

2D Gameplay Tutorial

Providing 2D Gameplay with Unity 2

Author: Graveck Interactive

Last Revision: 19-FEB-2008

Contents

1. Introduction	
Why 2D is the Bee's Knees	4
Prerequisites	6
2. Setting The Scene	
Getting in the 2D Mind-Set	7
A Closer Look	9
Setting the Level Attributes	10
Object Profile: Level Attributes	12
Object Profile: Death Zone	13
Setting Up the Platforms	13
Object Profile: Platform (Prefabs)	15
Object Profile: Pre-Assembled Platform	16
Introducing Lerpz, Our Lovely Character	17
Placing Lerpz	18
Directing the Camera	19
Object Profile: Character (Lerpz)	20
Lighting our Scene	22
Spicing up the Character	24
The Camera	26
Object Profile: Main Camera	27
Continuing our Level	28
Script Profile: CameraTargetAttributes	28
Handling Rigidbodies	29
Object Profile: Crate	30
Exercise	31
Moving Platforms	32
Object Profile: Moving Platform	33
The Spaceship	34
Object Profile: Spaceship	35
Well Done!	36
3. Delving Deeper: Scripting Examples	
Studying the Scripts	38
Moving Platform Particle Effects	38

The Camera Scrolling Script	40
Streamline Your Workflow	41
Spaceship Script Part 1: Defining Helper Classes	43
Spaceship Script Part 2: Controlling the Spaceship	44
Spaceship Script Part 3: Special Effects	47
Finished!	47
4. What's Next?	
Your Turn	48

Introduction

Sometimes 2D gameplay calls and you can't resist its alluring charm! Unity can handle it.



This tutorial shows the steps you need to create a basic 2D-style platform game.

Why 2D is the Bee's Knees

It is no secret that Unity can help you create great three-dimensional games with ease. Its raw power and flexibility allows even beginners to create impressive games without much difficulty. Unity is undoubtedly a great 3D game engine -- so why on earth would you want to create a 2D game with it?

For certain game types, that extra "D" in 3D can be a hindrance. Some games simply work better with two-dimensional gameplay mechanics. Think of those simple and fun 2D side-scrolling games you have played in the past, or maybe you have a simple puzzle game idea that would work best constrained to two dimensions -- the list keeps going but the bottom line is many games can benefit by the use of only two dimensions.

Two-dimensional games are also commonly easier for the casual gamer to understand, which is something to consider if you're a small game developer (see the article "Casual Games as a Business").

Luckily, Unity is extremely flexible and can easily handle 2D gameplay! Its world class Ageia PhysX is highly adaptable and can be constrained in many ways including two dimensions.

In this tutorial we define a 2D Game as a game where graphics are still in 3D, but restricted motion causes the physics and gameplay only to occur in a single 2D plane. This lets us use 3D models for our content and makes life easier for artists and developers alike. One could say that we're using the best of both worlds: the ease of 2D

gameplay and the beauty of 3D graphics -- with the added attraction that existing developers already have a good 3D asset production pipeline in place.

This tutorial will guide you in building a 2D platform game. We have designed this tutorial with beginner / intermediate-level users in mind; we only ask that you have a baseline knowledge of Unity. We have also included some advanced concepts that may be of interest to more seasoned users.

We begin by getting you into the 2D mind-set. Then we will walk you through building a 2D world in which your character will move around. After adding a few cool obstacles and scenery, we'll lastly add a rocket ship for the player to fly.

We hope you enjoy this tutorial as much as we had fun creating it!



The tutorial's demo level in action.

Prerequisites

Here are the tools and knowledge you should have before you begin this tutorial:

- Unity 2

Although you can benefit from this tutorial if you're a Unity 1.x user, there are some features we use within this tutorial that are only found in Unity 2.0 and above.

- Basic Scripting.

We assume that you already have a grasp of basic scripting principles.

- Familiarity with the Unity user interface.

You should also be familiar with Unity's key interface elements, such as the Inspector, the various Views, and basic drag-and-drop techniques. If you are unfamiliar with Unity, please take a look at our "Introduction to Unity" videos. You'll find these in our website's Resources area.

- 3D Modeling Tools (Recommended).

Although not required since we supply the models, it is recommended that you have tools to examine the 3D assets. Such tools include Autodesk Maya 8, 3D Studio Max, Cinema 4D, and Cheetah3D. (NOTE: Blender can export FBX files, but cannot currently import this format.)

- 2D Graphics Tools (Recommended)

Again, it is not required but we recommended that you have tools to examine the 2D assets we supply. Such tools may include Adobe Photoshop, Corel Painter, or one of the budget alternatives such as Pixelmator.

Also, be sure to download the project file meant to accompany this tutorial. You'll find the files in our Resources section: <http://unity3d.com/support/resources/>

Setting The Scene

It's time to prepare for our journey.

In this section we start putting our scene together and look at how Unity can be made to handle 2D games.



This section of the tutorial will mostly deal with how to set up GameObjects in the scene view, adding Components, and how to manipulate them in the Inspector.

We provide you with a relatively empty scene and your mission is to build onto that scene to make up a more complete level. Bottom line is, we are providing the tools you need and you need to use them to construct a level. Later on in this tutorial we delve a bit deeper and explain how some of the specific scripts work.

Getting in the 2D Mind-Set

Now we need to get in the proper mind-set and create some common conventions that will remain consistent throughout our project. First we need to define our plane of motion. In other words, we need to restrict motion to only two of the three dimensions, traditionally named X, Y, and Z. To do this we must specifically decide which axis of motion will have no movement.

TIP When in the Scene View you can remember which color correlates to which axis with the simple mnemonic device "RGB = XYZ"

The usual convention is for the X-axis to correspond to horizontal movement relative to the observer, while the Y-axis corresponds to vertical movement. The Z-axis therefore corresponds to movements towards and away from the observer -- for our purposes, this means the Z-axis corresponds to motion towards and away from our camera.

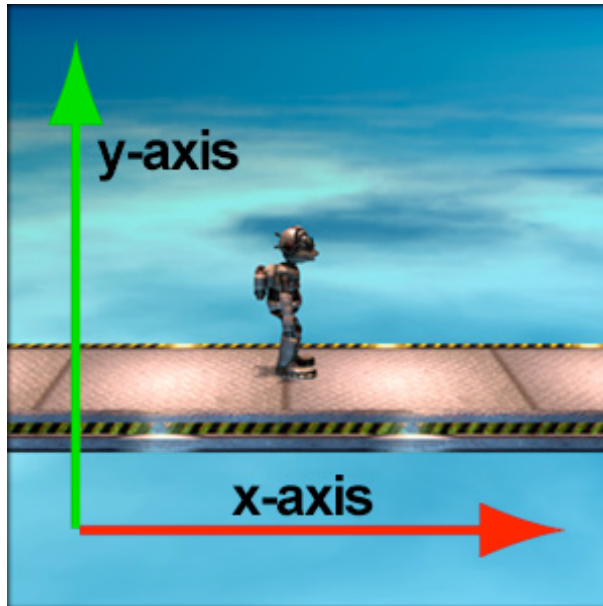
Throughout the tutorial let's keep the following in mind:

- Defining our Plane of Motion.

We are going to stick with common convention and have our motion be in the X-Y plane, meaning that no movement will occur in the Z-axis. Vertical movement will be in the Y-axis and horizontal movement will be in the X-axis.

- Restricted Rotation.

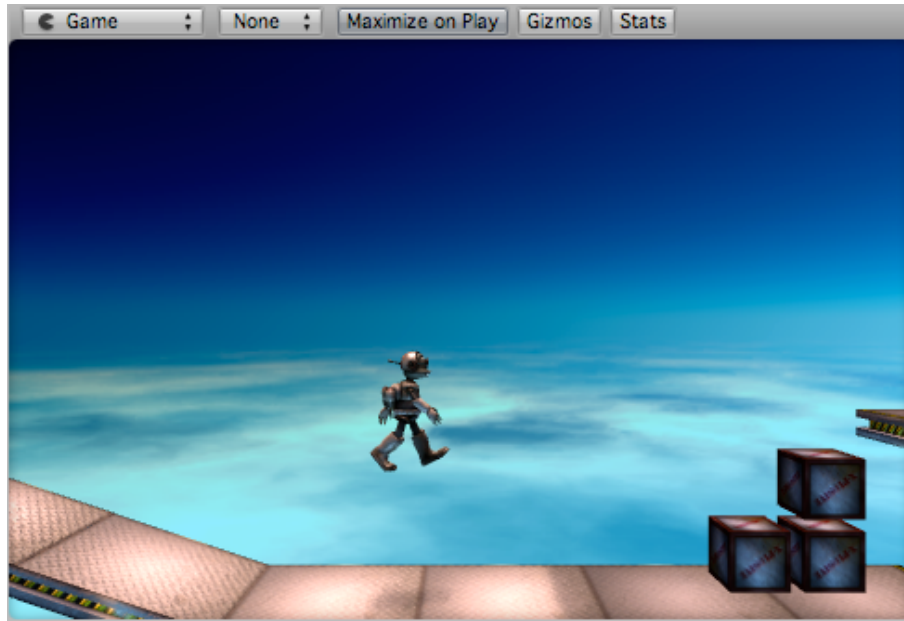
We also need to restrict our rotation. As a general rule, the only rotation that we will allow is rotation around the Z-axis. Remember the Z-axis passes through the camera; therefore rotation around the Z-axis will result in clockwise and counterclockwise rotation as the Main Camera sees it. There is one exception to this restricted rotation, however. The character is allowed rotation about the y-axis so he can turn from side to side.



A 2D game needs just two axes: X and Y.

With all this in mind, it's finally time to open our project and scene. If you haven't already done so, please download the project files now.

- 📁 Select File->Open Project and locate the 2D Tutorial project that you have downloaded.
- 📁 Once you have opened the project, find the scene 2D Platformer in the Project Pane and open it.
- 📁 Play the Scene.



The 2D Platformer Scene.

Controls

Use the cursor keys to move Lerpz. Hold down the Ctrl key to make him run and use the space bar to jump.

