてしまいます。きっと、Lerpzの宇宙船を捕らえている電磁フェンスも停止するはずです。Lerpzは宇宙船に乗り込む、オイルを注入、自由の世界へと飛び立つのです。

私たちのヒーローがしなければならないことは、オイル缶を集めることです。そうすれば、電磁フェンスは自動的に停止します。Lerpzは宇宙船に乗り込み脱出するでしょう。Mr.Biggerはロボット警備員を雇い、Lerpzをとめようとします。しかし、ラッキーなことに、彼らはそれほど強くありません。

さあ、ストーリーはわかりました。私たちのヒーローを具現化しましょう。

Lerpzを追加 (Introducing Lerpz)

 プロジェクトを開いて、Scenes->TheGame Sceneを見てください。

最初のステップはLerpzをシーンに追加することです。

 プロジェクトペイン(Project Pane)のObjectsフォルダを開きます。

 Lerpz Prefabをシーンビューか階層ビューにドラッグします。

 階層ビューに新しく追加されたLerpzのエントリをクリックし、名前をPlayerに変更します。

 Playerオブジェクトを選択したまま、マウスをシーンビューに動かし、F(focus)キーをタイプし、Lerpzモデルをビューの中心に表示します。

 Lerpzを少し高くなったプラットフォームの上に配置します。牢獄の近くの(黄色のV字にくぼんだ)ジャンプ台がある場所です。(次のページのスクリーンショットを見てください。)

Playをクリックすると、Lerpzが牢獄外の中庭に立っていることでしょう。現段階では、Lerpzは動くことができません。また、カメラもプレイヤキャラクタにリンクする必要があります。

 Playボタンをもう一度押してゲームを停止します。

Lerpzが動くようにしなければなりません。でも、まず、一步下がってカメラについて見てみましょう。



Lerpzをシーンに配置

第三者視点カメラ(Third Person Cameras)

1人称視点ショーティングでは、カメラはプレイヤーの視界であり、カメラがシーン中で他のオブジェクトを追跡することを考える必要はありません。プレイヤーがカメラオブジェクトを直接コントロールします。1人称視点カメラは、そのため、比較的実装が簡単です。

しかし、3人称視点カメラは、プレイヤーの周りに追従しなければなりません。とてもシンプルに思えますが、プレイヤーとカメラの間にシーン中の物体が入ることを考える必要があります。レイキャスティングを用いて、望まないオブジェクトがカメラとプレイヤアバタの間にあることをチェックできますが、いくつかの特殊な場合を考えなければなりません。たとえば：

- Lerpzが壁の前に立った場合はどうなるか？カメラは、プレイヤーの上に移動し、見下ろすべきか？横側に回り込むべきか？
- 敵が、プレイヤアバタとカメラの間に入った場合はどうするか？
- プレイヤコントロールはどのように動作すべきか？カメラ位置によるべきか？もしそうなら、カメラが障害物をよけるために、予想外の動きをした場合混乱してしまいます。

3人称視点に関して、多くの解決策が、今までに試されてきました。どれも、100%の回答ではありません。1つの解決法は、敵や壁など間にあるものを半透明にし、フェードアウトさせるものです。また、プレイヤーの周りに付いていくカメラをつかう方法もありますが、必要であれば、壁やビルをすり抜けて、プレイヤーのビューを正常に保つことが必要です。

このチュートリアルで使うプロジェクトには、いくつかの異なるカメラスクリプトが含まれていますが、チュートリアルでは**SpringFollowCamera**のみを使用します。プロジェクトペインの**Script**フォルダ中、**Camera**サブフォルダから見つけることができます。

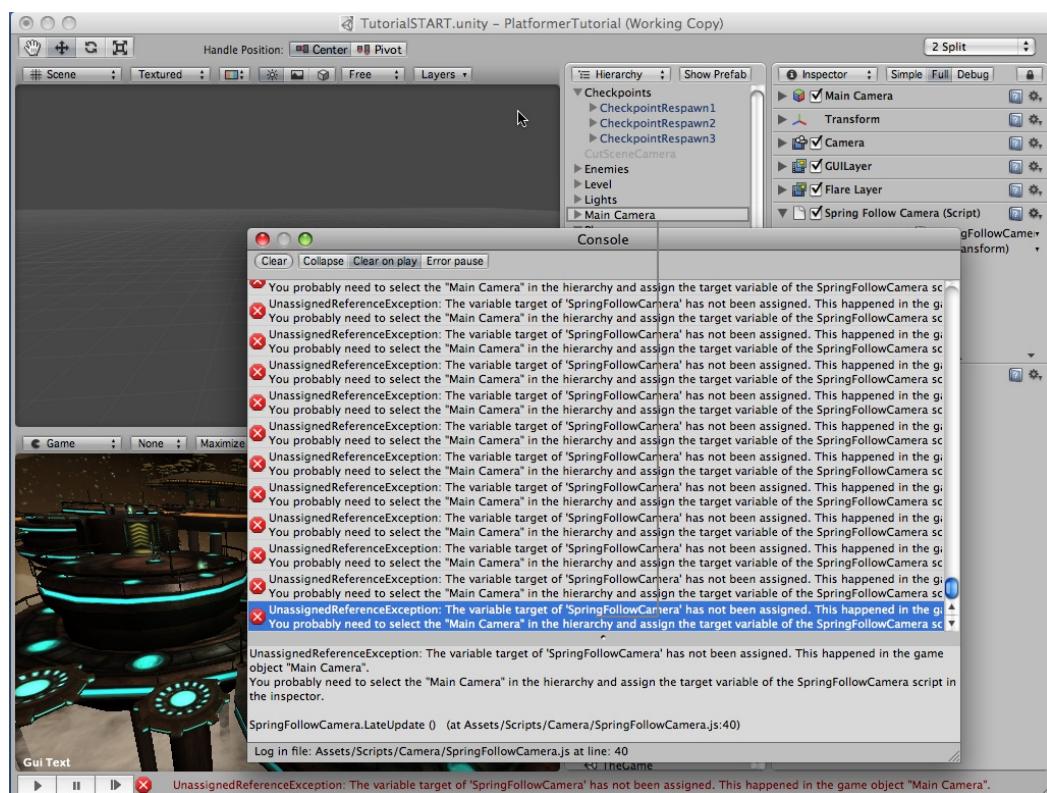
 SpringFollowCameraスクリプトをプロジェクトペインから、階層ペインNearCameraオブジェクトにドラッグしてください。

 プレイをクリックします。

エラーメッセージを受け取るでしょう。Play、Pause、Stopボタンのすぐ右側、Unityのウィンドウの最下部に表示されます。

 デバッグコンソール(Debug Console)が開いてなければ、開きます。 (Shift+Cmd+C[mac] / Shift+Ctrl+C[wIn])

これで、警告、エラーや、ゲーム中でのその他のデバッグ情報が表示されます。ログ中には多くの同じメッセージが、連続して表示されていると思います。図3.1に示されるように、ペインの下部分に、ハイライトされたログが、エラーメッセージに関する、少し詳しい情報を表示しています。



“No target”(ターゲットがありません)エラーメッセージ

TIP デバッグログウィンドウは、可能な限り、改裝中の原因となるゲームオブジェクト(または、プレハブやスクリプトが原因であれば、プロジェクトトペイン)へ、リンクを示す行を表示します。前ページのスクリーンショットを参照してください。

UnassignedReferenceExceptionは、Unityを始めたばかりのころは、最もよく目にするエラーの種類です。恐ろしく聞こえますが、スクリプトに値がセットされていないだけです。デバッグログにも同じことが表示されていますので、書かれている通りに行いましょう：

 階層ペイン中の**NearCamera**オブジェクトをクリックし**Spring Follow Camera**(スクリプト)コンポーネントの属性を見ましょう。

Targetの属性は**None(Transform)**に設定されています。これは、カメラが対象とするターゲットオブジェクトが設定されていないことを示しますので、それを設定しましょう：

 もしプレイ中であれば、ゲームを停止しましょう。

 もし、選択されていなければ、**NearCamera**オブジェクトを階層ペインから選択します。

 **Player**ゲームオブジェクトを階層ペインから**Target**設定にドラッグします。

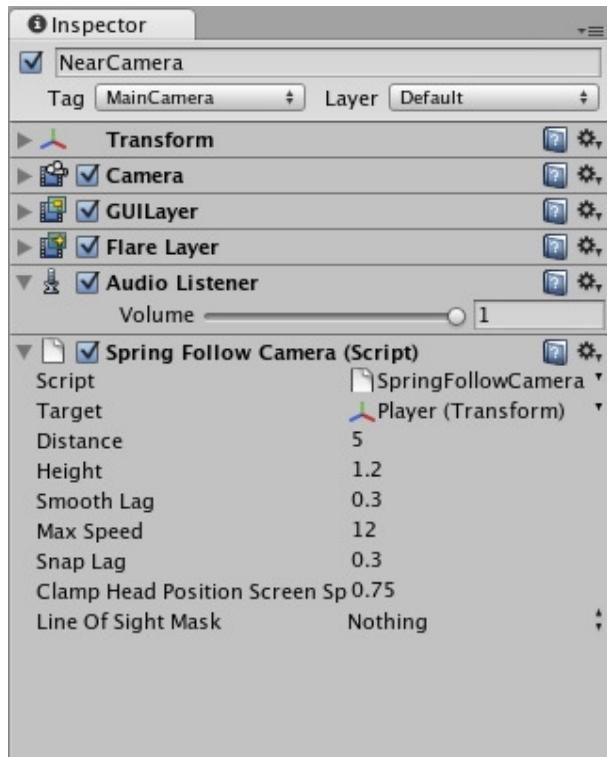
プレイ中の変更

ゲームをプレイしている間も、Unityでは、ゲームオブジェクトやコンポーネントの属性を変更できます。しかしながら、これは保存されません！ゲームを停止すると、全ての変更は破棄されてしまいます！

変更を破棄したくなければ、変更を行う前に、**必ずゲームを停止してください！**

ここでプレイをクリックしても、カメラは動きません。**SpringFollowCamera**スクリプトに関するエラーが表示されます。ターゲットには**ThirdPersonController**が付加されていなければなりません。三人称カメラは、プレイヤの操作と密接に関連しているため、プレイヤの動作を把握し、それに応じた動きをお分けなければならないからです。

最終的な設定は、次のページの画像に示されています：



Spring Follow Camera スクリプトの設定.

カメラの動きが気に入れなければ、いくつか、別の値を試してみてください。ここにあげているのは一例であり、唯一の正しい設定というものは存在しません。

依存関係のなかで、状況を考慮し、最適なものを選ばなければなりません。

ThirdPersonControllerスクリプトを、プロジェクトペインのScripts→Playerフォルダから、Playerゲームオブジェクト(階層ペイン中)にドラッグして、カメラとプレイヤの接続を行います。(プレハブ接続が破壊されます)

Third Person Controllerスクリプトにも、必要なオブジェクトや依存関係があります。最も重要なものはCharacter Controllerコンポーネントです。幸運なことに、スクリプトがUnityにそれを伝えているので、Unityがコンポーネントを追加してくれます。

接続と依存性

Unityは視覚的アセットを表示するのに優れていますが、ゲームにおいて期待する、インタラクティブ性を提供するには、それぞれが接続されていなければなりません。接続は視覚的に表示することが難しいものです。

接続は、依存性とも呼ばれます。これは、あるオブジェクトが機能するために、別なオブジェクトが必要なことを示します。別のオブジェクトが、また別のオブジェクトを必要とする場合もあるでしょう。

結果として、無数の文字列(つまりスクリプト)によって、アセットは相互に結合され、ゲームを作ります。

これらの依存性を定義することが、ゲームデザインの鍵となります。

次は、**Player**ゲームオブジェクトにタグをつけます。スクリプトはタグによって指定されたゲームオブジェクトを、シーン中から探すように、Unityに指示できるようになります。

 下図に示されるように、Playerオブジェクトがインスペクタに表示されている状態で、**Tag**ドロップダウンメニューを開いて、"Player"タグを選んでください。

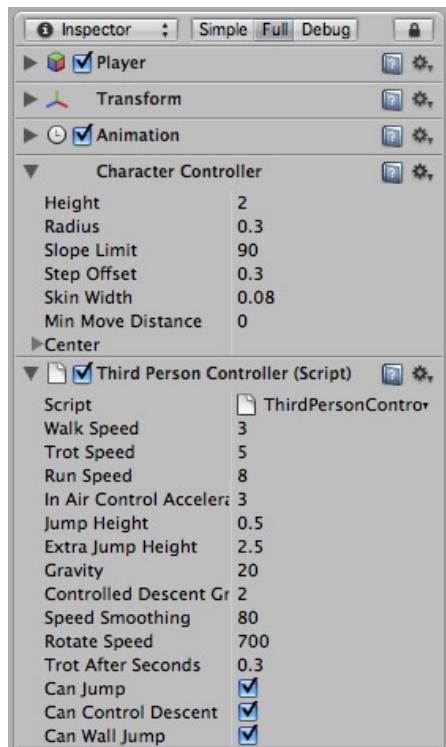


Playerタグを選択

NOTE メニューに用事されているタグのリストはUnityがデフォルトで用意したものです。独自のタグやレイヤを作る方法については、後ほど学習します。

タグは後で利用しますので、Character Controllerとスクリプトに戻りましょう。

 Playerオブジェクトを選択し、インスペクタを見ます。次のようになっているはずです：



Character Controller と Third Person Controller スクリプトコンポーネント

次のステップは、Character Controllerの調整です。Y軸方向のずいぶん下のほうにカプセルコライダ(Capsule Collider)がある(シーンビュー中のColliderの位置は、青色の長い円柱ワイヤフレームで表示されています)ため、Lerpzはうっすらと浮き上がっています。Center Yを変更します。



Character ControllerのCapsuleCollider(青いワイヤフレームで表示)を調整



Capsule Colliderをスクリーンショットに示される位置に調整します。(実験によると、Character ControllerのCenter Yを1.03にすると、Lerpzの足部分にぴったりと合いました。)

Playをクリックすると、コントローラによってLerpzが動き回る様子を確認できます。Lerpzの足はきちんと地面を捉えています。

Character Controller と Third Person Controller スクリプト

ほとんどのゲームにおいて、プレイヤーのアバタは、回ったり、突然止まったり、ものすごい距離をジャンプしたり、そのほか物理的には不可能な技能を持っています。これらは、古典的な物理法則ではモデル化することが難しいものです。そのため、CharacterControllerはプレイヤアバタを物理エンジンから切りはなし、基本的な動作のコントロールを提供します。

Character Controllerはプレイヤ(とプレイヤ以外の多く)の行動をシンプルにします。基本的な行動システムに結び付けられたカプセルコライダによって、キャラクタが動き回ったり、階段を上がったり、スロープを上り下りすることを可能になります。最大の段差やスロープのサイズはインスペクタで変更できます。

Character Controllerは通常スクリプトと共に用いられます。スクリプトはCharacter Controllerを使い、ゲームの要求を満たすように拡張します。このプロジェクトでは、Third Person Controllerスクリプトが、この機能を提供し、ゲームに必要なサポートを追加します。ジョイスティック、キーボード、マウス、その他の入力デバイスの操作に応じて、プレイヤアバタをコントロールすることを可能にします。

Unityの **Input Manager** (Edit->Project Settings->Input Manager)が、インプットデバイスに合わせてどのようにプレイヤをコントロールするかを定義します。

NOTE プレイヤに使用しているスクリプトに、特別なことは何もありません。このプロジェクトのために作られた、通常のUnityスクリプトです。デフォルトのCharacter Controllerスクリプトは存在しません。

Third Person Controllerスクリプトはプレハブの一部に含まれているので、追加する必要はありません。

次のステップはLerpzを正しく動かすために、ジャンプやパンチなど、動作を追加します。

Lerpzを動かす

現時点ではLerpzはシーンの中を滑っています。これは、Character Controllerがアニメーションをコントロールしていないためです。プレイヤの形状や、どのアニメーションシーケンスを、それぞれの動作に適用するかが指定されていません。Lerpzとアニメーションシーケンスを接続しなければなりません。これは、ThirdPersonPlayerAnimationで行われます。



コンポーネントメニューを使いThirdPersonPlayerAnimationスクリプトをPlayerゲームオブジェクトに追加します。

Playをクリックすると、Lerpzが正しくアニメーションするのを確認できます。

さて、ここでは何が起こっているのでしょうか？スクリプトは何をするのでしょうか？答えは、Unityがどのようにしてキャラクタアニメーションのデータを扱うかに隠されています。

キャラクタアニメーション

キャラクタアニメーションシーケンスは3D Studio Max, Maya, Blender, Cheetah3Dなどのモーリングパッケージで作成されます。Unityは保存されたシーケンスを自動的にインポートし、Animationコンポーネントに格納します。

アニメーションシーケンスは、基本的なアニメーションに用いられる、仮想の骨格（スケルトン）で定義されます。スケルトンは、モデルのメッシュ（モデル自体の視覚的な形状を定義するデータ）が、アニメーションによってどのように変化するのかを定義します。

スケルトンとアーマチュア(Skeletons & Armatures)

ストップモーションやクレイモーションのアニメーションに詳しければ、金属の骨格を使うイメージを持つでしょう。アニメーションされるモデルは、骨格の周りに作られます。3Dモデルで使用される仮想的なスケルトンは、それらとほぼ同じであり、それほど複雑になることは、めったにありません。

そのようなコンポーネントのメッシュは、スキンメッシュと呼ばれます。仮想骨格はメッシュの下の骨を定義し、どのようにアニメーションするかを決定します。

アニメーションブレンディング(Animation blending)

キャラクタのアニメーションは、ゲームに必要な柔軟性を提供するために、混合されます。たとえば、歩くアニメーションは、話すアニメーションと混合できます。結果的に、キャラクタは話しながら歩くことになります。

ブレンディング（混合）は、アニメーション間のスムーズな遷移を提供するのにも用います。たとえば、歩くアニメーションからパンチのアニメーションへの遷移などです。

アニメーションの切り替えや、アニメーションの混合が、いつ必要で、どのように行われるかをUnityに指示するには、スクリプトを使用します。

Third Person Player Animationスクリプト

Lerpzのモデルは、15個のアニメーションを含み、様々なプロジェクトで用いられるように作られました。このチュートリアルでは、そのうち11個のアニメーションを使います。Playerオブジェクトを階層ペインから選択し、インスペクタを見ると、15個のアニメーションシーケンスがAnimationコンポーネントの中で、リストアップされていることが確認できます。以下に載せるアニメーションが、チュートリアルで利用するものです：

- Walk -- 通常の歩行動作

- Run -- 走る動作(Shiftキーを押すと走ります)
- Punch -- 敵のロボットガードを攻撃するときの動作
- Jump -- Lerpzがジャンプしたときの動作 Played when Lerpz leaps into the air.
- Jump fall -- Lerpzがジャンプ後最高点に達した後に、落下する動作
- Idle -- Lerpzが何もしていないときの動作
- Wall jump -- Lerpzが壁を蹴って後方に飛んだときの動作
- Jet-pack Jump -- Lerpzがジェットパックを使ってゆっくりと落ちるときの動作
- Ledge fall -- Lerpzがプラットフォームの端から落ちたときの動作
- Buttstomp -- Lerpzがロボットガードに攻撃を受けたときの動作
- Jump land -- Lerpzがジャンプや落下後に着地したときの動作

LerpzのモデルとアニメーションはMayaで作成されて、Unityにインポートされました。メッシュとアニメーションの詳しい情報は[Unity Manual](#)を参照してください。

アニメーションの多くは、プレイヤーが使用しているコントロールと反応に応じて、**ThirdPersonPlayerAnimation**スクリプトによって扱われます。いくつかのアニメーションは他のアニメーションにかぶさって行われますが、シンプルに順番に実行されるものもあります。スクリプトはmessage responder関数の集まりです。入力デバイスや、キャラクタ状態の変更に伴って、関連するメッセージが**ThirdPersonController**によって、送出されます。

Lerpzの攻撃(パンチ)は、**ThirdPersonCharacterAttack**(後で追加します)という、異なるスクリプトで処理されます。不必要な分割に見えるかもしれません、実は必要です。ほとんどの基本動作である、歩く、走る、ジャンプするなどは、プレイヤーキャラクターの形状にかかわらず、ほとんど同じです。しかし、攻撃や防御の動きは、より多彩です。ある、プラットフォームゲームでは、プレイヤーキャラクターは銃を持っているかもしれませんし、他のゲームでは、マーシャルアーツで戦う場合もあるでしょう。

このチュートリアルではLerpzは、こぶしを使った西洋武術のマスターです。つまり、敵をこぶしで殴りつけます。アニメーションは、シンプルなパンチをするものです。

ギズモ(Gizmos)

ThirdPersonCharacterAttackは便利な特性を持っています。Lerpzのパンチアクションによって効果のある範囲が、球体のギズモによって表示されます。ギズモは2つのギズモ描画メッセージ処理関数の1つで描画されます。例では、黄色い球体でパンチの位置と、有効範囲を示すギズモがOnDrawGizmosSelected()関数への、返答として描画されます。

この関数は、静的でUnityエディタ自身によって呼び出されます。もうひとつ、`OnDrawGizmos()`があります。これは、Unityエディタによって、更新時に呼び出されますが、親ゲームオブジェクトの、選択、非選択に依存しません。

ジェットパック(The Jet-Pack)



Lerpzのジェットパックが作動中

さて、私たちのキャラクタは走ったり、ジャンプしたりしています。しかし、彼のジェットパックはまだ、動いていません。Lerpzはジェットパックを使って、ゆっくりと降りてきます。この動作は、すでに行われていますが、ジェットパックのジェットは、まったく、動きません。ジェットを動作させるには、2つのパーティクルシステムとポイントライトコンポーネントを追加する必要があります。パーティクルシステムは、炎のような効果を出し、ポイントライトは、炎が光っているような効果を生み出します。

TIP 左右のジェットそれぞれにポイントライトを配置することもできますが、2つは十分に近いので、1つのライトだけを用いることで、処理の負担を減らします。これは、簡単かつ効率の良い、最適化です。

パーティクルシステムとは?(What is a Particle System?)

