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

Gravity Guy 2014 - a little computer game...



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’ 5

3 Changing a ‘property’ 6

4 Add “guy” to our scene 7

5 Panning and Zooming your ‘scene’ window panel 9

6 Add some blue ‘platforms’ to our game 11

7 Run the game! 12

8 Adding a ‘red’ platform – learning to create solid ‘ground’ 13

9 Making a re-usable ‘prefab’ from a gameObject in the scene 17

10 Adding a script to make red platforms move up and down 20

11 Gravity Guy – defining the Document class 21

12 Gravity Guy – make this player sprite move! 22

13 Gravity Guy – position guy to random location 28

14 Gravity Guy – make this player sprite jump randomly each “frame” 29

15 Gravity Guy – make guy move the right each new frame 31

16 Gravity Guy – make guy move using dx/dy velocity variables 32

17 Gravity Guy – add some gravity 34

18 Gravity Guy – make guy land on the ground 36

19 Gravity Guy – keeping the program tidy with methods/functions 40

20 Gravity Guy – moving left/right with ARROW keys 44

21 Gravity Guy – replace hard-coded numbers with a variable 49

22 Gravity Guy – Jump upwards when SPACE key pressed 50

23 Gravity Guy – Jump upwards ONLY when standing on solid ground 51

24 Gravity Guy – Collision (stop falling) with platforms 55

25 Gravity Guy – Add more platforms 57

26 Gravity Guy – Create food object for guy to collect 59

27 Gravity Guy – Add a score counter for pieces of food eaten 61

28 Gravity Guy – Bonus step – add a moving platform 63

29 Gravity Guy – Bonus step – a “cheat” to reset object positoins 65

30 INFORMATION – the 3 special Unity folders 66

# Getting started

## UNZIP folder “gravity\_guy2D.zip”

Unzip the contents of folder “gravity\_guy2D\_QUICKSTART.zip” to wherever you want to store your work (e.g. desktop or your network/USB drive). This should result in the following folders:

**gravity\_guy2D/**

**unity\_project/**

**Assets/**

**Library/**

**ProjectSettings/**

**gravity\_guy2D\_assets/**

*(we’ll learn about what’s in here later ...)*

## 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 the “scene1” file
   1. Using your systems file explorer, navigate to the following folder:

**unity\_project/**

**Assets/**

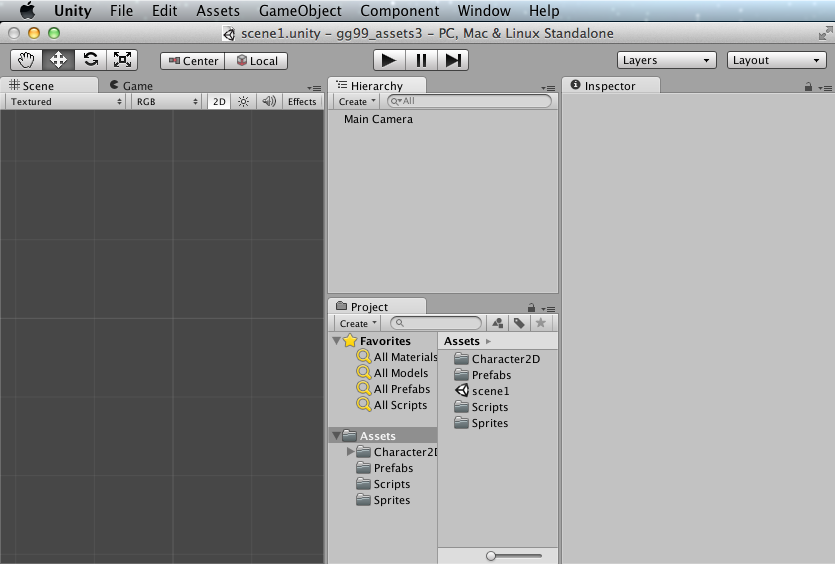
**scene1.unity**

*(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

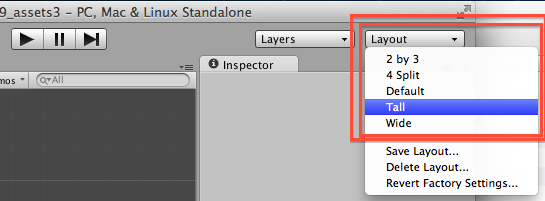
Your screen should look as follows:



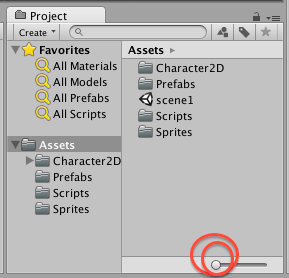
## Change screen layout to match this tutorial

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:

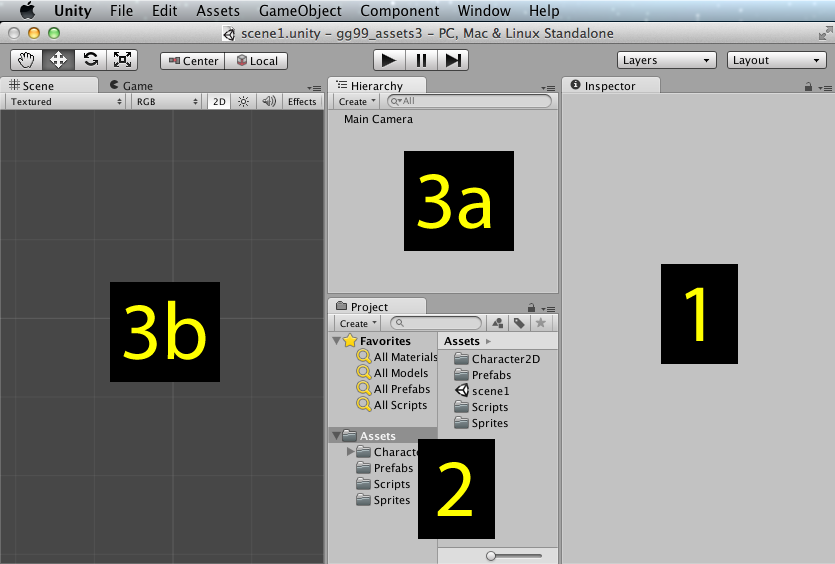


Note – this arrangement is just a personal preference of mine (.. matt ..) – you should fine out how you work best and arrange the Unity window panels however works best for you ☺

# 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’

## Locate the folder “gravity\_guy2D\_assets”

Unzip the contents of folder “gravity\_guy2D\_assets” to the desktop or your network/USB drive.

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’ are also included the assets for a project – as with Gravity Guy 2D.

­­­ ­­

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’.

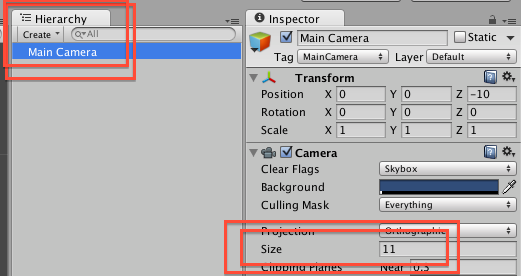
The ‘**size’** property of a camera determines how much of the ‘scene’ the camera can see (and so how much will be displayed on screen.

## 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 ‘resources’ 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
* copy of a ‘resource’ object from the **Project Panel** to the current scene

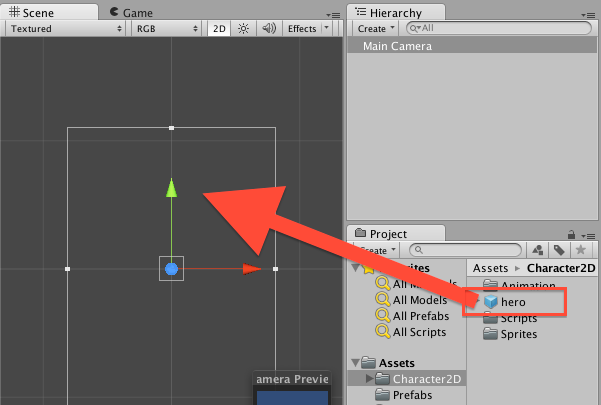
Remember, what we see in the **Project Panel** are all the files in our **‘Assets’** folder on the computer disk.