The GUI buttons at the top toggle between controlling the character and the spaceship. Everything in this scene has already been laid out for you. Feel free to explore and examine everything you can. The more familiar you are with how we set up our level, the easier it will be to create your own.

A Closer Look

Here are two things you can investigate to better understand how to set up a 2D game before you do it yourself:

Object Positions

If you bring up an element in the Inspector, you'll see that all objects are placed at zero in the Z-axis, keeping to the convention we defined above.

When modeling your 3D assets, it may make your life easier to model them in your modeling application with the same orientation as in Unity. Although not crucial, this avoids the inconvenience of having to rotate them after you import them.

Platform Tiling

If you investigate the platforms in this scene you'll notice that they are made up of smaller, tile-able pieces, with end-caps for the beginnings and ends of platforms. This can save hours of modeling time so platforms can be built within Unity and easily changed. Everything from the size to the textures are modeled with tiling in mind.

NOTE Tiling was a very common technique in the early days of computer games. It had the very useful advantage of reducing the amount of graphical assets required, keeping the project's overall size down. In the days when computer memory was measured in kilobytes, this was an important factor. Today, tiling remains a useful technique as it reduces both asset production time and download size.

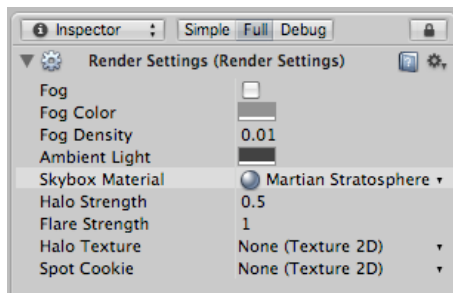
Setting the Level Attributes

After playing our pre-built level, it's time to begin our own.

Create a new scene by going to File->New. You now have a completely empty scene that is ready to rock. Before you begin, make sure that your interface is set up how you prefer. Be sure that the Scene View, Project Pane, Hierarchy Pane, and Inspector are all present.

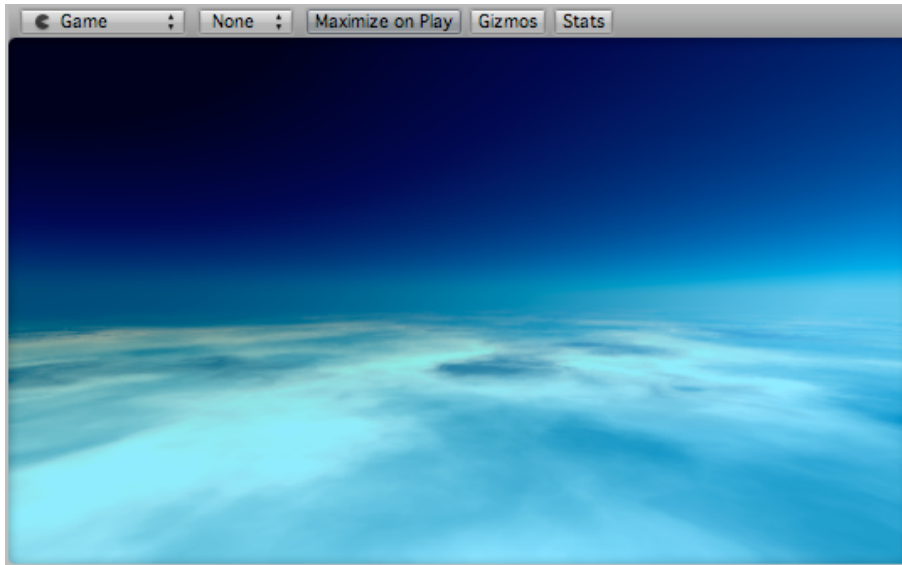
First thing to do is to define our Skybox:

- ✦ Go to Edit->Render Settings in the menu. In the Inspector, you should see an empty slot for a **Skybox**.
- ✦ In the Project Pane, expand the folder Standard Assets->Skyboxes.
- ✦ Drag Martian Stratosphere into the **Skybox Material** slot in the Inspector for the Render Settings.



The Render Settings

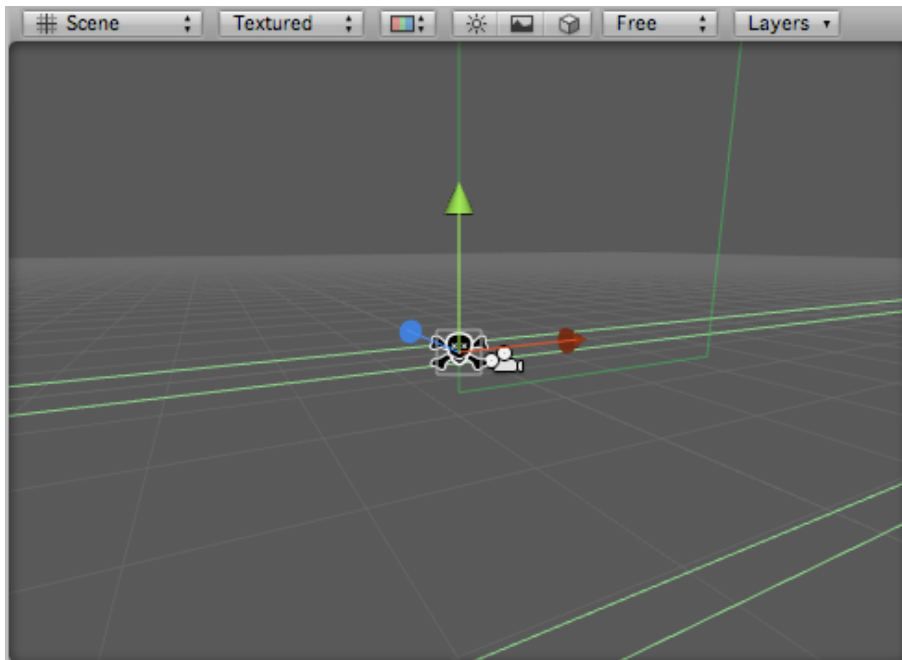
Now we should have a nifty skybox that we will see in the background at all times in our game.



The Game View, showing the Skybox.

Next, we need to add something which will manage some important attributes.

- ✎ Look in the Project Pane and find the Prefab named Level Attributes within the Level Prefabs folder.
- ✎ Drag the Prefab into the Hierarchy Pane so it is loaded in your scene.
- ✎ Ensure that its Position is at (0, 0, 0). This will make life easier in the long run for the child object "Death Zone."



The Level Attributes Prefab in the Scene View. We see two new objects here: the small rectangle rising upwards defines the level's boundaries. The long oblong lying flat is the "Death Zone" (displaying the skull).

In the Scene View you should now be able to see a green rectangle that represents the size of your level. If you cannot see the green rectangle, adjust your viewing position so that you can clearly see it.

🔍 Adjust the settings according to how big you want your level to be.

As a reference point, the completed level that we supplied was 44.25 units wide by 60 units tall. Make sure that everything is located at zero in the Z-axis. The size of your level can easily be changed later so don't worry about getting it absolutely right.

Finally...

🔍 SAVE your scene!

So what's going on here? Lets take a much closer look at the **Level Attributes** and **Death Zone** GameObjects...

Object Profile: Level Attributes

The Level Attributes object started as an empty GameObject at position (0, 0, 0). The script **LevelAttributes** has been applied to it.

Functions

- `OnDrawGizmos()`

Displays the level's dimensions. The level's size is denoted by the green-bordered rectangle drawn in the scene view.

- `Start()`

Creates physics Colliders at startup that act as borders around the world to prevent the character from leaving the level boundaries.

Script: LevelAttributes

This script defines the level's key attributes based on the values you set in the Inspector:

- **Bounds**
 - **X:** The x-coordinate that your level will start at.
 - **Y:** The y-coordinate that your level will start at.
 - **Width:** The width of your level starting at the X-coordinate above.
 - **Height:** The height of your level starting at the Y-coordinate above.

- **Fall Out Buffer:** This creates a buffer zone at the bottom of your level. Its purpose is to give your character some room to fall without the camera following until it hits the bottom Collider.
- **Collider Thickness:** The thickness of the Colliders that create your borders.

Further Analysis

If you select the **LevelAttributes** GameObject after pressing play, then look at the green boundary rectangle in the scene view, you will notice Colliders (represented by light green boxes) at the edges. These are the automatically-generated physics Colliders created by the **LevelAttributes** script.

Object Profile: Death Zone

The Death Zone object started as an empty GameObject at position (0, 0, 0). It is a child object of Level Attributes. It does not need to be a child of Level Attributes, but for organizational purposes works well there. The script "DeathTrigger" has been applied to it.

Function




Provides a Collider -- defined as a Trigger -- which causes the character to re-spawn if he/she falls onto it. This is useful for pits that the player can fall into.

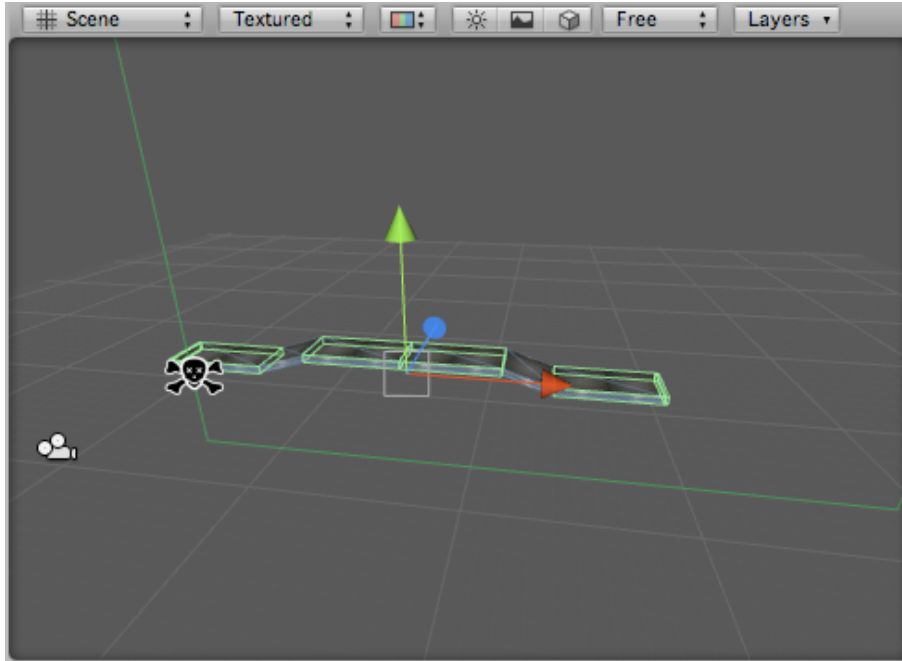
Script: DeathTrigger

This script draws the little skull and crossbones that you see in the scene view by taking advantage of Gizmo icons. Although it's not necessary, it is useful. ("Setting Up the Platforms" has more information on Gizmos.)