パーティクルシステムは、大量のパーティクルを3Dワールドに生成します。通常2Dのビルボードやスプライトです。それぞれのパーティクルは、速度と加速度を持ち、一定時間存在します。設定に応じて、ビルボードマテリアルが使われます。パーティクルは、火、煙、惑星の爆発などをシミュレートできます。

パーティクルシステムを追加 (Adding the Particle Systems)

ゲームオブジェクトメニューから、何も入っていないゲームオブジェクトを階層ペインに作成します。

ゲームオブジェクトの名前を"Jet"に変更します。

新しいゲームオブジェクトが選択された状態で、次のものを追加します：

Ellipsoid Particle Emitter

Particle Animator

World Particle Collider

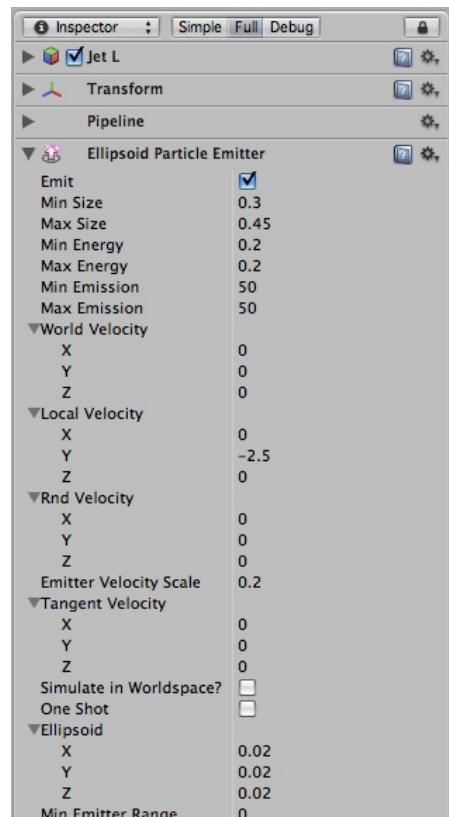
Particle Renderer

インスペクタで、**Particle Renderer**コンポーネントの“Enabled”のチェックをはずすと、一時的に、無効になります。

"Jet"をLerpzの右側のジェット噴射口の真下に、配置します。

Particle Rendererを有効化します(Enabledをチェック)

Ellipsoid Particle Emitterを、次の図のように設定します：



Ellipsoid Particle Emitterの設定

設定すると、ジェットの炎のシミュレートを使う、細いパーティクルのストリームになります。

TIP パーティクルが真下に向かって動いていない場合、Unityの回転ツールを使って、オブジェクトをジェットが、ジェットパックの噴射方向に向くように回転してください。

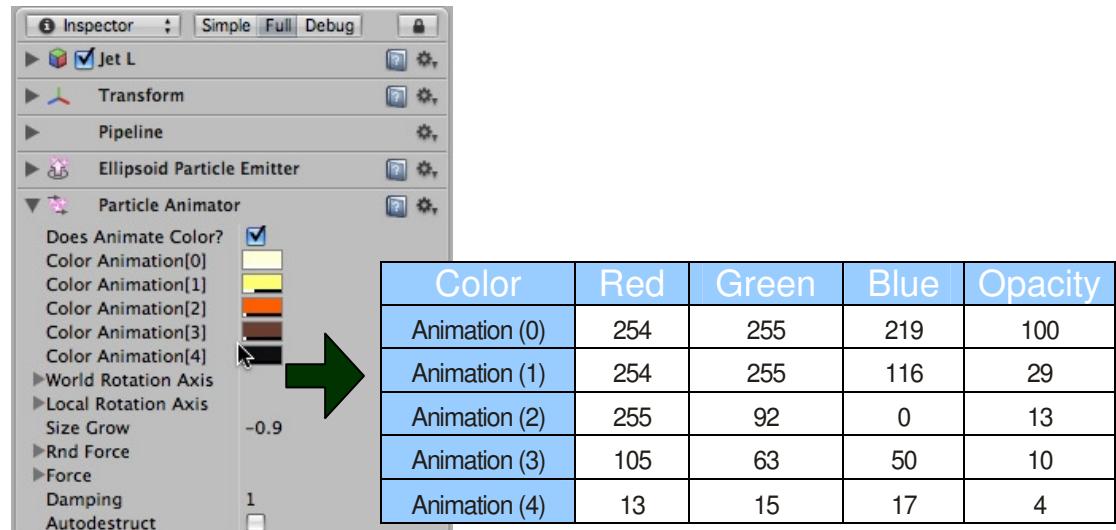
作業を終えると、パーティクルシステムは、プレイヤーの"torso"の子オブジェクトとして、階層に追加されます。これにより、ジェットがプレイヤーの移動に追従します。ここでは、ライトに見せることが目的なので、正確な位置にそれほどこだわる必要はありません。

Min Sizeと**Max Size**設定はパーティクルの幅を定義します。**Min Energy**と**Max Energy**設定は、最小と最大の、パーティクル生存期間を設定します。ここで用いているパーティクルは0.2秒という、短い時間生存し、消えます。

パーティクルの生成量(quantity)は50に設定しました。**Min Emission**と**Max Emission range**は、一度にスクリーンに存在するパーティクルの量を定め、どのぐらいの種類を生成するかを定めます。ここでは、MinとMax rangeを同じに設定したので、ジェットはスムーズに爆破というより、バチバチと火を噴いて見えます。最終的に、パーティクルはスムーズに流れるべきです。

NOTE "Simulate in Worldspace"は無効化しました。これにより、パーティクルが高速で動いていくよりも、ゆっくり揺らめく炎ではなく、勢いよく噴射するジェットが表現できます。ジェットがLerpzの回転やひねりに無関係に動作することで、温度の高い、一定した炎の噴射になります。

 Particle Animatorコンポーネントを、図のように設定します：



Particle Animatorの設定。表はColor Animationの設定。その他の設定も忘れない！

Partice Animatorは、パーティクルの色を生存期間に応じて変化させます。パーティクルは白から始まり、黄色、オレンジと暗い色に変化し、ジェット噴射をクールにします。それぞれのパーティクルにテクスチャを描画するため、**Particel Animator**はパーティクルに色合いをつけます。カラーアニメーションは微妙ですが、効果が期待できます。

色の上をクリックすると、色選択ダイアログが表示されます。また、"Opacity(不透明度)"を設定するスライダも表示されます。テーブルは色と不透明度の、アニメーションサイクルに応じた、設定を含んでいます。炎は格好よく現れて、消えていきます。

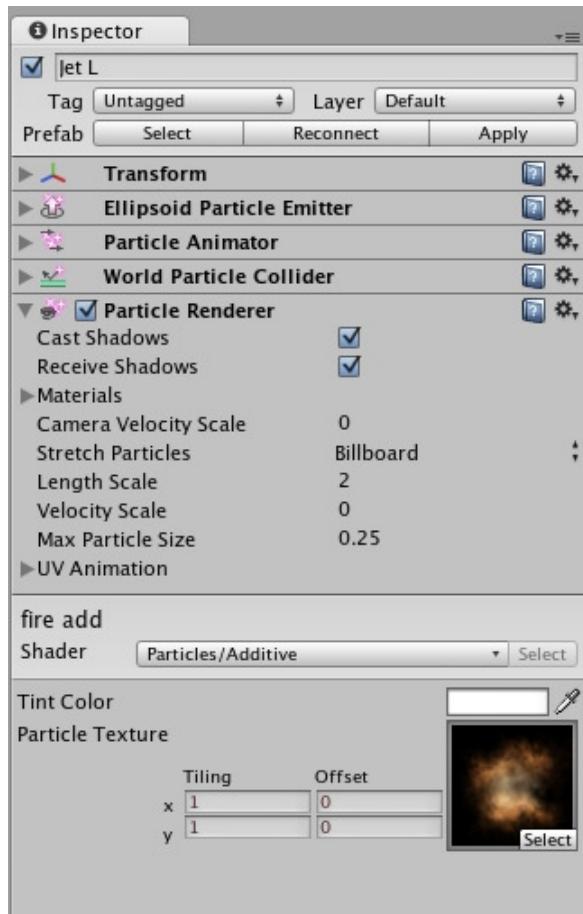
NOTE 不透明度の設定はパーティクルのアルファチャンネルを変更し、透過度を設定します。パーティクルのマテリアルが、すでに、アルファチャンネルを含んでいる場合、不透明度設定で変更されます。

次は、**Particle Renderer**です。このコンポーネントは、各パーティクルを描きます。そのため、どのようにパーティクルが出現するかを知る必要があります。また、それぞれのパーティクルを描くマテリアルも定義します。炎のようなジェットの効果を出したいので、"fire add"マテリアルを使用します。Particles->Sources->Materials->fire add にあります。

TIP このアセットは、Standard Assetsフォルダにも含まれています。



コンポーネントの値を以下のように設定します：



Particle Renderer設定

Stretch Particles設定は、Unityにパーティクルが高速で移動した場合、引き伸ばされてレンダリングされるかどうかを設定します。今回は速度に応じて少し、引き伸ばします。わずかな視覚的きっかけを与え、小さく、丸い形状にします。

NOTE **Cast Shadows**と**Receive Shadows**設定は、カスタムシェーダを使わない場合は効果がありません。このチュートリアルではカスタムシェーダは扱いません。

ライトの追加 (Adding the Light)

私たちのジェットはクールに見えますが、ただのイリュージョンです。パーティクルシステムが、小さな画像を吐き出しますが、結果的に炎のような効果は、光を発しません。イリュージョンを完成させるには、点ライトのオブジェクトを追加する必要があります。ジェットの噴射と同時に、点灯、消灯します。結果、炎のジェットは、ライトアップされます。(ここでは、それぞれのジェットに1つずつではなく、1つだけのライトを使い、計算にかかるパワーを削減します。)

 新しい、**Point Light**ゲームオブジェクトを作成しましょう。



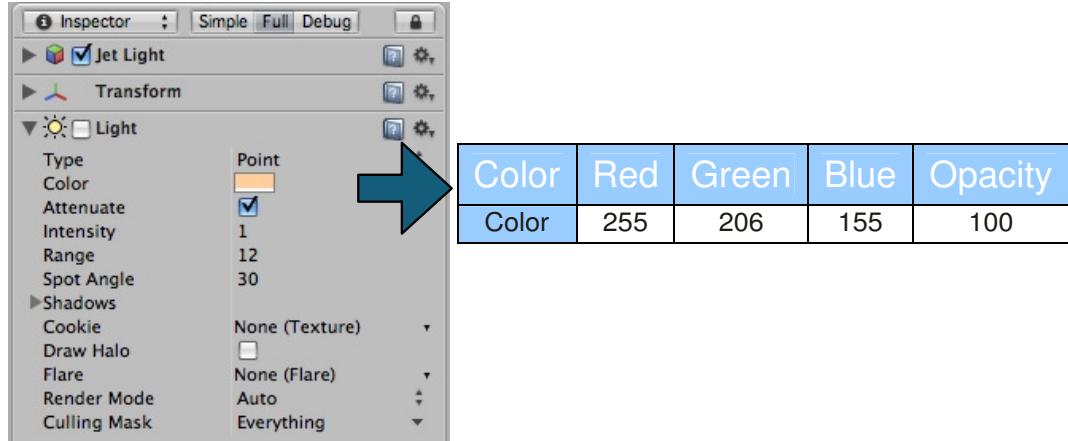
作成したオブジェクトの名前を“Jet Light”に変更し、Lerpzのジェットパック上の、2つのジェットモデルの間に配置します。このライトにより、ジェットが光を放っているような、効果が再現されます。(後ほど、Jetパーティクルシステムに戻ってきます)



この効果が動作するには、明るくて、高い強度の点ライトが必要です。



ライトを選択し、設定を次の図のように行ってください：



ジェットパックライトの設定

どうして影をつけないの？(Why no shadows?)

影は、多くのハードウェアで、計算負荷が大きくなります。避けられるものならば、使用を控えるべきです。ジェットは、それほど大きくありません。Lerpzの背中が光る程度で問題ありません。この、点ライトは近くの物体に反射するでしょうが、影を薄くするほどの明るさにはならないでしょう。

次のステップは、Playerゲームオブジェクトが、ジェットとライトを含むように、更新することです。初めに、ジェットをプレハブとして、プロジェクトペインに追加します：



プロジェクトペインで、(空の)Playerフォルダを選び、Create...をクリックします。



ドロップダウンメニューから、Prefabを選びます。これで、空のプレハブオブジェクトが作成されます。



空のプレハブの名前を Jet に変更します。



新しい、光り輝くJetオブジェクトを、階層ペインから新しいプレハブにドラッグします。

階層ペイン中のJetの名前が青に変わり、プレハブにリンクされたことを示します。Jetプレハブのインスタンスを2つ使ってLerpzのジェットパックを作ります。プレハブを使うため、オリジナルのプレハブを変更すれば、両方のジェットが変更されることになります。



オリジナルのJetオブジェクトを、階層ペインから削除します。(Jet Lightは消さないでください!)

Playerに、Jetを2度追加し、ライトを1度追加します。



階層で、Playerオブジェクトを選択します。



torso子オブジェクトが見つかるまで、フォルダを開きます。



Jetプレハブを、torsoオブジェクトに2回ドラッグします。これで、シーンに2つのJetオブジェクトが作られます。



Jetインスタンスの名前をJet LとJet Rに変更します。



Jet Lightを、同じく、torsoオブジェクトにドロップします。

次のような階層が出来上がっているはずです：



ジェットパック階層



Unityの操作ツールを使って、それぞれのJetプレハブをLerpzモデルの、Jet接続口に移動します。パーティクルが正しい方向に噴射されるように、回転する必要があるかもしれません。



Jet Lightを2つのJetプレハブの間に移動します。

ここまで終わると、Lerpzは2つの噴射しているジェットを、ジェットパックに装備して、動き回っているはずです。あと少しで完成です！

最後のステップは、JetプレハブとJet Lightオブジェクトを、Lerpzがジャンプしているときだけ、作動させることです。これは、スクリプトによって行われます：



Scripts->Playerの中にある、JetPackParticleControllerスクリプトを、階層ペインで、Playerオブジェクト階層の1番上に、ドラッグしてください。これで、スクリプトがプレイヤーキャラクターに追加されました。

これで、ジェットパックは期待通りに動作するはずです。スクリプトが2つのパーティクルシステムとライトを、Lerpzの動きに合わせて、コントロールします。プレイヤがジャンプボタンを押すと、3つの要素を連動させて、噴射を表示します。



プロブシャドウ(Blob Shadows)

Lerpzは常に、見つけやすい必要があります。そうすれば、プレイヤは、画面の中に多くのものが表示されていても、見失わずにすみます。これは主に、アーティストや、ゲームデザイナの仕事ですが、Unityによって行われる部分もあります。

最も重要なもののひとつは、影とライティングです。パフォーマンスのために、ライティングの効果は通常、アーティストによってテクスチャの中に書き込まれます。“焼きこみ(baking)”と呼ばれる技術です。この技術は、セットや小道具など、動かないオブジェクトには効果的です。街灯の下を歩くキャラクタは、ライトにリアルタイムで反応しなければなりません。キャラクタの下の道路は、焼きこまれたライトを持っていますが、キャラクタにはこのトリックが使えないため、道路も正しく反応しなければなりません。

解決策は、必要な場所に、ダイナミックなライトを配置することです。焼きこみテクスチャを用いると、追加したライトは、焼きこみライティングとして扱われます。ライトが動いているオブジェクトのみに、効果があるように設定します。ライトはすでに、シーン中に配置してあります。

これが、最後の要素である影を残します。

3Dプラットフォームゲームでは、影は、キャラクタの位置を把握するのに重要で、ジャンプや落下した際、どこに着地するのかを教えてくれます。つまり、Lerpzは良く見える影を持っていなければなりません。

影は、グラフィックエンジンにより、リアルタイムに計算され、レンダリングされるライトによって生成されます。しかしながら、このような影は、多くの計算パワーを使います。加えて、全てのグラフィックカードが影を早く計算する機能を持っているわけではありません。古いカードでは、一切影を描画することができないものもあります。

そのため、Lerpzにはプロブシャドウ(Blob Shadow)を使用します。

プロブシャドウの追加 (Adding a Blob Shadow)

プロブシャドウは、見せ掛けです。ライトの光線をつかって、何かに当たるかをチェックするのではなく、ただの暗い画像を、キャラクタの下に投影します。ここでは、ただの丸くて黒いプロブです。これは、グラフィックカードにとって簡単で早く行えますので、どんなハードウェアでも動作します。

UnityはBlob-Shadowプレハブを基本アセットの中に用意していますので、新しく作らずに、それを使うことにします。アセットは既にインポートされ、プロジェクトのBlob-Shadowフォルダに追加されています。フォルダを開いて、blob shadow projectorプレハブをクリックし、キャラクタオブジェクトの一番上にドロップします。階層ペインのPlayerです。これで、projectorがPlayerの直下に配置されるはずです。



blob shadow projectorプレハブがPlayerの階層に入った。

つぎは、blob shadow projectorのPositionとRotationデータを変更し、キャラクタの真上から、真下の地上に向くように調整します。

4分割レイアウト(4-Split layout)を選択します。



プロブシャドウプロジェクトのRotation値をそれぞれ90、180、0に設定します。



側面と上面ビューを用いて、プロジェクトをLerpzの頭の真上に移動します。また、影のサイズがちょうど良くなるまで、上下して調整してください。

新しいレイヤの作成(reating a new Layer)

現時点では、プロブがLerpzの上にも表示されいることがわかるでしょう。これは、望ましくありません。これを避けるには2つのオプションがあります。Near Clip Plane設定をプロジェクトから遠くに離すか、特定のレイヤにはプロブを投影しないようにするかです。2つ目の方法を用います。



なぜNear Clip Planeを調整しないの？(Why not adjust the Near Clip Plane?)

この方法は、簡単に見えるかもしれません、プレーンはLerpzのアニメーションを考慮したうえで、スクリプトによって制御しなければなりません。ジャンプしたときに足が前のほうにでたり、着地すると、元に戻したり。影は常に、Lerpzが立っている地面を照らしていないので、Near Clip Planeは同じ状態においておくことができないのです。



Playerゲームオブジェクトを開きます。



インスペクタからレイヤドロップダウンを開きます。

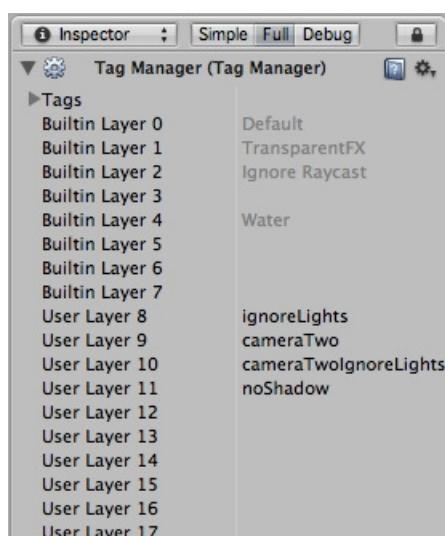


Add new layer... を選択します。



空のUser Layerエントリを選択し、名前をnoShadowとします。

インスペクタは次のようにになっているはずです。

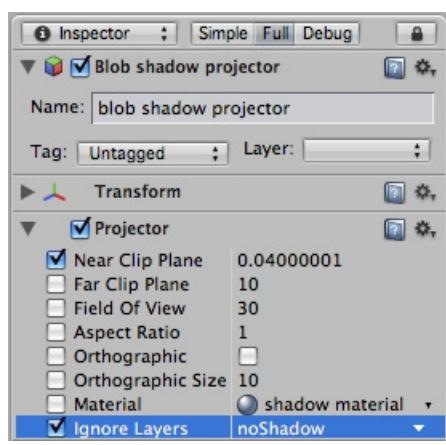


Tag Managerを使って、新しいレイヤを追加する。

- Playerオブジェクトを、階層ペインでクリックし、通常のインスペクタ設定を開きます。
- "Layer"ドロップダウンをクリックし、新しいレイヤの名前である、noShadowに設定します。Unityは子ゲームオブジェクトにも適用するかをたずねますので、"Change Children Layers"を選択します。

次に、Blob Shadow Projectorに、レイヤ中のオブジェクトには投影しないことを教えます。

- Blob Shadowの属性をインスペクタに表示し、プロジェクトコンポーネント中のIgnore Layersエントリを確認します。
- ドロップダウンメニューを使ってnoShadowレイヤ(子オブジェクトにも適用します)を、下図のように選択します。



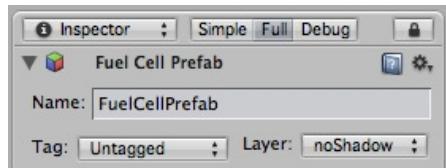
Blob Shadow Projector 設定

ここでゲームをプレイして、動き回ると、影が期待通りに動いているのを確認できます。ただし、落ちているオイル缶の近くでジャンプするとうまくいきません。アイテムの上にプロブシャドウが表示されてしまいます。

落ちているアイテムには影が映ってほしくありませんので、Blob Shadow Projectorに、それらを避けるように教えます。

これらの落ちているアイテムの詳細は、次の章で説明しますが、影の問題だけはここで解決しておきましょう。

- はじめに、ゲームをとめます。
- プロジェクトペインに行き、FuelCellPrefabとHealthLifePickUpPrefabオブジェクトを見つけます。Propsフォルダの中になります。
- それぞれのプレハブのルートオブジェクトを選択し、下図に示すとおり、レイヤをnoShadowに設定します。



レイヤが“noShadow”に変更された。

NOTE 親オブジェクトに変更を行った場合、Unityは子オブジェクトにも適用するかどうかを尋ねてきます。影響を考慮せずにやってしまうことは危険です。今回は、“FuelCellPrefab”と“HealthLifePickUpPrefab”ゲームオブジェクトの全てを、同じ“noShadow”レイヤに入れましたので、Unityの問い合わせに賛成し、変更を全てに適用しました。

スクリプトのコンセプト(Scripting Concepts)

歴史的に、先駆者たちがエンターテイメントのために作成した、驚くほど複雑な、オートマタと呼ばれる機械がいくつもあります。それらのいくつかは、とても巧妙で、パペットを使った簡単な舞台を行うこともできます。ユーザの入力に応じて、インタラクティブに動作を変更するものもあります。これらの機械は基本的には同じです。デザイナが、パペットやプロップ、背景などのアセットを作ります。それから、機械屋が思い通りの動作をさせます。

基本的な原則は年月を経ても変化せずに残っています。コンピュータは単に、鉄とバネで出来た物理的な機械を、手続きのリストによってコントロールされる、仮想的な機械に変えただけです。Unityはこの、手続きのリストを、スクリプトとして扱います。

ほとんどのスクリプトはゲーム開発において、有限状態機械と呼ばれ、中心におかれます。有限状態機械は、状態と呼ばれる、システムのインタラクティブな状況を定義するのに重要です。

オブジェクトが描画されても、されなくても、物理法則に従っていても、いなくても、照明でも、影を映しても、跳ねても、ディスプレイ上で位置を示しても、何でも状態として扱えます。インスペクタペインで、多くの状態を直接変更することができます。なぜならば、それらの状態はゲームにおいて一般的なものであるからです。

しかしながら、ゲームに特化された種類の状態もあります。Unityはプレイヤのアバタがエイリアンであることを知りません。どのぐらいのダメージをLerpzが受けても大丈夫なのか、Lerpzがジエットパックを持っていることも知りません。Unityは、どのようにしてロボットガードの動きを知り、Lerpzとインタラクトしたらいいのでしょうか。

ここに、スクリプトが登場します。ゲームに特別な状態の管理とインタラクションを追加するために、スクリプトを使用するのです。

ゲームでは複数の状態を管理しなければなりません。それらは：

- プレイヤの体力

- プレイヤが取得した、オイル缶の数
- プレイヤが、バリアを解除するのに十分なオイル缶を集めたかどうか
- プレイヤが、ジャンプパッドを踏んだかどうか
- プレイヤが、落ちているアイテムに触れたかどうか
- プレイヤが、宇宙船に触れたかどうか
- プレイヤが、復活ポイントに触れたかどうか
- ゲームオーバや、ゲームスタートの画面が表示されるかどうか
- その他、いろいろ、、、

これらの状態の多くは、他のオブジェクト状態をテストし、状態を更新することが必要です。中間状態を用いて、状態の変更をスムーズにする場合もあります。たとえば、オイル缶を収集すると、プレイヤがバリアを解除するのに、十分な数を集めたかどうかがチェックされます。

組織と構成(Organization & Structure)

このチュートリアルでは、プレイヤの状態機械、レベルと、敵がゲームオブジェクトにリンクしたスクリプトで操作されます。スクリプトはお互いに通信し、メッセージを送り合うことで機能を呼び出します。

リンクを設定するには、いくつかの方法があります：

- インスペクタを用いて、関連するオブジェクトにドロップして、リンクを作成する方法。

他の目的にも再利用しようとしている、一般的なスクリプトの使用に典型的です。

これは、特に探す必要がなく、単に関連する変数から値を取り出すようなスクリプトで、最も効果的です。しかし、事前にどのオブジェクトがリンクされるのかを知っておかなければなりません。

このオプションを、**LevelStatus**(このスクリプトは、今のところ短い切れ端で、すでにLevelゲームオブジェクトに負荷されています。)中のカットシーン(シーン切り替え)で用います。これにより、複数のカメラを柔軟に切り替えます。ひとつは"level exit unlocked"で、もうひとつは"level complete"です。しかし、それを変更するオプションが有ります。

- スクリプト中のAwake()関数で参照を設定する方法。

Awake()関数は、Update()イベントがゲームオブジェクトで実行される前に、すべてのスクリプトで呼び出されます。ここにリンクを設定すると、後で使うUpdate() 関数で用いる結果をキャッシュできます。通常、スクリプト中でアクセスする必要のある、他のゲームオブジェクトやコンポーネントへのリンクはプライベート変数で設定します。

`GameObject.Find()`を呼び出して、関連するオブジェクトを見つける必要があれば、1度だけ行うようにするべきです。`Awake()`中で`GameObject.Find()`を行うと、非常に処理が遅くなります。

この方法は、1つめの方法ほどの柔軟性は必要なく、毎ゲームサイクルに、複雑な探索を行いたくない場合に、もっと効果的です。解決法は、スクリプトが'起き上がる'時に探索し、結果を保持しておき、更新する場合に用いることです。

例えば、レベルの状態を扱う**LevelStatus**スクリプトはPlayerオブジェクトを含む、いくつかのオブジェクトへの参照を保持しています。これらは変更されないことがわかっているので、計算機にこれを行わせることができます。

- `Update()`関数で、参照を設定する方法。

この関数はゲームサイクルで、最低1度は呼び出されますので、重たい関数を呼び出すことは避けるべきです。`GameObject.Find()`や`GetComponent()`関数は、非常に重たいです。

この方法は、必要とするオブジェクトが、ゲームプレイ中に変わってしまうような場合に使用します。

例えば、チュートリアルのシーンのように複数の復活ポイントが有る場合に、どこで復活するべきでしょうか？これは、ゲーム実行中に変更される可能性があるので、それに応じて扱わなければなりません。

問題は、動作が重たいことなので、ゲームを設計する際に、頻繁に処理を行わずにすむように行うことが大切です。

ビジュアル開発環境でのスクリプト (Scripts in a Visual Development Environment)

Unityは、参照や接続よりも、視覚的アセットに重きをおく、特徴的なツールです。大きなUnityのプロジェクトは、階層に散らばる複雑なスクリプトを保持します。そのため、このチュートリアルは、オブジェクト指向のテクニックを用いて、複雑性を軽減しています。

プレイヤアニメーションなど、状態機械の特定の部分に作用するスクリプトは、それ自身が、関連する状態の変数を管理するべきです。これにより、スクリプトが、他のスクリプトに保持された状態変数にアクセスする際の、複雑性を軽減します。そのため、いくつかのスクリプトはローカルにキャッシュを保持し、情報へのアクセスを早めます。このテクニックは、スクリプト中の1つの関数が、他のスクリプトの同様の関数を呼び出す、コマンドの連鎖においても使用されます。プレイヤの死亡と体力の管理が、1つの例です。

NOTE Unityに使い慣れると、あなたのゲームにより適した、状態を管理する他の方法を見つけるでしょう。チュートリアルで使用されているデザインパターンは"なんにでも当てはまる"解決策ではないのですから！