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

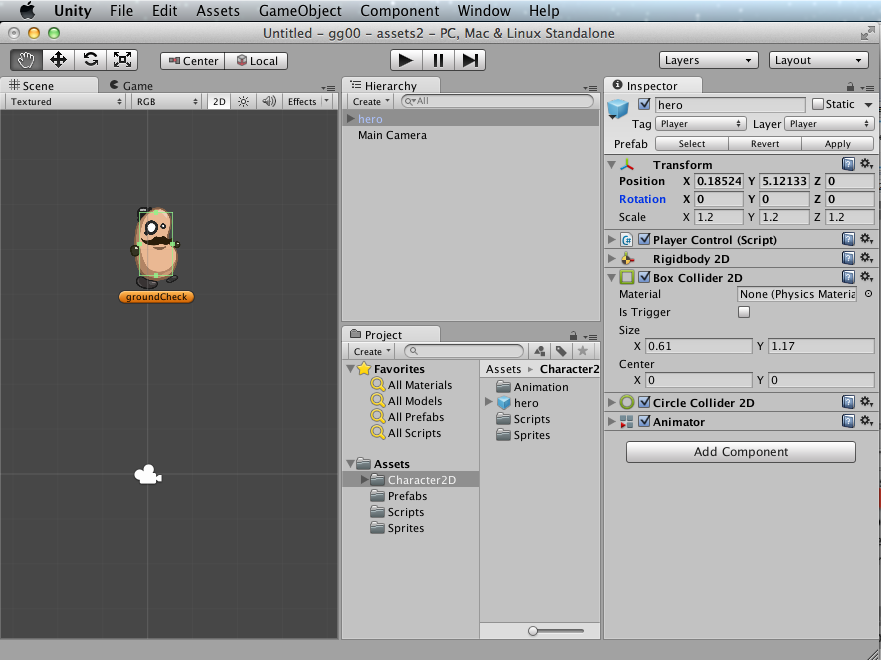
Add a copy of the pre-made ‘hero’ game object (prefab) to the current scene:

* + 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 COPY of the ‘hero’ in our current scene …

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

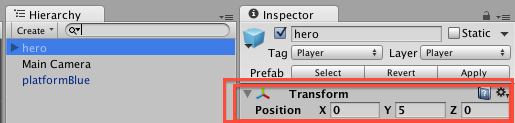


The screen should look similar to the following **once a copy of the ‘hero’ gameObject has been added to the scene:**



Set the ‘position’ property of our hero to exactly (0, 5, 0)

* + 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 …)



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:

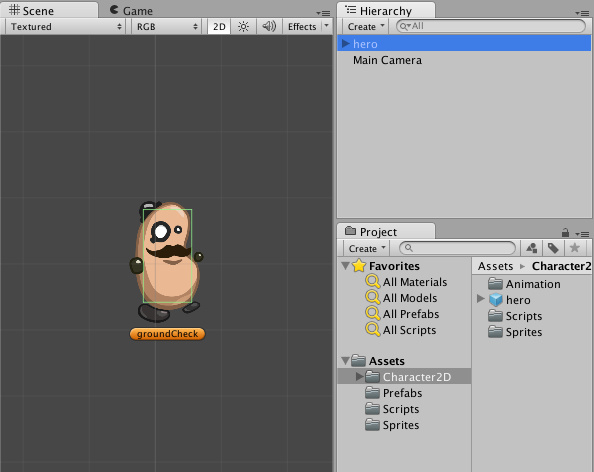
* Zoom into a specific gameObject
* Zoom in and out
* Panel up/down/left/right

## 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.

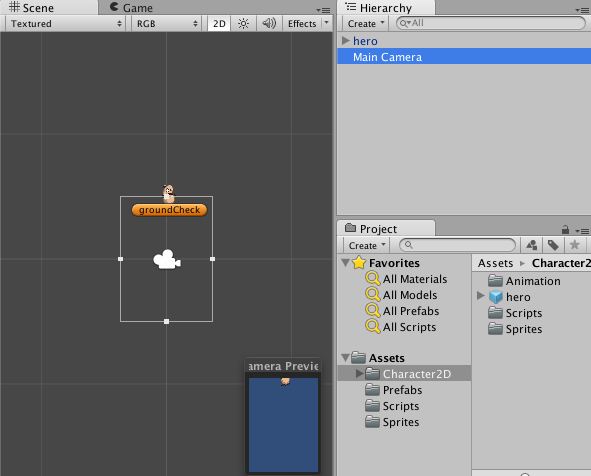
Zoom and centre on the **hero** gameObject:

* + 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 CTRL-key
  + 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

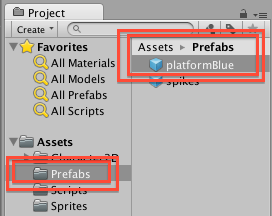
**We’re almost ready to play the first version of our Gravity Guy game !**

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

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

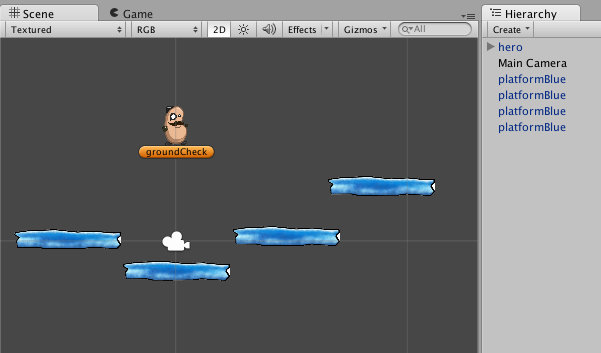
Add a copy of the pre-made blue ‘platform’ game object (prefab) to the current scene:

* + 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 …

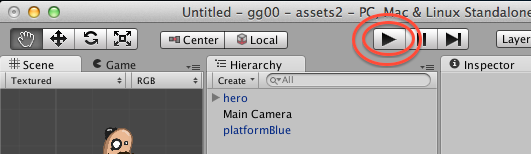
Drag a few more copies of **platformBlue** into our scene, in a stair-like arrangement, so your Scene should look something like the following:



Run the game!

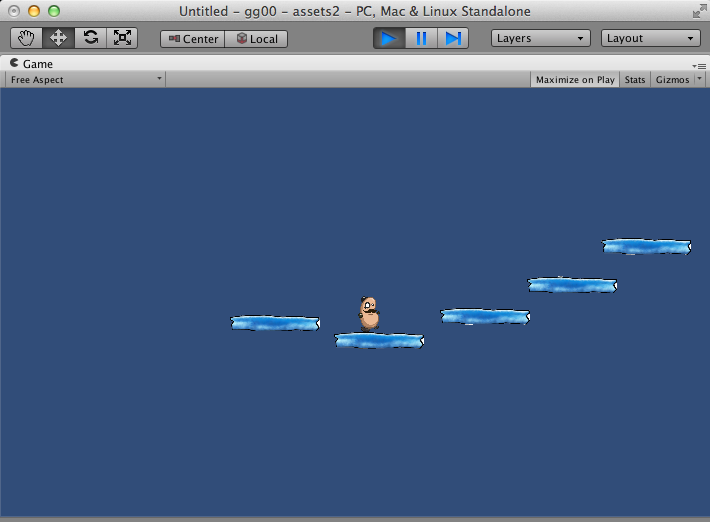
## Run the game by pressing the triangle PLAY GAME button

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:

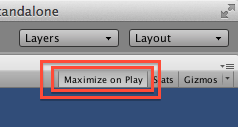
* He’ll fall due to gravity down the screen
* He will (hopefully) land on a blue platform
* You can move him left/right with the arrow keys – Press SPACE key to jump
* If he falls off all the platforms then just click the PLAY button a second time to STOP the game



**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)



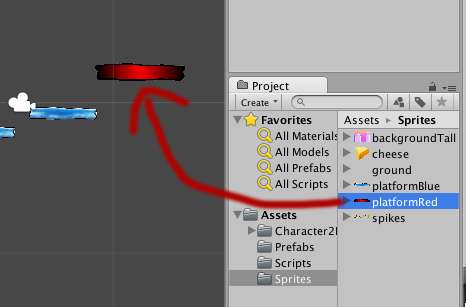
If you have “Maximize on play” selected,  
then the **Game** panel will fill the whole Unity application window.

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’:**



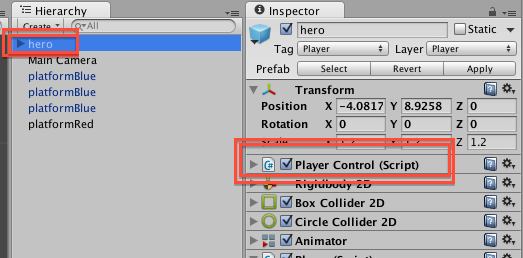
* + Drag the ‘**platformRed’** game object prefab from the **Project Panel** into the **Scene Panel** somewhere near the blue platform gameObjects

## 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’:

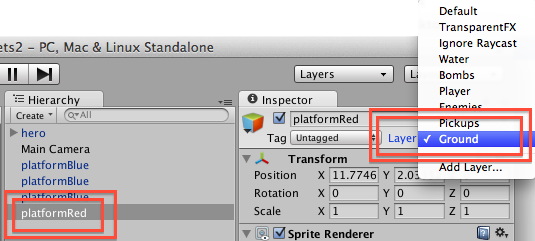
1. The gameObject must be on ‘Layer’ named ‘Ground’
2. The gameObject must have a 2D-collider physics component

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.

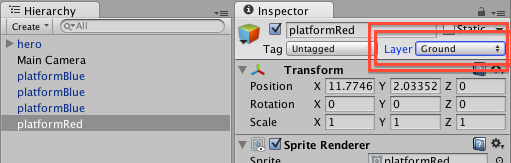
## Setting gameObject platformRed’s layer to ‘Ground’

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 and choose ‘Ground’



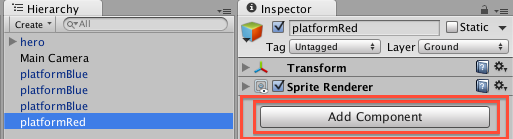
The layer of platformRed should now be ‘Ground’:



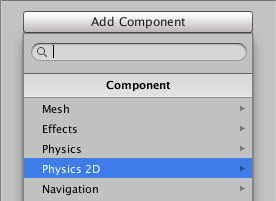
## Adding a ‘Box Collider 2D’ physics component to platformRed

Now add a collider to **platformRed**:

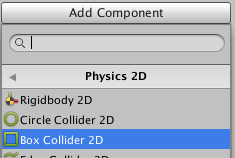
* + In the **Hierarchy** select gameObject **platformRed**
  + At the bottom of the Inspector click the ‘Add Component’ button:



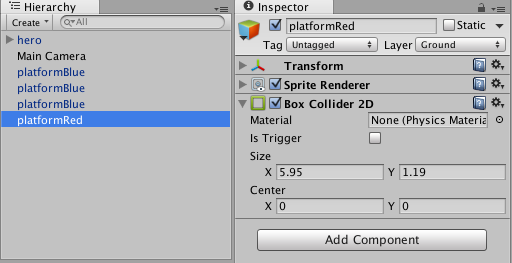
* + From the **Component** menu choose ‘Physics 2D’:



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



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



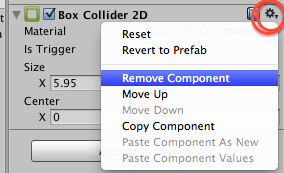
Run the game, and you should now be able to land / run / jump from the red platform just like the blue ones.