Setting Up the Platforms

Now it's time to create the foundation of our level: the platforms.

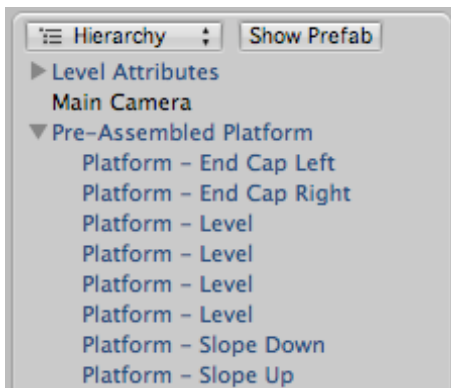
-  In the Project Pane you will see a prefab called "Pre-Assembled Platform." Drag it into the Scene.
-  Ensure the platform is located at zero along the z-axis.
-  Position the platform in a place you're happy with.



Placing the Pre-Assembled Platform Prefab.

This platform was pre-assembled for you, but you will have to make the rest yourself.

Look in the Hierarchy Pane and you'll see that the Pre-Assembled Platform has several child objects. These children objects are what make up the platform. If you investigate the platforms you'll notice that they are made up of smaller, tile-able pieces, with end-caps for the beginnings and ends:



Looking inside the Pre-Assembled Platform Prefab.

Using these platform tiles, create more platforms for your character to run wild!

Keep these pointers in mind:

- Always make sure the z-position is at zero.
- Look at the pre-assembled platform for reference.

- We highly recommend that you parent groups of platform tiles together, for reasons described in Object Profile: Pre-Assembled Platform, below.

To do this, create an empty GameObject by going to GameObject->Create Empty and name it appropriately. Make sure the GameObject is located at zero along the Z-axis. Also make sure that the X and Y coordinates are where you want the platform to be located. Then you can drag the platform tiles inside the empty GameObject you just created. Last, apply the script **CombineChildren** so you mimic the function of the Pre-Assembled Platform GameObject described above.

- The tiles can be precisely positioned easily because their lengths are nice whole-number values.
- The platforms must stay inside of the green level boundary you set up in the previous section! If your platforms are outside this area they will be unreachable. You can always change the size of the boundaries.
- Save frequently!

You can always create more platforms later, but you at least want an area that the character can run around.

Object Profile: Platform (Prefabs)

In the Project Pane you can find these prefabs inside Level Prefabs/Platforms.

Functions

These are the tile-able pieces that make up the larger platforms. You'll find five different pieces within the Platform Tiles folder:

- Platform - Level:
A level piece of platform that is 6 units long.
- Platform - Slope Up:
A piece of platform that slopes up moving from left to right. It is 3 units long.
- Platform - Slope Down:
A piece of platform that slopes down moving from left to right. It is 3 units long.
- Platform - End Cap Left:
This belongs at the left edge of an array of platforms to cap it off.
- Platform - End Cap Right:
This belongs at the right edge of an array of platforms to cap it off.

The length of these pieces, (with the exception of the end caps since it is not necessary,) are nice whole numbers so they can be precisely tiled by editing the numbers in the Inspector.

Components

- **Box Collider**

Without this, the character and spaceship would fall right through the platform! This is needed to physically interact with the character. The Box Collider size has been adjusted to fit the platform tile, otherwise all settings are default. Since the platform does not move at all, we do not have to apply a Rigidbody component.

- **Pipeline, Mesh Filter, and Material:**

These assign the proper mesh and texture so it renders in the scene.

Script: CollisionSoundEffect

The **foot** script attached to our player checks `GameObject` it comes into contact with for the presence of this script and plays the sound effect it holds (if you have set one). In this way our player character's footsteps can be made to sound different on different surfaces -- e.g. a metallic sound on metal platforms; a wooden footstep sound on wooden surfaces, and so on.

Script: CollisionParticleEffect

This script, like the one above, provides a similar service for particle effects. The player's **foot** script looks to see if we've set a particle effect `GameObject` and starts it if so. This is how the character raises the clouds of dust as he runs around the example level.

TIP If you're curious, examine the texture (which is the same for each tile) to see how it loops.

Object Profile: Pre-Assembled Platform

It's the parent object that contains platform tiles that create a single large platform.

Function:

This `GameObject` serves two purposes.

- As a container used to organize the level and
- To aid performance -- see the script described below.

It is recommended that you structure your entire level using this method. In other words, for each group of platform tiles that make up a large platform, create a parent object for them with the **CombineChildren** script applied.

Script: CombineChildren

In the Inspector you can see this script with its self-explanatory options. What this does is take all of the children GameObjects -- i.e. the platform tiles -- and combine their meshes so instead of rendering 6 different objects, only 1 object is rendered. This enhances performance, especially when you reach a large number of platforms.

TIP This script actually comes in the Standard Assets package, which is shipped with Unity.

Introducing Lerpz, Our Lovely Character



Lerpz, our zany character, is getting anxious for action!

Lets get our spawn point set up first. The spawn point is where the character will appear when you start the game. It's pretty simple to do, just follow these steps:

- 🔗 Create a new empty GameObject by going to GameObject->Create Empty. Rename it to "Character Spawn Point".

- 🔗 Apply the script titled **SpawnPoint** to the GameObject you just created. This script creates that little 2D icon of Lerpz inside the Scene View so you can see where the spawn point is. (This little drawing is called a Gizmo. The Death Zone also has one of these to display the skull-and-crossbones icon.)
- 🔗 Position the spawn point where you want Lerpz to appear. Make sure it is at z = 0 with a platform underfoot, so he actually has a place to stand when he appears.

What exactly is a Gizmo?

Gizmos are shapes, icons and other visual aids that appear in the Scene View such as lines, lamps, cameras, and audio sources to assist in visual debugging or layout. Gizmos drawn exclusively in this 2D tutorial are the Death Zone, and Character Spawn Point.

All it takes is a very simple script. Here's an example of the script **SpawnPoint** that is applied to the Character Spawn Point:

```
function OnDrawGizmos()
{
    Gizmos.DrawIcon(transform.position, "Player Icon.tif");
}
```

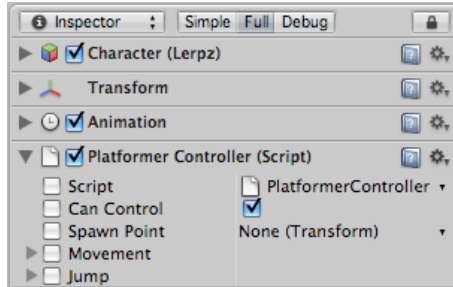
Gizmo icons must be placed in the Assets/Gizmos folder.

The **LevelAttributes** script also takes advantage of Gizmos: The green border you see representing the level boundaries is a Gizmo. For a complete reference on Gizmos, you can refer to the documentation (<http://unity3d.com/Documentation/ScriptReference/Gizmos.html>).

Placing Lerpz

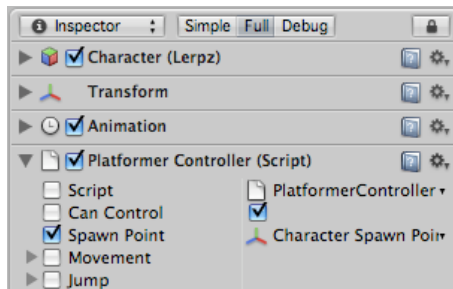
The next step is to add Lerpz himself to our Scene:

- 🔗 Drag the Prefab "Character (Lerpz)" in our Level Prefabs directory into the Scene. Since we have a spawn point set up, you can place him anywhere you'd like, just as long as he's somewhere in the Scene.
- 🔗 If it's not already visible, bring up our Character in the Inspector and open up the Platform Controller Script Component. Note the Spawn Point slot:



Character (Lerpz) with no Spawn Point set.

👉 Drag our earlier Character Spawn Point object into this slot, so it looks like this:



Character (Lerpz) with the Spawn Point set.

The next step is to get the Main Camera to follow Lerpz around...

Directing the Camera

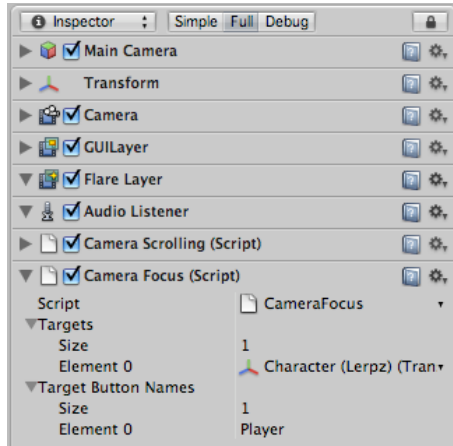
At the moment, our main camera will simply sit where it is and point at whatever it's aimed at. Much like in a movie or TV production, we need to direct the camera -- tell it how to move, when to do so and what it should be aimed at. We achieve this with two scripts:

- 👉 Drop the CameraScrolling script (from Scripts->2D->Camera) onto the Main Camera GameObject.
- 👉 Drop the CameraFocus script (from the same location) onto the Main Camera.

The next step is to give the scripts the information they need to work:

- 👉 Select the **Main Camera** in the Hierarchy Pane and open up the **CameraFocus** script Component in the Inspector.
- 👉 Open the **Targets** group and change **Size** to 1. **Element o** should appear below it.
- 👉 Drag our **Character (Lerpz)** GameObject onto the **Element o** slot.
- 👉 Open the **Target Button Names** group and change **Size** to 1. (Again, **Element o** should appear.)
- 👉 Type "Player" into **Element o's** slot.