Playerゲームオブジェクトをプレハブにすることもできます。そうすれば、他のプロジェクトでも、スタートポイントとして使用することができます。

-  プロジェクトペインでPlayerフォルダをクリックします。
-  新しいプレハブを作ります。(Playerフォルダの中に作られます)
-  新しいプレハブに適切な名前をつけます。LerpzPrefabが良いでしょう。
-  PlayerゲームオブジェクトをLerpzPrefabにドラッグし、作業を完了します。

死と再生 (Death & Rebirth)

プラットフォームゲームのキャラクタはリスクの高い人生を送ります。Lerpzも例外ではありません。もし、レベルから落下すれば、ライフを1つ減らすようにします。また、Lerpzが安全な場所(通常、復活ポイント呼ばれます!)に再出現するようにします。これで、Lerpzは冒険を続けることができます。

もしLerpzが地面から落ちるのだとすれば、レベルに登場するその他の住人も落ちるはずです。適切に設定しなければなりません。

最良の解決法は、ボックスコライダを使って、何かがレベルから落ちたことを検出する方法です。コライダを非常に長く、広くします。これで、プレイヤーが飛び降りるときにジェットパックを使ったとしても、捕捉することができます。しかしながら、Lerpzはどこか、復活する場所が必要です。復活ポイントは、少しあとに設定します。はじめに、ボックスコライダを作りましょう。

-  空のゲームオブジェクトを作成します。
-  新しいオブジェクトをFalloutCatcherに変更。
-  Box Colliderオブジェクトをそれに追加
-  Fallout DeathスクリプトをComponent->Third Person Propsから追加

インスペクタを使い、図に示されるように値を設定します：



Fallout Catcherの設定

落下死スクリプト(The Fallout Death script)

スクリプトはシンプルにThirdPersonStatusスクリプトに委譲するだけなので、短いものです。(Lerpzに追加する必要がありますが、今はまだ行いません)

コライダのトリガを扱うのはOnTriggerEnter()の中です。この関数は、Unityによって、ボックスコライダが、Lerpzや敵のような、Colliderコンポーネントを持つ他のゲームオブジェクトに当たった場合に呼び出されます。

3つのテストが有ります：1つはプレイヤ、1つは単純なリッジドボディ(Rigidbody)オブジェクト、もうひとつは、オブジェクトがCharacterControllerコンポーネントを保持しているかどうかです。2つ目のテストは箱や、木箱が(レベルにはそのようなアイテムは配置されていませんが、実験してみなければ配置できます) レベルから落ちた場合のテストです。3つめは、通常の物理が負荷されていない敵の検出に用います。

プレイヤがボックスコライダに接触すると、ThirdPersonStatusスクリプト中のFalloutDeath()を呼び出します。

他の衝突物がゲームオブジェクトにあたった場合、単純に破棄し、シーンから取り除きます。そうしなれば、永遠に落ち続けることになってしまいます。

加えて：

- Unityの関数Reset()は要求されたコンポーネントが存在することを保証します。コンポーネントがはじめに追加されるとき、この関数はUnityによって自動的に呼び出されます。

エディタから、コンポーネント右側のギアアイコンをクリックし、Resetを選択することで、呼び出すこともできます。



リセットメニュー命令

- @Scriptディレクティブはスクリプトを直接Unityのコンポーネントメニューに追加します。複雑なプロジェクトで、多くのアセットを持っている場合、プロジェクトペインから探し出す手間を省き、便利です。

現時点でゲームをプレイしようとすると、UnityはどこにLerpzを再出現したら良いか分からないと、不満を言うでしょう。再生ポイントの登場です。

再生ポイント(Respawn Points)

プレイヤーが死んだ時、どこか安全に再出現させる場所が必要です。このチュートリアルでは、Lerpzは3つの再生ポイントのうちの1つに出現します。Lerpzがポイントの1つに触ると、その再生ポイントがアクティブになります。



Lerpzがアクティブな再生ポイントに立っている。

再生ポイントは、RespawnPrefabプレハブオブジェクトのインスタンスです。(プロジェクトペインのPropsフォルダの中にはあります)

このプレハブは、テレポート基地のモデルで、3つのパーティクルシステムとスポットライト、その他の奇妙なものから成っています。基本的な構造は次のとおりです：

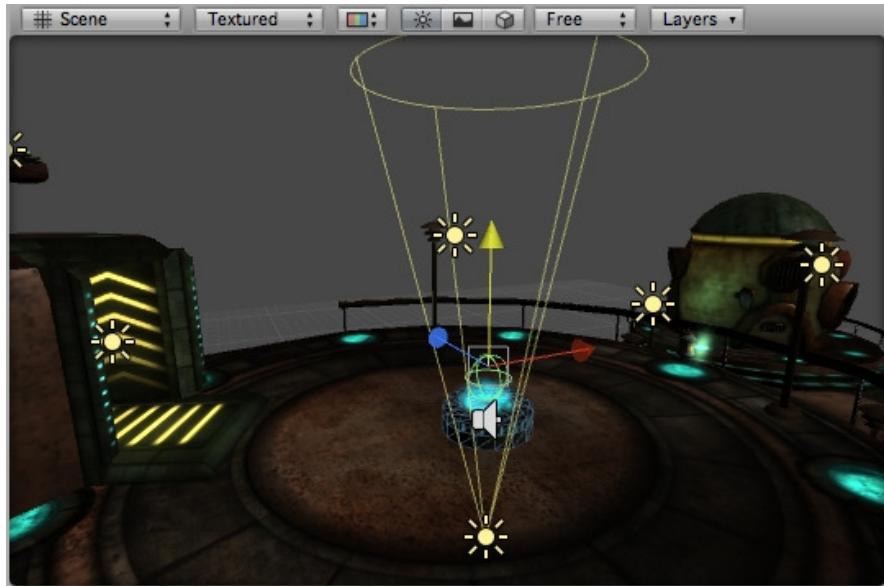
- **RSBase**はモデル自体を含みます。短い円柱の土台と、中心部に青色に光るディスクからなります。
- **RSSpotlight**はモデルの表面から、青いライトで照らすスポットライトオブジェクトです。青いテクスチャが光っているような効果を作り出します。
- 残りのゲームオブジェクトはパーティクルシステムです。**RespawnPrefab**オブジェクトに付加された**Respawn**スクリプトはパーティクルシステムをプレハブの状態に応じて切り替えます：
 - 再生ポイントがアクティブでなければ、小さな、微かなパーティクル効果が、明るい青い霧のように表示されます。これは、**RsParticlesInactive**に含まれています。
 - 再生ポイントがアクティブならば、大きな、仰々しい効果が表示されます。これは、**RsParticlesActive**に含まれています。
- 一度にレベル中の再生ポイントは、1つだけがアクティブになります。プレイヤーが再生ポイントに触れるとき、コライダオブジェクト(引き金としてセットされている)が検出し、再生ポイントのアクティビティ化を行います。
- 残り3つのパーティクルシステム、**RSParticleRespawn1**、**RSParticleRespawn2**、**RSParticleRespawn3**はプレイヤーが再生ポイントに再生すると、有効化されます。ワンショットパーティクルシステムが使われています。スクリプトがそれを実行すると、**RsParticlesActive**パーティクルシステムが再ストアされ、処理が完了します。

プレハブは、再生ポイントの状態をコントロールする、**Respawn**スクリプトを含んでいます。しかしながら、ゲームにおいて、プレイヤーが死亡したときに戻ってくる、特定の再生ポイントを知るために、マスター コントロールスクリプトの下に、再生ポイントを配置しなければなりません。やってみましょう：

-  **RespawnPrefab**をシーンビューにドラッグします
-  次のページにある図のような位置に配置してください。
-  インスタンスの名前を**Respawn1**に変更します。
上のステップをもう2回繰り返します。どこでも好きな場所に配置してください。もっとたくさん置いてもかまいません！レベルの奥のほう、アリーナの近くと、プラットフォーム上方の庭の木の近くに配置すると良いでしょう。

次のステップは、コンテナとなるゲームオブジェクトを作ることです。

-  **RespawnPoints**に名前を変更します。
-  全ての再生ポイントのインスタンスを**RespawnPoints**の子要素にします。



一つ目の再生ポイントを配置(Lerpzはショットから外れた位置に移動)

どのように動作するのか(How it works)

シーンがロードされるとUnityはStart()関数を、全てのRespawnスクリプトに対して呼び出します。ここで、いくつかの重要な変数が初期化され、他の要素へのポインタがキャッシュされます。

static変数が重要なメカニズムのキーです：

```
static var currentRespawn : Respawn;
```

これは、currentRespawnという、グローバル変数を定義します。

Staticキーワードは、スクリプトのインスタンスによって、その変数が共有されることを示します。これで、どの再生ポイントが現在アクティブなのかを追跡できます。しかし、シーンが開始した時点では、どのポイントもアクティブではないため、シーンにおいてデフォルトとなるものを設定しておく必要があります。Unityインスペクタはstatic型の変数は表示しませんので、スクリプトが、Initial Respawn属性を、それぞれのインスタンスで設定しなければなりません。

- デフォルト再生ポイントをInitial Respawn属性の上にドラッグします。
これを、シーン中の全ての再生ポイントで繰り返します。(チュートリアルプロジェクトの場合、デフォルトは、牢屋のすぐ近くにある、プレイヤーのスターとポイントとなる、Respawn1とします)

NOTE オリジナルのプレハブに直接これらの値を設定することはできません。

プレイヤのコライダによって、再生ポイントがアクティブになると、ポイントの**Respawn**スクリプトが、古い再生ポイントを非アクティブにし、次に、**currentRespawn**を新しいポイントにします。**SetActive()**関数がパーティクルシステムとサウンドエフェクトを操作します。

プレイヤキャラクタの再生は、プレイヤのゲーム状態のほとんどを管理する、**ThirdPersonStatus**スクリプトによって、扱われます：

 **ThirdPersonStatus**スクリプトをPlayerゲームオブジェクトに追加します。スクリプトは *Scripts->Player->ThirdPersonStatus*にあります。

再生ポイントスクリプトは、サウンドエフェクトも操作します。オーディオソースとして、それぞれの **Respawn** プレハブに付加されたものを除いて、ワンショットサンプルとして再生されるものです。このコンポーネントは、ループする、"アクティブ"なサウンドを保持しています。スクリプトは単純に、プレイヤが復活したり、復活ポイントをアクティブにしたり、復活ポイントが非アクティブ化されたときなど、必要に応じて、サウンドを有効化、無効化します。

NOTE Unityはサウンドエフェクトを追加する簡単すぎる方法を作りました。このようなアセットを追加したい場合、どのように使われるかを良く考えてください。たとえば、"respawn deactivated"サウンドを追加していないので、まだそれが再生されたのを聞いたことがありません。2つの復活ポイントを音の届く範囲には配置していないでしょう。もし、このプロジェクトをマルチプレイヤのゲームに変換したいと思えば、必要な音を追加し、スクリプトで操作することが必要になります。

スクリプトは複雑ではないので、簡単に読み進めることができるでしょう。復活ポイントには次のチャプタで、もう一度戻ってきます。

シーンの設定(Setting the Scene)

私たちのヒーローは動けるようになりました
ので、何か目的をあげましょう…。



初めの一歩 (First Steps)

このセクションでは、アクションが行われるゲームワールドの構築について見ていきます。映画においては、セットをつくり、小道具を配置し、台本を書くことで、ヒーローが行動できるようにします。

初めのステップはステージを用意することです。チュートリアルファイルには、すでに、基本的なレベルとなるメッシュを配置し、集めることができるアイテムをばら撒いてあります。ここでは更に、アイテムや要素を追加します。しかしこれらの作業を全て行うと、非常に時間がかかるので、事前に大部分は配置してあります。

ライティング (Lighting)

レベルにはすでに環境光が、多数のポイントライトとして配置されています。これらは、シーンやシーン内の物体、プレイヤーキャラクタ、敵、限られた範囲の拾えるアイテムをライトアップします。

このプロジェクトでは、ライトは、このレベルをモデリングしたアーティストによって、すでに配置されています。ライティングは、その場所の雰囲気を作り出す上で非常に重要ですので、モデリングを行ったアーティストによって、ライトは配置されるようにするのが、通常は良いでしょう。

アイテムの配置 (Placing Props)

数多くのアイテムがすでに配置され、最適なセットが準備されています。

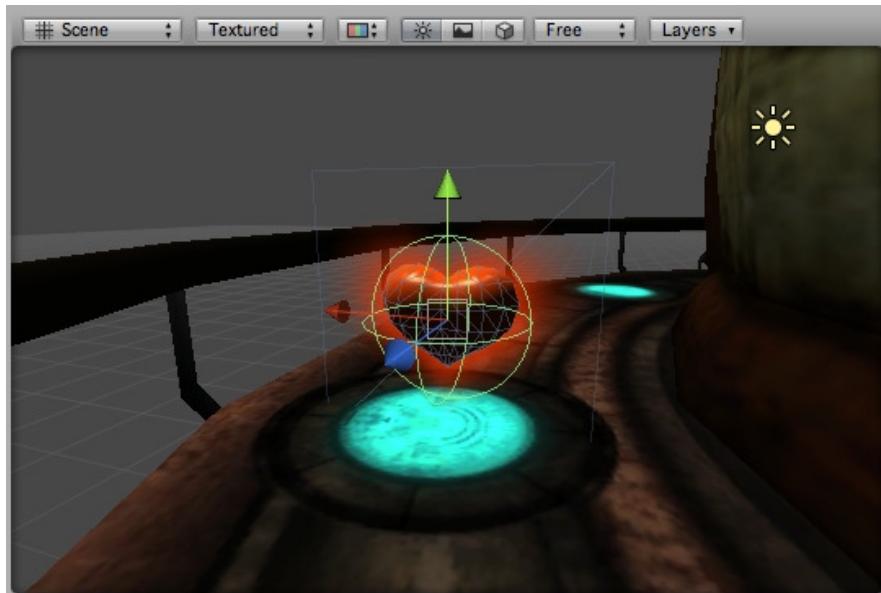
自分なりのレベルを作る(Building Your Own Levels)

チュートリアルのレベルはMayaによって、シーンのコンポーネントがアレンジされ、Unityにインポートされました。レベルの変更を試してみたり、追加のレベルを作ったりしたければ、プロジェクトペインの中の**Build Your Own!**フォルダの中に、個々のシーン要素が含まれています。

とても多くのオイル缶がアイテムとしてあるので、それをワールドに全て配置すると、非常に退屈なチュートリアルになってしまいます。ここまでチュートリアルを読んでいれば、どのようにして配置するかはわかっているはずなので、ここでは配置するものを限定します。"Health"アイテム、ジャンプパッド、復活ポイントを配置します。

体力アイテム (Health Pickups)

拾うことのできる体力アイテムを配置する、簡単な作業からはじめましょう。回転しながら光っているハート型のアイテムは、プレイヤに体力を追加します。アイテムはプレハブとして、プロジェクトペインに入っています。"Props"フォルダから、**HealthLifePickUpPrefab**オブジェクトを見つけられるでしょう。



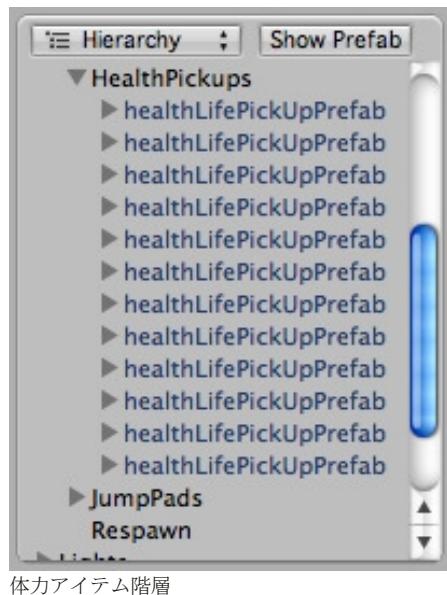
体力アイテムの配置

1つをシーンビューにドラッグし、Unityの位置設定ツールで、レベル中の好きなところに配置します。

この操作を、6つ程度のアイテムが配置できるまで繰り返します。

どこに配置するかは、あなたしたいですが、見つけたり集めたりするのに、簡単すぎないほうが良いでしょう。プレイヤがどのようにレベルをプレイするかを考え、どこにアイテムを配置するのが最も良いかを判断しましょう。あまりに気前よく配置しすぎると、ゲームが簡単になりすぎてしまいます。

最後に、アイテムをフォルダにグループ化し、階層ペインにおいて散らばってしまうのを避けましょう。*GameObject->Create Empty*メニューから、新しいゲームオブジェクトを作ります。新しいオブジェクトの名前をHealth Pickupsに変更し、体力アイテムをグループ化します。次のように：



体力アイテム階層

バリアフィールド (The Force Field)



バリアフィールド

現時点では、バリアフィールドが、我々のヒーローの宇宙船を閉じ込めていますが、アニメーションしていません。ただのメッシュテクスチャです。結果的に、残念なビジュアルになっています。

適切な視覚効果をさせるには、いくつかの方法がありますが、どれを選べばよいのでしょうか。単純な解決法が最も良いことは、良くあります。単純にテクスチャのUV座標を動かすことになります。これで、素晴らしいバリアフィールドの効果が得られます。

アニメーションは、短いスクリプトによって行われます。スクリプトはプロジェクトペインの **Scripts->Misc** フォルダにあります。 **FenceTextureOffset** と言う名前で、次のようになっています：

```
var scrollSpeed = 0.25;

function FixedUpdate()
{
    var offset = Time.time * scrollSpeed;
    renderer.material.mainTextureOffset = Vector2 (offset,offset);
}
```

はじめの行は、**Scroll Speed** 属性を Unity の GUI から直接編集できる状態にします。 **FixedUpdate()** 関数は、Unityにより、1秒間に設定された回数呼び出されます。 **Scroll Speed** の値と現在の時間を掛け合わせる短い式によって、テクスチャのオフセットを定めます。

素敵な属性 (Pretty Properties)

Unity のインスペクタが属性や、値を表示するとき、名前を見栄えが良くなるように調整します。Unity は、名前から大文字を見つけ出し、その直前にスペースを挿入します。加えて、名前の 1 文字目を大文字に変換します。そのため、**scrollSpeed** は **Scroll Speed** と表示されるのです。

テクスチャがレンダリングされる場合、通常、テクスチャはただの画像です。マテリアルの **mainTextureOffset** 属性は、Unity にテクスチャ画像の UI 空間ににおけるオフセットを教えます。これにより、複雑なアニメーションシークエンスを使うことなく、非常に効果的な結果を得ることができます。

levelGeometry ゲームオブジェクトを開き、レベルデータのことなる要素を見てみましょう。
impoundFence オブジェクトの上のフェンスをアニメーションさせる必要があるので、
fenceTextureOffset スクリプトを **impoundFence** オブジェクトにドロップします。

集取できるアイテムのスクリプト (Scripting the Collectable Items)

現時点では、Lerpz は レベル中のアイテムを拾うことはできません。なぜなら、Unity は私たちのヒーローが、アイテムを拾えることを教えられていないからです。2つの要素をそれぞれの取得できるアイテムに追加する必要があります：

- コライダコンポーネント
- コライダとプレイヤの体力更新などを扱うスクリプト

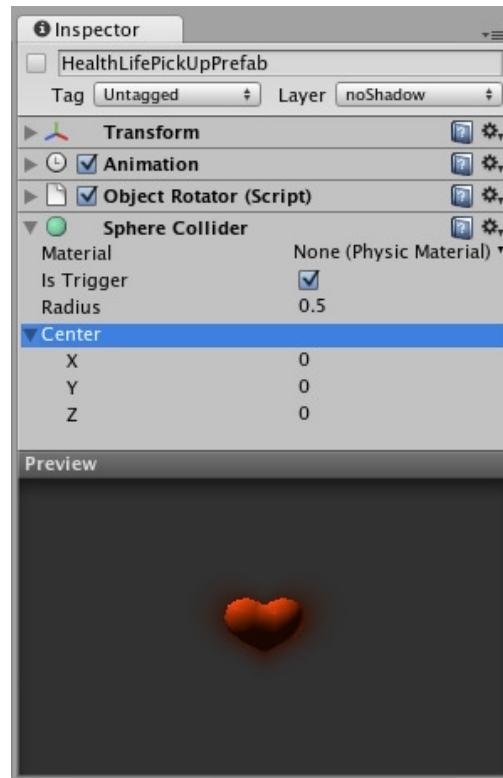
階層ペインの取得可能なアイテムは、すべてプレハブインスタンスであり、青く表示されています。オリジナルのプレハブを直接変更すれば、自動的に、ゲーム内すべてのアイテムを更新出来ます。

私たちのヒーローが取得できる2つのプレハブは、FuelCellPrefabとHealthPickUpPrefabです。プロジェクトトペインの中のPropsフォルダに置かれています。

 HealthPickUpPrefabオブジェクトのルートを選択します。

Component->Physics->Add Sphere Colliderを使って、スフィアコライダをプレハブに追加します。インスペクタに現れたことを確認してください。

 最後に、Triggerチェックボックスをチェックします。



インスペクタに表示されたHealthPickUpPrefab

NOTE ObjectRotaterスクリプトもプレハブに負荷されています。これは単に、アイテムをその場所で回転させるシンプルなものです。もっと複雑なスクリプトによるアニメーションを、後のチャプターで扱います。

コライダには2種類の使い方があります：何かとぶつけることも出来ますし、トリガとして使用することも出来ます。

トリガ (Triggers)

トリガは目に見えないコンポーネントで、名前が示すとおり、イベントを引き起こします。Unityではトリガはシンプルなコライダで、Trigger属性が設定されたものです。何かがトリガに衝突すると、物理的な動作をする代わりに、仮想的なスイッチのような働きをします。

トリガは、何かが衝突すると、次の3つのうち、1つのイベントメッセージを送出します。

OnTriggerEnter()、OnTriggerStay()、OnTriggerExit()です。

トリガのイベントメッセージは、トリガオブジェクトに付加された全てのスクリプトに送られます。さて、体力プレハブに適切なスクリプトを追加しましょう：

- ☞ コンポーネントメニューに行き、Third Person PropsサブメニューからPickupスクリプトを選択します。これで、Pickupスクリプトが、プレハブに追加されます。
- ☞ Pickup Type属性をインスペクタでHealthに設定します。
- ☞ 最後に、Amount属性を3などに設定します。この量は、プレイヤが取得したときに、獲得する体力の量です。

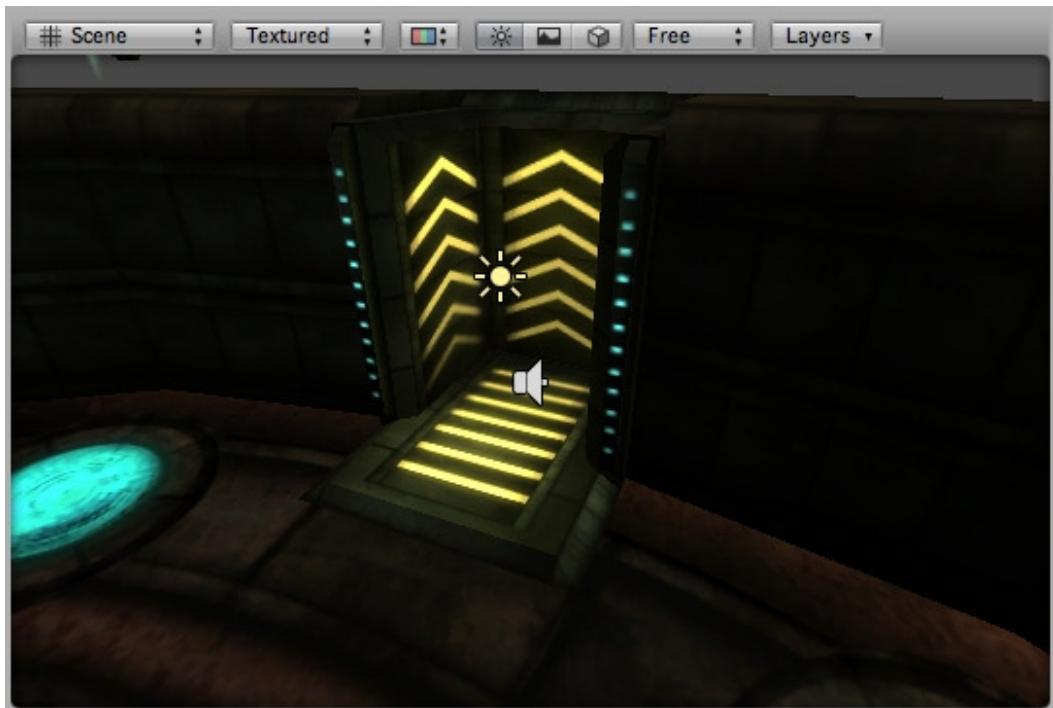
どのぐらいの体力？(How much health?)