## 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.

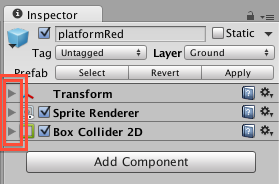
To **remove a component** in the Inspector:

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

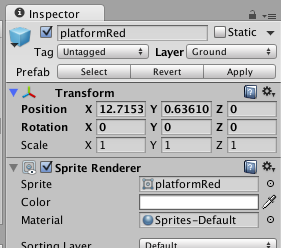


To **contract/collapse** the display of a component’s properties in the Inspector:

* + 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,

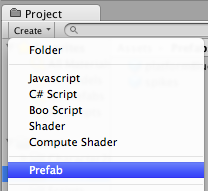
**manually** as we have been doing, or through **code**

**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**

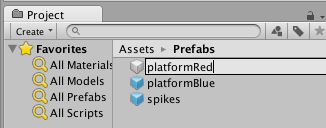
## Create a new, empty ‘prefab’ named ‘platformRed’

Create a new empty, appropriately named prefab:

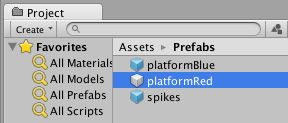
* + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Prefabs**
  + Select ‘Prefab’ from the drop-down ‘Create’ menu a the top of the **Project Panel:**

****

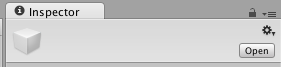
* + You should now see a new, white cube, named ‘New Prefab’



* + Rename this ‘platformRed’:



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

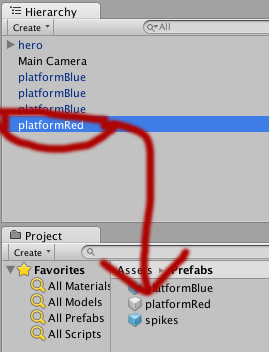


Empty prefabs are WHITE, prefabs containing a copy of a gameObjects components and properties are BLUE.

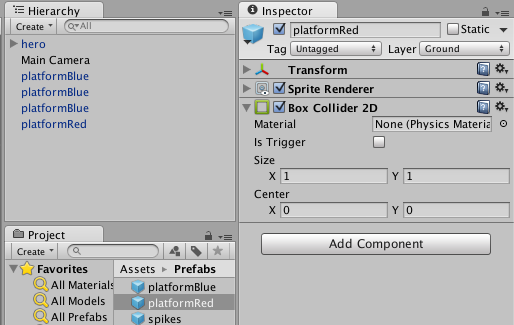
## Copy into the prefab all the components of scene gameObject ‘platformRed’

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

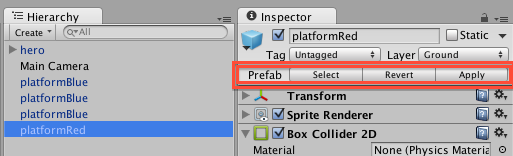
* + In the **Project Panel (**left side, lower section ‘Assets’), select folder **Prefabs**
    - So we can see our empty prefab in the Project panel
  + Drag gameObject **platformRed** from the **Hierarchy** onto the white cube empty prefab ‘**platformRed’**



* + 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 …

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.

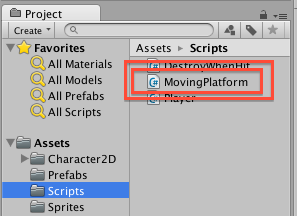
Now delete one of the platformRed gameObjects from the scene:

* + In the **Hierarchy** select a platformRed gameObject
  + Click CTRL-BACKSPACE 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

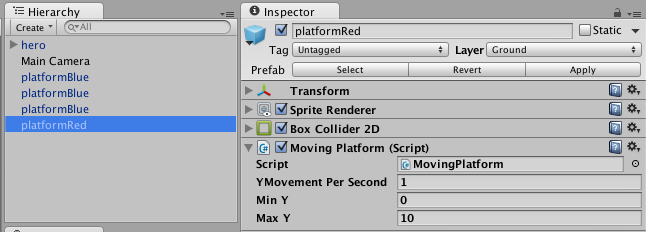
## Add a script component to a gameObject

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:**

****

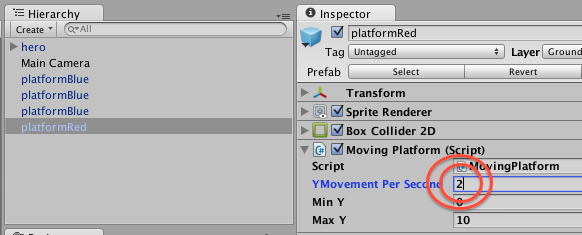
* + Drag script **MovingPlatform** from the **Project panel** onto the **platformRed** gameObject in the Hierarchy
    - There are 2 ways to add a script component to a game object – dragging the script over the gameObject’s name in the Hierarchy
    - The second method is to drag the script INTO the 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**

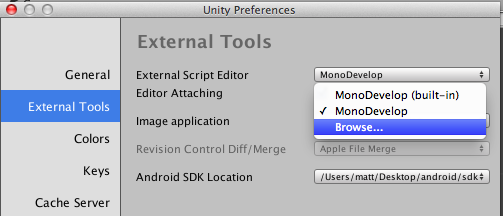
The editor should suggest methods and variable IDs as you type – this called code-completion

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.

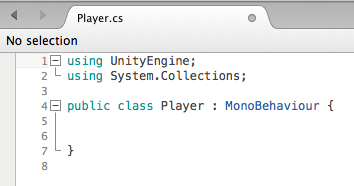
It is well worth ensuring that code-completion is working since it   
(a) speeds up coding   
(b) reduces errors in your code ☺

## Remove the default body 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 5 and 14 (i.e. remove the 2 empty methods named Start() and Update():



## 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;*

private void OnGUI(){

string scoreMessage = "Score = " + score;

GUILayout.Label(scoreMessage);

}

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

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

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

The second statement uses the default layout manager named GUILayout (i.e. start displaying things at the top left of the screen), and tells it to display a text ‘Label’ containing our score message.

Labels are non-interactive (they can’t be clicked like buttons – they just display text or images on screen.

## 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.

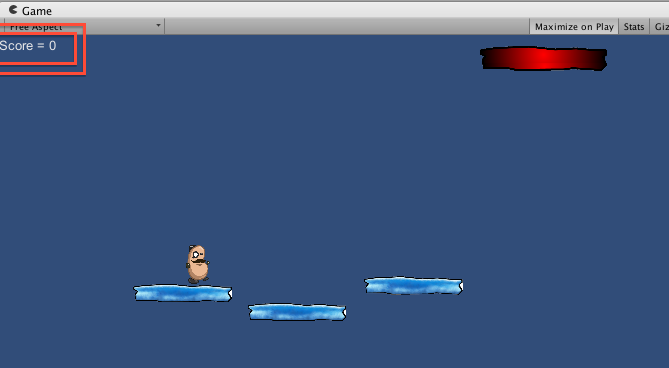
Remember when we ADDED the MovingPlatform script component to the red platform gameObject in our scene … Well, we have to add our Player script as a component of our ‘hero’ game object in our scene – so let’s do that next

## 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 top right of the screen!

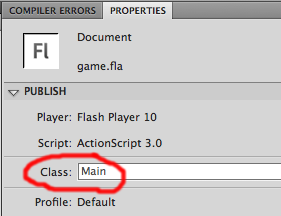


Gravity Guy – defining the Document class

## View the Properties of your Flash document

Ensure nothing on the stage is selected

1. Click in the gray area off the stage
2. Now look at the Properties panel
   * + If necessary you may need to drag the panel divider to make the Properties panel large enough to see the Document “Class” value



You should see that the Document “Class” is defined as “Main”. This means that when the Flash movie is run / loaded into a webpage, it will begin to execute the program statements found in the Main.as class file – this is where your program will be written (along with any additional class files needed).

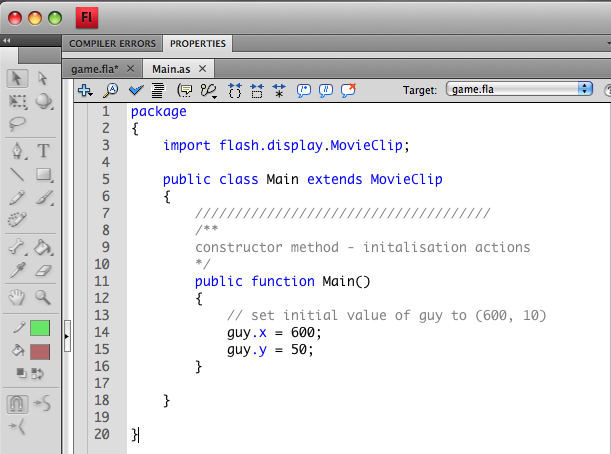
Gravity Guy – make this player sprite move!