The result should look like the image below:



Setting up the Camera Focus script component.

We should now have a working camera and character running and jumping over our platforms.

It was very easy to get the character in the level using Prefabs, but on closer inspection Lerpz is anything but simple:

Object Profile: Character (Lerpz)

Lerpz is the main character of our game. He can walk, run, jump, and fall. He moves around our level wreaking havoc wherever he goes.

Components

- **Animation**

This component stores a list of animations for the character that can be accessed by scripting. **PlatformPlayerAnimation** (listed below) makes use of this component by accessing different animations.

- **Character Controller**

The traditional side-scrolling character controls are generally not physically realistic. The character comes to a halt immediately and turns on a dime, making it difficult and impractical to use Rigidbody physics. As a result, we use an alternate method to move the character around by using the Character Controller. See more detail on this component in the documentation.

(<http://unity3d.com/Documentation/Components/class-CharacterController.html>)

Script Applied: PlatformerController

This script is a custom script made for this 2D tutorial. This script, in cooperation with the **CharacterController** component, is used to define its physical mechanics within our world. Most of the adjustable settings are self-explanatory and we will not go into much detail, but a few require some further discussion:

- **Speed Smoothing**
How quickly does the character change speeds? For example, how quickly can the character go from not moving to running. Higher means faster.
- **Rotation Smoothing**
How fast the character visually changes directions. It does not affect the motion.
- **Can Control**
This simply states whether the player can control the character. This can be turned off by another script if at any time you want user to lose or gain character control. An example of this being toggled is when you toggle between the spaceship and the player.
- **Spawn Point**
You must define the spawn point by dragging a Transform object into the slot. Then the character will spawn at that point.

Script: **PlatformerPlayerAnimation**

This script takes the animations listed in the Animation Component and plays them at the appropriate times. For example it detects when a player is on the ground and moving at a certain speed, and then plays the appropriate animation such as "run."

Script: **PlatformerPushBodies**

Since the character is not a Rigidbody, it does not automatically physically interact with everything around it. This script allows the character to apply forces to Rigidbodies such as the crates within the level.

The setting **Push Power** is the strength of the push applied by the character.

The **Push Layers** setting determines what layers the character can actually push. Sometimes, you may not want a specific Rigidbody to be pushed by the character but still interact physically with the world around it, so you can create a layer for that object make sure that the character cannot push that layer with this setting.

Script: **CameraTargetAttribute**

This defines some camera behaviors, and is also applied to the Spaceship. This script is more thoroughly discussed in the next section.

Further Analysis

We have just scratched the surface of explaining the character. To fully benefit you, it would be wise to examine and experiment with all of the scripts attached to the character to see how they work. We have supplied comments in the code to help guide you along.

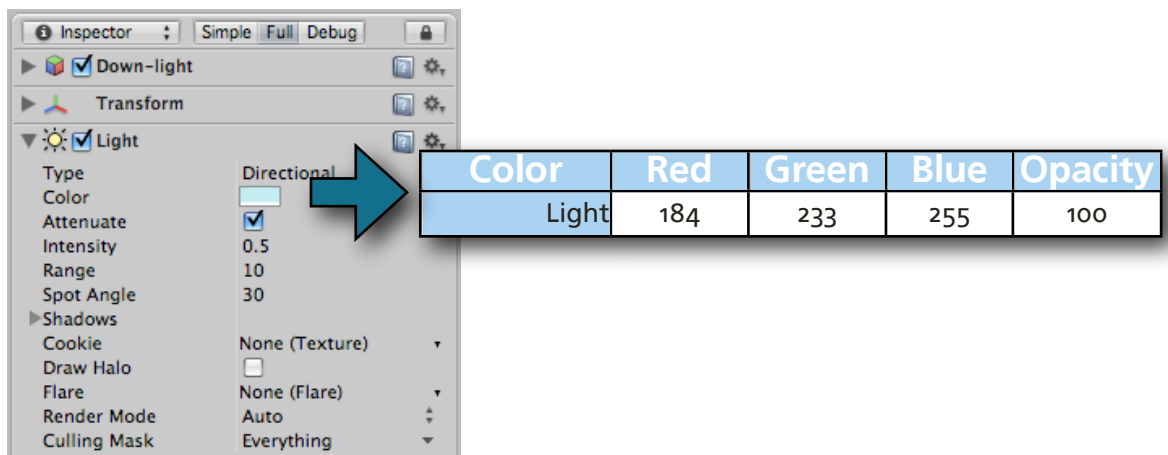
Take a few moments to play around with Lerpz's settings. Adjust the different settings to see what they do, and if things get too crazy you can always delete him from the scene and add a new one using the Prefab!

Lighting our Scene

At this stage we have our character running around our Scene, but it's all very dark. We need some lights to make everything easier on the eye!

We'll add two Directional Lights:

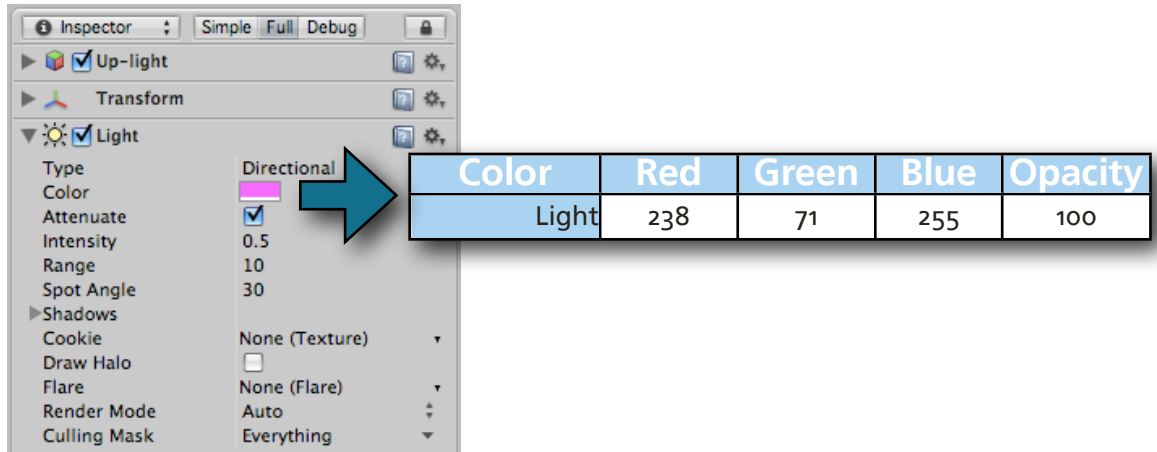
- ❏ Create an Empty GameObject. Name it "Main Lights". (We'll use this to organize our lights and keep them in one place.)
- ❏ Create a Directional Light. (GameObject->Create Other->Directional Light).
- ❏ Rename it "Down-light".
- ❏ Drag the light into our Main Lights object.
- ❏ Rotate our Down-light object to point about 60 degrees downwards.
- ❏ Set it as shown below (**Intensity** is also changed from the default, to 0.5):



The light blue down-light's settings.

Now we'll repeat the process again, but for our up-lighter. This will light the under-sides of the platforms:

- ❏ Create a Directional Light. (GameObject->Create Other->Directional Light).
- ❏ Rename it "Up-light".
- ❏ Drag the light into our Main Lights object.
- ❏ Rotate our Up-light to point about 60 degrees upwards.
- ❏ Set it as shown below (**Intensity** is again changed from the default, to 0.5):

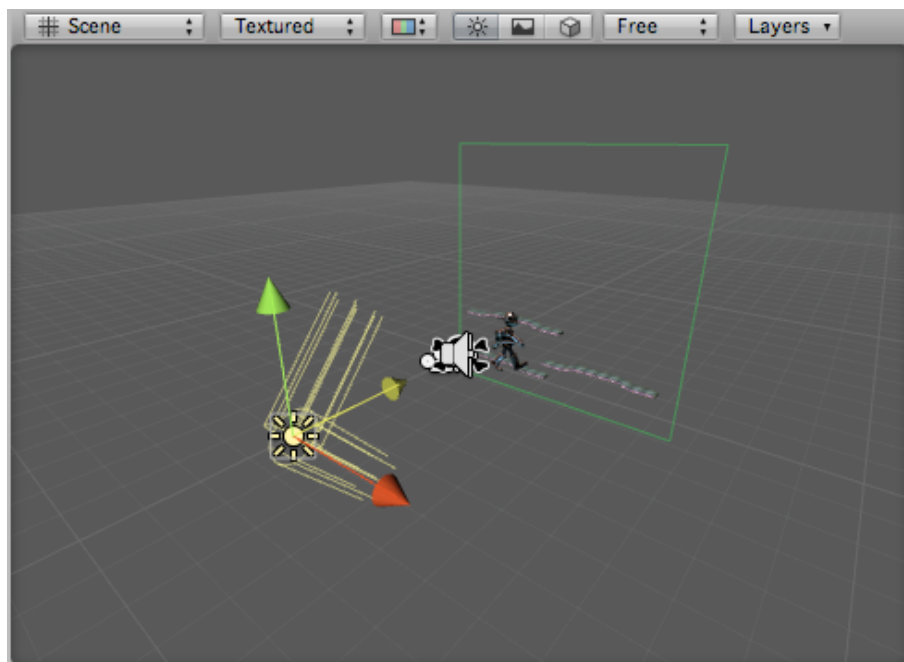


The purple up-light's settings.

The next step is purely for our own convenience:

- ☞ Select our Main Lights parent object and drag it backwards, away from the platforms (see screenshot below). Precision isn't important here: this is just to keep the lights out of the way while we work -- as they're Directional Lights, their distance from the platforms has no effect.

Our completed lighting rig should look something like this, (although your level's platforms will probably be very different):



Our lights, positioned away from our Camera and level, so we can see them more clearly. The "Main Lights" object is highlighted here, showing both the child lights.


TIP The light colors chosen in this example are hardly subtle, but they clearly show how each light affects the scenery. Feel free to change the colors!