ヘッドアップディスプレイ、またはHUDが、プレイヤの現在の体力レベルや、ライフを表示しています。これは、最大体力6までしか扱えません。もし、プレイヤが、すでに体力が最大の状態で、体力アイテムを取得したらどうなるのでしょうか？これは、好みの問題ですが、ここでは、追加のライフを与えることにしました。このためのロジックは、プレイヤの状態をチェックするスクリプトのThirdPersonStatusにおいて、定義されています。

オイル缶の取得は、ほぼ同じ方法で設定されますが、2点だけ異なります：

- Pickup Type属性はFuelCellに設定します。
- Amount値は、取得したオイル缶の数が反映されるように設定します。(1が良いでしょう)

ジャンプ台 (Jump Pads)



ジャンプ台

ジャンプ台は、レベル中の、明るく、黄色と黒の縞模様になった空間です。これは、**LerpZ**を加速させ、空中に投げ上げます。この目的のために、スクリプトを付加したコライダを利用します。

まず、空のゲームオブジェクトを作り、**Jump Pad Triggers**という名前になります。これを、ジャンプ台の鳥がオブジェクトを保持するフォルダとして用います。

では、プレハブを作りましょう：

- 【】 空のゲームオブジェクトを作ります。
- 【】 オブジェクトの名前を**JumpPad Trigger 1**に変更します。
- Box Colliderオブジェクトを、それに追加します。(Sphere Colliderも理論上は動作するはずですが、Box Colliderがジャンプ台の形状に最適です)
- 【】 コライダをトリガとして設定します。
- 【】 JumpPadスクリプトを追加します。

これで、オブジェクトが作成されました、これをプレハブにします：

- 【】 プロジェクトペインの上のCreateメニューから**Prefab**を選択します。



Jump Padゲームオブジェクトを新しいプレハブにドラッグアンドドロップします。



プレハブの名前をJumpPad Triggerに変更します。



オリジナルのゲームオブジェクトを階層から削除します。



JumpPad Prefabをシーンにドラッグし、ジャンプ台の位置に配置します。(6つの場所があります。4面ビューレイアウトを使って、位置を合わせると良いでしょう)

デフォルトのジャンプ台のJump Height設定は5ですが、これはLerpzを庭のレベルに投げ上げるには十分ではありません。だいたい、15から30の値をジャンプ台に設定するとよいでしょう。(最終的なプロジェクト例では、30に設定しています)



最後にプレイをクリックし、新たに設定したトリガが、正しく動作しているかをテストして、確認しましょう。



NOTE スクリプトはプレハブにも同様に働きます：*Props*フォルダの**Pickup**スクリプトとプレハブにリンクを追加しました。プロジェクトペイン中の1つを編集すると、シーン内のコピーすべてに影響します。

作業手順を、スムーズで、面倒で無くするためには、良く整理しておくことが大事です。

- 可能な限り、プレハブのインスタンスを使いましょう。
- 種類では無く、機能で整理しましょう。
- 空のゲームオブジェクトをコンテナとして利用しましょう。

たとえサイズの小さなゲームであっても、非常に多くのアセットが必要であることに、きっと驚くでしょう。

GUI

幾つのライフが残っているのでしょうか?
Lerpzの体力レベルは幾つですか? GUIの
時間です。



ユーザインターフェース

ゲームは通常、メニュー、オプションスクリーンなど、グラフィカルユーザインターフェース(GUI)を持っています。さらに、通常ゲームは自身に覆いかぶさるようなGUIを持ちます。スコアを角に表示するようなシンプルなものから、アイコンや、持ち物を表示したり、体力を表示したりするバーなど、こったるものまであります。

Unity 2 では新しいGUIシステムが導入され、ゲームのGUI構築が容易になりました。そのシステムを「Lerpz の脱出」に使います。

NOTE 古いシステムもUnityには残っていますが、将来的に非推奨になるでしょう。それについては、チュートリアルでは扱っていません。

Unity 2 の新GUIシステム

以前は、Unityにボタンを描画させ、Unityが関連するメッセージを送信し、スクリプトが、ユーザのボタン選択、クリック、解除に合わせて処理を行っていました。

古いシステムは、古典的なイベントドリブンGUIモデルでしたが、Unity 2 は全く新しいGUIシステムを導入しました。イミディエイトモードGUIと呼ばれます。古典的なGUIシステムを使っているならば、イミディエイトモードGUIのコンセプトはショッキングなものかもしれません。

ここに、例があります：

```
function OnGUI()
{
    If (GUI.Button (Rect(50, 50, 100, 20), "Start Game") )
        Application.LoadLevel("FirstLevel"); // load the level.
}
```

OnGUI()がゲームサイクルで、最低2回は呼び出されます。1度目の呼び出しで、UnityはGUIを構築し、それを描画します。ここでは、"Start Game"と書かれたシンプルなボタンが、指定された位置に描画されます。

二度目は、ユーザ入力が行われた場合です。ユーザがボタンをクリックすると、ボタン描画を囲んでいるIf(...)条件節が真(true)となり、Application.LoadLevel()が呼び出されます。

他のGUI要素である、ラベル、グループ、チェックボックスなども、同様に動作さいます。関数は、真偽値、またはユーザ入力を適切に返します。

明らかにアドバンテージは、無数のイベントハンドラをGUIのために必要としないことです。それらはすべて、OnGUI()関数に含まれています。

Unity 2 は2種類のイミディエイトモードGUI関数を提供します：基本となるGUIクラスは、上記の例のように使用されます。GUILayoutクラスは、GUIのレイアウトを配置するのに使用されます。

詳しい情報

Unityの新しいGUIシステムについての詳細は、以下のリンクから見ることができます：

- <http://unity3d.com/support/documentation/Components/GUI%20Scripting%20Guide.html>
- <http://unity3d.com/support/documentation/ScriptReference/GUI.html>

ゲーム内HUD

私たちのゲームではゲーム内GUIを使って、プレイヤの体力、残りライフと、取得しなければならないオイル缶の数を表示します。グラフィカルな要素はすでにプロジェクトに含まれています。

GUIは、新しいGUIコンポーネントを用いて様々な要素をレイアウトする、GameHUDスクリプトによって、操作されます。このスクリプトは、シーン特有の要素を保持する、Level GameObjectに負荷しなければなりません。(単純にNearCameraオブジェクトに追加したり、「GUI」ゲームオブジェクト自身に追加したりもできます。重要なゲームデザインと言うよりは、個人的な趣向の問題です)

Level GameObjectをレベル特有の状態やその他のスクリプトを管理する物として使用します。

GUI Skin オブジェクト

Unity 2 の新しいGUIシステムはスキンをサポートしています。これにより、GUI要素の完全なコントロールが可能になります。GUIのスキンを変更すれば、ボタンの形状、画像、フォント、色等をその他すべてのGUI要素に適用出来ます。また、スクロールバー、テキスト入力ボックス、ウィンドウに対しても行えます。

ゲーム内GUIはグラフィカル画像の上に、`GUI.Label()`関数をもじいてゲームのGUIを作成します。しかし、残りライフとオイル缶の数を表示するために、独自のフォントがHUDに必要です。

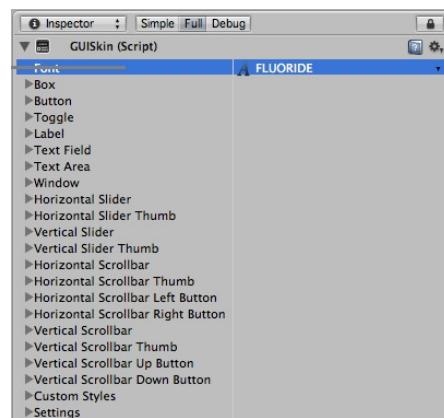
GUISkinアセットが、UnityのGUIの'見た目'を決定します。ウェブサイトに対するCSSファイルのような役割です。オブジェクトはデフォルトの見た目を変更したい場合に必要となります。フォントを変更しますので、**GUISkin**をシーンに追加する必要があります。

 Assetメニュー命令を使い、新しいGUI Skinオブジェクトを作成します。プロジェクトペインにデフォルトのUnity GUI スキンデータを保持した状態で追加されます。

 新しいGUI Skinオブジェクトの名前を、`LerpzTutorialSkin`に変更します。

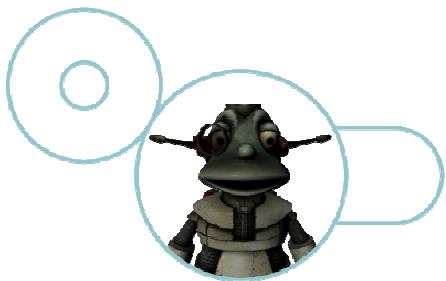
デコレーションされたフォントである"Flouride"をゲームに使用します。デフォルトスキンへの変更はこれだけです。

 Flourideフォントオブジェクトを新しいGUI SkinアセットのFont属性にドラッグします。



GUI Skinにフォントを設定

GUI Skinオブジェクトは階層ペインに追加しません。代わりに、GUI SkinをGameHUDスクリプトに、直接追加します。GUI Skinとともに、GameHUDスクリプトはどのアセットがGUIディスプレイを作るのに必要なかを知らなければなりません。これには、`GUIHealthRing`アセットが含まれます。



GUIHealthRing画像

NOTE Unityから画像に関して、イメージが”2の乗数”ではないことに、いくつかの警告が出ていることに気が付くでしょう。多くのグラフィックカードは画像が2の乗数であることを期待します。実際の画像データの大きさにかかわらず、それによって、計算速度が最適になるからです。GUIは通常このレベルの最適化は必要ありません。そのため、ここで用いるアセットは最適化されたものではありません。画像のサイズが、2の乗数であるかどうかを知りたければ、イメージをインスペクタに表示してください。最適化を行うことができます。

このイメージはLerpzの体力上方を表示するために使われます。右側のスペースはLerpzの残りライフを表示します。左側の円は、残り体力をパイチャートで表示します。パイチャートは6つの2次元画像の配列から、適切なものを表示します。名前はhealthPie1からhealthPie6です。次の画像です：



healthPie5画像

NOTE これらの画像は、アルファチャンネルによって透明度が指定されています。

分割された画像を使うことで、Lerpzの体力に応じて、適切な画像を表示するだけでよくなります。複雑なプログラムで、画像を回転させて描画させる必要がなくなります。

2つ目の重要なGUI要素は、オイル缶の状態を表すものです。メイン画像はGUIFuelCellです：



GUIFuelCell画像

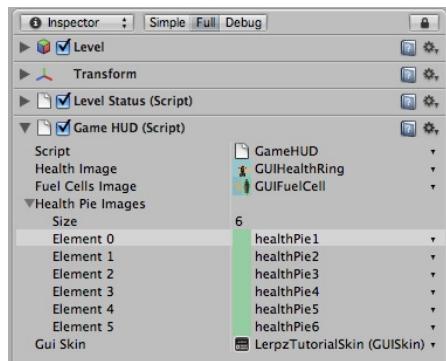
これは、ゲームスクリーンの右下に表示され、レベルを開錠するために必要な、オイル缶の残り数を表示します。

- 【A】 GameHUDスクリプトをLevelゲームオブジェクトに追加します。
- 【B】 Levelゲームオブジェクトを選択し、インスペクタを確認します。Game HUD(Script)コンポーネントが追加されているはずです。
- 【C】 GUIHealthRingとGUIFuelCell画像をGameHUDスクリプトに追加します。
- 【D】 Health Pie Images属性を開きます。

Health Pie Imagesは配列です。Unityは今のところ、大きさをいくつにするべきかわかりませんので0が設定されています。6つのパイチャート画像を追加しますので、値を変更する必要があります。

- 【E】 Size横の“0”をクリックします。“6”に変更します。Element 0からElement 5までの、6つの空エントリが表示されているはずです。
- 【F】 プロジェクトペインで、GUIアセットフォルダを開き、health pie 画像を見つけます。1から6まで6つの画像があります。
- 【G】 コンピュータは0から数えますので、healthPie1はElement 0に配置します。healthPie2はElement 1へ。それぞれの画像をドラッグして追加します。

最後にLerpzTutorialSkin GUIスキンを空のGui Skinスロットに追加します。設定は次のようになっているはずです。



GameHUDスクリプト設定

ゲームを実行すると、HUDがプレイエリアに表示されています：



ゲーム内HUD

解像度非依存

```
GUI.matrix = Matrix4x4TRS (Vector3.zero, Quaternion.identity, Vector3
(Screen.width / 1920.0, Screen.height / 1200.0, 1));
```

GUIの問題の1つはサイズです。上のスクリーンショットは24インチのiMacにおいて、解像度1920 x 1200で動作させたものです。

現在のディスプレイスイズと解像度に合わせて、HUDのサイズを動的に変更する必要がありますが、どうしたらよいのでしょうか？

Unity 2 の新しいGUIシステムは、トランスマット行列をサポートしています。この行列は全てのGUI要素のレンダリング前に適用されますので、位置を変えたり、回転したり、サイズを変更したり、どのような状況でも動的に行うことができます。

GameHUDスクリプトから抜粋した上記の行が、方法を示しています：

Game Viewから、"Maximize on Play"オプションを無効化し、アスペクト比を4：3に設定すると、GUIのサイズが変更されフィットするのが確認できます。

HUDを回転させたり、上下を反転したり、ズームインしたりも望みどおりにできます。ハイスコアを表示するゲームメニューなどでは、これらは便利な特性です。このトリックはレベル完了のシーケンスにおいて使用します。

スタートメニュー

全てのゲームにはスタートメニューが必要です。これは、ゲームが始まったときや、プレイヤが設定を変更、保存したゲームを読み込む場合に使います。最も重要な場合として、ゲームを始めるときに使います。この章では、スタートメニューをスクラッチから構築していきます。

NOTE スプラッシュスクリーン、メニューなどはすべて、Unityのシーンです。そのため、ゲームレベルは通常シーンですが、シーンは常にゲームレベルではありません。1つのシーンにおいてスクリプトを用いて、他のシーンを読み込んで実行し、シーンをつなぎます。

スタートメニューにはいかが必要です：

- 2つのGUIテキストボタン："Play"と"Quit"
- ゲームの名前。独自フォントを使ってレンダリングします。
- いい感じの音楽
- いい感じの背景

言い換えれば、次の画像のようなものです：



スタートメニュー

シーンの設定

最初のステップは新しい、空のシーンを作ることです。

【】 Command+N[Mac](Ctrl+N[Win])をタイプしてシーンを作成し、Command+S[Mac](Ctrl+S[Win])で保存します。

【】 名前をStartMenuとします。Unityは自動的にカメラをシーンに追加してくれますが、いまのところ、何も見るものはありません。

では、新しいGUIシステムを使い、メニューを構築しましょう：

【】 プロジェクトペインから空のJavaScriptファイルを作成します。

【】 名前をStartMenuGUIに変更し、エディタで開きます。

初めに、Unityスクリプトディレクティブを記述します。ディレクティブはUnityに、スクリプトに関する情報や、追加の指示を与えます。これらは、Javascriptのコマンドの一部ではなく、Unity自身に向けたものです。完全に記述されたスクリプトコードは付録の章にあります。

ここでは、Unityにスクリプトをエディタ中で実行させます。これによって、プロジェクトを毎回停止したり、再実行したりすることなく、結果をすぐに確認できます：

```
// Make the script also execute in edit mode
@script ExecuteInEditMode()
```

LerpzTutorialSkinアセットとリンクさせなければいけないので、初めのラインは次のようにになります：

```
var gSkin : GUISkin;
```

背景のためにTexture2Dオブジェクトが必要です。(インスペクタを利用して、背景画像をこの属性にドロップします)

```
var backdrop : Texture2D; // our backdrop image goes in here.
```

“Loading...”メッセージをプレイヤが“Play”ボタンをクリックしたときに表示したいので、これを管理するためのフラグが必要です：

```
private var isLoading = false; // if true, we'll display the "Loading..." message.
```

さいごに、OnGUI()関数を記述します：

```
function OnGUI()
{
    if (gSkin)
        GUI.skin = gSkin;
    else
        Debug.Log("StartMenuGUI: GUI Skin object missing!");
```

上記のコードは、有効なGUI Skinオブジェクトにリンクが張られていることを確認します。リンクが存在しなかつた場合、Debug.Log()関数がエラーメッセージを吐き出します。(外部的なリンクやデータをこの方法でチェックすることは、デバッグを容易にする上でよいことです)

背景

背景画像は、GUI.Label属性に設定された背景画像を要素の背景画像として使います。テキストは何も設定されず、サイズは常にディスプレイ全体です。これによってスクリーンサイズに適合します。

```
var backgroundStyle : GUIStyle = new GUIStyle();
backgroundStyle.normal.background = backdrop;
GUI.Label ( Rect( (Screen.width - (Screen.height * 2)) * 0.75, 0, Screen.height * 2,
Screen.height), "", backgroundStyle);
```

はじめに、新しいGUIStyleオブジェクトを定義し、デフォルトのGUIスキンを上書きします。このインスタンスでは、"normal.background"スタイル属性のみを、指定した背景画像を使うように変更します。

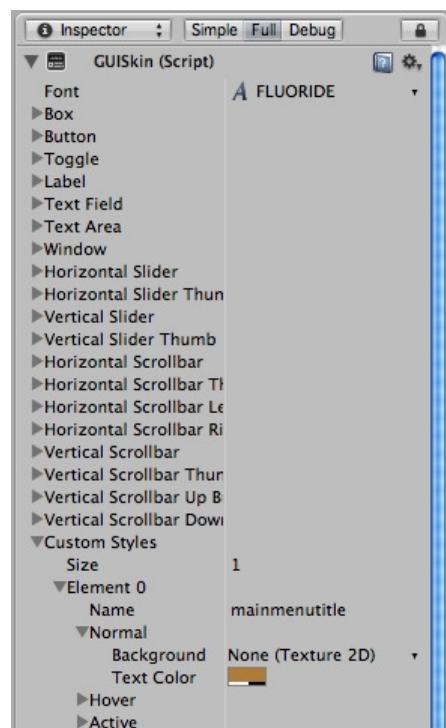
GUI.Label()関数はRectオブジェクトを引数として受け取ります。この矩形のサイズはディスプレイのサイズから決定されるため、画像は常にスクリーン全体に表示されます。画像のアスペクト比も考慮されますので、切り取られたり、サイズが変更されたりすることで、ゆがみなく追加されます。

つぎに、タイトルテキストです。スクリプトコードを書く前に、GUISkinを見直し、小さな変更を加える必要があります。

メニューはLerpzTutorialSkinアセットで定義されたデフォルトのフォントを使用します。ここで、テキストを表示するためのデフォルトスタイルは、大きなゲームタイトルを表示するに適していないので、新しい独自GUIスタイルをスキンに追加します。名前はmainMenuTitleです：

 プロジェクトペインに行き、LerpzTutorialSkinをクリックして、インスペクタに詳細を表示します。

独自スタイルをGUISkinに追加します。



独自スタイルをGUISkinオブジェクトで定義

 "Custom Styles"を開き、"Size"を"1"に設定します。"Element 0"のエントリが表示されます。



“Element 0”を開き、上の図のように設定します。特に：テキストの色を茶色いオレンジに設定します(表示されていない属性については、気にする必要はありません。デフォルトの値のまま残しておけば大丈夫です。)

最後に、残っているコードをスクリプトに記述します。

```
GUI.Label ( Rect( (Screen.width/2)-197, 50, 400, 100), "Lerpz Escapes",  
"mainMenuTitle");
```

NOTE GUIスタイルの名前は、スクリプト内でアクセスしようとするときには、大文字小文字が考慮されません。“mainMenuStyle”と“mainmenustyle”は同じ意味になります。(原文には上記のように書かれていますが、大小文字が原因でエラーが発生した場合がありました)

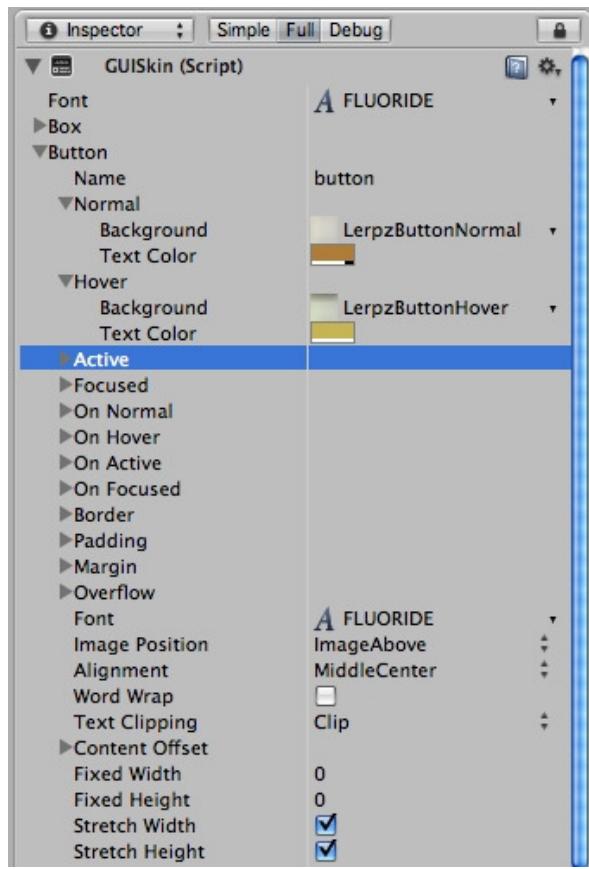
ボタン

LerpzTutorialSkinオブジェクトのGUI Button属性を変更して、いい感じのボタンを提供する必要があります。

メニューとゲーム内HUD(Heads-Up Display)において同じGUIスキンオブジェクトを使用しています。HUDではデフォルトフォントだけを変更しました。しかしながら、スタートメニューのボタンでは、ボタンのテキスト後ろにグラフィカルな画像が必要です。デフォルトのボタンデザインは、ゲームのスタイルに合っていません。変更する必要があります。

TIP GUI属性が、Hover、Focus、Active eventsに反応するためには、画像が空であっても、背景画像を設定しなければなりません。

プロジェクトペインでLerpzTutorialSkinアセットをクリックして、インスペクタに詳細を表示します。下に示されるように設定を変更します。使用しない他のGUI属性は無視してください：



ボタン画像をLerpzTutorialSkin GUIスキンオブジェクトに設定

さあ、"Play"ボタンを追加しましょう：

```
if (GUI.Button( Rect( (Screen.width/2)-70, Screen.height - 160, 140, 70), "Play"))
{
    isLoading = true;
    Application.LoadLevel("TheGame"); // load the game level.
}
```

上のコードで、"Play"ボタンを描画して、処理まで行えます。レンダリングとイベントハンドリングが同じ場所に記述されているので、わかりやすくなっています。GUI.Button()関数は、ボタンの位置とサイズを決めるRectオブジェクトとテキストを受け取ります。

ユーザがボタンをクリックすると、ボタン関数はtrueを返しますので、ゲームレベルを読み込むことができます。

isLoadingをtrueに設定することで"Loading..."テキストが表示されるようにし、続けてUnityにゲームレベルを読み込ませます。

TIP 新しいGUIシステムは要素を描画するための複数の関数をサポートします。テキストの代わりに画像を使うことや、テキストと画像とツールチップを組み合わせることもできます。詳細はドキュメントを参照してください。

“Quit”ボタンも同様に処理されますが、スタンドアロンで動作している(または、Unityのエディタ内)のか、チェックが行われます。

```
var isWebPlayer = (Application.platform == RuntimePlatform.OSXWebPlayer ||
    Application.platform == RuntimePlatform.WindowsWebPlayer);
if (!isWebPlayer)
{
    if (GUI.Button( Rect( (Screen.width/2)-70, Screen.height - 80, 140, 70), "Quit"))
        Application.Quit();
}
```

ここで解るように、“Play”ボタンと非常に似ています。ほんの少し下に描画し、最初に全てを書く必要があるかどうかをチェックします。

最後のステップは”Loading”テキストをユーザが”Play”ボタンを選択したときに表示することです。ゲームシーンの読み込みには数秒かかるため、この方法を用います。特に、Webプレイや内部で実行しインターネットからストリーミングする場合は時間がかかります。

ここでisLoadingを使用します：

```
if (isLoading)
    GUI.Label ( Rect( (Screen.width/2)-110, (Screen.height / 2) - 60, 400, 70),
                "Loading...", "mainMenuTitle");
}
```

再び、mainMenuTitle独自GUIスタイルを使用します。これで、テキストのスタイルがタイトルと一致します。(GUIスタイルは大小文字に依存しません。大文字にしてあるのは読みやすくするためです。)

“Quit”ボタンは、ゲームをスタンドアロンアプリケーションとして実行しているときのみ有効です。もし、WebプレイやダッシュボードウィジェットやUnityのエディタで実行されている場合、“Quit”ボタンは、何もしません。UnityのWebプレイは含まれているウェブページを終了できません。また、ブラウザを閉じてしまうのも良いアイディアではありません。同様に、“Quit”はダッシュボードウィジェットの場合、どうするべきでしょうか？終了してしまうとダッシュボードから消えてしまいます。(Webプレイを実行しているページを閉じることには議論の余地がありますが、これはWebプレイ自身が処理するほうが良いでしょう。)

ということで、ゲームがスタンドアロンで実行されていない場合は、Quitボタンを無効化し、表示しないようにしなければなりません。Unityエディタの内部で実行されている場合は1つの例外です。エディタ中では、ボタンが正しく表示されて、動作が正確に行われることをチェックしなければならないため、表示しないわけには行きません。

