|  |  |
| --- | --- |
| ITB logo portrait B&W | INSTITUTE OF TECHNOLOGY BLANCHARDSTOWN  A Taster of Computing  [[VERSION – Unity 2D – C# language]] |

Gravity Guy 2D (2015) - a little computer game...

Part 1



Welcome to “Gravity Guy”. In this multimedia programming exercise you will create a little 2D computer game.

CONTENTS

1 Getting started 3

2 Getting to know the 5 Unity window ‘Panels’ 6

3 Changing a ‘property’ 7

4 Add “guy” to our scene 8

5 Panning and Zooming your ‘scene’ window panel 11

6 Add some blue ‘platforms’ to our game 13

7 Run the game! 14

8 Adding a ‘red’ platform – learning to create solid ‘ground’ 16

9 Making a re-usable ‘prefab’ from a gameObject in the scene 20

10 Adding a script to make red platforms move up and down 24

11 Storing and display the player’s game ‘score’ 26

12 Create a piece of cheese tagged ‘Food’ 31

# Getting started

## UNZIP folder “gg\_part1\_assets.zip”

Unzip the contents of folder “gg\_part1\_assets.zip” to wherever you want to store your work (e.g. desktop or your network/USB drive). This should result in the following folders:

gg\_part1\_assets/

**Assets/**

**Library/**

**ProjectSettings/**

## Open the provided Unity project

The simplest way to open a Unity project is as follows:

1. Ensure the Unity application is **not already running**
   1. If it is running, then Quit the application
2. Locate and double-click a “scene” file (for simple projects there is often just one, named ‘scene1’)
   1. Using your systems file explorer, navigate to the following folder:

gg\_part1\_assets/

Assets/

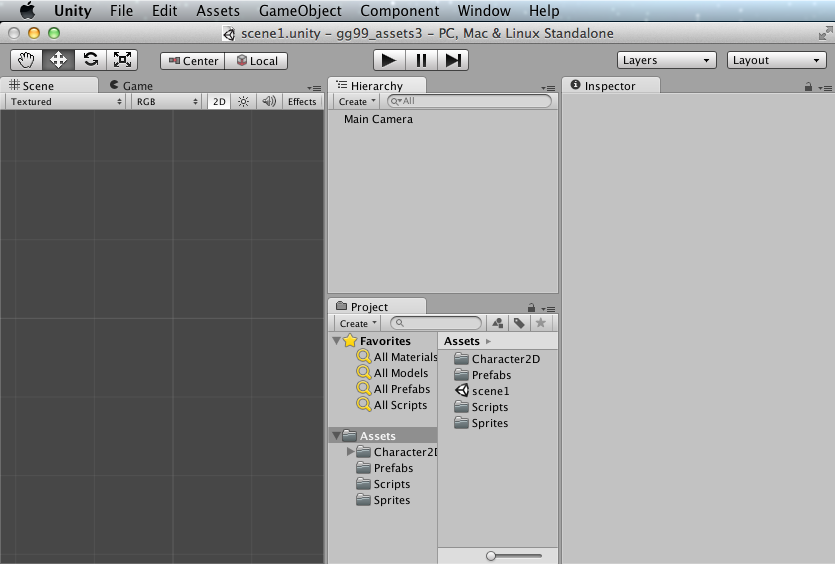
scene1.unity

(and some other files/folders – ignore those for now)

* 1. Now open the file **scene1.unity**
     + double-click file (left mouse button); or right click mouse and choose **Open**, etc.
  2. The Unity application should now start-up and load our game project
* If Unity asks you for any registration / logon details ask a member of staff what to do (or if you’re confident, then register/logon with Unity, and run the Free version to get going)

## Scene1 of the project should now be open in the Unity application

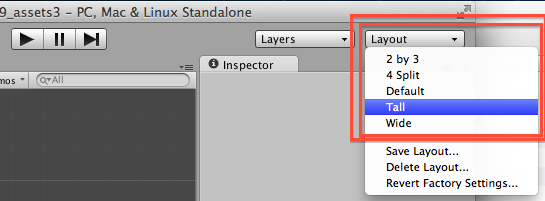
Your screen should look as follows:



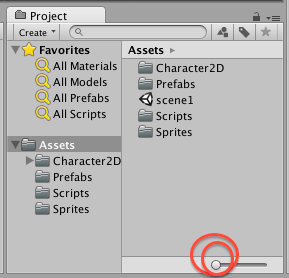
## Change screen layout to match this tutorial

If your screen looks different, do the following 2 actions to arrange the Unity Application window Panels to look the same as in the screenshots for this tutorial:

1. Reorganise the screen into the ‘Tall’ layout of window panels:



1. Drag the Project Assets icon size slider all the way to the LEFT, so all Assets are listed as **text names**, rather than large icons:



Note – this arrangement is just a personal preference of mine (.. matt ..) – you should fine out how you work best and arrange the Unity window panels however works best for you ☺

NOTE: ‘assets’

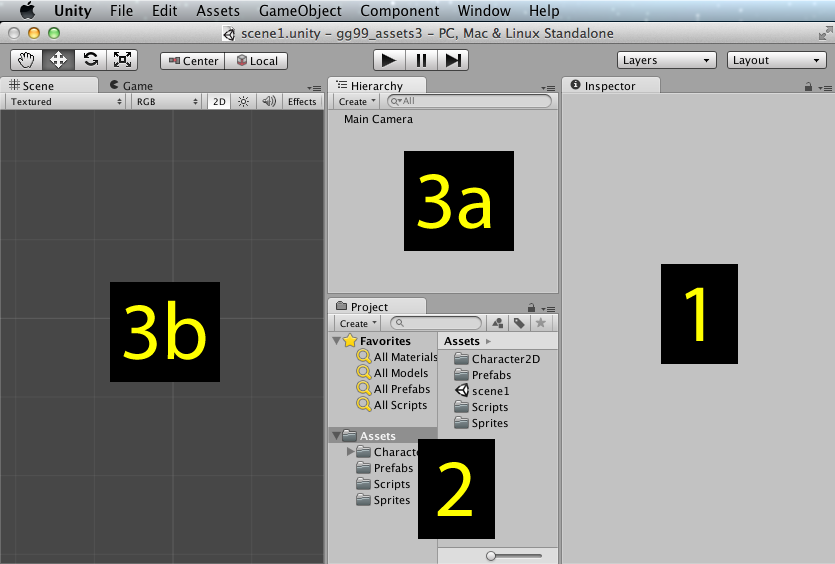
The different multimedia files (images, sounds, video clips) are know as ‘**assets’**, the content components needed to build a multimedia system.