Spicing up the Character

Lerpz has a couple of simple visual touches: a working jet-pack and dust clouds kicked up when he walks. Let's look at these in more detail:

Jet Pack Particle Effects

If you watch the character's jet pack it only fires when he jumps and is moving upwards. Additional scripting is needed to accomplish this. We also need to make some particle effects.

 In the Hierarchy Pane, expand the Character, then expand **rootJoint** and lastly expand **torso**. Here you will see where the **Rocket Jet** GameObjects are located.

The code to start the jets is inside the **PlatformerController** script within the `Update()` function.

TIP Unity's Particle System features are covered in depth in our Resources area. Check out Forest "Yoggy" Johnson's video on the subject, as well as our 3D Platformer Tutorial.

Footsteps in the dust

As Lerpz walks, we hear his footsteps and see him kicking up plenty of dust. (Clearly it's been some time since these platforms were given a good clean!)

If you expand the many levels of Lerpz's legs in the Hierarchy Pane, you'll notice that each foot has a **FootEffect** GameObject attached to it. This contains three Components and a script:

- **RigidBody**
This component is required if we want Unity to detect collisions.
- **AudioSource**
Essentially a placeholder for a footstep sound. When our script detects that our foot has struck another object, it checks if that object has a **CollisionSoundEffect** script. If it does, it grabs that sound effect, drops it into this **AudioSource** component and plays it, adding a subtle, random variation to the pitch each time for variety.
- **Sphere Collider**
Unity's physics engine uses this to determine when the foot has touched something. This is set as a trigger. When this Collider touches the collider in a platform object, the foot script reacts accordingly by scanning the object we've struck for **CollisionSoundEffect** (see above) and **CollisionParticleEffect** scripts, playing the relevant effects accordingly.

The foot script does nothing other than the tasks mentioned above and is therefore very short and simple:


```
var baseFootAudioVolume = 1.0;
var soundEffectPitchRandomness = 0.05;
```

These two variables define the volume of the footstep sound effect, as well as how much variation is applied to its pitch. (This variation is used to prevent the footstep sounding too artificial.)

The first function we define is `OnTriggerEnter()`. This is called whenever the Sphere Collider touches another Collider, such as that attached to a platform element:

```
function OnTriggerEnter (other : Collider) {
    var collisionParticleEffect : CollisionParticleEffect =
    other.GetComponent(CollisionParticleEffect);

    if (collisionParticleEffect) {
        Instantiate(collisionParticleEffect.effect, transform.position, transform.rotation);
    }
}
```

The above code looks for a `CollisionParticleEffect` script in the object we've collided with. If it finds one, it fetches the particle system Prefab we've stored in there and instantiates it at the foot's position. This is how we produce the little clouds of dust.

The next section of code does something very similar for the footstep sound effect:

```
var collisionSoundEffect : CollisionSoundEffect =
other.GetComponent(CollisionSoundEffect);

if (collisionSoundEffect) {
    audio.clip = collisionSoundEffect.audioClip;
    audio.volume = collisionSoundEffect.volumeModifier * baseFootAudioVolume;
    audio.pitch = Random.Range(1.0 - soundEffectPitchRandomness, 1.0 +
soundEffectPitchRandomness);
    audio.Play();
}
}
```

This code looks for a footstep sample in the object we've collided with. If it finds one, it grabs it, copies it into the **FootEffect** GameObject's own `AudioSource` component, adds a bit of random variation to its pitch, then plays it.

The only other function in this script is `Reset()`, which simply ensures two key properties are set to the correct default values. This function is called automatically by



Unity when the script is first added to a GameObject, and it can also be called manually by clicking on the cog-wheel icon to the right of the script component's name in the Inspector.

```
function Reset() {  
    rigidbody.isKinematic = true;  
    collider.isTrigger = true;  
}  
  
@script RequireComponent(AudioSource, SphereCollider, Rigidbody)
```

The very last line in the code is a Unity directive which ensures the three components listed are present in our GameObject. This saves having to remember to add them manually.

The Camera

It's now time to look at operating the Camera. We need it to follow our player around the level.

-  Save the scene you have built so far and reopen the Scene named 2D Platformer that came pre-made with this tutorial.
-  Play the Scene for a moment and observe the camera movement.
- You should see that not only does the camera follow the character, but the character's position shifts as he walks further in a single direction. This subtle movement allows the player to more easily view what's ahead. This is a technique which can be seen in games as old as Williams' classic arcade game, "Defender".
- In addition to this basic movement, we also added a "Springiness" property to our camera's script: When the character jumps, it is preferable not to have the camera follow the entire jumping motion because it makes for some jerky camera movement. To solve this problem, we provide a setting that allows the camera to lag a little so it doesn't follow every single motion.
- The third and final point to note is the camera's behavior at the boundaries of the world. Notice how the camera does not move past the boundaries.

Having a well-placed camera is almost always difficult no matter what type of game you are making, and therefore can often get complicated. The camera motion we provide in this tutorial may not be perfect in every way, but it is a good place to start.

Object Profile: Main Camera

This provides the player's view of our Scene and is an important part of our game.

Components

- **Camera**
This is the core component for a Unity camera. It is a complex component which relies on the following Components, (which Unity will normally add automatically):
 - **GUI Layer**
This allows 2D GUIs to be rendered.
 - **Flare Layer**
This allows lens flare effects to be rendered. These are beyond the scope of this tutorial.
 - **Audio Listener**
This is basically a microphone attached to the camera. Traditionally, the microphone is attached to the camera because it allows for more realistic stereo sound as it represents the player's point of view. Without this component, we would not hear any audio while playing our game.

Script: CameraScrolling

This is the script that makes the camera move how we want. There isn't much to adjust here except for:

- **Distance**
This is the distance in the z-direction that the camera is from the target object.
- **Springiness**
As we stated earlier, when the character jumps it is preferable not to have the camera follow the entire jumping motion because it makes for jerky motion. This setting defines how responsive our camera will be to the target's motion.

Script: CameraFocus

This script allows us to easily switch between different targets for the camera. In our case, the targets will be Lerpz and the Spaceship.

NOTE This script also takes advantage of the new GUI features in Unity 2.0. In the pre-made scene we have set up for you, this script is what allows you to toggle between the Lerpz and the Spaceship.

Let's take a look at the adjustable settings:

- **Targets**
You can set as many targets as you like. Our pre-assembled scene has two. You need to drag a Transform object from the Hierarchy into the slot.

- **Target Button Names**

This should have as many names as there are targets. These are the names that appear on the buttons of the GUI display panel.

Further Analysis

We will take a closer look at the camera scripts in the next section of this tutorial.

Continuing our Level

Now that we have an idea what components and scripts we need, it's time to get the camera working in our new Scene...

- 🔗 Reopen your scene, if it's not already open.
- 🔗 Experiment with the **Springiness** and **Distance** settings in the **CameraScrolling** script. Set them according to your preference.

There is another camera-related script which is not applied to the camera itself. We need to examine this too. It is the **CameraTargetAttributes** script.

This script is applied to the Character, and later will be applied to the Spaceship.

Script Profile: CameraTargetAttributes

Provides a few additional camera movement properties that aren't applied onto the camera itself.

Applied To:

Character (Lerpz) and Spaceship

Key Properties

Height Offset

When at 0, the target will be sitting vertically in the center of the screen. If it is set to a positive number, that means the camera will shift upwards, resulting in a vertically off-center target.

Distance Modifier

The distance of the camera from the target is defined in the **CameraScrolling** script, but this setting adds on to that value. In the pre-assembled scene the rocket has a higher distance modifier, so the camera pulls back a little when you choose to control it.

Velocity Look Ahead

This defines how quickly the camera shifts to "look ahead" when a character is moving.

Max Look Ahead

This distance is where the "looking ahead" will stop so the character doesn't leave the screen. X represents the horizontal distance, Y represents the vertical distance.

- ⚡ Before you move on, play around with the Lerpz's **Camera Target Attributes** settings to understand what this script does.

Orthographic Projection

Even though we are working with 2D mechanics, the graphics are still 3D and everything still has a bit of perspective to it. For you traditionalists out there, just follow these steps so it's strictly side-view orthographic with no perspective:

- ⚡ In the Hierarchy, select your Main Camera, and check the box in the Inspector that says "Is orthographic."

In the Game View you should see a strict orthographic view of your level without perspective. But there's a problem now, the Skybox has disappeared because it doesn't work for orthographic views. No need to worry, a few quick adjustments and we're back in business!

- ⚡ Adjust a setting of the Main Camera in the Inspector. By default, the setting **Clear Flags** is set to **Skybox** but we want it to be set to **Depth Only**.

Now we need to add a new camera to render the backdrop.

- ⚡ Go to GameObject->Create Other->Camera.

This new camera will be set to only render the Skybox and nothing else.

- ⚡ Change the **Depth** setting to "-2", which sets it to render before the Main Camera and therefore will appear in the background.
- ⚡ Now the Culling Mask setting needs to be changed to **Nothing**, otherwise your level may be rendered more than once.
- ⚡ You can also adjust the **Field of View** so that you can see as much or as little of the skybox as you desire.

Now you have a side-scroller that has a more traditional feel without perspective!

Handling Rigidbodies

Rigidbodies are the gateway for applying physics to your objects. (If you are unfamiliar with the concept of a Rigidbody, please view the documentation.)

Our job is to tell the Rigidbodies how to behave in this restrictive 2D plane of motion. The platforms will never move so the Rigidbody component is not necessary, but the

rocket ship and the crates need to move and physically interact with their surroundings. This section explains how to restrict motion of these Rigidbodies.

If you were to examine both the crate and the rocket in the Inspector View of our pre-assembled scene, you would notice two major things that are specific to our 2D game:

- They are both located at 0 along the Z-axis. This is important because every object needs to be at the same place along the Z-axis.
- They also have a component called "Configurable Joint." This is a new component in Unity 2.0 which offers countless possibilities, but for our purposes it is used for restricting motion.

Notice that **ZMotion** is locked, which disallows motion in the Z-axis. **Angular XMotion** and **Angular YMotion** are also locked, preventing any forbidden rotations conforming with what we decided in the previous chapter. The **Configure In World Space** checkbox is also selected. All other settings are left at their default values.