次のステップは、音楽の追加です。

- 🔊 プロジェクトペインに行き、Soundsフォルダを開き、StartMenuオーディオファイルを、階層ペイン中のMain Cameraオブジェクトにドラッグします。これでAudio Clipコンポーネントが追加されます。
- 🔊 Main Cameraオブジェクトをクリックし、インスペクタで新しいAudio Clipコンポーネントを見つけます。Play On AwakeとLoopのチェックボックスを有効化します。
- 🔊 StartMenuGUIスクリプトをMain Cameraに追加します。
- 🔊 最後に、StartMenuGUIコンポーネント中で、GSkinをLerpzTutorialSkinに追加、BackdropをStartSplashScreenに追加します。

シーンを実行すると、下図のようなものがゲームビューに表示され、短いオーケストラ調の音楽が繰り返し演奏されるはずです。



実行したスタートメニュー

これで終わりです！スタートメニューが完成しました！

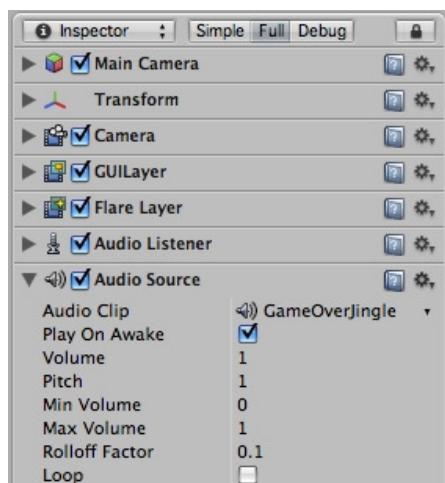
TIP 音楽はAppleのOrchestral Jam PackよりApple Loopsをつかって作られました。その後Garagebandでアレンジされました。これらはMacユーザには便利な音を組み合わせて作るツールです。どのような音楽スタイルがゲームに合うかを考えて作れます。PCユーザにはAudacityをお勧めします。

自分のプロジェクトでは、オプションメニュー・ハイスクアメニュー、マルチプレイヤーロビー等を追加したくなるでしょう。Unity 2 GUIを使用すれば、構築することが可能です。

ゲームオーバー

プレイヤーがゲームを完了するか、挑戦に失敗した場合に、ゲームオーバースクリーンが表示されます。スタートメニューと異なり、このシーンにはボタンや視覚的なユーザインタラクションはありません。ただ、"Game Over"メッセージを背景の上に表示し、短いジングルを再生するだけです。ジングルが終了するか、ユーザがマウスをクリックすると、自動的に"Start Menu"シーンを読み込みます。

- ▶ **はじめに**、新しいシーンを"GameOver"という名前で作成します。
- ▶ **GameOverJingle**オーディオファイルをデフォルトの**Main Camera**オブジェクトにドロップし、次のように設定します：



GameOverJingleの設定

このシーンにはエディタを使って追加するものは他にありません。デフォルトのカメラだけで十分です。

次のステップはスクリプトの構築です(完全なスクリプトは付録の章で見つけられます。)：

- ▶ **新しいJavascriptスクリプトアセットを作成し、名前を"GameOverGUI"**とします。
- ▶ **エディタで開き、下記のコードを追加します。**

スタートメニューと同様、Unityエディタでプロジェクトの実行中も見れるようにしたいので、次の二文を追加します：

```
@script ExecuteInEditMode()
```

スタートメニューでは、LerpzTutorialSkin GUIスキンアセットを使いました。GUIスキンは、GUIスタイルを決定し、それをGUI全体に適用することができます。

それとは違う方法として、個別のGUIスタイルオブジェクトを直接定義することができます。ゲームオーバースクリプトでは、3つの`GUILayout`変数を定義します。インスペクタを用いて、テキストのサイズ調整に使う2つの変数とともに、設定することができます：

```
var background : GUIStyle;
var gameOverText : GUIStyle;
var gameOverShadow : GUIStyle;
```

これらのGUIスタイルオブジェクトは、すぐ後でインスペクタを用いて設定します。

次にテキストの拡大縮小値を”Game Over”メッセージに設定しデフォルトフォントサイズより大きくレンダリングするために定義します。

メッセージは、影の付いたテキスト効果を出すために、違う色で2度書きます。そこで、2つのサイズ変更値を定義します：

```
var gameOverScale = 1.5;
var gameOverShadowScale = 1.5;
```

最後に`OnGUI()`関数です：

```
function OnGUI()
{
```

まず、背景です。スタートメニューで使用したのと同様にサイズ変更します：

```
    GUI.Label ( Rect( (Screen.width - (Screen.height * 2)) * 0.75, 0, Screen.height * 2,
        Screen.height), "", background);
```

次は、影付きの”Game Over”メッセージを表示します。テキストのサイズを拡大し、スクリーンの中心に表示する必要があります。運のいいことにGUIシステムのビルトイントランスクローム行列を使って、拡大縮小を操作できます。

TIP GUIトランスクローム行列は、気ままに望みどおりの変換を行えます：拡大縮小、回転、反転、ぐるぐる回すなど、思うがままです。

テキストを暗い影の付いた色で表示するために、`gameOverShadow` GUIスタイルを`GUI.Label`関数に渡します。

```

    GUI.matrix = Matrix4x4TRS(Vector3(0, 0, 0), Quaternion.identity, Vector3.one *
gameOverShadowScale);
    GUI.Label ( Rect( (Screen.width / (2 * gameOverShadowScale)) - 150,
(Screen.height / (2 * gameOverShadowScale)) - 40, 300, 100), "Game Over",
gameOverShadow);
}

```

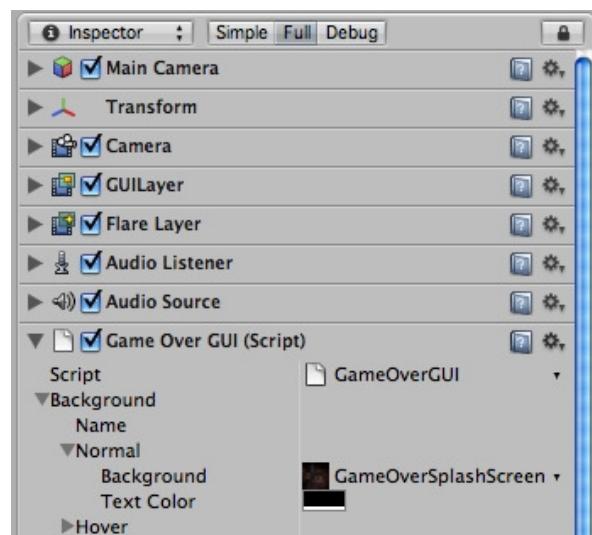
最後に、同じテキストを明るい色でもう一度書きます。UnityのGUIシステムは常にコードに現れた順番で要素を描画します。そのため、影の上にテキストが表示されます。`gameOverScale`値を使っても`gameOverTextGUI`スタイルを使っても差はありません。

```

    GUI.matrix = Matrix4x4TRS(Vector3(0, 0, 0), Quaternion.identity, Vector3.one *
gameOverScale);
    GUI.Label ( Rect( (Screen.width / (2 * gameOverScale)) - 150, (Screen.height / (2 *
gameOverScale)) - 40, 300, 100), "Game Over", gameOverText);
}

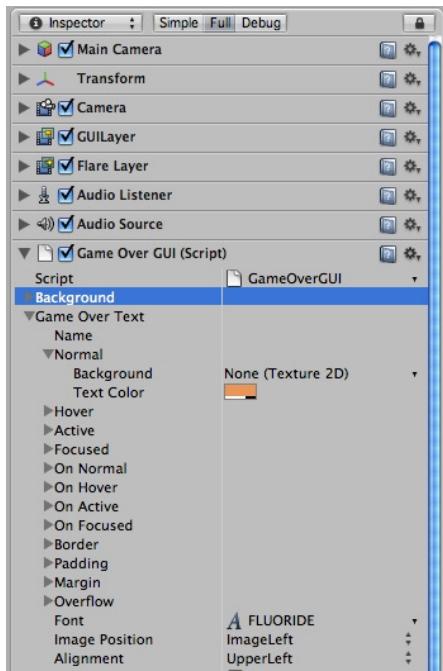
```

- ➡ スクリプトを保存して、Main Cameraにドロップします。
- ➡ Main Cameraオブジェクトをクリックしてインスペクタに表示します。値を設定するときです。
- ➡ 初めは背景GUIスタイルです。”GameOverSplashScreen”画像をNormal->Backgroundスロットに、次のようにドロップします：



適切な背景GUIスタイルの設定

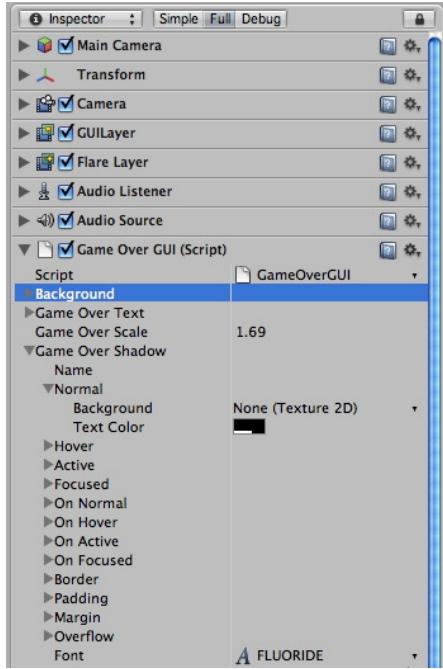
次に、Game Over Text GUIスタイルを次の図に示すように設定します。(その他の設定は、いつもどおり、そのままにして置いてください。)



適切な Game Over Text GUIスタイルの設定

ついに、Game Over Scaleを1.69に設定します。

Game Over Shadow GUI設定を行います：



適切な、Game Over Shadow GUIスタイル設定

最後に、Game Over Shadow Sizeを1.61に設定します。

下図のようなGUIがゲームビューに表示されるのを確認できるはずです：



最後にMain Cameraに2つ目のスクリプトを追加します。これで、音楽の再生が終了したことを確認し、スタートメニューのシーンを読み込みます。

新しいJavascriptスクリプトアセットを作成します。**GameOverScript**と名前をつけます。

スクリプトをエディタで開き、次のコードを追加します(完全なスクリプトコードは付録の章で見つけられます。)：

```
function LateUpdate ()  
{  
    if (!audio.isPlaying || Input.anyKeyDown)  
        Application.LoadLevel("StartMenu"); }
```

このコードはオーディオの再生が終了しているか、何かのキーがプレイヤによって押されたかをチェックし、"StartMenu"を読み込みます。

GameOverScriptを**Main Camera**に追加すれば終わりです。GUIが完成しました！

敵対者

敵対者無しには、いかなるゲームも完成しません。このチャプタでは、Lerpzが戦う敵を追加します。



敵対者と衝突

これら 2 つの要素はどんなゲームでも鍵となる要素です。Lerpz を油断させずにおく何かが必要です。ゲームデザイナの仕事は、障害をプレイヤの通り道に配置することですが、克服できるようにします。

Lerpz は 2 つの敵対者に遭遇します：ロボットガードとレーザバリアです。

レーザトラップ

レーザトラップはレーザ通路に配置されて降り、プレイヤがビームに触るとダメージを与えます。下の図が、そのうちの 2 つを示しています。これらを、牢屋を出てからレベルの最後の方にある、アリーナの両側にある、短い通路構造に配置します。



Lerpzが2本のレーザトラップに遭遇

レーザは上がったり下がったりします。プレイヤがレーザに当たってしまうと、体力をいくらか失ってしまいます。

レーザトラップの実装

それぞれのレーザトラップは、同じ縦の平面を、単純に上がったり下がったりするビームです。Lerpz(または、敵)がビームにあたると、ダメージを受けます。

レーザビーム自体は、**Line Renderer**コンポーネントがそれ自身のゲームオブジェクトに含まれたものによって、実現されます。移動とロジックは**LaserTrap**スクリプトに定義されます。1つめのレーザトラップを作成しましょう：

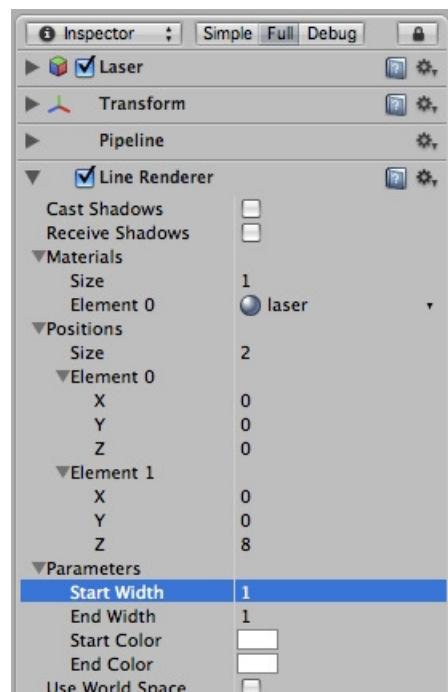
- 【】 空のゲームオブジェクトを作成します。
- 【】 名前を”Laser”に変更します。
- 【】 **Line Renderer**コンポーネントを追加します。*(Component->Miscellaneous->Line Renderer)*

NOTE レーザはまだ配置しないでください！”Use World Space”チェックボックスを先に無効化しなければなりません。すぐに行います。

- 【】 **LaserTrap**スクリプトを追加します。



Line Rendererコンポーネントの設定を図のように行います。(レーザマテリアルは *Particles->Sources->Materials* の中にあります。)



Line Rendererの設定



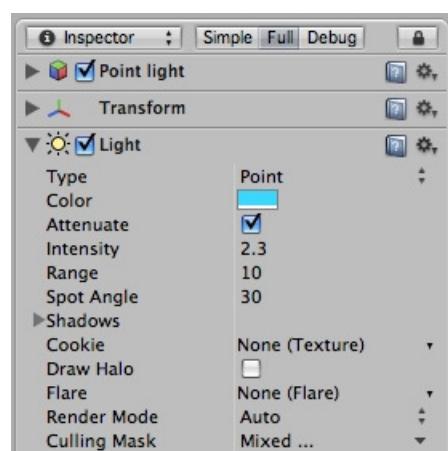
完成したオブジェクトをレーザトンネルに配置します。(屋根の付いた通路が、アリーナの両側にあります。牢屋からずっと離れた、レベルの奥のほうです。)



Point Lightゲームオブジェクトを**Laser**オブジェクトの子として追加します。



Point Lightを図のように設定します：



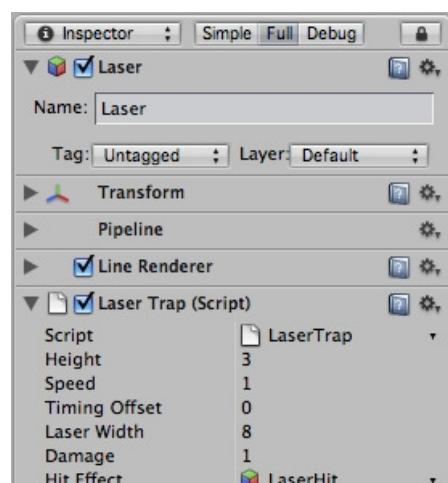
Laser Trapの Point Light設定

ポイントライトはレーザの光源として働き、レーザの上下に合わせて、上がったり下がったりします。これが、ラインレンダラによって書かれたレーザビームが光を発しているような効果を生み出します。

Line Rendererコンポーネント

ラインレンダラは、名前の通り、3次元空間に線を描きます。線が描かれるべき点が配列として含まれています。線自体はTrail Renderer コンポーネントと同じ手法で描かれます。レーザのように見せるために、ライティングを行います。

次のLaserTrapスクリプトの属性を次のように設定します：



Laser Trapスクリプト属性

TIP LaserHitアセットは、プロジェクトペインのPropsフォルダにあります。

最後に、Laser Trapを複製(Cmd+d[Mac](Ctrl+d[Win]))、または、プレハブを作成)し、幾つかをレベルに合うように配置します。全部で4つ配置すると良いでしょう。それぞれのトンネルに2つずつです。

次のLaserTrapスクリプトの属性を自由に変化させて、満足するまで試してみてください。

Laser Trap スクリプト

レーザトラップの鍵となるのはLaserTrapスクリプトです。詳しく見てみましょう。

概要

レーザトラップはスクリプトによって上下するラインレンダラコンポーネントです。このスクリプトは、衝突を検知し、視覚的効果を引き起こします。

はじめに、レーザトラップに関する幾つかの基本的な属性を定義します：

- `height`は、振幅の幅を定義します。レーザビームが開始点からどの程度上下に移動するかを決定します。
- `speed`は、ビームの移動速度を定義します。
- `timingOffset`は、それぞれのレーザトラップオブジェクトが、振幅を開始する位置を異なるものにします。
- `laserWidth`は、レーザビームの、開始点から終端までの幅を定義します。
- `damage`は、プレイヤがビームに触れたときに、与えられるダメージの量を決定します。
- `hitEffect`は、様々なゲームオブジェクトにリンクされ、何かがレーザトラップに触れた際にインスタンス化されます。これは、レーザトラップの効果をハードコーディングするよりも柔軟で、アセットを他のプロジェクトで再利用することを可能にします。

スクリプトは2つの関数を定義します。それぞれを見てみましょう：

`Start()`関数は、ラインレンダラコンポーネントをインスタンス化し、レーザの初期位置(Y軸上)を設定し、ラインレンダラの2つ目の頂点が、`laserWidth`で定義された幅に合うように設定します。これにより、通路の幅に合わせて、簡単にレーザトラップの幅を調節出来ます。

NOTE それぞれのラインの終端座標を、ラインレンダラの配列に手動で設定することも出来ますが、スクリプトで処理することで、ゲーム内でレーザビームがアニメーションする事において、多少の柔軟性が得られます。

さて、メインとなる`Update()`関数です。ここでは、興味深いことが行われています：

はじめに、レーザビームの移動が計算されます。`Mathf.Sin()`関数を使います。現在の時間を`Time.time`で取得(ゲームが始まってからの時間を秒で返します)し、アニメーションさせたいスピードと掛け合わせ、`timingOffset`値に加えます。これで、レーザの位置がサインカーブに沿って動きます。最後に、配置したい`height`と、結果のオフセットを組み合わせます。

次のステップは、衝突の検出です。スクリプトは、レイ(直線)をビームのパスに沿って撃ち、**Collider**コンポーネントを持ったゲームオブジェクトが衝突したことを、確認します。

(反応時間を得るために、時間ベースのテストを行います。そうしなければ、プレイヤは、レイに当たっている間ずっと、体力を失い続けることになります。)

何かに衝突したら、プレイヤか敵かをチェックします。そうであれば、"ApplyDamage"メッセージを送信します。同時に、hitEffectゲームオブジェクトをインスタンス化します。これにより、魔法が演出されます。これが、エネルギーバーストの効果を生成する、LaserHitゲームオブジェクトです。



レーザにあたるのは、健康上良くありません。

このチュートリアルをプレイすれば、レーザにヒットすると、Lerpzの体力が失われるのを確認できるでしょう。

ロボットガード

動いている敵は、ロボットガーデです。レベルの様々な場所に戦略的に配置されています。Lerpzが敵の範囲に入ると、攻撃に向かってきます。



ロボットガード

Lerpzのゲーム内の敵は、ロボットガードです。このモデルは'古典的'ですが、保全するには十分です。コレクタや熱狂的なファンは、不運なことに、このような古典的な過ぎし日の宝石を、Ford社の売れなかった大型車Edsel、AppleのLisa、Sinclair C5と言う電気自動車等と比較します

このモデルは、狡猾にデザインされていることで知られています。何度かの足部分への良く狙われたパンチに敏感です。無能力化されると、持っている幾つかのアイテムを吐き出した後、ロボットはBIOSが再起動するまで、嬉しそうに地面に倒れます。実際は、距離センサが、付近に何もいないことを感知するまで、ですが。このセキュリティロボットは、今は庭の装飾品として使われています。

上記のことより、次のことが決定出来ます：

- ロボットは、非常に初歩的なAIを持っている
- ロボットは、倒されると、取得可能なアイテムを吐き出す
- ロボットは、プレイヤが見えなくなると再出現する

探索と破壊

ほとんどのゲームのAIは、知性よりも行動のモデル化に焦点が当てられています。私たちのロボットは、少し賢く見えますが、プレイヤの存在に基づいて、予想できる反応を行います。これは、それほど悪いものではありません。多くのゲームプレイヤはこのような行動を好みます。パターンを知ることが出来、それによって、どのようにロボットを倒せばよいのかが学習できるからです。

ロボットガードは、とてもシンプルな行動パターンを持っています。AIスクリプトである **EnemyPoliceGuy** を反映します。

Idle (アイドル) -- このモードでは、ロボットは立ち止まってロボットなりの考えを巡らせています、カチカチと静かに音を刻みながら、侵入者が範囲に入ってくるのを待っています。

Threaten (警告) -- 侵入者が壇に入ると、ロボットはスタンバイをやめ、チラシに”警告状態”と宣伝されていた通りに、バトンを回転させ注意が向いていることを知らせます。

Seek & Destroy (探索と破壊) -- アクティベートされると、ロボットガードは侵入者に向かっていき、攻撃を加えます。侵入者を攻撃範囲に捉えると、バトンを使って殴りかかります。

NOTE 1つのアニメーションシーケンスがそれぞれのモードで使われます。モデル及びスクリプトでの名前は”turnjump”です。

Struck (停止) -- プレイヤがロボットガードを倒すと、このアニメーションが再生されます。(モデルとスクリプト中では”gothit”)

プレイヤがロボットの探索範囲から外に出ると、ロボットは”Idle”モードに戻ります。

上記の状態は**EnemyPoliceGuy**スクリプトによって扱われます。スクリプトはアニメーションシーケンスの変更を行います。(プレイヤで行ったように2つにスクリプトを分解するほど多くのアニメーションを使用していません)

ロボットガードの追加 (Adding the Robot Guards)

レベルに何体かのロボットが必要ですので、、、



プロジェクトペインを開き **Copper** プレハブを **Enemies** フォルダから見つけます。



レベルの中でいい場所を見つけてプレハブをシーン上にドロップします。地面に正しく配置してください。(プレハブは**Character Controller**コンポーネントを持っています。カプセルコライダの最下部が、ちょうど地面に触るか、ほんの少しだけ上になる事を確認してください。)

フェンスのあるエリアや主に閉じられている場所であれば、ロボットが落ちてしまうことがないので、配置するには良いでしょう。

ゲームをプレイすると、ロボットがアイドル状態で立っているのを見ることが出来ます。ロボットがなにか興味深いことをするように、スクリプトを追加しましょう。

2つのスクリプトをロボットのプレハブに直接追加します：

- 🔊 **Copper** プレハブを選択し、インスペクタに表示します。
- 🔊 **EnemyDamage** と **EnemyPoliceGuy** スクリプトを **Character Controller** コンポーネント上に、インスペクタでドラッグします。

もし、すでに追加していなければ **ThirdPersonCharacterAttack** スクリプトを **Player** オブジェクトに追加します。これは、プレイヤのパンチ行動を扱います。(これがなければ、Lerzはロボットをパンチしません！)

ゲームをプレイすると、ロボットは、プレイヤの接近に反応するはずです。

死の青い火花 (Blue Sparks Of Death)

プレイヤが、ロボットをノックダウンすると、死のシーケンスが初期化されます。ロボットは倒れながら火花を出し、プレイヤが範囲から離れるまで横たわっています。

さらに、ロボットが倒れると、幾つかの取得可能アイテムを吐き出すようにします。

新しいスクリプトやアニメーション、その他の要素を現在のプレハブに追加するのではなく、新しいものを、我々の電気的な敵の断末魔のために作成します。

分割と征服 (Divide & Conquer)

ロボットが死ぬと、プレイヤへの反応を停止し、スクリプトを停止しなければなりません。実は、それほどシンプルではありません。スクリプトはそれぞれが独立して動いているため、それぞれのスクリプトにメッセージを送って状態値を管理します。メッセージはサイクルごとにチェックされる必要があります。

それよりも、単純に我々のロボットプレハブを他のものに入れ替えてしまう方が簡単です。特別な効果とスクリプトを持ち、アイテムを吐き出しながら、倒れるためだけに用意された仮想的なスタントです。ここでは、この方法を使います。



ひざまずくロボット

上のイメージが、プレハブの置換えを表示しています。

これをもってくるのはEnemyDamageスクリプトなので、詳しく見てみましょう。

ApplyDamageメッセージがGameObjectに送られると、ApplyDamage()関数が呼び出されます。ロボットがダメージを受けすぎると(この例では3回に設定されています)、Die()関数が呼ばれ、次の楽しみを開き始めます。

Die()関数は、はじめにCopperゲームオブジェクトを破壊としてマークします。(破壊はUpdate()関数が完了するまで、動作しませんので、関数のはじめに行っておきます。)

次に、差し替えるプレハブがインスタンス化され、ロボットの現在位置(胴体から腕までの子オブジェクトを含む)がコピーされます。

火花の散る爆発は、単純にもうひとつのパーティクルシステムアセットです。これがインスタンス化され死につつあるロボットゲームオブジェクトが正しい位置と向きに現れるように調整します。