Sometimes ‘scripts’ (C# or JavaScript/UnityScript classes) are also included the assets for a project – as with Gravity Guy 2D.

# Getting to know the 5 Unity window ‘Panels’

## Scene1 of the project should now be open in the Unity application

Your screen should look as follows:



Note the following:

INSPECTOR

* [ 1 ] The window Panel on the RIGHT of the screen (panel 1 above) is the **Inspector Panel**
  + - This shows the properties of whatever has been selected

PROJECT

* [ 2 ] The window Panel CENTER BOTTOM of the screen (panel 2 above) is the **Project Panel**
  + - This is like a file explorer window, showing the contents of the project ‘Assets’ folder

HIERARCHY

* [ 3a ] The window Panel CENTER TOP of the screen (panel 3a above) is the **Hierarchy Panel**
  + - This shows text list of the objects (called gameObjects) in the current ‘scene’

SCENE

* [ 3b ] The window Panel CENTER BOTTOM of the screen (panel 3b above) is the **Scene Panel**
  + - This shows a 3D view of some or all of the objects in the current ‘scene’

­­­ ­­

# Changing a ‘property’

## Unity project concepts – TV studio analogy cameras and scenes

When a Unity game is running, it decides what to display by deciding what each ‘**camera’** in the current ‘**scene’** can see.

For simple games (like Gravity Guy) we only have 1 camera – the **Main Camera**.

At first we will also have only 1 scene, to keep things simple (we’ll add menu / game over screens later in this tutorial).

For 2D games this camera can ‘see’ a rectangle of the ‘scene’.

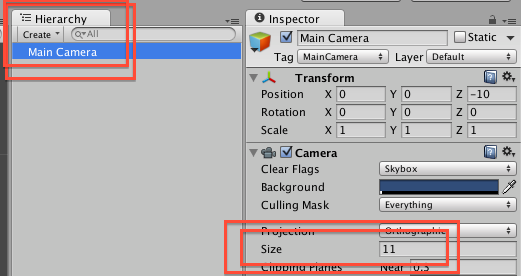
The ‘**size’** property of a camera determines how much of the ‘scene’ the camera can see (and so how much will be displayed on screen.

## Making the ‘Main Camera’ show more of the ‘scene’

The default setting of a camera is to have size ‘5’. We are going to change this to 11, so that we’ll be able to see more of the game before any camera scrolling has to happen.

Set the ‘size’ property of the Main Camera in the scene to 11:

* + In the Hierarchy Panel, select the Main Camera
  + In the **Inspector Panel**, for the **Camera** component section, change property **size** from 5 to 11



Once you have completed more of this tutorial, try changing the size of the camera, this will allow you to ‘zoom’ the players view closer / further away from the action. (of course you’ll also have to change the maximum up/down-left-right settings for when the camera scrolls – this kind of ‘tweaking’ of game parameters once you have something playable is an important part of the testing and improving of a game BEFORE it gets released into the real world – alpha and beta testing with game players helps you find settings that maximise the gaming experience for the player … but all that is for another day …

5 what ? 11 what?

Elephants? Meters? Pixels? What ‘units’ exist in a ‘scene’ in Unity?

It is best to think of each unit as a ‘meter’ when working with Unity.

Unity units can represent other measurements – but why make life complicated ☺

So think of a camera of size 5, will be 5 meters wide and 5 meters tall in a 2D Unity game.

When you create game characters, they will be around 2 units tall (like a human adult – around 2 meters tall …)

# Add “guy” to our scene

Project – the ‘resource’ files on disk you can use anywhere in your project

Scene – the resources you have added to the current scene

A core ‘workflow’ in Unity is to add a new gameobject to the current scene.

There are 2 ways we usually add a gameobject to the scene:

* Create a new empty object
* drag a ‘resource’ object from the **Project Panel** to the current scene

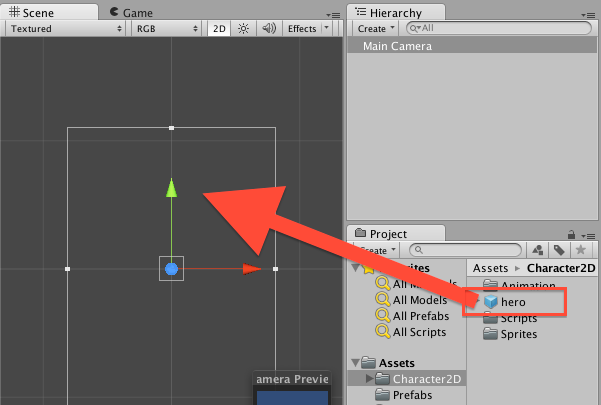
Remember, what we see in the **Project Panel** are all the files in our **‘Assets’** folder on the computer disk.

## Drag a copy of our ‘hero’ guy from Project folder into our scene

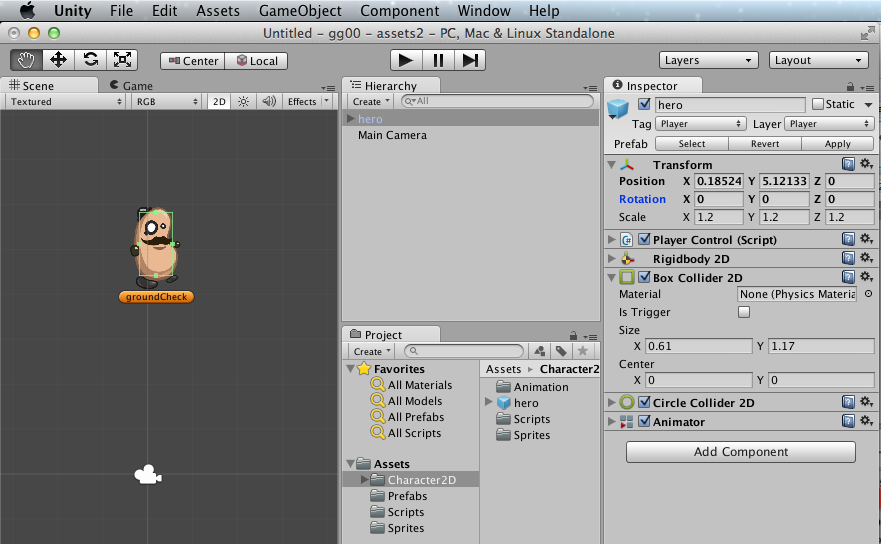
Add a copy of the pre-made ‘hero’ game object (prefab) to the current scene:

* + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Character2D**
  + In the right-hand-side of the **Project Panel**, you should now see a small blue cube labelled ‘**hero’**
  + Drag the ‘**hero’** game object prefab from the **Project Panel** into the center of the **Scene Panel**
    - The exact location doesn’t matter – we’ll set the position of this game object next …
    - Note – ‘hero’ is still in the Project Folder – we have just made a INSTANCE of the ‘hero’ in our current scene …

The screen should look similar to the following **as you drag ‘hero’ from the Project Panel**:

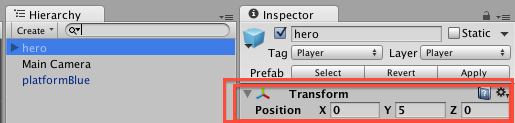


The screen should look similar to the following once the ‘hero’ gameObject has been added to the scene:



Set the ‘position’ property of our hero to exactly (0, 5, 0) (*Note: the third, Z, parameter is for 3D game effects*)

* + In the Hierarchy Panel, select hero
  + In the **Inspector Panel**, for the **Transform** component section, change property **Position** to:
    - X=0 Y=5 Z=0
    - (HINT – you can use the TAB key to move to the next text entry box once you’ve set X to zero …)



NOTE: Unity special term ‘gameObject

An object in the scene (i.e. in the Hierarchy and Scene panels) is called a ‘gameObject’. A gameObject is an instance of the class MonoBehaviour.

So when you drag a 2D image or 3D model onto the scene, you are creating a gameObject, which contains components linking to the image/model file in your Project folder.

GameObjects can send and receive many event messages (such as object collisions, and mouse clicks, and new frame updates etc.). The superclass MonoBehaviour defines all these interfaces.

# Panning and Zooming your ‘scene’ window panel

I can’t see the thing(s) I want to work on in the ‘scene’ panel?

A core skill in Unity is manipulating the view of what you can see in the ‘scene’ window panel

On this page you will learn the fundamental 3 actions you need to manage what you see:

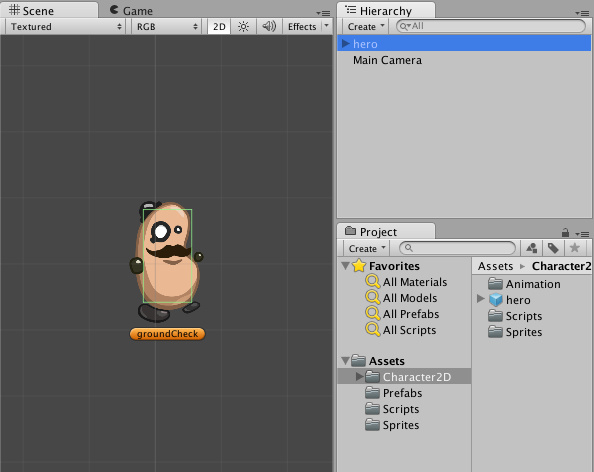
* Zoom into a specific gameObject
* Zoom in and out
* Panel up/down/left/right

## Double click a gameObject in the Hierarchy to make the Scene panel centre and zoom

Once you have lots of gameObjects in the scene, you will want to change which one you are viewing from time to time. Double clicking the name of the gameObject in the Hiearchy will make the view in the Scene window panel zoom to view the gameObject, and arrange the center of the selected object to be the center of the Scene Panel.

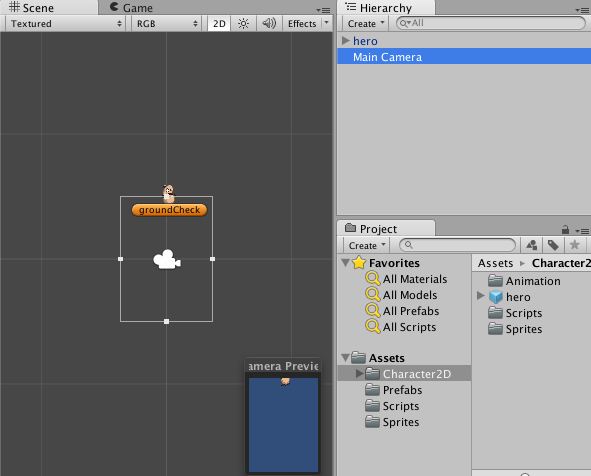
Zoom and centre on the **hero** gameObject:

* + In the **Hierarchy Panel** double click gameObject **hero**
  + The Scene panel should look as follows:



Zoom and centre on the **Main Camera** gameObject:

* + In the **Hierarchy Panel** double click gameObject **Main Camera**
  + The Scene panel should look as follows:



## ‘hand’ tool – to pan left/right/up/down

Being about to change what we see in the Scene window panel left/right/up/down is called ‘panning’. To do this we simply select the ‘hand’ tool (top left of Unity application window) and then mouse-drag with our ‘hand’ cursor in the Scene window panel.



(we’ll learn about the other tools later in this tutorial …)

## Mouse wheel / trackpad zoom – zoom in and out of Scene panel contents

To zoom in or out of what is currently being viewed in the Scene panel, use either the mouse-wheel, or your track-pad zoom method (e.g. on my laptop I use the 2-finger open-close method to zoom).

* NOTE: An alternative method to ZOOM is as follows:
  + Hold down the ALT-key
  + RIGHT-Mouse click AND drag
    - Up/right zooms in
    - Down/left zooms out (you’ll also see that while doing this the ‘hand’ tool changes to a magnifying glass !)

# Add some blue ‘platforms’ to our game

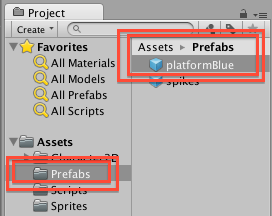
We’re almost ready to play the first version of our Gravity Guy game !