Lets take a closer look at the crate. (The Spaceship has additional components added, and we will look at that later.)

Object Profile: Crate

In the Project Pane you can find this prefab inside of the directory Level Prefabs. The crate is not meant to be stationary, but rather interact with the character and the world around it.

Components

- **Box Collider**
Without the Box Collider, the object would not be able to physically interact with its environment. All of the settings here are the default settings.
- **Rigidbody**
This also allows the crate to physically interact with its environment. The settings here are also the default values, but you can adjust them to make it more massive, to ignore gravity, or to be immovable.
- **Configurable Joint**
This is the component that restricts our movement to our 2D plane. All settings are default except for the following four:
 - **ZMotion:** Locked
 - **Angular XMotion:** Locked

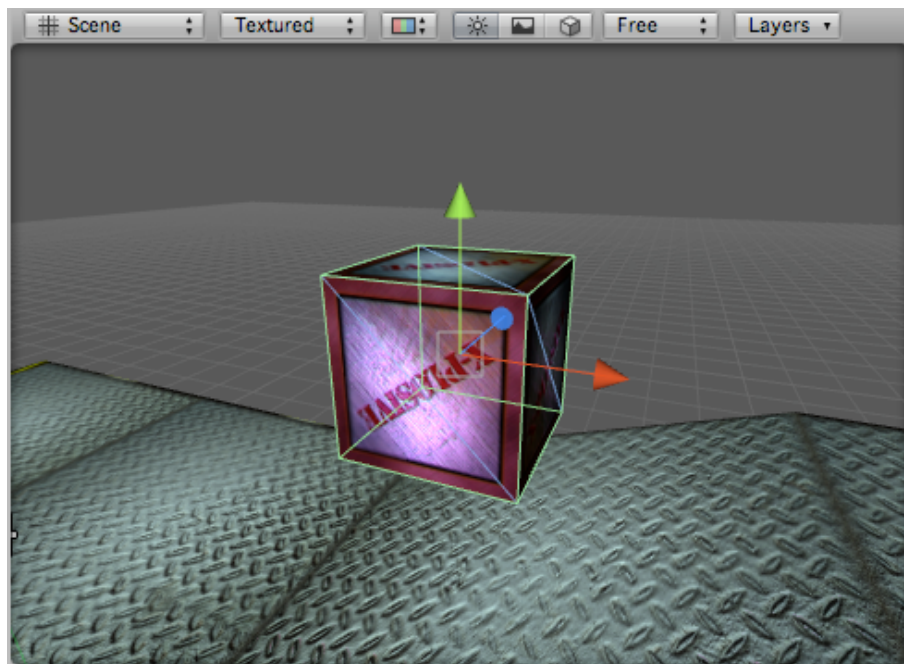
- ▶ **Angular YMotion:** Locked
- ▶ **Configure in World Space:** Checked

- **Pipeline, Mesh Filter, and Material**

These assign the proper mesh and texture so it renders in the scene.

Script: CollisionSoundEffect

This script assigns a sound for when an object collides with it. The colliding object must activate the sound through scripting.



The humble crate. So versatile! So easy to model! Where would the games industry be without them?

Exercise

- 🔧 Add as many crates to your scene as you want. You can orient them however you want as long as they're located at zero along the z-axis.
- 🔧 After you have added crates, step it up by creating your own Rigidbody to be loaded in the scene.

Try creating a sphere that the character can roll when it is pushed. Apply the same components and settings to the sphere, but make sure you add a Sphere Collider instead of a Box Collider.

- 🔧 Don't forget to save your scene!

Moving Platforms

A common ingredient in a side-scrolling 2D game is a moving platform. To create a moving platform we're going to position two waypoints in the Scene that the moving platform will oscillate between.

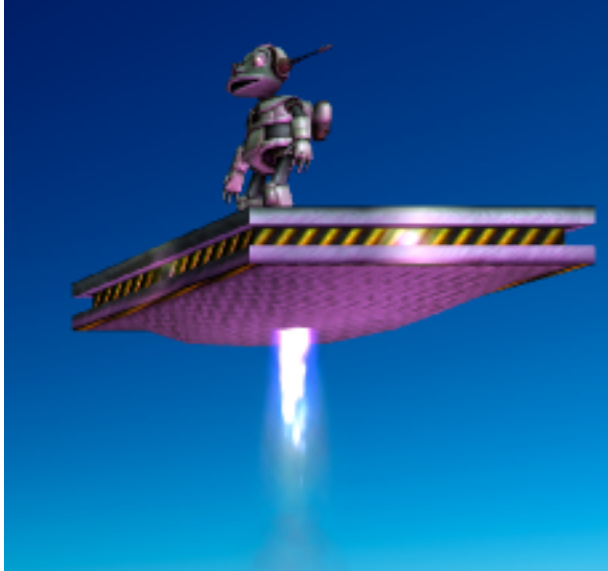
- 📌 Drop the **Moving Platform** Prefab into your scene.
- 📌 Drop a **PlatformWayPoint** Prefab into your scene and position it where you want the **Moving Platform** to start from.
- 📌 Duplicate the **PlatformWayPoint** and position it where you want the **Moving Platform** to move to.
- 📌 In the **Moving Platform's** Inspector there is a script named **Platform Mover** with two slots, **Target A** and **Target B**. Drop your two **PlatformWayPoint** GameObjects from the Hierarchy into their respective slots in the Inspector.
- 📌 There is also a setting for **Speed** which controls how fast the **Moving Platform** will move between the two **PlatformWayPoints**.

Now if you hit play, the platform should be moving how you'd expect. If it isn't behaving correctly, adjust your two **PlatformWayPoints** and the **Speed** variable until you achieve the desired result.

TIP You can add as many moving platforms as you'd like. Each moving platform should have its own set of two **PlatformWayPoints** to move between.

Object Profile: Moving Platform

Description: The moving platform that the character can stand on. The moving platform in the pre-assembled 2D Tutorial scene moves vertically, but they can be made to move whichever direction you desire.



Components

- **Box Collider**
Without the Box Collider, the character and other objects would fall right through it!
- **Rigidbody**
This also allows the crate to physically interact with the character and other objects. A few settings deviate from the default. "Use Gravity" setting is off. The "Is Kinematic" setting is on. The "Is Kinematic" setting prevents the character or other Rigidbodies from moving the platform somewhere besides its predefined path.
- **Configurable Joint**
Like the Rigidbodies described in the previous section, this component restricts our movement. Everything is the same except now ALL rotation is forbidden:
 - **ZMotion:** Locked
 - **Angular XMotion:** Locked
 - **Angular YMotion:** Locked
 - **Angular ZMotion:** Locked
 - **Configure in World Space:** Checked

- **Pipeline, Mesh Filter, and Material**

These assign the proper mesh and texture so it renders in the scene.

Script: **MovingPlatformEffects**

This script is used to activate the particle effects on the bottom of the platform. The **Horizontal Speed to Enable Emitters** property defines how fast the platform needs to move horizontally for the jets to fire. It automatically fires for any vertical movement. We'll take a more in-depth look at this script later.

The Spaceship

By now you should have almost everything set up. Lerpz, the character, should be moving around and navigating over crates and movable platforms. The last thing you have to set up is the Spaceship. Adding a spaceship in this tutorial will show you how to add forces to a Rigidbody and make it move.

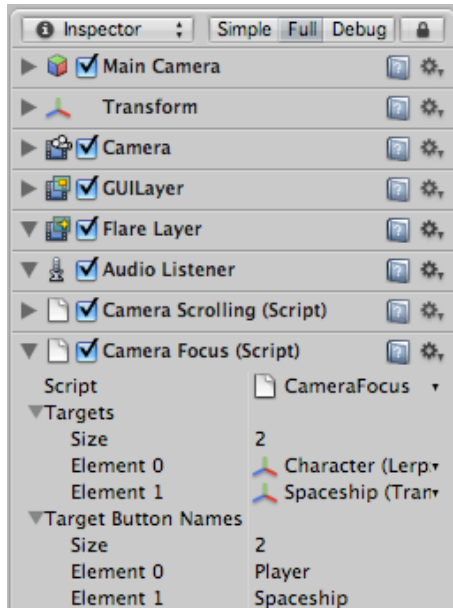
There are only two things you need to do to get the Spaceship fully functional:

- 🔗 First drag the **Spaceship** Prefab into the Scene.
- 🔗 Position it where you want in your level.

Now we need to adjust a few settings in the **Main Camera** so that the camera will switch between the Spaceship and Lerpz. It's time to revisit the **CameraFocus** script:

- 🔗 Change the number of **Targets** to 2 with **Element 0** as **Character (Lerpz)** and **Element 1** as **Spaceship**. This is easily achieved by dragging the respective GameObjects onto the "Element" slots.
- 🔗 You also need to change the size of **Target Button Names** to 2, with **Elements 0** and **1** named **Player** and **Spaceship** respectively. You can see the final settings in the following screenshot.

Now you're good to go!



The updated Camera Focus script settings.

You should by now have a complete and working level! If anything isn't working properly, read back and see if you missed anything. Also remember that you can always open the scene 2D Tutorial and use that as a reference.

⚡ Don't forget to save!

Object Profile: Spaceship

The Spaceship prefab can be found in the director Level Prefabs. It's the Spaceship you fly around the level with.

Components

- **Mesh Collider**
Allows the object to physically interact with its surroundings. It is important that **Convex** is checked. A convex Mesh Collider is able to move in real-time, otherwise it is too computationally expensive to run. To see the definition of a Convex Mesh, you can view the documentation (<http://unity3d.com/Documentation/Components/class-MeshCollider.html>).
- **Rigidbody**
This also allows the Spaceship to physically interact with its environment. The settings used are the default settings when you apply a Rigidbody to an object.
- **Configurable Joint**
The standard settings that we apply to other Rigidbodies in the scene as seen on our crate:
 - **ZMotion:** Locked

- **Angular XMotion:** Locked
- **Angular YMotion:** Locked
- **Configure in World Space:** Checked
- **Pipeline, Mesh Filter, Mesh Renderer, and Material:** These assign the proper mesh and texture so it renders in the scene.