インスタンスが配置されたので、次のステップは、プレイヤーを押しやることです。それによって、Unityの物理エンジンはロボットを倒して回転させることができます。

最後に、最大2つの取得可能アイテムを生成します。50%の確率で体力かオイル缶が選ばれます。空中のランダム方向に投げ出しておしまいです。

これらの値は**Copper**ゲームオブジェクトを選択し**Enemy Damage**コンポーネントで変更出来ます。幾つかの値を試してみることをおすすめします。

取得可能アイテムと物理 (Droppable Pickups & Physics)

ロボットが倒されたときに現れる取得可能アイテムは、先に作ったオリジナルのプレハブから派生したものですが、パーティクル効果と、移動と衝突をハンドルする**DroppableMover**スクリプトが追加されています。

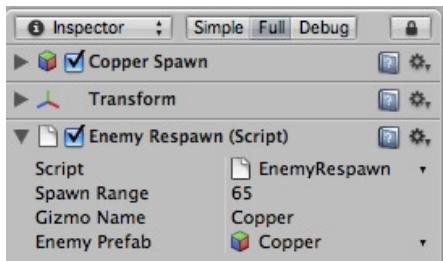
コライダが取得をトリガするために設定された物理エンジンによって無視されるために、スクリプトが必要でした。取得可能アイテムは、初期速度が与えられ、下向きに光線を照射(レイをキャスト)することで、アイテムが、床に衝突したかを検出します。検出されると、スクリプトは無効化します。(このコードは、1つ問題点があります。Y軸方向下向きにしかチェックしないため、取得可能オブジェクトは壁にめり込んでしまう場合があります。)

出現と最適化 (Spawning & Optimization)

プレイヤが特定の距離に来たときに敵を出現させるのは、コンピュータやゲームの初期に行われていた古いトリックです。これを使うと、敵の状態を保持しておく必要がなくなります。また、プロセッサの処理を軽減することも出来ます。単純に、プレイヤから見えないロボットを消してしまうことで、AIやスクリプトの不必要的実行を避けることが出来ます。

やってみましょう：

- 【】 空のゲームオブジェクトを階層ペインのトップに作成します。
- 【】 名前を**CopperSpawn**に変更します。。
- 【】 **EnemyRespawn**スクリプトをオブジェクトの上にドロップします。
- 【】 ロボットを出現させたい位置に**CopperSpawn**オブジェクトを配置します。(小さなロボットアイコンと、出現範囲を示す球体がギズモとして表示されるのを確認します。これらは**EnemyRespawn**スクリプトによって描画されます。)
- 【】 オブジェクトの設定を図のように行います。



CopperSpawnの設定

幾つか、このゲームオブジェクトが必要となりますので、親のゲームオブジェクトを作成し、**CopperSpawn**オブジェクトを子オブジェクトにしましょう。

- ☞ 空のゲームオブジェクトを、階層ペインのトップに作成します。
- ☞ ゲームオブジェクトの名前を**Enemies**に変更します。.
- ☞ **CopperSpawn**オブジェクトを**Enemies**の子オブジェクトに設定します。**CopperSpawn**オブジェクトを**Enemies**オブジェクトにドラッグします。
- ☞ **CopperSpawn**ゲームオブジェクトを用いて、プレハブを作成します。インスタンスをレベルのあちこちに配置しましょう。

NOTE この方法の副作用は、ロボットを倒して範囲から離れると、プレイヤがその場所に戻ったときにロボットが新しく出現することです。

どのように動くのか (How it works)

CopperSpawnゲームオブジェクトは、プレイヤが範囲に入ったかをチェックするスクリプトを含んでおり、範囲に入ると**Copper**プレハブのインスタンスを生成します。プレイヤが範囲の外にでると、ロボット出現スクリプトは自動的に、ロボットをシーンから削除します。

これを行うスクリプトは**EnemyRespawn**です。このスクリプトには沢山コメントが有りますが、重要な関数は次の2つです：

`Start()` -- プレイヤゲームオブジェクトのトランスフォームのリンクを、後で利用するためにキャッシュします。

`Update()` -- はじめに、プレイヤが範囲に入るとロボットをインスタンス化します。プレイヤが範囲から出るとロボットプレハブは破棄されます。

2つのギズモ関数もエディタの中で使用されます：

EnemyRespawnはまた、Unityの”ギズモ”を利用します。ギズモは通常シーンビューのみに表示される、視覚効果です。ゲーム中には表示されず、出現範囲などを球体で表現します。このスクリプトでは、2つの種類のギズモを使用しています。

一つ目は、ロボットのアイコンを描画し、シーン中の出現オブジェクトを、マウスでクリックして選択できるようにします。

NOTE アイコン画像は *Assets->Gizmos* フォルダに格納されています。



OnDrawGizmos() 関数がロボットのアイコンを表示

アイコンを表示するギズモのコードは次のようにになります：

```
function OnDrawGizmos ()  
{  
    Gizmos.color = Color(1, 1, 1, 1);  
    Gizmos.DrawIcon(transform.position, gizmoName + ".psd");  
}
```

OnDrawGizmos() 関数はUnityエディタGUIが更新されたり、リフレッシュされたりする度に呼び出されますので、アイコンは常に表示されています。関数がどのアイコンイメージを使うかを知るために、

ギズモのName属性をインスペクタで"Copper"に設定します。

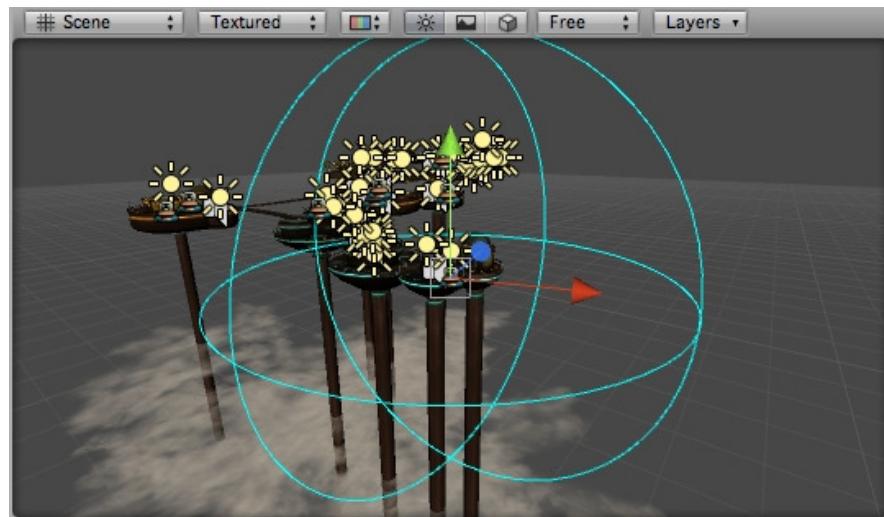
OnDrawGizmosSelected() 関数はオブジェクトが選択された場合のみ、UnityのエディタGUIによって呼び出されます。選択されている間はずっと、UnityのエディタGUIが更新や再描画される度に呼び出されます。

この例では、関数は **spawnRange** で定義された半径を用いて球体を描画します。プレイヤが侵入したときに、敵ロボットがインスタンス化されたり、プレイヤが離れたときに、インスタンスを破棄したりする範囲を球体で描画します。

```

function OnDrawGizmosSelected ()
{
    Gizmos.color = Color(0, 1, 1);
    Gizmos.DrawWireSphere(transform.position, spawnRange);
}

```



OnDrawGizmosSelected() 関数が、Spawn Range値により定義された球体を描画している

別な最適化手法(Alternative Optimizations)

上記のテクニックに加えて、UnityはOnBecameVisible()とOnBecameInvisible()関数を提供します。しかし、我々の再出現テクニックと違い、これらの関数はカメラの向きとその他の設定に基づいているため、プレイヤオブジェクトに基づいていません。このため、OnBecameInvisilbe()は、カメラが別の方向を向くだけで呼び出されます。この動作は要求するものと異なってしまします。

我々の方法より最適なもう一つのテクニックは、スクリプトではなく、Colliderコンポーネントをトリガとして利用して、プレイヤの位置をチェックすることです。UnityはOnTriggerEnter()とOnTriggerExit()関数をこの目的のために提供しています。しかし、コライダを他の目的に使用しているオブジェクトに、再出現スクリプトを付加したい場合には、利用することは出来ないでしょう。

オーディオ&最終調整 (Audio & Finishing Touches)

この章では、サウンドエフェクトと、2つのシーン切り替えをゲームに追加します。



導入(Introduction)

CDを聴いたり、映画を観たりするとき、耳から聞く音は、様々ステージに定常的に流れるものが有ります。マスタリングと呼ばれます。マスタリングはオーディオを聞かせるための技術です。ボリュームレベルを合わせ、周波数をフィルタし、様々なテクニックを用いて、最終的なミックスが最大限に良く聞こえるように調整します。しばしばこのプロセスは、音楽やサウンドトラックの明らかな歪を修正します。例えば、ある楽器が大きくなりすぎたり、騒がしかったりする場合です。その他にも、サウンドトラックが、安いスピーカーでも十分なクオリティで聞くことができるよう、しかし、ハイエンドのシネマオーディオシステムでの音を損なうことのないように調整します。

ゲームの音楽でもこれは変わりません。良い調整を行い最良の状態を持っていくことが、非常に大切です。

オーディオ(Audio)

ゲームにおいて、オーディオを完成させ、マスタリングを行うことは難しい作業です。ゲームは受動的で一方向のエンターテイメントではなく、双方向的です。つまり、プロジェクト中の個々のオーディオアセットがどのように相互作用し、必要ならば、先手を打ってマスタリングしたり、ミキシングしたりが必要になります。マスタリングは、ゲームロジック自体によって扱われるリアルタイムのタスクです。

理想的な世界では、すべての開発者が気心の知れたオーディオエンジニアに連絡が取れます。現実の多くの小さなプロジェクトでは、そうもいきません。

サウンドをプロジェクトに他の物と共に存させることが、最も重要な事です。これは、主観的なプロセスであり、バスの聞いた音を好む人や、高い周波数の音を好む人もいますので、アルファやベータバージョンでテスト要員を使いフィードバックを、客観的視点からプロジェクトの音を取り込んでいくことが重要です。(オーディオモニタリング設定は便利ですが、高価であるだけでなく、特別な部屋を容易出来ない場合、共同作業者に迷惑です。)

理想的には、ひとつの音源からのオーディオを同じ両耳でミックスしたものを使うことで、一貫した音を得ることができます。しかし、このチュートリアルでもそうしていますが、ゲーム全体のミックスに合うように、調整と設定のために準備された、幾つかの異なる音源を使わなければなりません。大きすぎるサウンドエフェクトを頻繁に聞かせることは、プレイヤーを苛立たせてしまうでしょう。テストプレイを何度も行い、このプロセスのための時間を確保しておいてください。

サンプルノート (Sample Notes)

Unityはオーディオに関してとてもシンプルなインターフェースを持っていますが、幾つか重要なことがあります。

- サンプルが同じレベルであることを確認します。これは音量のレベルとロールオフ設定を一貫したものにします。ノーマライゼーションを使うと良いですが、音源がどのようにミックスされるのかを考慮する必要があります。耳障りなサウンドエフェクトが多くなると、ユーザは混乱してしまいます。
- 3Dワールドでリアルに音源を位置させたい場合は、モノサンプリングの音源を使ってください。
- Ogg Vorbis圧縮システムで音源を圧縮してください。(Unityが行ってくれます)
- ウェブでの公開を行う場合は、常にOgg Vorbisで圧縮しておくべきです。
- 短くて、良く使うサウンドエフェクトでは、"Decompress on load"ボックスをチェックしてください。
- 特定の位置に配置しなくて良い長い音楽にのみ、ステレオ音源を使用してください。このような音源はデフォルトのAudio Listener音量で再生されます。

Lerpzの脱出にサウンドを追加 ! (Adding Sound to Lerpz Escapes!)

我々はすでに、幾つかのスポット効果をゲームに追加しました。この章では、さらにオーディオを追加しゲーム構築のプロセスを完了します。

多くのゲームジャンルで、サウンドエフェクトは実際の音源から取得したり、サウンドエフェクト集から取得したりします。しかし、Lerpzの脱出！では、簡単に手に入らないサウンドエフェクトが必要になります。通り過ぎる宇宙船にマイクを向けたり、便利なロボットガードを見つけたりすることはできません。そのため、想像力を働かせなければなりません。

可能なサウンドエフェクト全てをゲームに追加することはせず、この章ではUnityのオーディオ機能を紹介するのに十分なものだけを用います。この章を読み終わると、自分なりのサウンドエフェクトを追加できるようになります。

ゲームで必要となるサウンドエフェクトのリストは次のようにになります：

プレイヤ

- 歩く/走るサウンド
- 攻撃サウンド
- 攻撃を受けたサウンド
- 死亡サウンド
- ジェットパックの発火さうんど

ロボットガード

- 待機サウンド
- 攻撃サウンド
- 攻撃を受けたサウンド
- 死亡/爆発サウンド

取得可能アイテム

- オイル缶取得さうんど
- 体力取得さうんど

環境サウンド

- 霧囲気を出すための、長い、繰り返されるサウンド
- ジャンプ台に乗った時のサウンド

宇宙船を囲むフェンス

- "Active"のサウンド
- "Shutdown"のサウンド

宇宙船

- シーン切り替えアニメーションと共に再生される離陸サウンド

幾つかのサウンド効果は、すでにゲーム中に追加してありますが、さらに15個を追加する必要があります。そのうちの幾つかは、足音やジェットパックの噴射など、明らかなフォーリイサウンド(シーンに同期してなるサウンド)です。フォーリイサウンドのために、別の音源を利用したものもあります。(例えば、pickupFuelサウンドは、ゴルフボールがクラブによって打たれたサウンドです。)

これらのサウンドは、多くのサウンドエフェクト集に含まれています。AppleのGrage BandやLogic Studio 8、Soundtrack/Soundtrack Proソフトウェアでは、これらのサウンドがライブラリに含まれており、このチュートリアルで用いる殆どのサウンドは、そこからのものです。その他多くの音源はオンラインで見つけることが出来ます。大抵フリーです！

環境サウンド(Ambient Sounds)

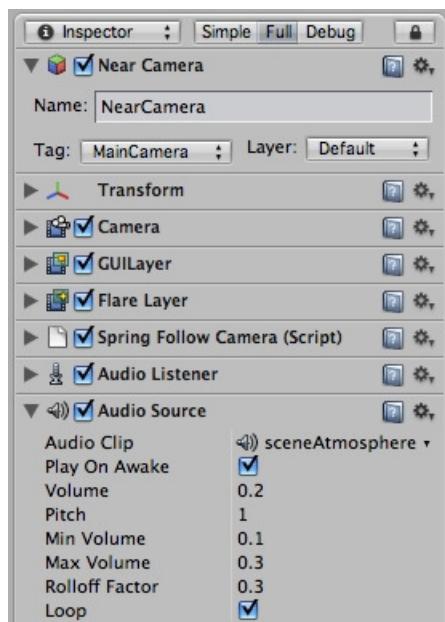
私たちのゲームには環境として、雲の上に近くの惑星が見える感動的な景色が設定されています。このような場所では、ゲームに没入するために、現実離れしたサウンドが必要でしょう。

このチュートリアルプロジェクトは、繰返される環境サウンド音源として**sceneAtmospher**を含んでいます。これはステレオのOgg Vorbis音源で、エイリアンにぴったりな、1960年代の幾つかの、古いモノラルのBBC Radiophonic Workshopサウンドエフェクトから作られています。

音源は、Near Cameraオブジェクトに階層ペインで追加します。

 音源をNearCameraオブジェクトにドロップします。Unityは自動的にAudio Sourceコンポーネントを作成します。

 設定を図に示されるように行ってください：



sceneAtmosphere音源をNear Cameraオブジェクトに追加

Audio Sourceコンポーネントは、Unityがインスペクタやスクリプトで、どのようにサウンドを扱うかを定義する、基本的なコントロールを含んでいます。

このインスタンでは、Play On Awakeスイッチを設定し、サウンドが自動的に開始されるようになります。音量は、バックグラウンドの環境音源であり、その他の音源を邪魔しないため、小さめに設定します。

その他の設定を簡単に見てみましょう：

Pitch設定は、音源が再生される速度を決定します。1が通常の速度です。基本的に使われるアルゴリズムは、テープレーコダと同じですので、時間による引き伸ばしは行われます。2に設定すると音源は2倍の速度で再生され、ピッチも2倍になります。

TIP Pitch設定は、一度だけ再生するサウンド効果に有効です。値を調整することで、それぞれの再生を様々に変化させることができます。これは、銃の発射やレーザ、足音に有効で、複数の音源を保持する必要を減少します。

次の設定は、音源の音量範囲を決定します。遠く離れたときに、その音を聞こえなくしたい場合、**Min Volume**を0に設定します。

名前が示す通り、**Max Volume**はオーディオの最大音量を定義します。音源の真上に来たときに最大となる音量の値です。

次は**Rolloff Factor**です。これは、音源の音量が、距離に応じてどの程度素早く変化するかを決定します。小さい値は、遠い距離でも音が聞こえることを示します。設定は、ゲーム全体の現実感を出すために重要です。

最後に、**Loop**設定があります。宇宙船のフェンスにおいて、停止するまでサウンドが鳴り続ける用にするために、これを有効にしています。

NOTE ループする音源はループがスムーズに行われるようデザインしておくことが必要です。そうしなければ、繰り返されるたびに、小さな破裂音を聞くことになります。多くの音源エディタは、このために、"ゼロクロス点の探索"機能を備えています。

ジャンプ台 (The Jump Pads)

JumpPadプレハブを作ったとき、サウンドエフェクトについては扱いませんでした。ここで、追加します。しかし、どのように再生するとよいでしょうか？

運の良いことにJumpPadのスクリプトはすでにサウンドエフェクトをサポートしています。スクリプトをエディタで開くと、次のような再生のコードがあるでしょう：

```
...
if (audio)
{
    audio.Play();
}
```

audio変数は、スクリプト中では定義されていません。どこから来たのでしょうか？

これはUnity自身によって定義されています。幾つかある、便利な変数の一つで、**Audio Source** コンポーネントが付加されていれば、どのようなゲームオブジェクトであれ、付加されたスクリプト内で、**Audio Source** コンポーネントを指し示します。これらのコンポーネントを追加していないので、**audio**変数はnullとなり、スクリプトは再生をスキップします。

- ▶ ジャンプ台インスタンスの1つをシーンでクリックし、インスペクタに表示します。
- ▶ **jumpPad**サウンドエフェクトをインスペクタでドラッグします。(自動的に**Audio Source**コンポーネントが作成されます。)

変更をオリジナルのプレハブに適応することで、すべてのジャンプ台がサウンドエフェクトを共有出来ます。

- ▶ **Audio Source**の設定をいろいろ変更して、望みのサウンドになるまで試してみましょう。

取得可能アイテム

取得可能アイテムは簡単です。サウンドエフェクトはそれぞれ、**pickupFuel**と**pickupHealth**です。**Pickup**スクリプトはすでに、オーディオをサポートしていますので、これらの効果を追加するには、単純にプレハブごとに付加されたスクリプトのサウンドエフェクトスロットに追加すれば大丈夫です。

下の画像は、レベル中にある1つの**FuelCellPrefab**インスペクタの詳細を表示しています。**pickupFuel**サウンドエフェクトが**Pickup**スクリプトの**Sound**変数に追加されています。



fuelCellPrefabのサウンドエフェクト設定



pickupFuelサウンドエフェクトを**Sound**スロットにドロップするか、リストから使用可能な音源を選択して、オーディオを追加します。スロット右側の小さい三角形をクリックするとリストが表示されます。



Sound Volumeを2に設定して、サウンドエフェクトが目立つようにします。



Health pickupについても、**pickupHealth**音源を用いて同様に行います。



TIP 時間を節約するために、プレハブに直接サウンドエフェクトを追加することができます。

ゲームをプレイすると、それぞれの取得可能アイテムを取得した際に、適したサウンドエフェクトが再生されるのを確認出来ます。

バリアフェンス(The Impound Fence)

宇宙船を囲んでいるフォースフィールドは、アクティブな間、ブンブン、シューシューとノイズを出します。GarageBandでループする環境ノイズとともに、サウンドエフェクトを作成しました。

TIP AppleのGarageBandはAAC(".M4A)音声フォーマットでのみ書き出せるので、Audacityを使いモノラルのAIFFフォーマットに変換したものを、プロジェクトでは使用しています。

音源は**activeFence**と言う名前です。



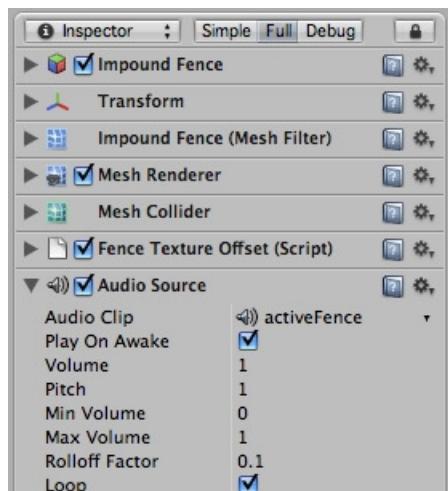
階層ペインで**levelGeometry**を開き**impoundFence**オブジェクトを探します。



activeFenceサウンドをオブジェクト上にドロップします。自動的に、**Audio Source**コンポーネントが**impoundFence**オブジェクトに追加されます。



最後に、**Audio Source**の設定を次のように変更します：



バリアフェンスのAudio Source設定

プレイヤ(The Player)

現時点ではLerpz自身は何も音を発していません。音を追加するのは当然に思えますが、どれを追加すればよいのでしょうか？

このチュートリアルで実装するサウンドエフェクトは：

- パンチ音
- “打撃”音。Lerpzがロボットから攻撃された場合に再生されます。
- ジェットパックの噴射音

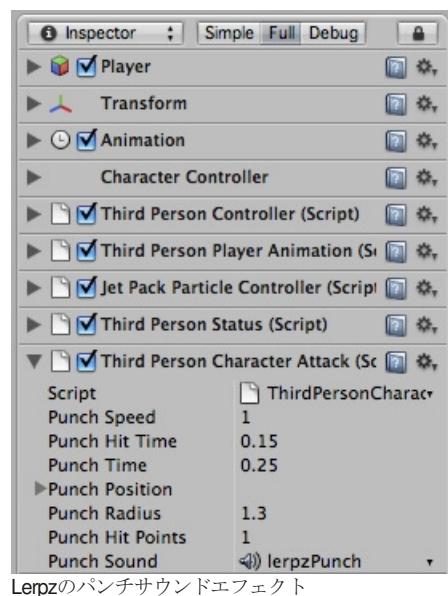
- ・プレイヤが死んだり、復活したりした場合に再生される音。

(足音は、皆さんとの練習のために残しておきます。)

これらの音はスクリプトによって再生されます。

パンチ音 (Punching)

パンチの動作とアニメーションは**Third Person Character Attack**スクリプトで操作されます。サウンドエフェクトの属性がインスペクタに表示されています。**Punch Sound**属性を次のように設定してください：



Lerpzのパンチサウンドエフェクト

このサウンドを再生するスクリプトは以下のコードを使用しています：

```
if (punchSound)
    audio.PlayOneShot(punchSound);
```

はじめに、パンチサウンドが設定されているかをチェックします。設定されていれば、`PlayOneShot()`関数を使って、サウンドを再生します。間数は、`Audio Source`コンポーネントの付加された一時的なゲームオブジェクトを作成し、シーンに追加し、再生します。サウンドエフェクトが終了すると、ゲームオブジェクトはシーンから削除されます。

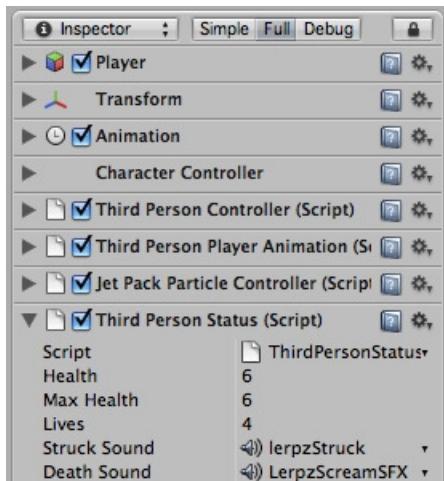
NOTE ゲームの再生中、ワンショットサウンドが階層ペインに短時間だけ現れるのが確認出来ます。これは通常の動作です。

打撃音と悲鳴 (Struck sound & Scream sound)

`Third Person Status`スクリプトはさらに2つのサウンド効果を操作します。1つはLerpzが敵に攻撃された場合、もう1つはLerpzが死んだ場合です。(Lerpzが復活する直前かゲームが終了する直前です)