We just need to give our ‘hero’ something to land / walk / jump on …

## Drag copies of a ‘blue platform’ into our scene

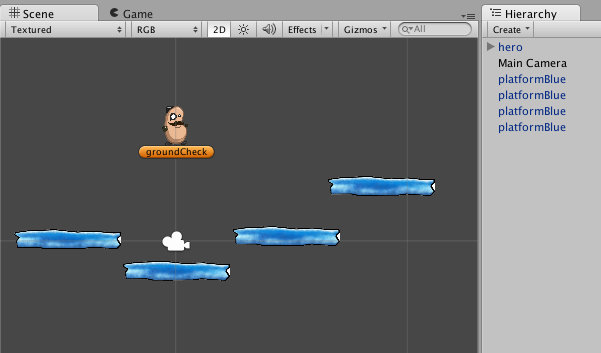
Add a copy of the pre-made blue ‘platform’ game object (prefab) to the current scene:

* + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Prefabs**
  + In the right-hand-side of the **Project Panel**, you should now see a small blue cube labelled ‘**platformBlue’;**



* + Drag the ‘**platformBlue’** game object prefab from the **Project Panel** into the **Scene Panel** somewhere BELOW the ‘hero’ (so with gravity he’ll fall down onto this platform)
  + NOTE – each time you add a gameObject to the scene, you should see a corresponding new entry appear in your **Hierarchy** window panel too …

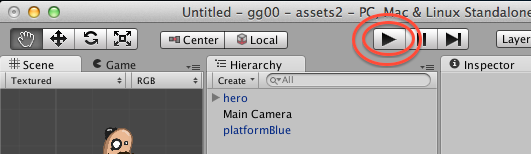
Drag a few more copies of **platformBlue** into our scene, in a stair-like arrangement, so your Scene should look something like the following:



# Run the game!

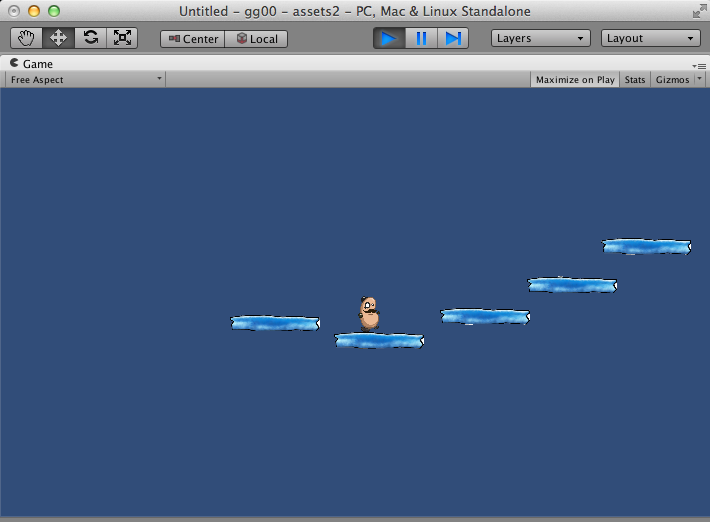
## Run the game by pressing the triangle PLAY GAME button

You may have notices the music playhead style buttons (play / pause / fast forward) buttons at the center top of the Unity application window – now it’s time to make use of them …



Click the triangle ‘PLAY’ button, and the hero character should come to life:

* He’ll fall due to gravity down the screen
* He will (hopefully) land on a blue platform
* You can move him left/right with the arrow keys – Press SPACE key to jump
* If he falls off all the platforms then just click the PLAY button a second time to STOP the game

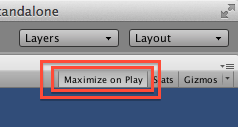


When you run the game

The **Game** window panel will come to the front (if not already visible)

Since the default color for the background of what the camera shows is BLUE,   
then most of the **Game** panel will be blue (except for the hero guy and the platforms)

If you have “Maximize on play” selected, then the **Game** panel will fill the whole Unity application window.

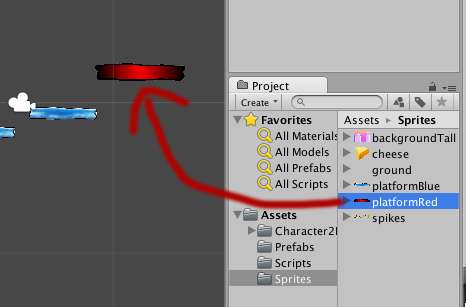


# Adding a ‘red’ platform – learning to create solid ‘ground’

## Drag the ‘platformRed’ image ‘sprite’ onto the stage

Let’s see how to turn an image ‘sprite’ into a solid platform that the player’s character can walk / jump from:

* + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Sprites**
  + In the right-hand-side of the **Project Panel**, you should now see a small blue cube labelled ‘**platformRed’:**



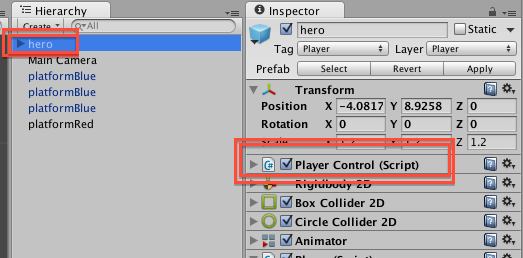
* + Drag the ‘**platformRed’** sprite from the **Project Panel** into the **Scene Panel** somewhere near the blue platform gameObjects

## Play-test your new scene gameObject

Run the game, and try to land on the red platform – the hero guy just falls past the image!

## How does the ‘controller’ script for the player’s ‘hero’ character detect ‘ground’?

There is a Csharp script attached to our ‘hero’ character named ‘PlayerControl’.



This script needs gameObjects to have 2 special characteristics to consider something as solid ‘ground’:

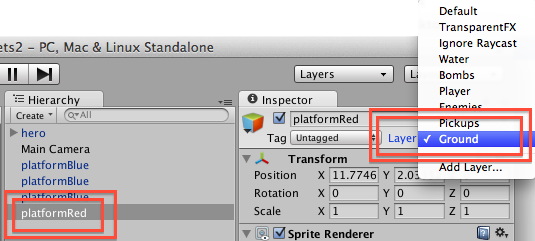
1. The gameObject must be on ‘Layer’ named ‘Ground’
2. The gameObject must have a 2D-collider physics component

Unity uses ‘colliders’ to detect collisions. A collider is a 2D or 3D boundary, which when it overlaps with another object’s collider boundary generates a collision event sent to both objects. For simple objects (like our platforms) we can use a simple rectangular box for our collider boundary, and Unity will even automatically make the rectangle the right size to enclose every coloured in pixel of our sprite image.