Script: Spaceship

So far the Spaceship is not much different from the crate, but this script is what sets it apart from an inanimate physics object.

The Spaceship script defines how it moves with the input from the keyboard. The adjustable settings on the script are:

- Forward Direction
- Positional Movement
- Rotational Movement.

For both **Positional** and **Rotational Movement**, you'll see that there is a **max speed** and, by default, if that speed is exceeded the drag drastically increases, making it much more difficult to accelerate.

The setting **Can Control** is toggled with the camera script **CameraFocus**, so you need not worry about that setting.

Well Done!

Congratulations! You are finished with building your 2D level. This was no easy task -- throughout the tutorial so far you have been introduced to a handful of fundamental concepts . . . and some more advanced concepts. Here's what you should have accomplished so far:

- Learned how to restrict movement to a 2D plane;
- Built your level with tileable platforms;
- Set your level boundaries;
- Been introduced to Gizmos;
- Explored the fundamentals of setting up a Character;
- Explored the complexities of setting up your camera;

- Seen how to make a strict orthographic camera view;
- Learned how to place and make Rigidbodies;
- Used waypoint system to create moving platforms;
- Set up the Spaceship.

Take a moment for a deep breath and enjoy playing your level. Then, if you're ambitious you can dive into the next chapter.

Delving Deeper: Scripting Examples

Now we've learned how the project is organized and how to use the tools provided, it's time to peer under the hood and see how these tools work.



Studying the Scripts

There are a lot of scripts within this project, and we will only cover a few of the more important ones. For the ones we do not cover in the tutorial text, you will see that the scripts are thoroughly commented so you can walk through them by yourself. These scripts cover both basic and advanced concepts, so whether you're new to Unity or a seasoned user you will benefit from reading them.

Moving Platform Particle Effects

If you watch the moving platform, you'll notice that the particle effect only occurs when it is moving upwards. This effect was accomplished with the script **MovingPlatformEffects**.

In a nutshell, this script compares the previous position with the current position and if it's moving upwards the particle emitters turn on. Note the usage of the `LateUpdate()` function instead of `Update()` -- this is to stop the particle emitter from reacting one frame late.

```
// We will turn on our special effects when the platform is raising, but if it is moving side
// to side, how fast does it have to be moving to cause our special effects to turn on?
var horizontalSpeedToEnableEmitters = 1.0;

// A true/false (boolean) variable to keep track of whether or not our special effects are
// currently turned on.
private var areEmittersOn : boolean;
```

```

// A reference to our other Moving Platform script that is handling all our movement.
// We'll need it to query the current platform velocity for our special effects.
private var movingPlatform : MovingPlatform;

// We are going to use these later to calculate the current velocity.
private var oldPosition : Vector3;
private var currentVelocity : Vector3;

function Start() {
    // Grabs the initial position of the platform.
    oldPosition = transform.position;
}

function Update() {
    // Remember if our emitters were on, then we'll see if they are currently on.
    wereEmittersOn = areEmittersOn;

    // The emitters are on if the vertical (y) velocity is greater than 0 (positive), or if the
    // horizontal velocity in either direction (positive or negative speed) is greater than
    // our horizontalSpeedToEnableEmitters threshold.
    areEmittersOn = (currentVelocity.y > 0) || (Mathf.Abs(currentVelocity.x) >
horizontalSpeedToEnableEmitters);

    // We only have to update the particle emitters if the state of them has changed.
    // This saves needless computation.
    if (wereEmittersOn != areEmittersOn) {
        // Get every child ParticleEmitter in the moving platform.
        for (var emitter in GetComponentsInChildren(ParticleEmitter)) {
            //Simply set them to emit or not emit depending on the value of
areEmittersOn
            emitter.emit = areEmittersOn;
        }
    }
}

function LateUpdate () {
    currentVelocity = transform.position - oldPosition;
    oldPosition = transform.position;
}

// This line tells Unity to nicely place this script in a submenu of the Component menu.
@script AddComponentMenu("2D Platformer/Moving Platform/Moving Platform Effects")

```

The Camera Scrolling Script

Now let's examine how we get the camera to follow the character. To accomplish this we use scripting. A simple yet functional camera script could be applied to the main camera, read the character's X and Y coordinates, and use those values for its own position while keeping a certain distance away in the Z-axis. This works perfectly well, but if you closely examine popular side-scrolling games you'll notice that it's never quite that simple: there are a handful of subtle touches to be made in order to get effective camera scrolling.

All of this movement is defined in **CameraScrolling** found in Scripts->2D->Camera. Open the script and examine it. Use the comments we supplied to understand how it works. Here we will briefly describe what each function does:

- `SetTarget ()`

Look at the top of the script and you'll notice a private variable called **target**. The `SetTarget ()` function is used by other scripts to change the private variable **target**. More specifically, the **CameraFocus** script uses this function when you click on the buttons at the top of the screen to choose whether to control Lerpz or his spaceship.

Why not allow setting of the variable directly? Simple: this way we can ensure that the variable is always set by calling our `SetTarget ()` functions, making it easier to check that the value we're setting it to is a valid one for our purposes.

Note also that we define the `SetTarget ()` function more than once. Each definition accepts a different number of arguments (otherwise Unity will complain). We can use these multiple definitions as an easy way to default the second argument to `FALSE` and also to make our scripts easier to follow. (This programming technique is most commonly referred to as "Overloading".)

- `GetTarget ()`

This is a simple accessor function, (sometimes called a "getter"). It is a publicly callable function that returns a private variable. Notice how **target**, defined at the top of the script, is marked "private"? We can not access it directly from another script, but another script can access this function.

Functions like this enforce a concept called "encapsulation". In Unity, this means our script is completely self-contained. Other scripts don't need to know how it works internally and this helps us reduce bugs, as well as making our scripts easier to re-use in other projects.

- `LateUpdate ()`

We're using `LateUpdate ()` instead of `Update ()` to ensure that the camera isn't

lagging a frame behind of the motion. This function uses the function `GetGoalPosition()` to interpolate between the current camera position and the goal position.

- `GetGoalPosition()`

This function calculates where the camera should be when the next frame is drawn.


(This part of our script seems fairly long-winded, but much of it deals with how the camera behaves at the boundaries of the level, which involves performing similar, but not identical, tests for each edge.)

Streamline Your Workflow

An important concept to keep in mind for any project is how to optimize your workflow.

One way to optimize your workflow is included within this tutorial. By harnessing the flexibility of Unity, you can create scripts that automate certain procedures for you. More specifically, in this tutorial you'll notice a menu at the top of the screen titled "2D" that contains two items that automate the process we just described.

Give one of these menu items a try by dragging a crate prefab into the Scene View.

 Select the crate in the Scene View, go to 2D->Move Onto 2D Plane. If it wasn't already, you'll notice that the crate's Z-position has changed to zero.

The other menu item is not necessary for the crate since that prefab already has the Rigidbody (and other required components) attached.

Creating custom menu commands can be surprisingly simple! Below we explain the script used to create these menu items. The script below, titled **TwoDHelper** can be found in the Project Pane within the Editor folder.

NOTE Scripts that add custom menu items must be located in the Editor folder.

First up, we have the 2D menu's Move Onto 2D Plane function.

This menu item has two functions: `MoveOnto2DPlane()` itself, which performs the function we want; and `ValidateMoveOnto2DPlane()`, which tells the Unity Editor when to enable the menu item. (Not all menu commands will need the latter function.)

```
@MenuItem ("2D/Move Onto 2D Plane ^2")
static function MoveOnto2DPlane () {
    for (var transform in Selection.transforms) {
        transform.position.z = 0;
    }
}
```

```

}

@MenuItem ("2D/Move Onto 2D Plane ^2", true)
static function ValidateMoveOnto2DPlane () {
    return (Selection.activeTransform != null);
}

```

The validation function merely checks that we have a `GameObject` selected. (We actually check if the currently selected object has a `Transform` as all `GameObjects` have one of these `Components`.)

The second menu item performs the same move-to-2D-plane action by calling `MoveOnto2DPlane()` first, then goes ahead and adds the **Rigidbody** and **Configurable Joint Components** we need for a `GameObject` to work in our 2D game. It even ensures the **Configurable Joint Component's** `zMotion`, `angularXMotion` and `angularYMotion` properties are set to "Locked" for us.

NOTE Both our menu commands will loop through whatever `GameObject` we've selected and apply the same instructions on any child `GameObjects` it contains.

```

@MenuItem ("2D/Make Selection 2D Rigidbody")
static function MakeSelection2DRigidbody () {
    MoveOnto2DPlane();

    for (var transform in Selection.transforms) {
        var Rigidbody : Rigidbody = transform.GetComponent(Rigidbody);

        if (!Rigidbody)
            transform.gameObject.AddComponent(Rigidbody);

        var configurableJoint : ConfigurableJoint =
            transform.GetComponent(ConfigurableJoint);

        if (!configurableJoint)
            configurableJoint =
                transform.gameObject.AddComponent(ConfigurableJoint);

        //configurableJoint.configuredInWorldSpace = true;
        configurableJoint.xMotion = ConfigurableJointMotion.Free;
        configurableJoint.yMotion = ConfigurableJointMotion.Free;
        configurableJoint.zMotion = ConfigurableJointMotion.Locked;
        configurableJoint.angularXMotion = ConfigurableJointMotion.Locked;
        configurableJoint.angularYMotion = ConfigurableJointMotion.Locked;
        configurableJoint.angularZMotion = ConfigurableJointMotion.Free;
    }
}

```

```


    }
}

@MenuItem ("2D/Make Selection 2D Rigidbody", true)
static function ValidateMakeSelection2DRigidbody () {
    return (Selection.activeTransform != null);
}

```

Spaceship Script Part 1: Defining Helper Classes

The next script we will be examining is called **Spaceship** and can be found in the Project Pane in Scripts->2D->Spaceship. For the sake of saving space the entire script will not be listed in this text, but can be examined on your own by using the comments we provide throughout.

 While the spaceship is selected in the hierarchy, look in the Inspector at the **Spaceship** component.

Notice the **Positional Movement** settings and **Rotational Movement** settings are identical. They both have **Max Speed**, **Negative Acceleration**, **Drag While Coasting**, etc.

