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

Gravity Guy 2D (2014) - a little computer game...

Part 2 – the story continues …



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

CONTENTS

1 Recap, and aims of this part of the tutorial 3

2 Play a sound when a piece of food is eaten 4

3 Respawn the hero character when it falls down too low … 8

4 Separate the GUI ‘view’ from the Player ‘model’ 9

5 Add killer ‘spike’ objects to the scene 11

6 Congratulations … 14

# Recap, and aims of this part of the tutorial

## Features to date

In the first part of this tutorial you implemented the following:

* Hero guy controlled by the user
* Fixed blue platforms
  + Acting as solid ‘Ground’ which can be landed on / jumped from
* Moving red platform
  + Acting as solid ‘Ground’ which can be landed on / jumped from
  + Moving at a customisable speed, between customisable max/min Y values
* A piece of cheese, tagged ‘Food’
  + Which when hit adds to Player’s score
  + And also removes itself from the scene
* An integer score
  + Stored in the player
  + And displayed using a GUILabel

## Skills and knowledge learned

In the first part of this tutorial you learned the following:

* The roles of the different window panels in the Unity editor
  + Project
  + Inspector
  + Hierarchy and Scene
  + Game
* How to:
  + play / stop a game
  + add instances of a prefab into the current scene
  + add a copy of an image sprite to the scene, the copying becoming a gameObject
  + change the properties of a selected gameObject or Project asset
  + add a Box Collider 2D to a gameObject
  + create and add a new string ‘tag’ to a game object
  + activate collision ‘trigger’ event messages for collisions between colliders
  + create a new C# script and add an instance as a component to a gameObject
  + create a prefab, populated with the components and properties of a gameObject in the scene

## New features / skills to be learned in this part of the tutorial

In this part of the tutorial you will add the following features to our game:

* **Play a sound each time we add to the score**
* **Respawn the user when they fall too far down the screen**
* **Separate the score ‘view’ display to the user, from the observed score property inside the Player object**
* **Add killer ‘spike’ gameObjects to the scene**
  + **And have these make the player lose lives and respawn each time they are hit**
  + **This will involve adding a ‘lives’ property to our Player, and adding corresponding GUI display of this property**
* *Add a “Game Over” scene to our game*
  + *And have the user see this scene when they lose their last life*
* *Display scores using GUIText gameObjects, rather than scripted Labels*

# Play a sound when a piece of food is eaten

**The 3 audio object types**

Unity has 3 special sounds related types of objects:

* **AudioClip** – a sound file, such as our ‘yum.mp3’
* **AudioSource** – a component of a game object, allowing a gameObject to play a sound clip file

- this can be empty in a gameObject

or it can be populated with an AudioClip

* **AudioListener** – this is like an electronic ‘ear’ – usually we just work with the default setup, which is that when a new Scene is created in Unity, the Main Camera automatically has an AudioListener

**Other media assets – import them just the same way**

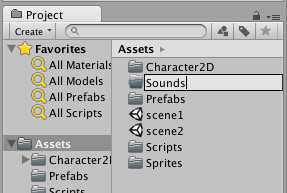
The methods shown in this section can be used to import 2D images, 3D models, audio files, text files, video files, C# scripts etc.

So know you know how to get NEW media/scripting assets into your Unity project …

## Add a sound clip file to a new Project folder “sounds”

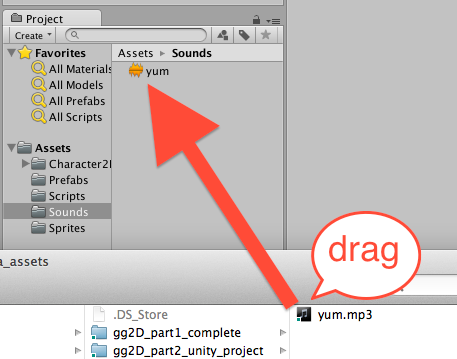
Create a folder “sounds” in the Project panel:

* In the Project panel select the Assets folder
* From the ‘Create’ dropdown menu choose ‘Folder’
* Rename the new folder ‘Sounds’

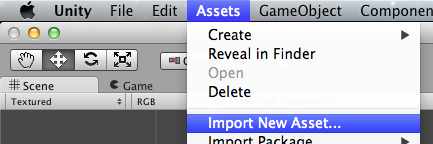


Add a copy of file sounds ‘yum.mp3’ into this new Sounds folder. You can either use the drag-and-drop method, or the Assets| Import menu approach …