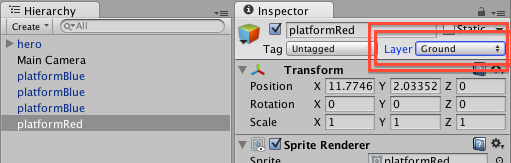
## Setting gameObject platformRed’s layer to ‘Ground’

Set the Layer of gameObject **platformRed** to ‘Ground’:

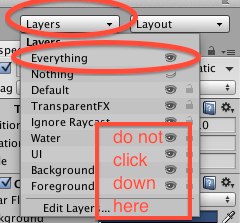
* + In the **Hierarchy** select gameObject **platformRed**
  + In the very top section of the **Inspector**, use the Layer dropdown menu (which is to the Right of the Tag drop-down menu) and choose ‘Ground’



The layer of platformRed should now be ‘Ground’:



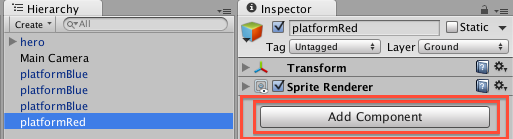
**WARNING** – do NOT hide items on Layer ground be clicking the wrong menu. You should not need to click the layer visibility menu (which is Left of the Layout menu) – if you do click this menu ensure Everything is selected with an ‘eye’ icon – so all layers are visible:



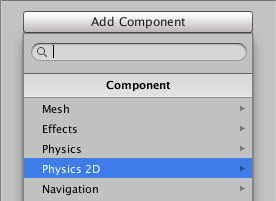
## Adding a ‘Box Collider 2D’ physics component to platformRed

Now add a collider to **platformRed**:

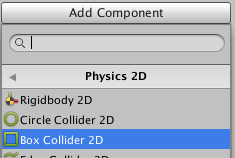
* + In the **Hierarchy** select gameObject **platformRed**
  + At the bottom of the Inspector click the ‘Add Component’ button:



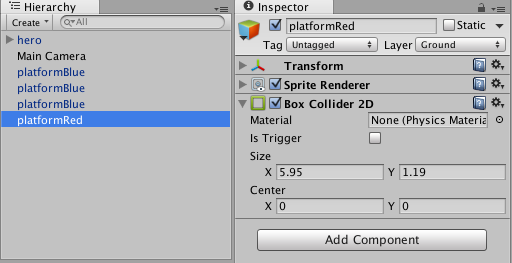
* + From the **Component** menu choose ‘Physics 2D’:



* + From the **Physics 2D** menu choose ‘Box Collider 2D’:



That’s it – we don’t need to change any settings for this new component:



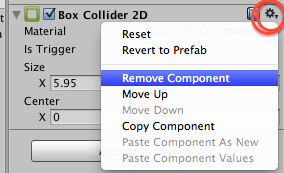
Run the game, and you should now be able to land / run / jump from the red platform just like the blue ones.

## Working in the Inspector – expand/contract/remove components

Whenever you learn how to ADD something to an object in computing, you should always learn how to REMOVE it as well – so you can get back how things were before your action.

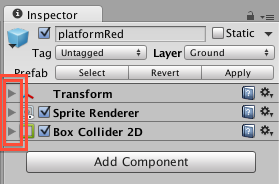
To **remove a component** in the Inspector:

* + Click the ‘cog’ icon on the right to display the component drop-down action menu
  + Then select ‘Remove Component’ from this menu

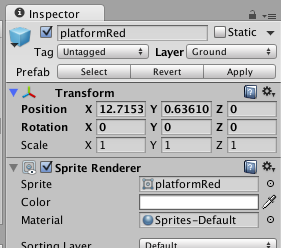


To **contract/collapse** the display of a component’s properties in the Inspector:

* + Click the tiny triangle icon to the LEFT of the component’s name
  + Try collapsing every component of a gameObject:



To **expand** component content display click the triangle again (it’s a reversing ‘toggle’ switch):



# Making a re-usable ‘prefab’ from a gameObject in the scene

Avoid repeating the same actions again and again …

A fundamental workflow concept with 2D and 3D Unity games is the ‘prefab’

Prefab stands for ‘pre-fabrication’ – literally “here’s one I made earlier…”

A Unity gameObject prefab is a COPY of all the components and properties

once you have a prefab (stored in your Project panel) copies can be added to the scene,

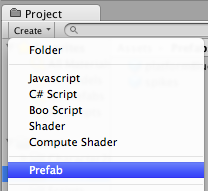
**manually** as we have been doing, or through **code**

IN A NUTSHELL – do not move on until you understand and can perform this exercise of creating a PREFAB of the platformRed gameObject in the scene

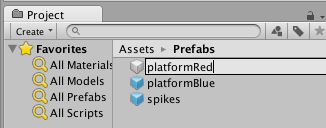
## Create a new, empty ‘prefab’ named ‘platformRed’

Create a new empty, appropriately named prefab:

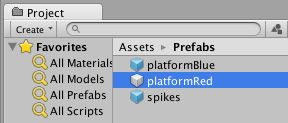
* + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Prefabs**
  + Select ‘Prefab’ from the drop-down ‘Create’ menu a the top of the **Project Panel:**



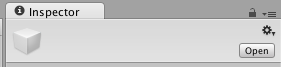
* + You should now see a new, white cube, named ‘New Prefab’



* + Rename this ‘platformRed’:



* + Note – this object has NO components in the Inspect at this point in time …

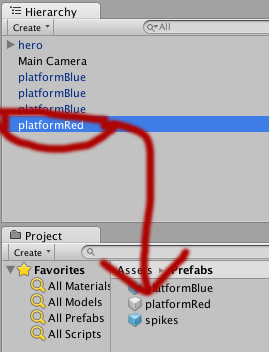


Empty prefabs are WHITE, prefabs containing a copy of a gameObjects components and properties are BLUE.