## View the “Main.as” file in the stage/class panel

Open file “Main.as”

1. Double click the file in the Windows File Explorer, or open the file from inside Flash from the File menu:   
   menu **File | Open**
2. HINT:
   * + If you minimise the Properties panel when working with “.as” files, you will have more screen “real esate” to work with – i.e. you’ll be able to see more lines of code

You should now be able to see the contents of file “Main.as”



As you can see there is just a little code so far. The code provided does the following:

* A new class “Main” is declared – a subclass of “MovieClip”
* A constructor method “Main” is defined
  + So far this does one thing – it sets the position of the “guy” movie clip to (300, 50)

Test your game:

* 1. there are 2 ways to test your game:
     + Menu: **Control | Test Movie**
     + Keyboard shortcut (this is recommended – it’s easiest!):

Mac: **<COMMAND><ENTER>**

PC**: <CTRL><ENTER>**

* 1. You should see the SWF window appear with the background, and the stationary guy graphical object at the top right of the window



## Understanding the code in the Main.as class file

Some program code has been provided. Before adding to this, it is important to understand every line you are starting from. Notice how many parts of the program are “sandwiched” between pairs of braces (curly brackets)…

package

{

import flash.display.MovieClip;

public class Main extends MovieClip

{

/////////////////////////////////////

/\*\*

constructor method - initalisation actions

\*/

public function Main()

{

// set initial value of guy to (600, 10)

guy.x = 600;

guy.y = 50;

}

}

}

* First a “package” is declared
  + A package is a “home” classes, allowing collections of classes to be named for re-use by other programs/programmers at a later date
  + Don’t worry too much about this, but understand that EVERY class needs to be inside a package

package

{

...

}

* Next is an “import” statement
  + Our Flash “Document class” (which we have named Game) is an example of a “MovieClip”
  + All the features and properties of MovieClips are defined in another file (written by the Adobe team), the are stored in the package “fash.display” and in the class MovieClip
  + This important statement tells Flash that we wish to refer to / make use of the MovieClip class from this package flash.display
  + Whenever we wish to make use of classes (bit of program) provided in the Adobe libraries, we need to “import” them to make them available
  + So you will see this list of “import” statements get longer as we make more and more use of pre-written classes from the Adobe libraries

import flash.display.MovieClip;

* Our class Game is now declared
  + All of our game management code will be written inside this “class”

public class Main extends MovieClip

{

...

}

* Next is a single line comment
  + Anything following // is ignored by Flash
  + Comments allow us to put in notes for Humans to understand the program
    - They also allow us to easily “disable” lines when we’re trying things out
    - This is called “commenting out” a line of code to stop it working
  + A long line of ///// is an easy way to indicate a new method function – helping break up the list of a program visually help us focus on just the lines of code we are working on at any one time

/////////////////////////////////////

* Next is a multi-line comment
  + Everything between /\* and \*/ is ignored by Flash, even on different lines
  + Multi-line comments are useful for more detailed comments, such as describing several steps that will be performed in a method
  + This example starts with /\*\* (2 asterisks) – this is a special comment that can be recognised by a documentation generator – this is called Javadoc-style commenting
    - It is good practice to write a Javadoc-style comment just before each method you write
    - This will make it very easy to generate documentation web pages to help other programmers re-use your programs in the future

/\*\*

constructor method - initalisation actions

\*/

* Next the method Main() is declared
  + A package is a “home” classes, allowing collections of classes to be named for re-use by other programs/programmers at a later date
  + Don’t worry too much about this, but understand that EVERY class needs to be inside a package

public function Main()

{

...

}

* Finally we get to look at the 2 lines of the program that actually do something !
  + The 2 lines inside the method function Main set the X-position of guy to 600, and the X-position of guy to 50
  + This should move “guy” somewhere on the right (x=600) and the top (y=50) of the screen

public function Main()

{

// set initial value of guy to (600, 10)

guy.x = 600;

guy.y = 50;

}



NOTE:

Flash documents are defined to take up a rectangle of a certain size. This size can be changed in the Document properties. For gravity guy the size of the rectangle for the Flash movie has been set to 800 by 400 (800 wide by 400 high). This was set to be the same size as the background picture – so if you change it you may have white areas that the background does not cover …

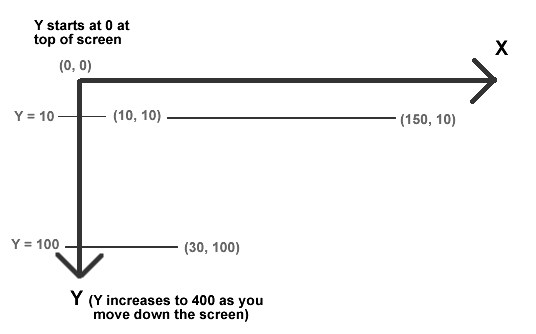
## Why is a small Y value at the top of the screen?

On computers the top-left of the screen is (0, 0), X increases to the right, and Y increases **downwards**. The reason for this is that the original computer graphics programs controlled the Cathode Ray Tube (CRT) directly, and the rays were scanned starting at the **top** of the TV screen and working downwards. So Y=0 at the top, and Y=800 (or whatever – depending on the screen resolution) at the bottom of the screen.

So remember the following:

* The close Y is to zero, the higher up the screen
* The larger the Y value, the further down the screen
* Adding a small positive value will make something go UP
* Adding a small negative number will make something go DOWN

This will all become relevant once we start making objects move around the screen with the arrow keys … so pressing the UP arrow key means we should add a small **negative** number to the Y position ….



Gravity Guy – position guy to random location

## Change the 2 statements inside the Main() method

Replace the 2 statements inside Main so you code now looks as follows:

package

{

import flash.display.MovieClip;

public class Main extends MovieClip

{

/////////////////////////////////////

/\*\*

constructor method - initalisation actions

\*/

public function Main()

{

// position guy in random location

guy.x = Math.random() \* 800;

guy.y = Math.random() \* 400;

}

}

Now each time you run the Flash movie guy is positioned in a different location.

Gravity Guy – make this player sprite jump randomly each “frame”

## Change method Main() to register calls to new method “gameLoop”

Flash is a frame-based multimedia system. The frame rate can be set to values from 1 to 120. Frame rates greater than 12 are usually required to give the impression of smooth movement. Every new frame an “event” is generated. This event is called “ENTER\_FRAME”. We can make our program “register” a method to listen for the event, and actions to be executed when the event occurs.

Most computer games are driven from a “game loop”, and so this is a sensible name for the method that will be repeatedly called each time there is an ENTER\_FRAME event message.

We need to make 3 changes to our Main class:

1. We need to import all classes defined in the flash.events package
2. We need to change method Main() to register gameLoop for ENTER\_FRAME messages
3. We need to write the code for the new method gameLoop()

package

{

import flash.display.MovieClip;

// step 1:

// import all classes defined in the flash.events package

//

import flash.events.\*;

public class Main extends MovieClip

{

/////////////////////////////////////

/\*\*

constructor method - initalisation actions

\*/

public function Main()

{

// step 2:

// We need to change method Main() to register

// method gameLoop for ENTER\_FRAME messages

//

stage.addEventListener( Event.ENTER\_FRAME, gameLoop );

}

// step 3:

// write the code for the new method gameLoop()

//

/////////////////////////////////////

/\*\*

gameloop - actions to be executed each "tick" of the game clock

\*/

public function gameLoop(event: Event):void

{

// position guy in random location

guy.x = Math.random() \* 800;

guy.y = Math.random() \* 400;

}

}

}

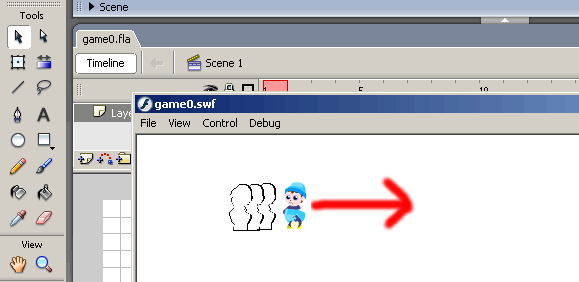
Guy should now be jumping around the screen according to the frame rate (initally 12 f.p.s. – try changing this).

**TRY THIS ….** Change the value from 800 to 400, and guy’s will only be positioned in the left hand side of the screen.

Gravity Guy – make guy move the right each new frame

## Add 5 to the x-position each frame

Rather than moving randomly, to make guy float across the screen to the right, we can add 5 to guy’s x-position each frame.



To achieve this all we need to do is replace the statements inside method gameLoop() to:

guy.x += 5;

This statement means *“add 5 to the value of guy.x and place the new total inside guy.x”.* So the full code for method gameLoop() now looks as follows:

public function gameLoop(event: Event):void

{

// add 5 to guy's position each "tick"

guy.x += 5;

}

**TRY THIS ….** We can make the guy move faster or slower by changing the value added to **guy.x**. A smaller value of 1 would make the sprite move slower, and a larger value like 20 would move it faster.

Gravity Guy – make guy move using dx/dy velocity variables

## Remove hard-coded value of 5 – introduce velocity variables

It is poor programming practice to have numbers like 5 in statements – we don’t’ know what they stand for, and to change them means going through the code line by line looks for each number to be changed.

It is much better practice to place values inside “variables” with meaningful names. Then the code makes more sense to human beings, and to change something we can just change the value inside the variable and the change will be implemented wherever the variable is referred to.

Movement in 2-dimensions are referred to as “velocity” movements. So the 2 variables we introduce (one for change in X-position, and one for change in Y-position) are referred to as velocity variables. Following the convention from mathematical calculus, we shall name these variables “dx” and “dy” – the “d” refers to “Delta” meaning a small change in mathematics. So “dx” means the small change in X-position, and “dy” means the small change in Y-position, each frame.