Since these settings are identical, this is a perfect opportunity to create a helper class.

If used correctly, a helper class can be used to create much cleaner and reusable code. You will see later where and how we use this class, but for now we will just define it.

(Another advantage to defining a helper class such as this is its properties will be grouped together by the Inspector, making for a much cleaner interface.)

```

...
class MovementSettings {
    var maxSpeed : float;
    var positiveAcceleration : float;
    var negativeAcceleration : float;
    var dragWhileCoasting : float;
    var dragWhileBeyondMaxSpeed : float;
    var dragWhileAcceleratingNormally : float;

    //A function that determines which drag variable to use.
    function ComputeDrag(input : float, velocity : Vector3) {
        var drag = 0.0;

        //Is the input not zero (the 0.01 allows for some error since we're working with
floats)
        //If the input is zero, use dragWhileCoasting
        if (Mathf.Abs(input) > 0.01) {
            //Are we greater or less than our max speed? and assign the appropriate
drag

```

```

        if (velocity.magnitude > maxSpeed)
            drag = dragWhileBeyondMaxSpeed;
        else
            drag = dragWhileAcceleratingNormally;
    }
    else
        drag = dragWhileCoasting;

    return drag;
}
}

```

The function `ComputeDrag()` defined in the **MovementSettings** class above will be used later in the script. Computing the drag is the same for both angular and positional movement, so putting it in the helper class means we don't have to write the same code twice when determining which drag to use.

Next we will define one more helper class to use later for adding particle effects to the spaceship. This one is a bit shorter and sweeter.

```

//...continued from Spaceship

//Generally always need this next line for your helper classes.
@script System.Serializable
class SpecialEffects {
    var positiveThrustEffect : GameObject;
    var negativeThrustEffect : GameObject;
    var positiveTurnEffect : GameObject;
    var negativeTurnEffect : GameObject;
    var collisionVolume = 0.01;
}

```

Spaceship Script Part 2: Controlling the Spaceship

The next chunk of code shown below is simply defining our variables that will show up in the Inspector.

We're putting our **MovementSettings** helper class into good use. We define both **positionalSettings** and **rotationalSettings** as a **MovementSettings** class type.

```

//...continued from Spaceship

//What is our forward direction? Our spaceship moves in the positive y direction.
var forwardDirection : Vector3 = Vector3(0.0, 1.0, 0.0);

//We create two instances using our MovementSettings helper class.

```

```

//One will be for translational (position) movement, the other for rotational.
var positionalMovement : MovementSettings;
var rotationalMovement : MovementSettings;

//We create an instance using our SpecialEffects helper class.
var specialEffects : SpecialEffects;

```

Now we finally get to the heart of the script. Above we have defined everything that we need, and now we put those items to use.

```

//...continued from Spaceship

//FixedUpdate() is advantageous over Update() for working with Rigidbody physics.
function FixedUpdate() {
    // Retrieve input. Note the use of.GetAxisRaw(), which in this case helps
responsiveness of the controls.
    // GetAxisRaw() bypasses Unity's builtin control smoothing.
    thrust = Input.GetAxisRaw("Vertical");
    turn = Input.GetAxisRaw("Horizontal");

    //Use the MovementSettings class to determine which drag constant should be used
for the positional movement.
    //Remember the MovementSettings class is a helper class we defined ourselves. See
the top of this script.
    Rigidbody.drag = positionalMovement.ComputeDrag(thrust, Rigidbody.velocity);
    //Then determine which drag constant should be used for the angular movement.
    Rigidbody.angularDrag = rotationalMovement.ComputeDrag(turn,
Rigidbody.angularVelocity);

    //Determines which direction the positional and rotational motion is occurring, and
then modifies thrust/turn with //the given accelerations.
    //If you are not familiar with the ?: conditional, it is basically shorthand for an
"if...else" statement pair. See
//http://www.javascriptkit.com/jsref/conditionals.shtml
    thrust *= (thrust > 0.0) ? positionalMovement.positiveAcceleration :
positionalMovement.negativeAcceleration;
    turn *= (turn > 0.0) ? rotationalMovement.positiveAcceleration :
rotationalMovement.negativeAcceleration;

    // Add torque and force to the Rigidbody. Torque will rotate the body and force will
move it.
    // Always modify your forces by Time.deltaTime in FixedUpdate(), so if you ever
need to change your Time.fixedTime
    // setting, your setup won't break.
    Rigidbody.AddRelativeTorque(Vector3(0.0, 0.0, -1.0) * turn * Time.deltaTime,
ForceMode.VelocityChange);
    Rigidbody.AddRelativeForce(forwardDirection * thrust * Time.deltaTime,
ForceMode.VelocityChange);

```

```
}  
  
// The Reset() function is called by Unity when you first add a script, and when you  
// choose Reset on the  
// gear popup menu for the script.
```

Next, we define some default values that appear if you choose Reset in the Inspector.

```
//...continued from Spaceship  
  
// The Reset() function is called by Unity when you first add a script, and when you  
// choose Reset on the  
// gear popup menu for the script.  
function Reset() {  
    // Set some nice default values for our MovementSettings.  
    // Of course, it is always best to tweak these for your specific game.  
  
    positionalMovement.maxSpeed = 3.0;  
    positionalMovement.dragWhileCoasting = 3.0;  
    positionalMovement.dragWhileBeyondMaxSpeed = 4.0;  
    positionalMovement.dragWhileAcceleratingNormally = 0.01;  
    positionalMovement.positiveAcceleration = 50.0;  
    // By default, we don't have reverse thrusters.  
    positionalMovement.negativeAcceleration = 0.0;  
  
    rotationalMovement.maxSpeed = 2.0;  
    rotationalMovement.dragWhileCoasting = 32.0;  
    rotationalMovement.dragWhileBeyondMaxSpeed = 16.0;  
    rotationalMovement.dragWhileAcceleratingNormally = 0.1;  
    // For rotation, acceleration is usually the same in both directions.  
    // It could make for interesting unique gameplay if it were significantly  
    // different, however!  
    rotationalMovement.positiveAcceleration = 50.0;  
    rotationalMovement.negativeAcceleration = 50.0;  
}
```

Last, we tell Unity which components are needed when this script is attached. This is actually found at the very bottom of the script. It is not required to be at the bottom, but for the sake of consistency you will notice that every time `RequireComponent()` is used in our scripts it is at the bottom.

```
//...continued from Spaceship  
  
// In order for this script to work, the object its applied to must have a Rigidbody and  
// AudioSource component.
```

```
// This tells Unity to always have the components when this script is attached.  
@script RequireComponent(Rigidbody, AudioSource)
```

Spaceship Script Part 3: Special Effects

With the script above, we have a fully functional spaceship that is controlled by the player. However, to give it a little more mojo we want to add some special effects.

The remaining sections of the **Spaceship** script deal with the special effects.

Briefly, all the particle effects are taken care of within the `Update()` function. It ends up being partitioned nicely since all of the physics and controls are taken care of within the `FixedUpdate()` function.

The sound effects are executed with the `OnCollisionEnter()` function.

Read through the script to understand how it works, using the comments we included to assist you.

NOTE The spaceship currently has only one particle effect that occurs when forward thrust is occurring, but the script allows for particle effects in all directions. This is a good example of reusable code for various different purposes.

As an exercise, we suggest adding more particle effects for thrust in the other directions.

Finished!

If you have reached the end of this section you should be proud. You didn't really add much to your level in this section, but you did expand your knowledge base. You can use some of the tools and concepts you learned here and apply them to countless other projects. Even if you didn't completely grasp all of the scripts, that's okay -- just store this project away and keep these scripts in mind to use as a resource later on.

Now, prop your feet on the desk and enjoy a nice tall glass of beer lemonade -- you've earned it.

(Disclaimer: Unity Technologies and Graveck Interactive do not condone drinking on the job or any other inappropriate or illegal alcoholic consumption.)

What's Next?

Here are a few challenges we give you to make this 2D platform game more complete.



Your Turn

In this tutorial we have looked into the basics of building 2D games using 3D tools. Now it's time to take off those little stabilizing wheels from the bicycle of wisdom and ride off on our own! Here are some suggestions...

Improve the game's UI

At the moment, the only way to switch between Lerpz and the Spaceship is by clicking on a button with the mouse. This breaks the continuity of the game world by requiring the use of a completely different input device. There are two possible solutions to this:

☞ Use the keyboard instead.

Replace the GUI with a keyboard command to toggle between Lerpz and the Spaceship. (Adding and changing controls in Unity is covered in the User Guide.)

☞ Allow Lerpz to enter the Spaceship.

Have the control automatically switch to the Spaceship when Lerpz touches it. The tricky part is to make it look as if Lerpz has stepped inside -- you'll need to hide the Character (Lerpz) model while the Spaceship is being controlled.

You'll also need to add a way to exit the Spaceship -- perhaps by having Lerpz reappear when it lands, or by adding a new keyboard command.

Add some bling

What side-scrolling platformer would be complete without some sort of coin, ring, or other sort of bling.

Here's a starting point if you need it:

- 🔗 Create an object (a cylinder perhaps) that has Rigidbody and Collider components.
- 🔗 Don't forget to apply the usual 2D restrictions.

One method of completing this task would be to have the bling be a trigger. When the character hits the trigger, a message is sent from the character to the object that in turn triggers forces/torque to make it spin and fly off the screen.

If you're really ambitious you could have it trigger sound and particle effects.

Add more pizzazz to the spaceship

Currently, the space ship has one particle effect for the forward thrust. Try adding particle effects for turning the spaceship. This will be good practice working with particle effects. Don't just reuse the same flame effect -- experiment and change it up a little! Additionally, you could have the spaceship thrust backwards. This entire task requires no additional scripting!

Have the Death Zone re-spawn the Spaceship

Look through the scripts to explore how the character is re-spawned and apply the same method to the Spaceship through scripting.

Turn it into a game

Lerpz doesn't really have a goal at the moment. This tutorial was designed to educate and hasn't dipped into game design issues. Consider the tutorial as a starting point and try making a complete game with it! Add some enemies to fight, puzzles to solve and all the other trappings of a platformer.