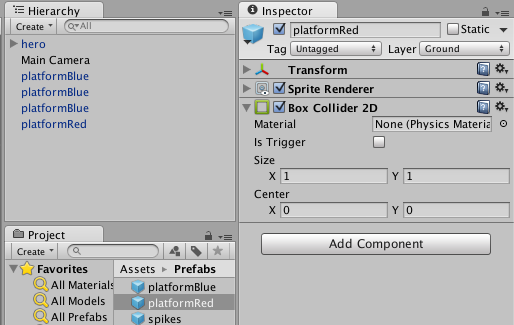
## Copy into the prefab all the components of scene gameObject ‘platformRed’

We will now ‘populate’ the contents of this empt prefab with all the details from our platformRed gameObject in our scene

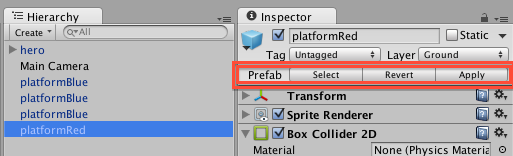
* + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Prefabs**
    - So we can see our empty prefab in the Project panel
  + Drag gameObject **platformRed** from the **Hierarchy** onto the white cube empty prefab ‘**platformRed’**



* + The prefab should now turn BLUE
    - And when selected should have a copy of all the components and properties of the gameObject from the scene



NOTE – gameObjects in the scene (in Hierarchy/Scene window panels) that are based on prefabs have their names coloured BLUE in the **Hierarchy**, and also 3 special buttons in the **Inspector**:



## Create new (ground) platformRed in the scene by dragging from new prefab

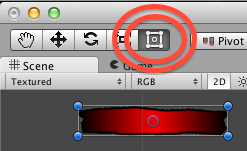
To add a new platformRed gameObejct to our scene, with layer set to ‘Ground’ and with a Box Collider 2D component, all we have to do now is make a copy of our new prefab – so we don’t need to refer to our sprite again.

Drag a couple more **platformRed** gameObjects into the scene by dragging the **platformRed prefab** into the **Scene** window panel.

When you run the game your player should be able to land / run / jump from all these red platforms too …

## New in 4.6: the “rect tool” – easy now to drag and resize 2D objects

To make it easier to move and scale 2D objects, a 5th tool has been added to the core Unity toolbar. This tool is called the ‘Rect Tool’. When in this editing mode, a selected 2D object will have 4 blue circle “handles” at each corner (for resizing – as always holding down SHIFT will scale width and height proportionally). Clicking and dragging anywhere inside the rectangle will allow you to drag/move the object around in the scene.



NOTE: The purple/blue circle (usually at the center of the rectangle for an object) is the position of the objects ‘pivot point’. Scaling and rotations are relative to the location of this point – usually we leave this at the centre unless after some special effect…

# Adding a script to make red platforms move up and down

## Remove all but 1 red platform from the Scene

Ensure you have more than one platformRed gameObject in the scene.

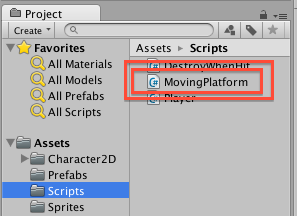
Now delete one of the platformRed gameObjects from the scene:

* + In the **Hierarchy** select a platformRed gameObject
  + Click the DELETE key to permanently delete the gameObject from the Scene
    - On a Mac use COMMAND-BACKSPACE to perform this action
    - ALTERNATIVELY **right mouse click** over the object to delete and choose ‘Delete’ from the **menu** that is offered …
  + Continue doing this until there is only a single platformRed gameObject remaining
* Position this last red platform to the right hand side of the scene

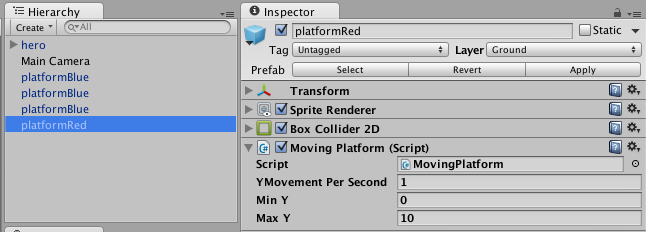
## Add a script component to a gameObject

A script has been written named ‘MovingPlatform’. Let’s add this script to our red platform in the scene:

* + In the **Hierarchy** select gameObject **platformRed**
  + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Scripts:**



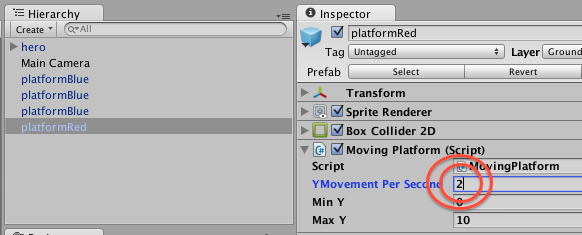
* + Drag script **MovingPlatform** from the **Project panel** onto the **platformRed** gameObject in the Hierarchy
    - There are 2 ways to add a script component to a game object – dragging the script over the gameObject’s name in the Hierarchy
    - The second method is to drag the script INTO the Inspector (ensuring the gameObject is selected in the Hierarchy already)
    - Find which way you prefer and make that your personal workflow …
  + With **platformRed** selected in the **Hierarchy**, we can now see in the **Inspector** that it has a new component named ‘Moving Platform (Script)’:



## Run the game, and see the magic in action!

Run the game, and you should see the red platform moving slowly up and down the screen.

Stop the game, and change the speed (Y Movement Per Second) to 2:



Run the game again, and the platform should move twice as fast.