To make guy move to the left, we shall set “dx” to -5 (moving 5 pixels to the LEFT each frame), and “dy” to zero – so guy will not change its vertical position when moving.

/\*\* small change in X position next "tick" \*/

var dx:Number = 5;

/\*\* small change in Y position next "tick" \*/

var dy:Number = 0;

GameLoop() needs to be changed to add “dx” to guy’s X- and “dy” to guy’s Y-position.

public function gameLoop(event: Event):void

{

// add (dx,dy) to guys position

guy.x += dx;

guy.y += dy;

}

The modified full program listing incorporating these changes is shown below.

package

{

import flash.display.MovieClip;

import flash.events.\*;

public class Main extends MovieClip

{

/////////////////

/// variables

/////////////////

/\*\* small change in X position next "tick" \*/

var dx:Number = 5;

/\*\* small change in Y position next "tick" \*/

var dy:Number = 0;

/////////////////////////////////////

/\*\*

constructor method - initalisation actions

\*/

public function Main()

{

stage.addEventListener( Event.ENTER\_FRAME, gameLoop );

}

/////////////////////////////////////

/\*\*

gameloop - actions to be executed each "tick" of the game clock

\*/

public function gameLoop(event: Event):void

{

// add (dx,dy) to guys position

guy.x += dx;

guy.y += dy;

}

}

}

**TRY THIS ….** We can make the guy move faster or slower by changing the “dx” value. A smaller value of 1 would make the sprite move slower, and a larger value like 20 would move it faster.

**TRY THIS ….** We can make the guy move faster or slower by changing the frame rate for the document.

**TRY THIS ….** By seting the “dy” variable to -5 or 7 or whatever, you can make guy move upwards/downwards as well.

Gravity Guy – add some gravity

## Introduce a “gravity” variable to the program

Gravity in computer games means making a sprite move down the screen faster and faster. To make things move down the screen we need to add positive numbers to their Y-position. To make them move faster and faster, we need to increase the amount added to their Y-position more and more each frame.

This “simulated gravity” is easily programmed by simply adding a small positive value to “dy” each frame – thus each frame the amount guy moves downwards is increased. E.g.

Frame 1 (guy.x, guy.y) = (50, 50) (dx,dy) = (0,0) gravity = 1.5

* Add dx to guy.x, add dy to guy.x, add gravity to dy

Frame 2 (guy.x, guy.y) = (50, 50) (dx,dy) = (0,1.5) gravity = 1.5

* Add dx to guy.x, add dy to guy.x, add gravity to dy

Frame 3 (guy.x, guy.y) = (50, 51.5) (dx,dy) = (0,3.0) gravity = 1.5

* Add dx to guy.x, add dy to guy.x, add gravity to dy

Frame 4 (guy.x, guy.y) = (50, 54.5) (dx,dy) = (0,4.5) gravity = 1.5

* Add dx to guy.x, add dy to guy.x, add gravity to dy

Frame 5 (guy.x, guy.y) = (50, 59) (dx,dy) = (0,6.0) gravity = 1.5

* Add dx to guy.x, add dy to guy.x, add gravity to dy

Frame 6 (guy.x, guy.y) = (50, 65) (dx,dy) = (0,7.5) gravity = 1.5

* Add dx to guy.x, add dy to guy.x, add gravity to dy

Frame 7 (guy.x, guy.y) = (50, 72.5) (dx,dy) = (0,9) gravity = 1.5

* Add dx to guy.x, add dy to guy.x, add gravity to dy

Frame 8 (guy.x, guy.y) = (50, 81.5) (dx,dy) = (0,10.5) gravity = 1.5

* Add dx to guy.x, add dy to guy.x, add gravity to dy

As we can see, after 3 frames guy is moving down the screen at a rate of dy=3.0 pixels per frame. This increases every frame because gravity is being added to dy, so by frame 8 guy is moving down the screen at a rate of 10.5 pixels per frame.

and this is increasing.

To code this simulated gravity into our game we need to do 2 things:

1. Define a variable “gravity” with value 1.5   
   (note – by setting the initial dy to -20, guy will first fly upwards, then start to go down as gravity kicks in)

/\*\* small change in X position next "tick" \*/

var dx:Number = 5;

/\*\* small change in Y position next "tick" \*/

var dy:Number = -20;

/\*\* simulated gravity

makes sprite accelerate down screen by this amount \*/

var gravity:Number = 1.5;

1. Add gravity to dy after it has been moved each “tick”

public function gameLoop(event: Event):void

{

// add (dx,dy) to guys position

guy.x += dx;

guy.y += dy;

// add gravity acceleration to "dy"

dy += gravity;

}

The modified full program listing incorporating these changes is shown below.

class Game extends MovieClip

{

package

{

import flash.display.MovieClip;

import flash.events.\*;

public class Main extends MovieClip

{

/////////////////

/// variables

/////////////////

/\*\* small change in X position next "tick" \*/

var dx:Number = 5;

/\*\* small change in Y position next "tick" \*/

var dy:Number = -20;

/\*\* simulated gravity

makes sprite accelerate down screen by this amount \*/

var gravity:Number = 1.5;

/////////////////////////////////////

/\*\*

constructor method - initalisation actions

\*/

public function Main()

{

stage.addEventListener( Event.ENTER\_FRAME, gameLoop );

}

/////////////////////////////////////

/\*\*

gameloop - actions to be executed each "tick" of the game clock

\*/

public function gameLoop(event: Event):void

{

// add (dx,dy) to guys position

guy.x += dx;

guy.y += dy;

// add gravity acceleration to "dy"

dy += gravity;

}

}

}

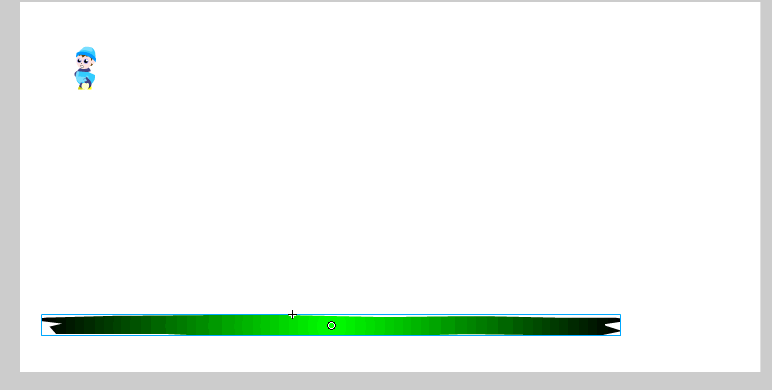
**TRY THIS ….** We can simulate extra strong gravity by setting it to 2 or 3 etc. (i.e. number bigger than 1.5).

# Gravity Guy – make guy land on the ground

## Create (drag) a “GroundTemplate” graphic on your screen

Find the “GroundTemplate” long green shape in the Library and drag from this in the Library onto the stage

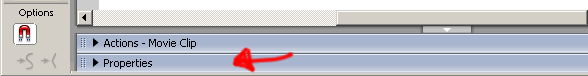
Position it at the bottom right of the stage. The screen should look as follows:



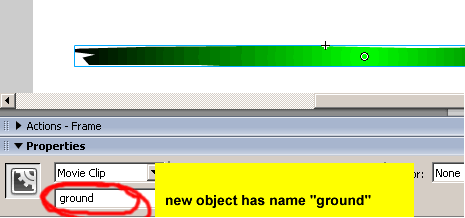
## Give your GROUND object the property of having the name “ground”

Display the “**Properties**” window (you may need to click the word “Properties” in the blue bar) at the bottom of the screen

(OR press **CTRL-F3** OR choose menu **Windows** | **Properties**)



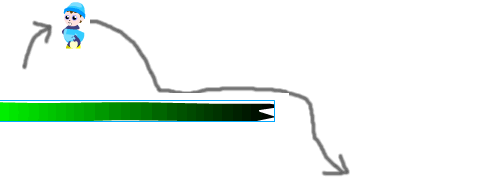
Underneath the words “Movie Clip” replace “<instance name>” with the object name “ground”



HINT: If you can’t see “Movie Clip” and “<instance name”> in the Properties window, ensure you have selected the “Arrow” tool and clicked on your green ground (so that the green ground object has a blue selection rectangle around it)

**- if that doesn’t work, ask a tutor!**

## Add program instructions to make guy stop falling when it hits ground



We now need to add a “conditional” statement (an IF-statement) to our program. I.e. we need to make the program test a condition – if the condition is true it will do something, if false it will not do anything. The logic for our test is as follows:

* IF
  + Guy hits the ground
* THEN
  + Make guy stop falling downwards

The ActionScript code for these statements is as following:

* IF-statements are written with the word “if” (lower case), an expression that should evaluate to “true” or “false”, and actions to be executed if true in braces (curly brackets), i.e.

if ( condition\_expression )

{

// actions to be executed if condition is TRUE

}

* Flash ActionScript movie clips provide the method hitTestObject() to allow us to test if one sprite is touching another. This method returns a truth value “true” or “false”, so we can use this method as out IF-statement condition. So we can ask if “guy” is touching “ground” as follows

guy.hitTestObject(ground)

* The actions we want to execute if guy is touching ground is to set the value of dy to zero:

dy = 0;

This test needs to be placed in the gameLoop() method, to change the value of “dy” – i.e. we need to set “dy” to zero to stop guy falling if he is touching the ground.