どちらのサウンドエフェクトも、先に追加した"lerpzPunch"サウンドと同じ方法で扱われます。

lerpzStruckと**LerpzScreamSF**を以下の図のように追加します：



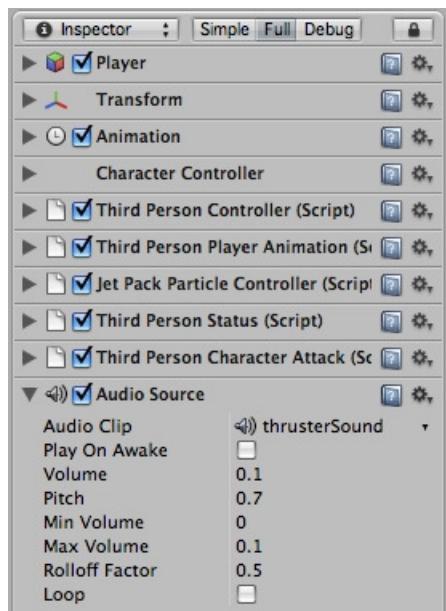
打撃と死亡のサウンドエフェクトを追加

ジェットパック(The Jet-pack)

ジェットパックは1度だけではなくループするサウンドエフェクトです。そのため、Playerゲームオブジェクトに直接、Audio Sourceコンポーネントとして追加します。Jet Pack Particle Controllerスクリプトが自動的にAudio Sourceコンポーネントを生成してくれますが、オーディオファイルは手動で追加しなければなりません。

thrusterSoundオーディオファイルをPlayerゲームオブジェクトのAudio Sourceコンポーネントにドラッグします。

Audio Source設定を次のように行います：



ジェットパックのオーディオ設定

サウンドエフェクトを操作する2つの部分がスクリプトにあります。両方ともJet Pack Particle Controllerスクリプト内に定義されています。はじめの部分はStart()関数でAudio Sourceコンポーネントを初期化します：

```
audio.loop = false;
audio.Stop();
```

2つめの部分は同じ関数の下のほうにあります：

```
if (isFlying)
{
    if (!audio.isPlaying)
    {
        audio.Play();
    }
}
else
{
    audio.Stop();
}
```

TIP audio変数はUnity自身によって生成されます。つねにゲームオブジェクトの AudioSourceコンポーネントをさしておきます。

このコードは解りやすいと思います。Play()関数は、再生されていようがいまいが関係なく、常にサウンドエフェクトを最初から開始するため、すでに再生されているかをチェックします。Unityがサウンドの音源をPlay()関数のたびに頭から再生を繰り返すと、つかえるような音が聞こえてしまいます。

ロボットガード(The Robot Guards)

ロボットガードは複数のスクリプトを持っています。そのうちの幾つかは属性にオーディオ音源も持っています。加えて、それぞれのロボットガードはAudio Sourceコンポーネントを保持しています。

Audio Sourceコンポーネントをジェットパックの音のために単一で使用するPlayerと異なり、EnemyPoliceGuyスクリプトはCopperプレハブのAudio Sourceコンポーネントを複数の繰り返されるサウンドや、それらの切り替えに必要に応じて使用されます。これを実現するためのサンプルコードは以下のようになります：

```
if (idleSound)
{
    if (audio.clip != idleSound)
    {
        audio.Stop();
        audio.clip = idleSound;
        audio.loop = true;
        audio.Play();
    }
}
```

上記の例はEnemyPoliceGuyスクリプトのIdle()関数のはじめの部分です。audio.Stop()の呼び出しは、音源を再生中に、気づかれることなく切り替えるために重要です。

このサウンドエフェクトを追加するために：

-  Copperプレハブをプロジェクトペインで選択し、インスペクタに表示します。
 -  CopperIdleLoopサウンドエフェクトをAudio Sourceコンポーネントにドラッグします。
 -  同じオーディオファイルをEnemyPoliceGuyスクリプトコンポーネントのIdle Sound属性に追加します。
 -  CopperActiveLoopサウンドエフェクトを、同じコンポーネントのAttack Sound属性に追加します。
- 最後に、MetalHitサウンドエフェクトをEnemy Damageスクリプトコンポーネントの、Struck Sound属性に追加します。

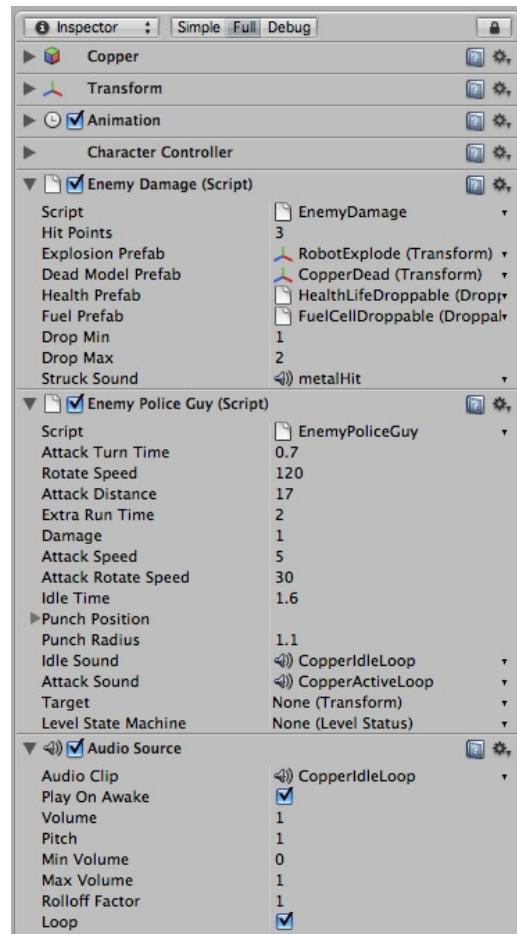
(インスペクタペインの最終的な見た目は、次のページの図のようになります。)

MetalHitサウンドは、**EnemyDamage**スクリプトの**ApplyDamage()**関数の中で再生されます。
コードは次のようにになります：

```
if (audio && struckSound)
    audio.PlayOneShot(struckSound);
```

Jet Packと同様、個々で使用しているaudio変数はUnity自身によって作られたショートカットの変数です。これは、GetComponent (

NOTE 同じ方法はPlayerのサウンドエフェクトにも使用されます。



オーディオファイル追加後の、Copperプレハブのインスペクタ設定

シーン切り替え(Cut Scenes)

シーン切り替えはプレイヤが知るべきイベントやストーリー要素をフィルインするのに便利です。プレイヤから切り離されるのは最小にするべきです。そこで、ここでは2つだけ、シーン切り替えを使います。

1つ目は、プレイヤが要求される量のオイル缶をレベルにおいて取得した時で、宇宙船のロックが解除されます。このシーン切替は、ピクチャインピクチャの手法で行われますので、プレイヤはシーンが再生されている間もプレイを継続出来ます。

このシーン切替で画面全体を占有しない、もうひとつの非常に大きな理由は、すべてのゲームオブジェクトである、ロボット、プレイヤコントロールなどすべてを、シーン切り替えが再生されている間、停止しなければならないからです。さもなくば、シーン切り替えが終了するまで、プレイヤは目が見えない状態になってしまいます。

NOTE 近くの木箱に登れば宇宙船にたどりつくことはできますが、この時点では宇宙船はロックされているので飛び立つことはできません。(バリアフェンスのロックが解除されるまで、メッシュコライダはトリガに変わりません)

2つめのシーン切り替えは、フェンスが無効化された後、プレイヤが宇宙船に触ると発生します。このシーンはフルスクリーンで再生され、宇宙船が離陸し、自由と新たな冒険に向かって飛び立つ様子を見ることが出来ます。これは、ゲームオーバーケンスに移る直前に再生されます。

はじめのシーンを詳しく見てみましょう、、、

バリアフェンスの解除 (Unlocking the impound fence)

バリアフェンスについては、はじめに幾つかのアニメーションを設定したときに触れました。このチャプタの早い段階で、サウンドエフェクトも追加しました。ここでは、アニメーションさせます。

しかしあくまで、すべてのオイル缶を取得した場合にのみ、これを実行しなければなりません。(完全なスクリプトコードは付録の章にあります。)



ThirdPersonStatusスクリプトを開きFoundItem()関数を探します。



コメントブロックの後に、次のコードを追加します。(必ず、関数の閉じカッコである" }"の後に追加してください。そうでなければ、動作しません!)

```
if (remainingItems == 0)
{
    levelStateMachine.UnlockLevelExit(); // ...and let our player out of the level.
}
```



スクリプトを保存します。



LevelStatusスクリプトを開きます。2つのシーン切り替えはここで操作されます。(完全なスクリプトコードは付録の章にあります。)



スクリプトの先頭に、以下のコードを追加します：

```
var exitGateway: GameObject;
var levelGoal: GameObject;

var unlockedSound: AudioClip;
var levelCompleteSound: AudioClip;

var mainCamera: GameObject;
var unlockedCamera: GameObject;
var levelCompletedCamera: GameObject;
```

NOTE 上のコードは、2つ目のシーン切り替えで必要な変数も含んでいます。



つぎに、`Awake()`関数に以下のコードを追加します：

```
levelGoal.GetComponent(MeshCollider).isTrigger = false;
```

このラインは重要です。2つ目のシーン切り替えが、早く起こることを防ぎます。**isTrigger**スイッチを無効にすると、バリアフェンスが有効な間、宇宙船はシーン上のただのオブジェクトとして扱われます。

解除シーケンス自身です：



LevelStatusスクリプトに次の関数を追加します。(続けて説明します。)

```
function UnlockLevelExit()
{
    mainCamera.GetComponent(AudioListener).enabled = false;
```

Unityはシーンに付き、1つだけの**Audio Listener**コンポーネントをサポートします。通常これは、シーンのメインカメラに附加されますが、ここでは複数のカメラをシーン内で使用しているので、一度に**Audio Listener**を複数持っていないことを確認しなければなりません。“フェンスが無効化された”サウンドエフェクトを聞きたいので、メインカメラの**Audio Listener**を一時的に無効化します。

次に、シーン切り替えカメラと、**Audio Listener**コンポーネントを有効化します。

```
unlockedCamera.active = true;  
unlockedCamera.GetComponent(AudioListener).enabled = true;
```

バリアフェンスには繰り返すサウンドエフェクトが付加されています。ここでは、再生を停止する必要があります：

```
exitGateway.GetComponent
```

これで、“フェンスが無効化された”サウンドを再生することができます：

```
if (unlockedSound)  
{  
    AudioSource.PlayClipAtPoint(unlockedSound,  
unlockedCamera.GetComponent(Transform).position, 2.0);  
}
```

サウンドエフェクトが開始されると、アニメーションシーケンスも開始させます。これらはスクリプトコードを使って手続き的に行います。次の数行がこのシーケンスを実現します。はじめのラインは、プレイヤが気付くまでの時間を確保するために、シーン切り替えが始まってからの待機時間を与えます。(コメントを残しておきましたので、アニメーションシーケンスを追いかけることができるでしょう)：

```
yield WaitForSeconds(1);  
  
exitGateway.active = false; // ... フェンスが消灯します  
  
yield WaitForSeconds(0.2); //... 一瞬停止します...  
exitGateway.active = true; //... フェンスが再び点灯します...  
yield WaitForSeconds(0.2); //... 一瞬停止します...  
exitGateway.active = false; //... フェンスは永久に消え去ります！
```

ついに宇宙船にアクセス出来ます！最後にしなければならないことは、宇宙船のメッシュコライダコンポーネントをトリガにすることです。

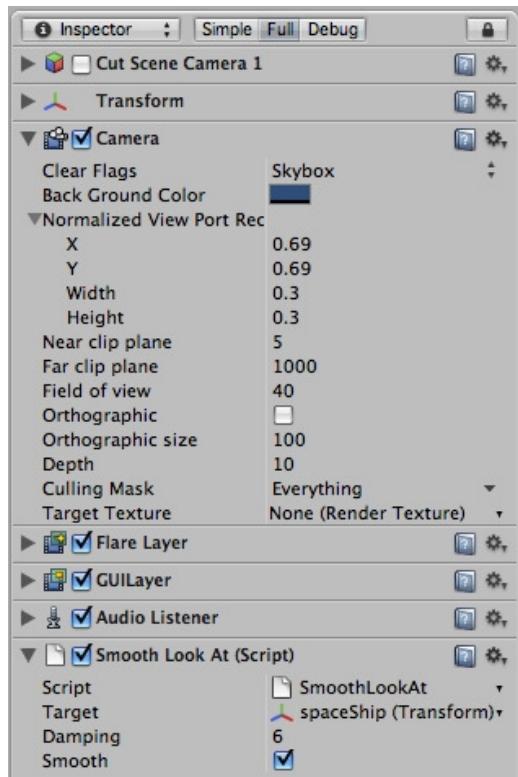
```
levelGoal.GetComponent
```

シーン切り替えカメラを終了し、**Audio Listener**コンポーネントを**NearCamera**に戻す前に、もう数秒停止し、プレイヤに結果を確認する時間を与えます：

```
yield WaitForSeconds(4); // give the player time to see the result.  
  
// swap the cameras back.  
unlockedCamera.active = false; // this lets the NearCamera get the screen all to  
itself.  
unlockedCamera.GetComponent(AudioListener).enabled = false;  
mainCamera.GetComponent(AudioListener).enabled = true;  
}
```

シーン切り替えカメラ自身の作成が次のタスクです。これはただのカメラゲームオブジェクトです。これまでゲームで用いていたものと何ら代わりはありません。さあ、設定しましょう：

- ☞ 新しいカメラをシーンに追加します。
- ☞ 名前を**CutSceneCamera1** に変更します。
- ☞ **SmoothLookAt**スクリプトをカメラに追加します。
- ☞ **spaceShip**モデルへのリンクを**SmoothLookAt**スクリプトの上にドロップすることで、カメラが向くべき対象を教えます。
- ☞ 残りの属性を図のように設定します：



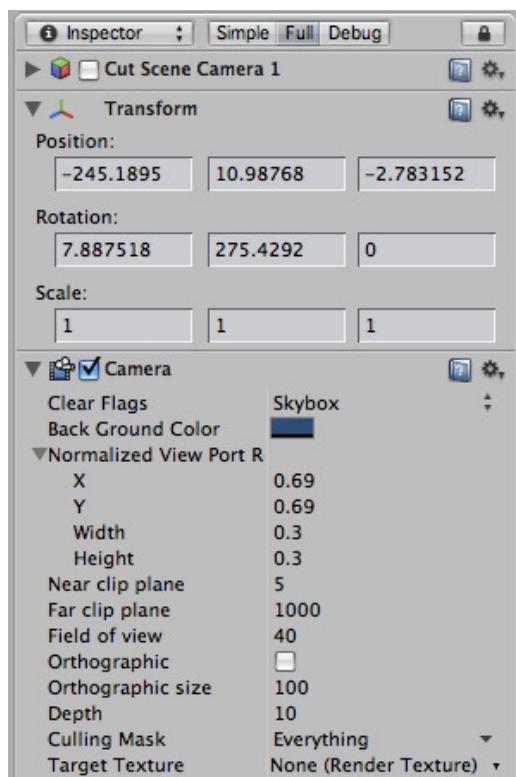
Cut Scene Camera 1 の属性。

このカメラは宇宙船を取り囲んでいるフェンスを写しているべきです。上の設定で大丈夫だと思いま
すが、自由に変更してかまいません。



Cut Scene Camera 1の位置合わせ

次に、CutSceneCamera1カメラ設定を以下のようにします：

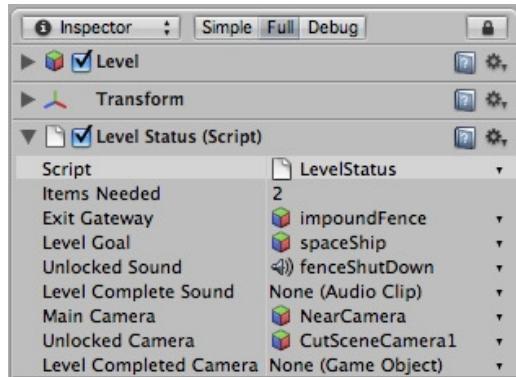


Cut Scene Camera 1カメラコンポーネントを設定します。

カメラはデフォルトで無効化されています。スクリプトで有効化されるまでは、何もレンダリングしてほしくありません。カメラを無効化するには、単に、インスペクタ右上の、**Cut Scene Camera 1**横のチェックボックスをオフにすればよいでしょう。

また、**Normalized View Port Rect** 設定を上に示すように行ってください。これは、カメラを出力するスクリーン上の位置を表します。ここでは、ディスプレイの右上に表示されます。カメラ深度は10に設定されています。Near Cameraより高い値であるため、シーン切り替えはメインのゲーム画像より上にレンダリングされます。

このシーケンスをテストする必要がありますので、**LevelStatus**スクリプトの属性を、下のように設定します：



Level Statusスクリプト属性

spaceShipオブジェクトはバリアフェンスの中にある宇宙船モデルです。

TIP スクリーンショットではItems Neededの値を一時的に2に設定しています。これで、シーン切り替えのテストが、2つのオイル缶を取得するだけ行えますので、レベルの大部分を走り回って時間を無駄にする必要がなくなります。テストが終了したら、値を20など、十分に高い値に忘れずに戻しておいてください。

ここでゲームをプレイすると、シーン切り替えが次のように表示されるはずです。



初めてのシーン切り替え。

NOTE メインビューとシーン切り替えビュー両方で表示されているFPSカウントは最適化の章で説明されています。

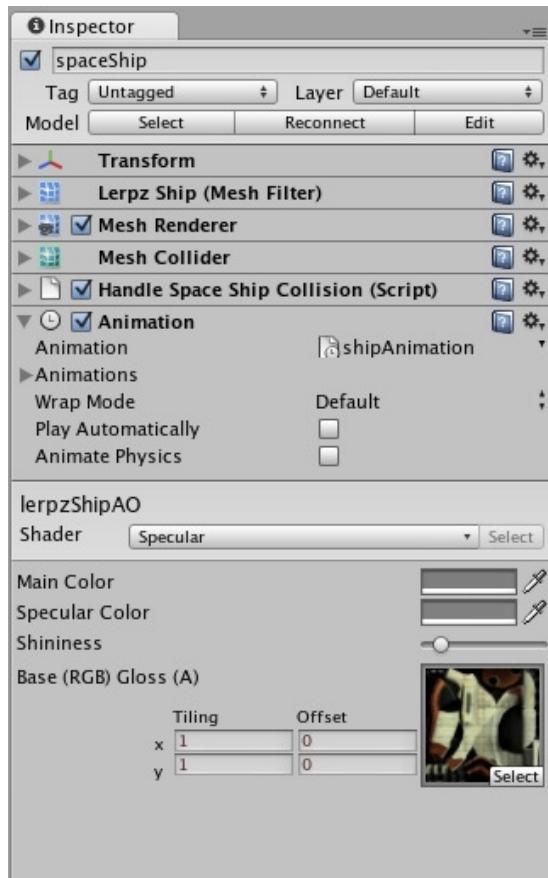
もうひとつのシーン切り替えは少し複雑です。宇宙船を離陸させ、遠くへ飛ばさなければなりません。スクリプトで行うこともできますが、AnimationClipを換わりに使うほうが簡単です：

バリアフェンス内の宇宙船モデルをクリックして選択します。(または、階層ペインでクリックします。)

ShipAnimationアニメーションクリップをプロジェクトビューのAnimationフォルダからドラッグし、インスペクタにおいて、AnimationコンポーネントのSpaceship Animation属性に追加します。

宇宙船のインスペクタにアニメーションコンポーネントがあるのを確認できます。プレイをクリックすると、宇宙船が飛び立つ様子が見られるでしょう。しかしながら、レベルを完了したときにのみ、宇宙船には飛び立ってほしいものです。スクリプトを利用しましょう。Unityはデフォルトではアニメーションを自動的に開始します。それは望ましくないので、ゲームをストップした状態で：

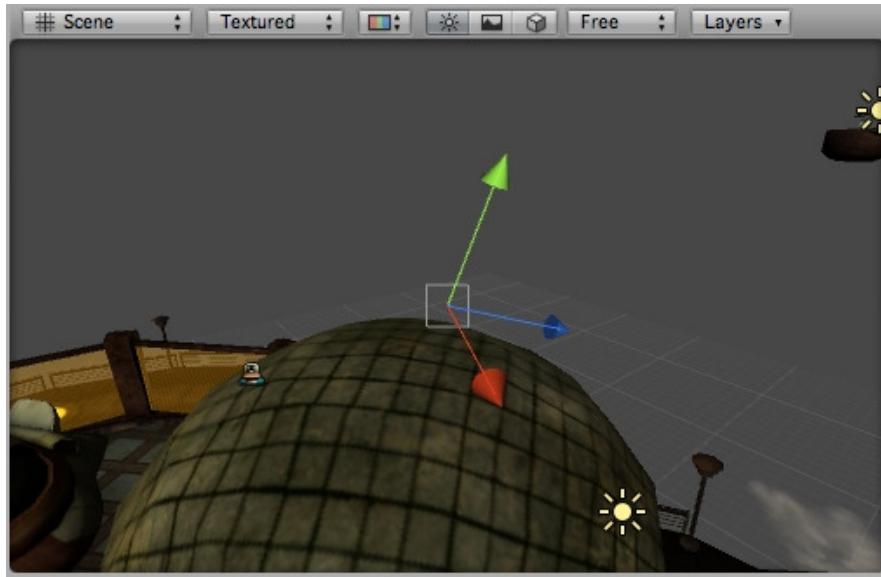
spaceShipアニメーションコンポーネントの、“Play Automatically”チェックボックスを無効化します。



spaceShipオブジェクトのアニメーション設定

次のステップで、2つ目のシーン切り替えカメラを作成します：

- ▣ 新しいCameraオブジェクトを作成します。.
- ▣ 名前をCutSceneCamera2に変更します。
- ▣ 図に示すように、バリアフェンスのオフィスビルの頂上に配置します：

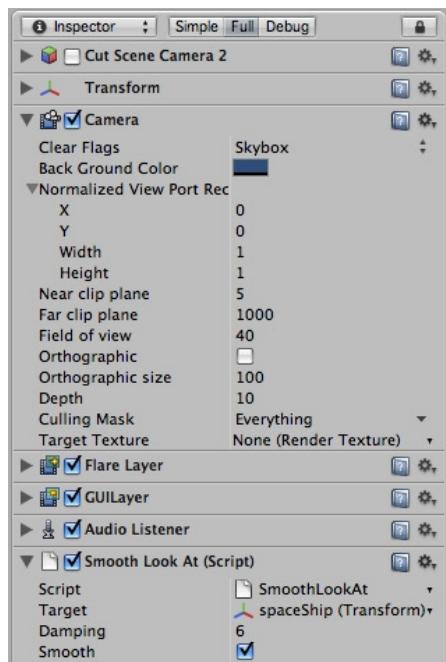


CutSceneCamera2を配置

向きは気にする必要はありません。場所だけが重要です。残りの部分はスクリプトによって処理されます。

- 【】 SmoothLookAtスクリプトをカメラに付加します。
- 【】 spaceShipモデルへのリンクをSmoothLookAtスクリプトにドロップし、カメラが向くべき方向をスクリプトに知らせます。

CutSceneCamera1と同様、デフォルトで無効化されていることを確認しなければなりません。残りの設定は、少し違います。次のページのスクリーンショットで確認できます。重要な点は、カメラが1部ではなく、画面全体を覆わなければならないということです。Normalized View Port Rect属性はそれに応じて設定します。



CutSceneCamera2の設定

次にシーン切り替え自身です。これは、1つ目のシーン切り替えに比べて、少し複雑です。シーンをトリガするメッセージを鎖に沿うようにして伝播しなければなりません：

初期トリガは、プレイヤがspaceShipモデルに触れると発生します。(初めのシーン切り替えが再生されていれば、spaceShipモデルはトリガとして動作しているはずです。)

そのため、spaceShipモデルにはスクリプトを付加し、トリガのイベントを処理する必要があります。

- 新しいJavaScriptのスクリプトアセットを作ります。
- 名前を**HandleSpaceshipCollision**に変更します。
- 次のコードを追加します(完全なスクリプトは付録の章にあります)：

```
private var playerLink : ThirdPersonStatus;

function OnTriggerEnter (col : Collider)
{
    playerLink=col.GetComponent(ThirdPersonStatus);

    if (!playerLink) // not the player.
    {
        return;
    }
}
```

```
    }
    else
    {
        playerLink.LevelCompleted();
    }
}
```

上記のコードでは、プレイヤが宇宙船に触れたかをチェックし、触れていれば**ThirdPersonStatus**スクリプト中の**LevelCompleted()**関数を呼び出します。

 新しいスクリプトを**spaceShip**オブジェクトに追加します。

ThirdPersonStatusスクリプトの**LevelCompleted()**関数は更に短いですが、ほぼ同じ事を処理します。

 次の関数を**ThirdPersonStatus**スクリプトに追加します：

```
function LevelCompleted()
{
    levelStateMachine.LevelCompleted();
}
```

levelStateMachineは**LevelStatus**スクリプトにリンクする属性です。**LevelStatus**には、レベルに関連するスクリプトだけが知しておく必要のある、レベル完了アニメーションのアクションが保持されます。

 次の、**LevelCompleted()**関数を**LevelStatus**に追加します(続けて説明します)：

```
function LevelCompleted()
{
```

初めに、1つ目のカットシーンと同様に、**Audio Listener**を切り替えます：

```
mainCamera.GetComponent(AudioListener).enabled = false;
levelCompletedCamera.active = true;
levelCompletedCamera.GetComponent(AudioListener).enabled = true;
```