If the red platform seems to move too far up or down before changing direction, try chaning the Min Y and Max Y properties of the MovingPlatform script, until it is moving just how you want it to …

# Storing and display the player’s game ‘score’

## Create a new Csharp script named ‘Player’

Let’s create a brand new script class from scratch:

* + In the **Project panel** select folder **Scripts**
  + Choose ‘Csharp’ from the **Create** menu at the top of the **Project panel**
    - A new Script file is created named ‘NewBehaviourScript’
  + Rename this new Script ‘Player’

## Startup the Monodevelop code editor application

Start the Monodevelop script editor application:

* Double-click on your new script ‘Player’
* Wait for a few seconds, and the Monodevelop editor should open up
  + Or up to a minute, if you have a slow computer ☹
* Note – with some operating systems (e.g. Windows 7 I think) Monodevelp may startup, but you will have to manually select the application to jump to the front of your computer
  + On a Mac the Monodevelop application will always jump to the front after starting up ☺

NOTE – if code-completion isn’t working for you – try this fix

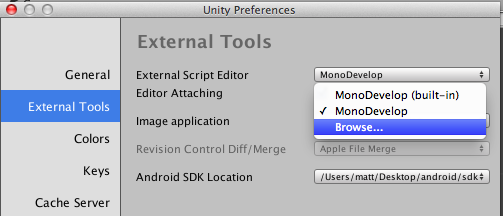
The editor should suggest methods and variable IDs as you type – this called code-completion

If this isn’t working for you (which happens for Matt with Unity 4.3 on a mac …) the following action solved the problem:

(1) open up the Preferences for Unity (should be on the File menu)

(2) use the Browse.. option to locate the MonoDevelp editor application on your hard disk (in Applications on a Mac, in Program Files in Windows)

(3) you should now see a tick by ‘Monodevelop’, rather than by ‘Monodevelop (built-in)’



I’ve no idea why this solved the problem (forced reload of code dictionaries perhaps?) – but it has worked for the author and only took 30 seconds to complete.

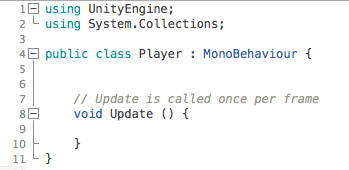
It is well worth ensuring that code-completion is working since it   
(a) speeds up coding   
(b) reduces errors in your code ☺

## Remove the empty default Start() method code in script class ‘Player’

You should see 2 ‘using’ statements to import the core packages UnityEngine and System.Collections, then the statement of a public class ‘Player’ (i.e. the same name as the Csharp script text file you have created in Unity – **the class name MUST match the file name** …). Plus there are 2 empty methods Unity has added (without us asking!) named Start() and Update():



Remove everything between lines 6 and 12 – i.e. remove the empty method named Start():



## Define a private integer property (variable) named ‘score’, set to zero

We are going to have whole number (integer) values for our player’s score, so lets declare such a variable for private use in our script class.

In between the 2 curly brackets (braces) write the following:

private int score = 0;

## Write a GUI (graphical user interface) method to display the score when the game is playing

Unity script classes have a special method named OnGUI(), that is executed (if it exist) EVERY frame – i.e. 20 or 35 or 60 or 102 times a second. The number of frames per second changes depending on how hard the computer processor is working and how fast it is.

Instruct Unity to display a score message each frame in the form: **Score = <n>**

Where <n> is the value of our integer score variable.

Add the following code to our class (after our integer score variable line):

private int score = 0;

void Update(){

string scoreMessage = "Score = " + score;

print(scoreMessage);

}

now SAVE YOUR CODE with **CTRL-S**

(**COMMAND-S** on a mac!)

The first statement in our Update() method creates a text ‘string’ of what we want to display.

The second statement tells Unity to print out the score message text to the ‘Console’ window.

NOTE – Update() is called every frame (50-100 times per second)

For debugging printing every frame via Update() is very handy (like Java System.out.println()).

However, since Update() is called EVERY FRAME, in a final published game we avoid putting logic into Update() since it has a major impact on game performance (frames-per-second).

Later in this tutorial we’ll learn a better way to display the score to the user, and only update the display each time the score changes. But for now, its easy, and also teaches you about **print()** – which is very handy when debugging/testing out a new piece of code.

## Run the game and see what happens!

When you run the game do you see a score displayed ?

* No ?
* Why – why didn’t our code execute?

Well, creating a Csharp script class in the Project panel has simply created a text file on the hard disk of our computer – Unity doesn’t know that we want this script EXCUTED in the scene we are currently creating with the hero character and blue and red platforms.

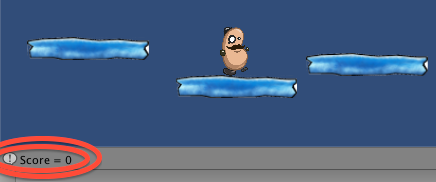
Remember when we ADDED the **MovingPlatform** script component to the red platform gameObject in our scene … Well, we have to add our **Player** script as a component of our ‘hero’ game object in our scene – so let’s do that next

## Add our Player script component to our ‘hero’ gameObject in the scene

Add script Player to the ‘hero’ gameObject in the scene Hierarchy:

* + In the **Hierarchy** select gameObject **hero**
  + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Scripts:**
  + Drag script **Player** from the **Project panel** onto the **hero** gameObject in the Hierarchy

Now run the game – and you should see a little white ‘Score = 0’ message in the Console status bar at the bottom of the screen.



RECAP: The full listing of your Player.cs class should be as follows:

using UnityEngine;

using System.Collections;

public class Player : MonoBehaviour

{

private int score = 0;

void Update()

{

string scoreMessage = "Score = " + score;

print(scoreMessage);

}

}

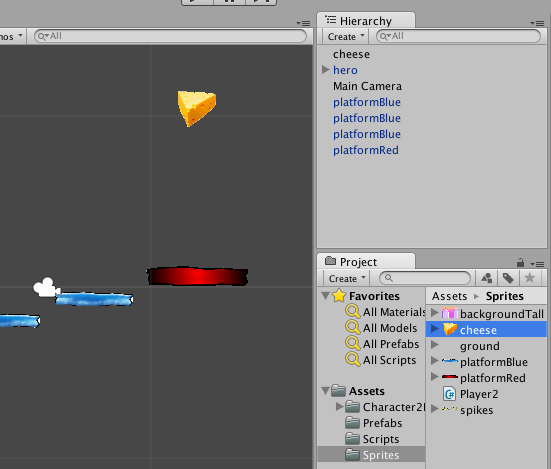
# Create a piece of cheese tagged ‘Food’

## Add a pie of cheese gameObject to the scene

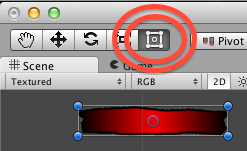
You’ll find an image of a piece of cheese in folder **Sprites** in the **Project panel**.