So add the new bold statements in the code for gameLoop():

public function gameLoop(event: Event):void

{

// check collision with ground

if (guy.hitTestObject(ground))

{

// stop falling down

dy = 0;

}

... as before

}

## Why is our guy stuck halfway into the ground?

Sometimes the following happens – the guy falls **below** the level of the ground, halfway into it!



Why does this happen?

* Our “dy” makes our guy move in steps of 5 or 10 pixels at a time (or whatever dy becomes as guy falls). So the when guy hits the ground he may be partly below the top – i.e. the previous step he may have been 2 pixels above the top of the ground, and if dy was 12 (say) then next frame guy will be 8 pixels **below** the top of the ground.
* This problem would not happen if guy was moving down the screen with a dy value of 1 pixel. But, due to gravity, the value of dy increases each frame, so unless guy started at the vertical position of the ground, (or only 1 pixel above it) then he will always be falling down the screen with a dy value bigger than 1.

We can fix this easily, since the graphics have been carefully created to solve this problem. What we need to do is set the Y-position of the guy to be the same as the Y-position of the ground – this will position his feet just above the ground to make things look right. When graphical “movie clips” are created (e.g. for PersonTemplate and GroundTemplate, from which instances “guy” and “ground” were created) the “reference point” can be specified to which the (x,y) position of the object refer. For PersonTemplate (guy) the reference point is at his feet – so whatever “y” position is assigned to guy will determine the vertical position of his feet. For GroundTemplate (ground) the reference point is at the **top** of the green drawing, so the “y” position of ground refers to the **top** of the ground. So if the “y” position of guy is set to be the same as the “y” position of the thing he touches (e.g. ground) then guy’s feet will be positioned on the top of the thing he touches – so it will look as though he is standing on the ground, rather than buried half way down it.

This can be very simply programmed by adding one extra line inside our “if” statement, to set the “y” position of guy to be the same as the “y” position of ground:

// check collision with ground

if (guy.hitTestObject(ground))

{

// stop falling down

dy = 0;

// set "guy"s feet (y) to meet top of "ground" (y)

guy.y = ground.y;

}

The modified full program listing incorporating these changes is shown below.

package

{

import flash.display.MovieClip;

import flash.events.\*;

public class Main extends MovieClip

{

/////////////////

/// variables

/////////////////

/\*\* small change in X position next "tick" \*/

var dx:Number = 5;

/\*\* small change in Y position next "tick" \*/

var dy:Number = -20;

/\*\* simulated gravity

makes sprite accelerate down screen by this amount \*/

var gravity:Number = 1.5;

/////////////////////////////////////

/\*\*

constructor method - initalisation actions

\*/

public function Main()

{

stage.addEventListener( Event.ENTER\_FRAME, gameLoop );

}

/////////////////////////////////////

/\*\*

gameloop - actions to be executed each "tick" of the game clock

\*/

public function gameLoop(event: Event):void

{

// check collision with ground

if (guy.hitTestObject(ground))

{

// stop falling down

dy = 0;

// set "guy"s feet (y) to meet top of "ground" (y)

guy.y = ground.y;

}

// add (dx,dy) to guys position

guy.x += dx;

guy.y += dy;

// add gravity acceleration to "dy"

dy += gravity;

}

}

}

Gravity Guy – keeping the program tidy with methods/functions

## Tidy up gameLoop() by moving statements into separate methods

Although we can see most of our program in one 1-page, or 1-screen at the moment, almost all non-trivial programs end up many 10s or even 100s of lines long. A way to manage this complexity is to separate different actions and decisions in your program by placing them into named methods.

At present our gameLoop() method does the following 2 things:

1. it checks collisions between guy and ground
2. it moves guy and updates dy with gravity

Therefore we can simplify the structure of our program by placing each of these sets of statements in their own method, and then making gameLoop() “call” each method in the right sequence.

So the new method to check collisions with ground can be written as follows:

public function checkGoundCollisions():void

{

// check collision with ground

if (guy.hitTestObject(ground))

{

// stop falling down

dy = 0;

// set "guy"s feet (y) to meet top of "ground" (y)

guy.y = ground.y;

}

}

So the new method to move guy and updatesdy with gravity can be written as follows:

public function updatePosition():void

{

// add (dx,dy) to guys position

guy.x += dx;

guy.y += dy;

// add gravity acceleration to "dy"

dy += gravity;

Method gameLoop() is now much simpler – it simply calls these two methods in the right order:

public function gameLoop(event: Event):void

{

checkGoundCollisions();

updatePosition();

}

The modified full program listing incorporating these changes is shown below.

package

{

import flash.display.MovieClip;

import flash.events.\*;

public class Main extends MovieClip

{

/////////////////

/// variables

/////////////////

/\*\* small change in X position next "tick" \*/

var dx:Number = 5;

/\*\* small change in Y position next "tick" \*/

var dy:Number = -20;

/\*\* simulated gravity

makes sprite accelerate down screen by this amount \*/

var gravity:Number = 1.5;

/////////////////////////////////////

/\*\*

constructor method - initalisation actions

\*/

public function Main()

{

stage.addEventListener( Event.ENTER\_FRAME, gameLoop );

}

/////////////////////////////////////

/\*\*

gameloop - actions to be executed each "tick" of the game clock

\*/

public function gameLoop(event: Event):void

{

checkGoundCollisions();

updatePosition();

}

/////////////////////////////////////

/\*\*

do appropriate actions if guy is touching solid ground

\*/

public function checkGoundCollisions():void

{

// check collision with ground

if (guy.hitTestObject(ground))

{

// stop falling down

dy = 0;

// set "guy"s feet (y) to meet top of "ground" (y)

guy.y = ground.y;

}

}

/////////////////////////////////////

/\*\*

update position of guy, and add gravity to dy

\*/

public function updatePosition():void

{

// add (dx,dy) to guys position

guy.x += dx;

guy.y+=dy;

// add gravity acceleration to "dy"

dy+=gravity;

}

}

}

Gravity Guy – moving left/right with ARROW keys

## Add instructions to move guy with left/right arrow keys

We are now going to add some user control – move the guy left and right with the arrow keys.

What we want to happen is that when the LEFT arrow key is pressed down, guy’s dx value will be set to -5. And when the RIGHT arrow key is pressed down, guy’s dx value will be set to 5. So pressing LEFT moves guy 5 pixels to the left each “tick”, and pressing RIGHT moves guy 5 pixels to the right each tick. And releasing the keys stops guy moving horizontally – so his dx value will be zero.

Each key on the computer keyboard has its own unique “key code”. The key code for the LEFT arrow is 32, and the key code for the RIGHT arrow is 33. Flash’s “Keyboard” class has given meaningful names to variables so that we can refer to the codes for keys using meaningful names. “Keyboard.LEFT” refers to the code 32, and “Keyboard.RIGHT” refers to the code 33.

They way Flash ActionScript 3 lets us detect key presses is through the events KEY\_DOWN when a key is pressed down, and KEY\_UP when a key is released. We can record when keys are down in an “array” by recording a “true” for a key’s KEY\_DOWN event and a “false” for its KEY\_UP event. An array is an ordered collection of values, so we will use an array called “isDown” organised by the “key code”.

The code we need to add to our program is the following. We need to write code to:

1. Import (make available) Flash’s Keyboard class

import flash.ui.Keyboard;

1. Declare the array “isDown” (to store “true” for keys hat are pressed)

var isDown:Array = new Array();

1. Register methods to respond to the events KEY\_DOWN and KEY\_UP

public function Main()

{

// register "gameLoop()" function to respond to "ENTER\_FRAME" events

stage.addEventListener( Event.ENTER\_FRAME, gameLoop );

// register methods to respond to key down/up events

stage.addEventListener( KeyboardEvent.KEY\_DOWN, myKeyDown );

stage.addEventListener( KeyboardEvent.KEY\_UP, myKeyUp );

}

1. Define a new method to set “dx” if LEFT or RIGHT arrow key is pressed

function checkArrowKeys():void

{

// default is not to move

dx=0;

// update (vx,vy) according to keys that are down

if (isDown[Keyboard.LEFT])

{

dx = -5;

}

if (isDown[Keyboard.RIGHT])

{

dx = 5;

}

}

1. Modify method gameLoop() to call the new method to check arrow keys, before testing for collisions or moving guy:

public function gameLoop(event: Event):void

{

checkArrowKeys();

checkGoundCollisions();

updatePosition();

}

The modified full program listing incorporating these changes is shown below.

package

{

import flash.display.MovieClip;

import flash.events.\*;

import flash.ui.Keyboard;

public class Main extends MovieClip

{

/////////////////

/// variables

/////////////////

/\*\* small change in X position next "tick" \*/

var dx:Number = 0;

/\*\* small change in Y position next "tick" \*/

var dy:Number = 0;

/\*\* simulated gravity

makes sprite accelerate down screen by this amount \*/

var gravity:Number = 1.5;

/\*\* array storing which keys are current pressed down \*/

var isDown:Array = new Array();

/////////////////////////////////////

/\*\*

constructor method - initalisation actions

\*/

public function Main()

{

// register "gameLoop()" function to respond to

// "ENTER\_FRAME" events

stage.addEventListener( Event.ENTER\_FRAME, gameLoop );

// register methods to respond to key down/up events

stage.addEventListener( KeyboardEvent.KEY\_DOWN, myKeyDown );

stage.addEventListener( KeyboardEvent.KEY\_UP, myKeyUp );

}

/////////////////////////////////////

/\*\*

gameloop - actions to be executed each "tick" of the game clock

\*/

public function gameLoop(event: Event):void

{

checkArrowKeys();

checkGoundCollisions();

updatePosition();