次に、プレイヤが宇宙船の中にいるような幻を見せたいので、プレイヤを隠します。

"HidePlayer"メッセージをPlayerのThirdPersonControllerスクリプトに送ることで実現します。関数は、プレイヤのレンダリングを無効化しますので、透明になります：

```
playerLink.GetComponent<ThirdPersonController>.SendMessage("HidePlayer");
```

安全のために(ロボットはプレイヤが見えているかどうかをチェックしないので、消えていても攻撃していく場合が考えられます)、プレイヤをカメラに写らない位置に、物理的に移動します。ここでは、十分すぎるほど安全と考えられる位置として、プレイヤを500ユニット上方に移動します。

```
playerLink.transform.position+=Vector3.up*500.0; // just move him 500 units
```

次に、レベル完了サウンドエフェクトを再生します。ここでは、宇宙船が離陸するサウンドです。

```
if (levelCompleteSound)
{
    AudioSource.PlayClipAtPoint(levelCompleteSound, levelGoal.transform.position,
2.0);
}
```

ここで、事前に録画しておいた、タイムラインベースのアニメーションを実行し、終了するまで待機します：

```
levelGoal.animation.Play();

yield WaitForSeconds (levelGoal.animation.clip.length);
```

最後に、"Game Over"シーンを読み込みます。：

```
Application.LoadLevel("GameOver"); //...just show the Game Over
sequence.
}
```

続いて、レベルを開始したときに、宇宙船のモデルがトリガとなっていない状態に戻します。また、playerLinkがプレイヤゲームオブジェクトを指すように設定します。playerLink属性はUnityがスクリプトをロードしたときに自動的に呼び出す、Awake()関数の内部で定義します。

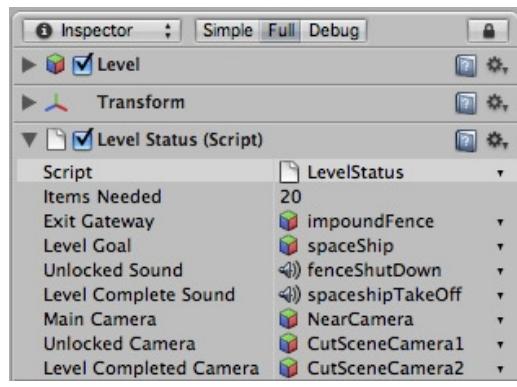


`Awake()`関数を、下記のスクリプトにあうように変更します。初めの関数である、`LevelStatus`の真上にあるはずです。

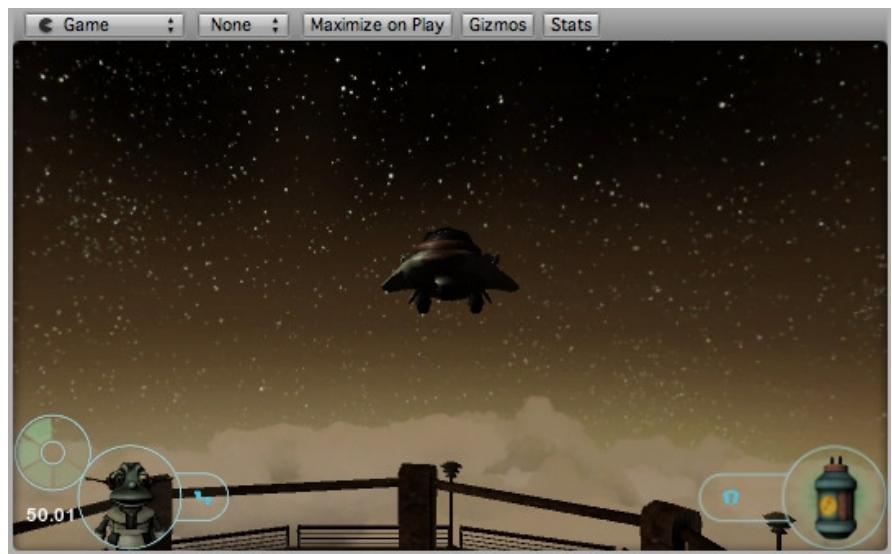
```
private var playerLink: GameObject;

function Awake()
{
    levelGoal.GetComponent(MeshCollider).isTrigger = false;
    playerLink = GameObject.Find("Player");
    if (!playerLink)
        Debug.Log("Could not get link to Lerpz");
    levelGoal.GetComponent(MeshCollider).isTrigger = false; // make very sure of this!
}
```

最後に、更新したスクリプトの属性を次のように設定します：



全てのオイル缶を集めて、宇宙船に飛び乗ると、結果として、次のような映像が再生されます：



作戦終了！私たちのヒーローは、次の冒険に向かって飛び立ちました。

次の章では、最適化のテクニックについて説明します。

最適化 (Optimizing)

ゲームは、G4 iBookから、最新の大容量メモリと、高速なグラフィックカード(2枚の場合も)をつんだ、Intel Mac Proまで、どのような場所でも動かなければなりません。そのためには、最適化を行います、、、



私たちはこれから、ステージに最後の包装作業を行います。最適化は、プロジェクトの最後に行われます、全てのキー要素が配置された後です。何を、どこで、どのようにしてプロジェクトを最適化するのかは、プロジェクトの設計と中身に依存します。このチャプタでは、もっとも一般的な最適化に関して主に扱います。

なぜ最適化するのか？(Why Optimize?)

Unityのプロジェクトは、主流や高性能なものではなく、古いコンピュータをターゲットにしています。パズルゲーム、カジュアルなタイトルや他の種類のプロジェクトでは、256Mbのメモリと古ぼけたグラフィックチップを積んだG4 iBookから、最新のメモリとハイエンドグラフィックカードを積んだIntel Mac Proでも動作しなければなりません。

この理由により、最終リリースのために最適化を考える必要があります。私たちの場合は、レンジに入ったときのみ存在させることによって、敵ロボットを最適化してあります。しかしながら、更に深い部分を行わなければ、シーンのレンダリングは非常に遅くなってしまいます。

レンダリングのFPSをモニタする (Optimizing Rendering Monitoring Frames Per Second)

ゲームを最適化する必要があるかどうかを決定する最良の方法は、フレームレートを確認することです。フレームレートとは、1秒間に描画され、表示される画像の枚数です。

既に“FPS”的文字が、チュートリアル中のスクリーンショットのいくつかに表示されていたのに気が付いていると思います。これは非常に重要ですが、短いスクリプトによって実現されています。

ここで、見てていきましょう。Frames-per-second リポータースクリプトです。スクリプトの名前はシンプルにFPSです。プロジェクトペインのScript->GUIフォルダにあります。

スクリプトはしっかりとドキュメントが書かれています、ここでは詳細について詳しくは述べませが、GUITextコンポーネントの追加を行う方法のみ、説明します。(これはUnity2GUIでは無く、Unity 1.xのGUIコンポーネントです。)

FPSカウンタによって、最適化の良いアイディアを得ることが簡単になります。

NOTE FPSスクリプトはUnityのエディタ内で実行されている場合は、制限された値です。なぜなら、エディタのレンダラは、ディスプレイのフレームレートによってロックされているからです。また、シーンビューを更新し、シーンをプレイしながらより多くのエラーチェックをしなければなりません。正確な値を知るには、プロジェクトをスタンドアロンビルドする必要があります。

状態ディスプレイの理解 (Making sense of the Stats display)

Unityの新機能として"Status"ボタンがゲームビューの上部にあります。これを有効にすると、追加のゲーム評価値を取得できます。これは、ポリゴンの数や、オブジェクトの複雑さを解決する方法を決めるのに役立ちます。



状態パネル

これらの情報は、カメラがレンダリングしているものに基づいています。シーン中を動き回ると、様々に変化するでしょう。重要な要素は：

Draw Calls -- レンダリングのパスの数です。シーン中の要素は複数回レンダリングされる場合があります。影や複数のカメラ、テクスチャのレンダリング、ピクセルライトなどです。複雑なシェーダは追加の描画呼び出しを伴う場合もあります。反射や屈折が計算される場合などです。

Tris -- 描画された3角形の数

全ての3Dモデルは3角形によって構成されています。3角形が少ないほど、早くレンダリングできます。丸や曲線のオブジェクトは、立方体や平面など直線で構成されたものに比べて、基本的に多くの三角形を必要とします。

Verts -- グラフィックチップに送られた頂点の数

頂点は3次元空間の点です。多くの点を3角形で共有できれば、より多くの3角形が描画できるため、複雑なモデルを使用することができます。

Used Textures -- 描画に使用されたテクスチャの数

マテリアルは定義やシェーダスクリプトによって、2つ以上のテクスチャを使うことができます。シェーダはテクスチャがどのように結合され、バンプマップや光沢のあるハイライト、反射や屈折の効果を生み出すかを定義します。

TIP パーティクルシステムは、パーティクルに2つの三角形を使用し、少なくとも1つのテクスチャ(テクスチャは通常、パーティクルシステム内の全てのパーティクルで共有されます。)を使用します。これらの効果は簡単に使うことができますが、使いすぎないように注意が必要です。

Render Textures -- ディスプレイに直接ではなく、テクスチャに出力されているカメラの数。これは、期待されるほど明確ではないでしょう。

Render texturesはいくつかの効果に使われます。ポストプロセッシング効果、CCTVスクリーンをレベルの別のエリアに表示、水中での反射、鏡やガラスの反射効果などです。加えて、ほとんどの影はこのテクニックで提供されます。そのため、シーンビューの中で追加カメラを見る必要はありません。

レンダリングの最適化：2カメラシステム (Optimizing Rendering: The Two-Camera System)

完成したプロジェクトをプレイすると、2つのNearCamera(Near CameraとFar Camera)が存在していることに気が付くでしょう。この、2カメラシステムはそれぞれのフレームでレンダリングされる量を減らすことができます。

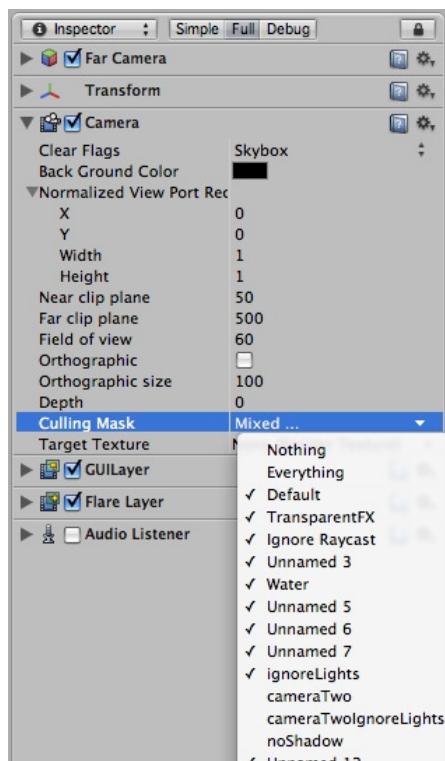
Near Cameraは近いものを描画します。ここでは、0.4から50ユニットの範囲です。

TIP 理論的には、ユニットは自由に決めることのできる長さですが、多くの開発者は”1ユニットが1メートル”として扱います。重要なポイントは、モデリングやテクスチャを作成するアーティストに、あなたが使用しているユニットの長さを教えておくことです。

Far Cameraは50ユニットから500ユニットまでの範囲をレンダリングします。しかしながら、これはシーンデータの要素のみをレンダリングします。要素はレイヤによって定義されます。

シーン中のそれぞれのオブジェクトは、存在するべきレイヤを与えられています。Near Cameraは全ての要素をレイヤによらずレンダリングしますが、下のスクリーンショットで見られるように、Far Cameraは"cameraTwo"と"cameraTwolgnoreLights"レイヤにあるものは無視するように設定されています。更に、オイル缶と体力の取得可能アイテムも無視させます。これらは"noShadow"レイヤに含まれていますので、カメラにおいてチェックがはずされています。

NOTE "noShadow"レイヤのチェックをはずすと、プレイヤもレンダリングされないことになりますが、プレイヤがNear Cameraからそれほど離れた位置に移動することは考えられないので、問題ありません。



Far Camera カリングマスク設定

NOTE Far Cameraはまた、Skyboxをレンダリング(上の画像においてClear Flags設定を見てください)するカメラでもあります。Near CameraのClear Flags設定は"Depth Only"になっていますので、Far Cameraにスーパーインポーズされたものになっています。Far Cameraが先にレンダリングされます。

このセクションは、カメラコンポーネントのCulling Mask属性によって定義されます。チェックされたレイヤがレンダリングされ、チェックされていないものはレンダリングされません。レイヤはインスペクタにおいてゲームオブジェクトに割り当てることができます。

ロボットガードと取得可能アイテムについて、これらの最適化を実際に確認することができます。それらのアイテムに近づくと、アイテムが現れている間、周りのシーンが相対的に近くにレンダリングされるのを確認できるでしょう。

終わりに(End of the road)

ここで学んだこと

次に進むために



あまり探索されていない道 (The Road Less Travelled)

このチュートリアルを書くにあたって、これまでのUnityの製品の中で最も長くなるのではないかとう、ありがたくない名誉を持っていました。3次元プラットフォームと言うジャンルに分類される1つのレベルのゲームをどのように構築するかを見てきましたが、私たちはUnity2で出来ることの表面を、ほんの少し引っかいただけに過ぎません。特定のジャンルについてさえ、その程度です。

私たちがともに歩む冒険は終わりました。次はあなたが自分で自転車にあったタイヤを見つけ、1人で続けて行く番です。でもそのまえに、次に試してみるべきことを幾つか提案しておきましょう、、、

より良いゲームにするための提案 (Suggested Improvements)

Lerpzの脱出は意図的に未完成のまま放置されています。非常に基本的なスタートメニューとゲームオーバースクリーンを作りましたが、たった1つのレベルしかありませんし、ゲームの最後を意識したものです。どのようにしてより良くすることが出来るでしょうか？

意図的なミスの修正 (Fixing the deliberate mistakes)

ゲームに関する小さな要求が沢山あります。これらは、あなたの技術アップのために意図的に残されたものです。それらは：

- ロボットを倒すと、あなたがの外にでる前に、近くに再出現します。しかし、古いロボットが残されたままになっています。
- レーザトラップがプレイヤーを弾きだしません。そのため、プレイヤーはすぐに、体力をすべて失ってしまいます。

両方の問題とも今までにチュートリアルで学んだことを使えば解決できるでしょう。

レベルの追加 (More levels)

プロジェクトペインは“Build Your Own”フォルダを含んでおり、レベルに使用されるそれぞれのアセットを含んでいますので、新しいレベルを追加することは難しくありません。

`DontDestroyOnLoad()`関数を使うことで、ゲームの状態情報をレベルの間で持ち運ぶことができます。例えば、現在のスコアや残っている体力などです。

敵の追加 (More enemies)

ゲームにはロボットガードの形をした動き回る敵が1種類しかいません。もう少し増やしてもいいのではないかでしょうか？AIとアニメーションに関して理解するために、良い勉強となるでしょう。また、モデルを作成し、Unityにインポートすることに関しても学習出来ます。

得点の追加 (Add scoring)

`Lerpz`の脱出にはスコアリングシステムがありません。追加することはそれほど難しくありませんが、プレイヤーが何かをしたときにスコアを増加させたりする事は、すこし挑戦が必要になるでしょう。(もちろん、シーンをまたいでスコアを保持することも可能です)

ネットワークハイスコアシステムの追加 (Add a networked high-score system)

Unity2はしっかりととしたネットワークやウェブサイトへの組み込みをサポートしています。ハイスコアを中央サーバにアップロードする以外に、自慢する方法があるでしょうか？ネットワーキングの基礎を理解するには、この機能を実装してみると良いでしょう。

複数プレイヤーサポート (Add multiplayer support)

ネットワークで接続された複数プレイやゲームのサポートは、どのようなゲームでも利用できる最も技術的なものでしょう。Unityはこれに関してもサポートしていますが、スクリプトを駆使して実現する必要があります。これは、上級レベルとして挑戦するとよいでしょう。

より詳しく (Further Reading)

情報としてはじめに確認すべきは、つねに、Unityのドキュメントです。

多くのチュートリアルも準備されています (Unityウェブサイトには動画もあります) :

<http://unity3d.com/support/documentation/>

加えて、Unity Wikiではユーザによって投稿された素晴らしい情報源です：

<http://www.unifycommunity.com>

最後に、フォーラムを使ってUnityの上級者と初心者の間で意見交換が行えます：

<http://forum.unity3d.com/>

スクリプト付録 (Script Appendix)

チュートリアルで使用したスクリプト



StartMenuGUI script

StartMenuGUI スクリプトのコードです :

```
//@script ExecuteInEditMode()

var gSkin : GUISkin;
var backdrop : Texture2D;
private var isLoading = false;

function OnGUI()
{
    if(gSkin)
        GUI.skin = gSkin;
    else
        Debug.Log("StartMenuGUI : GUI Skin object missing!");

    var backgroundStyle : GUIStyle = new GUIStyle();
    backgroundStyle.normal.background = backdrop;
    GUI.Label ( Rect( ( Screen.width - (Screen.height * 2)) * 0.75, 0, Screen.height * 2,
    Screen.height), "", backgroundStyle);

    GUI.Label ( Rect( (Screen.width/2)-197, 50, 400, 100), "Lerpz Escapes",
    "mainMenuTitle");
```

```

if (GUI.Button( Rect( (Screen.width/2)-70, Screen.height -160, 140, 70), "Play"))
{
    isLoading = true;
    Application.LoadLevel("TheGame");
}

var isWebPlayer = (Application.platform == RuntimePlatform.OSXWebPlayer ||
Application.platform == RuntimePlatform.WindowsWebPlayer);
if (!isWebPlayer)
{
    if (GUI.Button( Rect( (Screen.width/2)-70, Screen.height - 80, 140, 70),
"Quit")) Application.Quit();
}

if (isLoading)
{
    GUI.Label ( Rect( (Screen.width/2)-110, (Screen.height / 2) - 60, 400, 70),
"Loading...", "mainMenuTitle");
}
}

```

GameOverGUI

GameOverGUIスクリプトのコードです :

```

@script ExecuteInEditMode()

var background : GUIStyle;
var gameOverText : GUIStyle;
var gameOverShadow : GUIStyle;

var gameOverScale = 1.5;
var gameOverShadowScale = 1.5;

function OnGUI()
{
    GUI.Label ( Rect( (Screen.width - (Screen.height * 2)) * 0.75, 0, Screen.height * 2,
Screen.height), "", background);

    GUI.matrix = Matrix4x4.TRS(Vector3(0, 0, 0), Quaternion.identity, Vector3.one *
gameOverShadowScale);
    GUI.Label ( Rect( (Screen.width / (2 * gameOverShadowScale)) - 150,
(Screen.height / (2 * gameOverShadowScale)) - 40, 300, 100), "Game Over",
gameOverShadow);
}

```

```

        GUI.matrix = Matrix4x4.TRS(Vector3(0, 0, 0), Quaternion.identity, Vector3.one * gameOverScale);
        GUI.Label ( Rect( (Screen.width / (2 * gameOverScale)) - 150, (Screen.height / (2 * gameOverScale)) - 40, 300, 100), "Game Over", gameOverText);
    }
}

```

GameOverScript

GameOverScriptスクリプトのコードです :

```

function LateUpdate ()
{
    if (!audio.isPlaying || Input.anyKeyDown)
        Application.LoadLevel("StartMenu");
}

```

ThirdPersonStatus

ThirdPersonStatusスクリプトのコードです :

```

// ThirdPersonStatus: Handles the player's state machine. //
// Keeps track of inventory, health, lives, etc.

var health : int = 6;
var maxHealth : int = 6;
var lives = 4;

// sound effects.
var struckSound: AudioClip;
var deathSound: AudioClip;

private var levelStateMachine : LevelStatus; // link to script that handles the level-
complete sequence.

private var remainingItems : int; // total number to pick up on this level. Grabbed from
LevelStatus.

function Awake()
{
    levelStateMachine = FindObjectOfType(LevelStatus); if
    (!levelStateMachine)
        Debug.Log("No link to Level Status");

    remainingItems = levelStateMachine.itemsNeeded;
}
// Utility function used by HUD script:

```

```

function GetRemainingItems() : int
{
    return remainingItems;
}

function ApplyDamage (damage : int)
{
    if (struckSound)
        AudioSource.PlayClipAtPoint(struckSound, transform.position); // play the 'player
was struck' sound.

    health -= damage;
    if (health <= 0)
    {
        SendMessage("Die");
    }
}

function AddLife (powerUp : int)
{
    lives += powerUp;
    health = maxHealth;
}

function AddHealth (powerUp : int)
{
    health += powerUp;
    if (health>maxHealth) // We can only show six segments in our HUD.
    {
        health=maxHealth;
    }
}

function FoundItem (numFound: int)
{
    remainingItems-= numFound;

    if (remainingItems == 0)
    {
        levelStateMachine.UnlockLevelExit(); // ...and let our player out of the level.
    }
}

function FalloutDeath ()
{
}

```

```

        Die();
        return;
    }

    function Die ()
    {
        // play the death sound if available. if
        (deathSound)
        {
            AudioSource.PlayClipAtPoint(deathSound, transform.position);
        }

        lives--;
        health = maxHealth;

        if(lives < 0)
            Application.LoadLevel("GameOver");

        // If we've reached here, the player still has lives remaining, so respawn.
        respawnPosition = Respawn.currentRespawn.transform.position;
        Camera.main.transform.position = respawnPosition - (transform.forward * 4) +
        Vector3.up; // reset camera too
        // Hide the player briefly to give the death sound time to finish...
        SendMessage("HidePlayer");

        // Relocate the player. We need to do this or the camera will keep trying to focus on the
        // (invisible) player where he's standing on top of the FalloutDeath box collider.
        transform.position = respawnPosition + Vector3.up;

        yield WaitForSeconds(1.6);    // give the sound time to complete.

        // (NOTE: "HidePlayer" also disables the player controls.)

        SendMessage("ShowPlayer"); // Show the player again, ready for... // ...
        the respawn point to play it's particle effect
        Respawn.currentRespawn.FireEffect ();
    }

    function LevelCompleted()
    {
        levelStateMachine.LevelCompleted();
    }

```

LevelStatus

LevelStatusスクリプトのコードです :

```

// LevelStatus: Master level state machine script. var
exitGateway: GameObject;
var levelGoal: GameObject;
var unlockedSound: AudioClip;
var levelCompleteSound: AudioClip;
var mainCamera: GameObject;
var unlockedCamera: GameObject;
var levelCompletedCamera: GameObject;

// This is where info like the number of items the player must collect in order to
complete the level lives.

var itemsNeeded: int = 20; // This is how many fuel canisters the player must collect.

private var playerLink: GameObject;

// Awake(): Called by Unity when the script has loaded.
// We use this function to initialise our link to the Lerpz GameObject.
function Awake()
{
    levelGoal.GetComponent(MeshCollider).isTrigger = false;
    playerLink = GameObject.Find("Player");
    if (!playerLink)
        Debug.Log("Could not get link to Lerpz");
    levelGoal.GetComponent(MeshCollider).isTrigger = false; // make very sure of this!
}

function UnlockLevelExit()
{
    mainCamera.GetComponent(AudioListener).enabled = false;
    unlockedCamera.active = true;
    unlockedCamera.GetComponent(AudioListener).enabled = true;
    exitGateway.GetComponent(AudioSource).Stop();

    if (unlockedSound)
    {
        AudioSource.PlayClipAtPoint(unlockedSound,
            unlockedCamera.GetComponent(Transform).position, 2.0);
    }
    yield WaitForSeconds(1);
    exitGateway.active = false; // ... the fence goes down briefly...
    yield WaitForSeconds(0.2); //... pause for a fraction of a second...
    exitGateway.active = true; //... now the fence flashes back on again...
    yield WaitForSeconds(0.2); //... another brief pause before...
    exitGateway.active = false; //... the fence finally goes down forever!
    levelGoal.GetComponent(MeshCollider).isTrigger = true;
    yield WaitForSeconds(4); // give the player time to see the result.
}

```

```

// swap the cameras back.
unlockedCamera.active = false; // this lets the NearCamera get the screen all to
itself.
    unlockedCamera.GetComponent(AudioListener).enabled = false;
    mainCamera.GetComponent(AudioListener).enabled = true;
}

function LevelCompleted()
{
    mainCamera.GetComponent(AudioListener).enabled = false;
    levelCompletedCamera.active = true;
    levelCompletedCamera.GetComponent(AudioListener).enabled = true;
    playerLink.GetComponent(ThirdPersonController).SendMessage("HidePlayer");
    playerLink.transform.position+=Vector3.up*500.0; // just move him 500 units

    if (levelCompleteSound)
    {
        AudioSource.PlayClipAtPoint(levelCompleteSound, levelGoal.transform.position,
        2.0);
    }
    levelGoal.animation.Play();
    yield WaitForSeconds (levelGoal.animation.clip.length);
    Application.LoadLevel("GameOver"); //...just show the Game Over sequence.
}

```

HandleSpaceshipCollision

HandleSpaceshipCollisionスクリプトのコードです：

```

function OnTriggerEnter (col : Collider)
{
    playerLink=col.GetComponent(ThirdPersonStatus); if
    (!playerLink) // not the player.
    {
        return;
    }
    else
    {
        playerLink.LevelCompleted();
    }
}

```