Let’s add this as a game object to the scene:

* + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Sprites**
  + In the right-hand-side of the **Project Panel**, you should now see a small yellow triangle-shaped image labelled ‘**cheese:**



* + Drag the **cheese** sprite from the **Project Panel** into the **Scene Panel** somewhere above the red platform
  + You may need to make the piece of cheese smaller
    - Do this with the blue circle ‘drag handles’ of the rectangle around the **cheese** gameObject in the **Scene** panel
    - Using the ‘Rect Tool’ while holding down the SHIFT key



## Adding a ‘Box Collider 2D’ physics component to our piece of cheese

We want our game to detect collisions between the hero character and this cheese gameObject, so we need to add a ‘collider’ to it.

Now add a 2D Box Collider to the **cheese** gameObject in the Hierarchy:

* + In the **Hierarchy** select gameObject **cheese**
  + At the bottom of the Inspector click the ‘Add Component’ button:
  + From the **Component** menu choose ‘Physics 2D’:
  + From the **Physics 2D** menu choose ‘Box Collider 2D’:

## Playtest your game

Play the game … the player bumps into the piece of cheese

When the hero guy character hits a piece of cheese we want:

* To add 1 to the player’s score
* For the piece of cheese to disappear (or perhaps re-appear somewhere random ☺

## Using a ‘trigger’ and a ‘tag’ to create logic for collisions with objects

Generally games have logic where DIFFERENT actions are taken depending on what KIND of object the player’s character has collided with, e.g.:

* Hit small piece of cheese, add 1 to score
* Hit large piece of cheese, add 2 to score
  + Or perhaps add 1 to score, and replace big piece of cheese with a little piece of cheese
* Hit a door
  + If carrying a key, door opens, and no longer carrying a key
  + If not carrying a key, a message / sound is played to indicate player cannot open door at this time

The two parts of this aspect of game logic are:

* The script on the player needs to know each time a collision has occurred
* When a collision occurs, the script needs to know what KIND of object (or WHICH object) the player has collided with

A collider by itself, by default, just prevents the player from moving into the same space as the object surrounded by the collider.

We need to do 3 things to the Cheese object:

* Tag the cheese gameObject with the string tag ‘**Food’**
* Enable the ‘Trigger’ on the cheese gameObject’s collider
  + So rather than defaulting to solid object physics when a collision occurs,   
    an **‘event’ message** is sent to each object involved in the collision
    - The piece of cheese (so it knows it has been hit – perhaps it will play an animation …)
    - The player’s character, so we can run logic to add to score when food item hit
    - One of the event messages broadcast is **OnTriggerEnter2D(…)** and the parameter is a reference to the collider of the object whose collider has been overlapped with

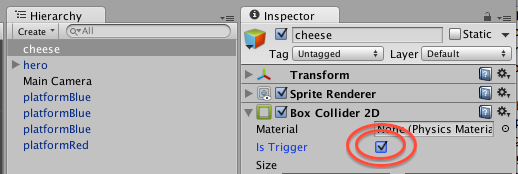
We need to add logic (script code) to the Player script class:

* add 1 to score when item hit that has the string tag equal to ‘Food’
* delete the gameObject whose collider we hit
  + in Unity the **Destroy()** method removes a gameObject from the scene at runtime …

## Enable collider ‘trigger’ to Cheese gameObject

Enable the Cheese gameObject’s trigger in its collider:

* + In the **Hierarchy** select gameObject **cheese**
  + In the details for component ‘Box Collider 2D’ tick the ‘Trigger’ checkbox



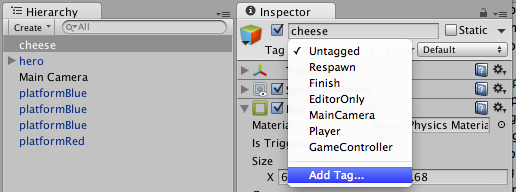
Trigger collision events will now be generated when something hits our piece of cheese…

## Create a new tag ‘Food’

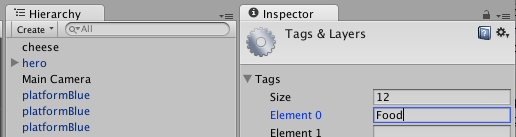
Before we can add tag ‘Food’ to the cheese object, first we have to add this word to the list of available tags in our Unity project.

Add ‘Food’ to the list of tags available:

* + In the **Hierarchy** select gameObject **cheese**
  + In the **Inspector Tag** drop-down menu select select ‘Add Tag…’



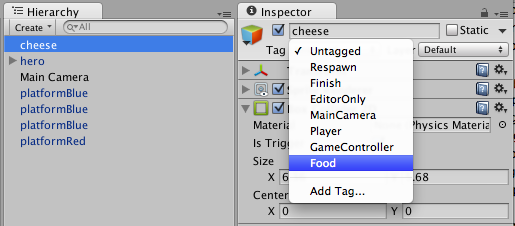
* + In the first available space, type ‘Food’
    - NOTE – the convention (which we’ll follow) is UpperCamelCase – the first letter of each word is capitalised, and there are NO SPACES in tag strings



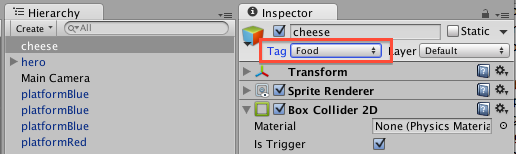
## Add tag ‘Food’ to the cheese gameObject in the Hierarchy

Enable the Cheese gameObject’s trigger in its collider:

* + In the **Hierarchy** select gameObject **cheese**
  + In the **Inspector Tag** drop-down menu select ‘Food’



You should now see that gameObject **cheese** has been tagged with string ‘Food’:



## Add new method to detect collision with food and add to score

Let’s add to the logic code for our Player script class:

* + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Scripts**
  + Double-mouse click the **Player** script, to open it up in the Monodevelop editor
    - On Windows you may now need to manually bring to the front the Monodevelop editor

Add a new method to the script classed, named OnTriggerEnter2D():

using UnityEngine;

using System.Collections;

public class Player : MonoBehaviour {

private int score = 0;

void Update(){

string scoreMessage = "Score = " + score;

print(scoreMessage);

}

void OnTriggerEnter2D(Collider2D hit)

{

if(hit.CompareTag("Food"))

{

score++;

Destroy (hit.gameObject);

}

}

}

Save these changes (CTRL-S / COMMAND-S), and run your game.

Now the score should increase to 1 when you hit a piece of cheese …



Congratulations

You have now created a complete platform game!