}

/////////////////////////////////////

/\*\*

do appropriate actions if guy is touching solid ground

\*/

public function checkGoundCollisions():void

{

// check collision with ground

if (guy.hitTestObject(ground))

{

// stop falling down

dy = 0;

// set "guy"s feet (y) to meet top of "ground" (y)

guy.y = ground.y;

}

}

/////////////////////////////////////

/\*\*

update position of guy, and add gravity to dy

\*/

public function updatePosition():void

{

// add (dx,dy) to guys position

guy.x += dx;

guy.y += dy;

// add gravity acceleration to "dy"

dy += gravity;

}

/////////////////////////////////////

/\*\*

set keycode isDown array entry to "true" when key is pressed down

\*/

function myKeyDown(event:KeyboardEvent):void

{

isDown[event.keyCode] = true;

}

/////////////////////////////////////

/\*\*

set keycode isDown array entry to "false" when key is released

\*/

function myKeyUp(event:KeyboardEvent):void

{

isDown[event.keyCode] = false;

}

/////////////////////////////////////

/\*\*

update dx/dy for arrow keys

\*/

function checkArrowKeys():void

{

// default is not to move

dx = 0;

// update (vx,vy) according to keys that are down

if (isDown[Keyboard.LEFT])

{

dx = -5;

}

if (isDown[Keyboard.RIGHT])

{

dx = 5;

}

}

}

}

Gravity Guy – replace hard-coded numbers with a variable

## Create a variable “speed” rather than write 5 or -5

It is a bad idea to write numbers directly in statements, such as setting dx to 5 or -5. The numbers represent something, and we should use variables containing those numbers instead. The use of variables make the program statements more meaningful, and also means the values can be changed easily, by changing the value in the variable rather than having to search through the program to replace values “hard coded” into statements.

We make the following changes to the program:

* The variable “speed” should be declared at the beginning of the class.

var speed:int = 5;

* Therefore the number 5 should be replaced with “speed” and the number -5 should be replaced with (-1 \* speed).

function checkArrowKeys():void

{

// default is not to move

dx=0;

// update (vx,vy) according to keys that are down

if (isDown[Keyboard.LEFT])

{

dx = (-1 \* speed);

}

if (isDown[Keyboard.RIGHT])

{

dx = speed;

}

}

Gravity Guy – Jump upwards when SPACE key pressed

## Add a JUMP feature when SPACE bar is pressed

Platform games provide characters with a way to “jump”. We are going to make guy jump upwards when the SPACE key is pressed down. To make guy jump we have to set “dy” to a large **NEGATIVE** number, e.g. -15.

We make the following changes to the program:

* A new method “checkJumpKeys()” is defined to check for the SPACE jump key

public function checkJumpKey():void

{

// only jump if SPACE AND no falling (o

if (isDown[Keyboard.SPACE])

{

dy = -15;

}

}

* The method gameLoop() is modified to call the new method “checkJumpKeys()” just before guy has his position updated (i.e. we check to see whether to make “dy” a value to jump upwards just before changing guy’s y-position)

public function gameLoop(event: Event):void

{

checkArrowKeys();

checkGoundCollisions();

checkJumpKey();

updatePosition();

}

Gravity Guy – Jump upwards ONLY when standing on solid ground

## Add a JUMP feature when SPACE bar is pressed

Your player should be able to jump ONLY when he is not falling. At the moment if you press SPACE anytime, even when “guy” is high in the air, he jumps even higher!

So we need to add a Boolean (true/false) “flag” to record whether or not guy is falling. This will usually be true, but will be set to **false** when we detect that “guy” is touching the ground (or in later steps, a platform).

We make the following changes to the program:

* First you need to declare a new variable called “isFalling” of type Boolean; setting the initial value of this variable to “true”:

var isFalling:Boolean = true;

* Now we need to make some additions to your gameLoop() method:
* Every time gameLoop() is called it will reset isFalling back to “true”

public function checkGoundCollisions():void

{

// default is not touching ground/platform

isFalling = true;

...

}

* Then each IF statement testing for “guy” touching the ground (or a platform) will set isFalling to FALSE when “guy” is standing on some form of ground

...

// check collision with ground

if (guy.hitTestObject(ground))

{

// stop falling down

dy = 0;

// set "guy"s feet (y) to meet top of "ground" (y)

guy.y = ground.y;

// record that guy is standing on solid ground

isFalling = false;

}

* Finally, we need to modify the checkJumpKey() method, so that “guy” will only be made to jump if both the SPACE key is pressed AND the isFalling flag is FALSE
  + i.e. jump if SPACE pressed and “guy” is standing on some form of solid ground

public function checkJumpKey():void

{

// only jump if SPACE AND not falling

if (isDown[Keyboard.SPACE] && !isFalling)

{

dy = -15;

}

}

The modified full program listing incorporating these changes is shown below.

package

{

import flash.display.MovieClip;

import flash.events.\*;

import flash.ui.Keyboard;

public class Main extends MovieClip

{

/////////////////

/// variables

/////////////////

/\*\* small change in X position next "tick" \*/

var dx:Number = 0;

/\*\* small change in Y position next "tick" \*/

var dy:Number = 0;

/\*\* amount you want sprite to move by each "tick"

in whatever direction\*/

var speed:int = 5;

/\*\* simulated gravity

makes sprite accelerate down screen by this amount \*/

var gravity:Number = 1.5;

/\*\* array storing which keys are current pressed down \*/

var isDown:Array = new Array();

/\*\* true/false "flag" to indicate if in the air (true)

or standing on something solid (false) \*/

var isFalling:Boolean = true;

/////////////////////////////////////

/\*\*

constructor method - initalisation actions

\*/

public function Main()

{

// register "gameLoop()" function to respond to

// "ENTER\_FRAME" events

stage.addEventListener( Event.ENTER\_FRAME, gameLoop );

// register methods to respond to key down/up events

stage.addEventListener( KeyboardEvent.KEY\_DOWN, myKeyDown );

stage.addEventListener( KeyboardEvent.KEY\_UP, myKeyUp );

}

/////////////////////////////////////

/\*\*

gameloop - actions to be executed each "tick" of the game clock

\*/

public function gameLoop(event: Event):void

{

checkArrowKeys();

checkGoundCollisions();

checkJumpKey();

updatePosition();

}

/////////////////////////////////////

/\*\*

do appropriate actions if guy is touching solid ground

\*/

public function checkGoundCollisions():void

{

// default is not touching ground/platform

isFalling = true;

// check collision with ground

if (guy.hitTestObject(ground))

{

// stop falling down

dy = 0;

// set "guy"s feet (y) to meet top of "ground" (y)

guy.y = ground.y;

// record that guy is standing on solid ground

isFalling = false;

}

}

/////////////////////////////////////

/\*\*

update position of guy, and add gravity to dy

\*/

public function updatePosition():void

{

// add (dx,dy) to guys position

guy.x += dx;

guy.y+=dy;

// add gravity acceleration to "dy"

dy+=gravity;

}

/////////////////////////////////////

/\*\*

set keycode isDown array entry to "true" when key is pressed down

\*/

function myKeyDown(event:KeyboardEvent):void

{

isDown[event.keyCode] = true;

}

/////////////////////////////////////

/\*\*

set keycode isDown array entry to "false" when key is released

\*/

function myKeyUp(event:KeyboardEvent):void

{

isDown[event.keyCode] = false;

}

/////////////////////////////////////

/\*\*

update dx/dy for arrow keys

\*/

function checkArrowKeys():void

{

// default is not to move

dx=0;

// update (vx,vy) according to keys that are down

if (isDown[Keyboard.LEFT])

{

dx = (-1 \* speed);

}

if (isDown[Keyboard.RIGHT])

{

dx = speed;

}

}

/////////////////////////////////////

/\*\*

check is SPACE key has been pressed for jumping

\*/

public function checkJumpKey():void

{

// only jump if SPACE AND not falling

if (isDown[Keyboard.SPACE] && !isFalling)

{

dy = -15;

}

}

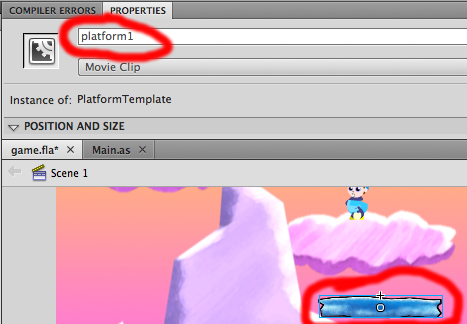
}

}

Gravity Guy – Collision (stop falling) with platforms

## Create “platform1” from the library

Just as guy can stand and jump from the ground, he should do the same from the various platforms to be added to the screen. Drag an object from the “PlatformTemplate” onto the stage and in the “Properties” panel name this object “platform1”.



The same actions need to be taken when guy touches platform1 as when he touches the ground:

* The change in Y is set to zero (so he stops falling/jumping)
* His feet are placed on top of the ground/platform he is touching
* The isFalling Boolean flag is set to FALSE (since “guy” is standing on solid ground)

This is achieved by adding a new IF-statement to our method checkGoundCollisions():

// check collisions with platforms

if (guy.hitTestObject(platform1))

{

dy = 0;

guy.y = platform1.y;

isFalling = false;

}

The modified full listing for the method checkGoundCollisions() is as follows:

public function checkGoundCollisions():void

{

// default is not touching ground/platform

isFalling = true;

// check collision with ground

if (guy.hitTestObject(ground))

{

// stop falling down

dy = 0;

// set "guy"s feet (y) to meet top of "ground" (y)

guy.y = ground.y;

// record that guy is standing on solid ground

isFalling = false;

}

// check collisions with platforms

if (guy.hitTestObject(platform1))

{

dy = 0;

guy.y = platform1.y;

isFalling = false;

}

}

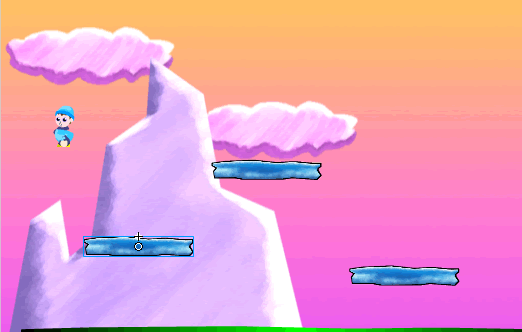
The guy should now stop falling when he lands on platform 1.

Gravity Guy – Add more platforms

## Add more platforms

Add platforms 2 and 3 in the same way (and name them “platform2” and “platform3” in the Properties panel). And add code to test if those platforms have been hit (collided with).

The screen should look as follows:



You code for method checkGoundCollisions() should look as follows:

public function checkGoundCollisions():void

{

// default is not touching ground/platform

isFalling = true;

// check collision with ground

if (guy.hitTestObject(ground))

{

// stop falling down

dy = 0;

// set "guy"s feet (y) to meet top of "ground" (y)

guy.y = ground.y;

// record that guy is standing on solid ground

isFalling = false;

}

// check collisions with platforms

if (guy.hitTestObject(platform1))

{

dy = 0;

guy.y = platform1.y;

isFalling = false;

}

if (guy.hitTestObject(platform2))

{

dy = 0;

guy.y = platform2.y;

isFalling = false;

}

if (guy.hitTestObject(platform3))

{

dy = 0;

guy.y = platform3.y;

isFalling = false;

}

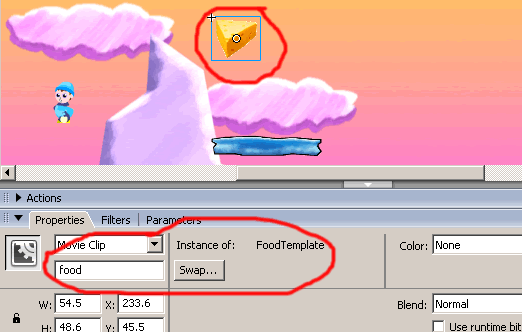
}

Gravity Guy – Create food object for guy to collect

## Create object “food” from the library “FoodTemplate”

Drag an object from the “FoodTemplate” onto the stage (it looks like a **piece of cheese**). While this object is still selected give the name “food” in the Properties panel window.

You screen should look as follows:



## Make food object move to random new location when “eaten”

We need to detect that guy has hit the cheese, then use some random mathematics to make the cheese food re-appear somewhere else on the screen Having tied up the structure of our code in the last exercise, let’s keep things tidy – so from now on statements will go into their own method if necessary.

So create a new method checkFoodCollisions() to do the following:

* Detect if “guy” has touched the “food” object

public function checkFoodCollisions():void

{

if (guy.hitTestObject(food))

{

...

}

}

* If so, then set the (\_x, \_y) position of the food to a random position on screen
  + NOTE the Flash screen is called the “Stage”, and this Stage object knows its own width and height
  + Therefore the random position can be set to a value based on the width and height of the Stage

food.x = Math.random() \* 800;

food.y = Math.random() \* 300;

The whole method checkFoodCollisions() should look as follows:

public function checkFoodCollisions():void

{

if (guy.hitTestObject(food))

{

food.x = Math.random() \* 800;

food.y = Math.random() \* 300;

}

}

The call to this method should be inside gameLoop(), just after the call to the checkGroundCollisions() method:

public function gameLoop(event: Event):void

{

checkArrowKeys();

checkGoundCollisions();

checkFoodCollisions();

checkJumpKey();

updatePosition();

}

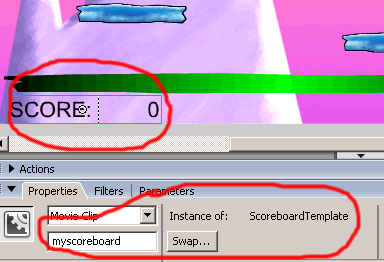
Gravity Guy – Add a score counter for pieces of food eaten

Finally, we will add an on-screen score to the game, that counts the number of pieces of “cheese” that “guy” has eaten.

## Create object “myscoreboard” from the library “ScoreboardTemplate”

Drag an object from the “ScoreboardTemplate” onto the stage (it looks like a **“Score: 0”**). While this object is still selected give it the name “myscoreboard” in the Properties panel window.

Your screen should look as follows:



## Add “score” counter variable

Add a new instance variable to your class by adding the following line to create and zero a score counter for your game:

var score:Number = 0;

## When food eaten add to score and update scoreboard on screen

Add the following 2 lines of code to the IF-statement that tests for when guy hits food. The first line adds 1 to the score (this is called “incrementing” a variable), the second line updates the “screenScore” text object in the scoreboard with the new score value.

public function checkFoodCollisions():void

{

if (guy.hitTestObject(food))

{

food.x = Math.random() \* 800;

food.y = Math.random() \* 300;

score++;

myscoreboard.score.text = score;

}

}

****

**Congratulations**

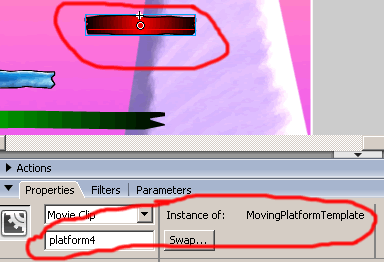
**You have now created a complete platform game!**

Gravity Guy – Bonus step – add a moving platform

## Create object “platform4” from the library “MovingPlatformTemplate”

Drag an object from the “MovingPlatformTemplate” onto the stage (it looks like a red platform). While this object is still selected give it the name “platform4” in the Properties panel window.

You screen should look as follows:

****

## Add another IF- test to checkGroundCollisions() so player can land on platform4

At the end of method checkGroundCollisions() write a new IF-test to detect when “guy” touches platform4:

function checkGroundCollisions():Void

{

...

// stop "guy" falling if he hits platform4

if( \_root.guy.hitTest( \_root.platform4 ) )

{

// set Y change to ZERO

dy = 0;

// set "guy"s feet to meet top of "platform4"

\_root.guy.\_y = \_root.platform4.\_y;

// set isFalling to FALSE

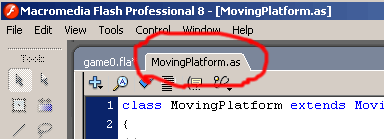
isFalling = false;

}

} // method

## How does the moving platform work?

If you are curious how the moving platform works open the file “MovingPlatform.as” from the **File | Open** menu.

****

This is a good example of one of the benefits of Object Oriented programming – you can use Classes that other people have written without having to understand the logic of the program code inside them. In this example, you can simply create an instance of a moving platform object on screen, and get all the benefits of its behaviour (bouncing up and down on the screen), and all you need to do is give the object a name and add a single IF-statement to do something when “guy” touches the object.

Gravity Guy – Bonus step – a “cheat” to reset object positoins

## Add a “cheat” in case food hard to get / guy falls out of sight

Sometimes our guy will fall off the ground, or the food will be positioned somewhere we can’t get to, so let’s add a “cheat” that puts things into easy places for us when we hit the SHIFT key.

Create a new method checkCheatKeys() that tests for the SHIFT key being pressed, and if found, then it resets the position of “guy” and “food”, and makes “guy” start falling gently”

/\*\*

reset game objects if SHIFT key is pressed

\*/

function checkCheatKeys():Void

{

// if SHIFT pressed reset game objects

if( Key.isDown( Key.SHIFT ))

{

// reposition "guy"

\_root.guy.\_x = 50;

\_root.guy.\_y = 50;

// reposition "food"

\_root.food.\_x = 100;

\_root.food.\_y = 100;

// start "guy" falling slowly

dy = 1;

isFalling = true;

}

} // method

Modify your gameLoop() method to call checkCheatKeys() after it calls the method to test for arrow keys:

function gameLoop():Void

{

// default is no Horizontal movement

dx = 0;

// default is that "guy" is falling

isFalling = true;

checkArrowKeys();

checkCheatKeys();

checkGroundCollisions();

checkFoodCollisions();

...

Since the “score” variable is not reset, you now have implemented a “cheat” for your game that will allow you to get as high a score as you wish ☺

# INFORMATION – the 3 special Unity folders

Unity projects all have the following folders (and sometimes some other loose temporary files):

**/unity\_project**

**/Assets**

**/Library**

**/ProjectSettings**

**NOTE: Make changes INSIDE the Unity application**

You should not change the contents of the 3 folders **Assets, Library** and **ProjectSettings**, except through actions in the Unity application itself. If you make changes to these folders, then Unity may lose track of the relationships between the different parts of the game/scenes, and you will have ‘broken’ your project.

So while you may find it interesting to ‘explore’ what is in the folders using the file browser of your Operating System, avoid moving / deleting / editing any of the contents of the Unity project folders.

**NOTE: These folder names are special – avoid them for your own folders**

Since Unity projects use the 3 special folders named: **Assets, Library** and **ProjectSettings**, you must NEVER create any sub-folders of your own with any of these 3 names.

For example, if you create a folder named ‘Assets’, inside the Unity folder named ‘Assets’, when you next try to open the Unity application it may become confused, and it could corrupt your saved project in a way that is impossible to recover from…