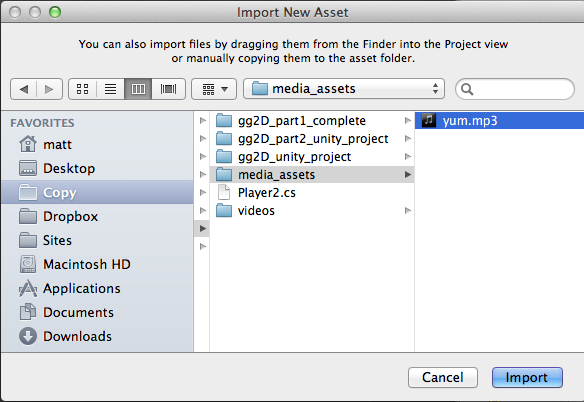
* **Drag-and-drop method:**
  + Ensure your **Sounds** folder is selected in the **Project** panel
  + Drag from an OS file window the file into the **Sounds** folder area (right hand side of **Project** panel)
  + You should now see the ‘yum’ sound clip file (with audio wave icon) in the **Project** panel **Sounds** folder



* **Drag-and-drop method:**
  + Ensure your Sounds folder is selected
  + Choose menu: Assets | Import New Asset …



* + Use the file dialog to navigate to and import the sounds file



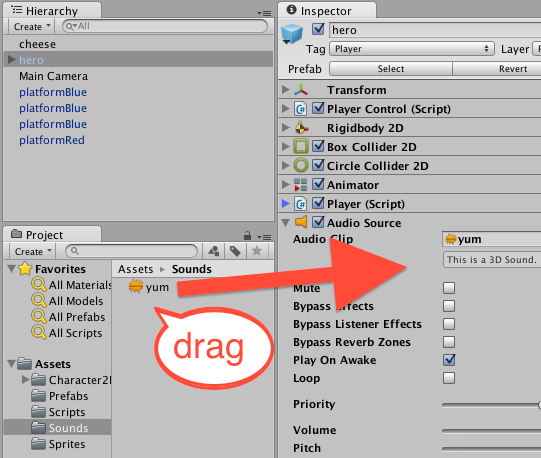
* + You should now see the ‘yum’ sound file (with audio wave icon) in the **Project** panel **Sounds** folder

## Add an AudioSource component, containing our ‘yum’ sound, to the ‘hero’ gameObject

Any Unity gameObject that wants to play a sound must have an ‘AudioSource’ component. The AudioSource component can be pre-loaded with a sound clip file (such as ‘yum’ …), or at run time a particular sound clip file can be loaded into it and then played.

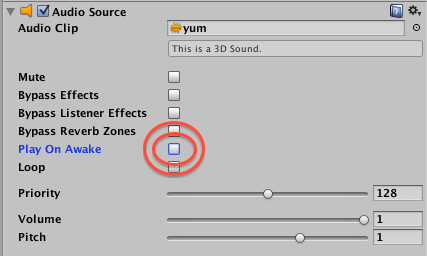
Unity makes things easy – if we drag an AudioClip sound clip file from the Project panel into a gameObject it will automatically add an AudioSource component to the gameObject, and populate it with the sound clip file that was dragged:

* In the **Project** panel select the **Sounds** folder
* In the **Hierarchy** select the **hero** gameObject
* Drag the **yum** audioClip sound file into the Inspector (which should be showing us the components and properties of gameObject **hero)**
* You should now see that the **hero** gameObject contains an **AudioSource** component, and its **AudioClip** property is our **yum** sound clip file



Finally, we do NOT want this sound to play at the beginning of the scene when the **hero** gameObject is instantiated, so we need to DESELET the “Play On Awake” option in the AudioSource component:

* In the **Hierarchy** select the **hero** gameObject
* In the Inspector un-tick the “Play on Awake” property of the AudioSource component



## Add to our Player script class, so that it plays a sound when “Food” items are collided with

Start editing the Player script class in Monodevelop

* In the **Project** panel select the **Scripts** folder
* Double click the **Player** script class file to load it into the **Monodevelop** editor

Recap – your Player script class code should look as follows:

using UnityEngine;  
using System.Collections;  
  
public class Player : MonoBehaviour {    private int score = 0;  
      
    private void OnGUI(){  
        string scoreMessage = "Score = " + score;  
        GUILayout.Label(scoreMessage);  
    }  
  
    private void OnTriggerEnter2D(Collider2D c){  
        string tag = c.tag;  
  
        if('Food' == tag){  
            score++;  
        }  
    }  
}

AFTER the statement where we increment (add 1 to) the score when hitting something with tag “Food”, we are going to add another statement, to tell the gameObject (**hero**) to send a “Play()” message to its AudioSource component (i.e. play the **yum** AudioClip in its AudioSource component). So each time we hit something tagged “Food” we play that sound:

*private void OnTriggerEnter2D(Collider2D c){  
        string tag = c.tag;  
  
        if('Food' == tag){  
            score++;*

            audio.Play();

*}  
    }*

## Playtest your game

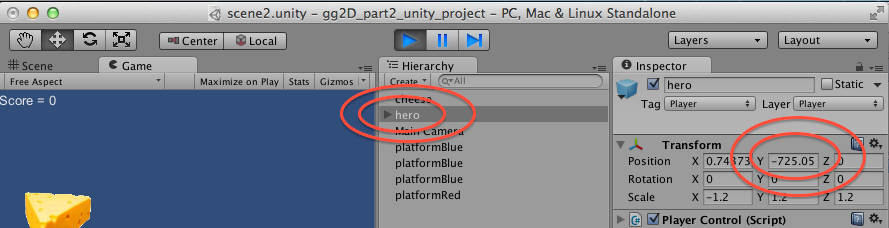
You should now hear the **yum** sound when you ‘eat’ a piece of cheese

# Respawn the hero character when it falls down too low …

## Add to our Player script class, to respawn when Y value too low

If our hero player character falls off a platform, it just keeps on falling …

See how – deselect the Game panel option “Maximise on play”, select **hero** in the Hierarchy (so we can see that gameObject’s properties in the Inspector), and run the game. When your character walks off a platform you can see its Y-position keep increasing as a larger and larger negative (downwards) value …



As well as OnGUI(), another message all scene gameObjects are sent EVERY FRAME is the Update() message. So any code in an Update() method will be executed every frame. So we are going to write an Update method that will return our **hero** character gameObject back to its starting postion (0,5,0) if its Y-position is less than -5 (below where all platforms should be).

* In the **Project** panel select the **Scripts** folder
* Double click the **Player** script class file to load it into the **Monodevelop** editor
* Add a new float property **DEATH\_Y** set to -5
  + This is the Y-position below which the **hero** character will be moved back to (0,5,0)
* Add a new Update() method to the C# code:

    private int DEATH\_Y = -5f;  
  
    private void Update(){  
        float y = transform.position.y;  
  
        if(y < DEATH\_Y){  
            Vector3 startPosition = new Vector3(0,5,0);  
            transform.position = startPosition;  
        }  
    }

## Playtest your game

When your **hero** character falls off a platform, after a short time it should reappear back at (0,5,0) – its start position.

**Note – watch our for speed / angle velocity …**

Our current code moves the **hero** back to the start position, but he will still be falling, and moving left/right with the same velocity as he was when off the platform.

A more advanced re-spawning of a character woud also require us to remove any downward speed, or sideways motion, when the character is moved back to its starting position.

But we’ll leave that ‘tweak’ for another tutorial …

# Separate the GUI ‘view’ from the Player ‘model’

## Separate View code from our Player behaviour script class

As a game becomes more complex, it becomes important to ensure each script class has a well defined boundary of responsibility. One example of this is our Player script class – it should maintain and update important properties of our player, such as the player’s SCORE and number of LIVES left, and any INVENTORY the player may be carrying.

However, the Player script class should NOT worry about HOW the values of these properties are COMMUNICATED to the user.

Each important property should be a PRIVATE variable, but should offer PUBLIC accessor methods (getters and setters) for any other objects that have responsibilities that require them to access these properties of the Player.

We are now going to REFACTOR our code, to separate the VIEW of player properties (our GIU to the user) from the Player script class itself.

## Remove GUI code from Player, but add public GET method for ‘score’

Let’s remove the GUI code from our Player script class, but add a public GETTER method for the important ‘score’ property:

* In the **Project** panel select the **Scripts** folder
* Double click the **Player** script class file to load it into the **Monodevelop** editor
* Edit the code as follows:
  + Remove method OnGUI()
  + Add a new method GetScore()

*using UnityEngine;  
using System.Collections;  
  
public class Player : MonoBehaviour {  
    private int score = 0;***public int GetScore(){  
        return score;  
    }***private float DEATH\_Y = -5f;  
  
    private void Update(){  
        float y = transform.position.y;  
  
        if(y < DEATH\_Y){  
            Vector3 startPosition = new Vector3(0,5,0);  
            transform.position = startPosition;  
        }  
    }  
  
    private void OnTriggerEnter2D(Collider2D c){  
        string tag = c.tag;  
  
        if("Food" == tag){  
            score++;  
            audio.Play();  
        }  
    }  
}*

## Create new GameGUI script class, and add it to the Main Camera

Create a new C# script named GameGUI:

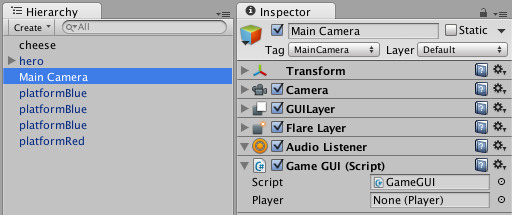
* In the **Project** panel select the **Scripts** folder
* From the **Project** ‘Create’ menu choose ‘C# Script’ and rename this new script “GameGUI”
* Double click your new **GameGUI** script class file, and in MonoDevelop edit its code to be the following:

**using UnityEngine;  
using System.Collections;  
  
public class GameGUI : MonoBehaviour {  
    public Player player;  
  
    private void OnGUI(){  
        int playerScore = player.GetScore();  
        string scoreMessage = "Score = " + playerScore;  
        GUILayout.Label(scoreMessage);  
    }  
}**

## Add an instance of our GameGUI script class as a component of Main Camera

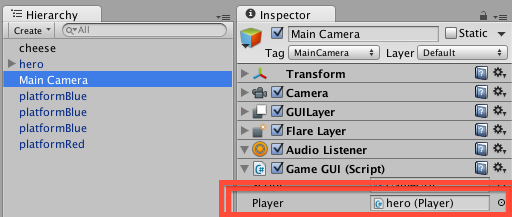
A script class does nothing, unless an instance of it has been added as a component of a gameObject in the scene. Do the following:

* In the **Project** panel select the **Scripts** folder
* In the Hierarchy select the **Main Camera** gameObject
* Drag the C# script class file from the **Project Scripts** folder into the **Inspector**
* You should now see that Hiearchy gameObject **Main Camera** now has a component Game GUI (Script)



We now need to make a link between the **hero** gameObject in our Hierarchy, and the ‘player’ variable in our GameGUI component in the **Main Camera**. Do the following:

* In the Hierarchy select the **Main Camera** gameObject
* Drag the **hero** gameObject from the Hierarchy into the Player property of the Game GUI (Script) component of the **Main Camera**
* The public ‘Player’ variable of the GameGUI script component of Main Camera should now indicate that it is a reference (link) to **hero (Player)**
  + The instance of the Player script class that is a component inside the **hero** gameObject



The game will look just the same to the user – but you know that the GUI ‘view’ has now been separated from the Player ‘model’ of player’s properties.

# Add killer ‘spike’ objects to the scene

## Add a new property ‘lives’ to our Player script class

Let’s start the player off with 3 lives. We need to add a property ‘lives’ to our Player script class, and also add a public ‘getter’, to allow our GameGUI to retrieve and display the value of the ‘lives’ variable:

Add a ‘lives’ inteter property to script class Player, and a corresponding public ‘getter’:

* In the **Project** panel select the **Scripts** folder
* Double click the **Player** script class file to load it into the **Monodevelop** editor
* Edit the code as follows:
  + Add a new private integer property ‘lives’ initialised to 3
  + Add a new public method GetLives()

*using UnityEngine;  
using System.Collections;  
  
public class Player : MonoBehaviour {  
    private int score = 0;***private int lives = 3;***public int GetScore(){  
        return score;  
    }***public int GetLives(){  
        return lives;  
    }**

## Edit our GameGUI to display the lives property to the user

Add code to our GameGUI script class to display the player’s lives integer on screen:

Add a ‘lives’ integer property to script class Player, and a corresponding public ‘getter’:

* In the **Project** panel select the **Scripts** folder
* Double click the **Player** script class file to load it into the **Monodevelop** editor
* Edit the code as follows:
  + We will move the Score display code into new method DisplayScore()
  + We will create a new Lives display code method DisplayLives()

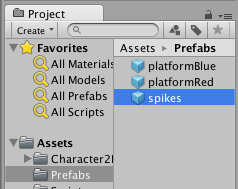
*using UnityEngine;  
using System.Collections;  
  
public class GameGUI : MonoBehaviour {  
    public Player player;***private void OnGUI(){  
        DisplayScore();  
        DisplayLives();  
    }  
      
    private void DisplayScore(){  
        int playerScore = player.GetScore();  
        string scoreMessage = "Score = " + playerScore;  
        GUILayout.Label(scoreMessage);  
    }  
      
    private void DisplayLives(){  
        int playerLives = player.GetLives();  
        string livesMessage = "Score = " + playerLives;  
        GUILayout.Label(livesMessage);  
    }  
}**

## Add some copies of the ‘spikes’ prefab as gameObjects to the scene

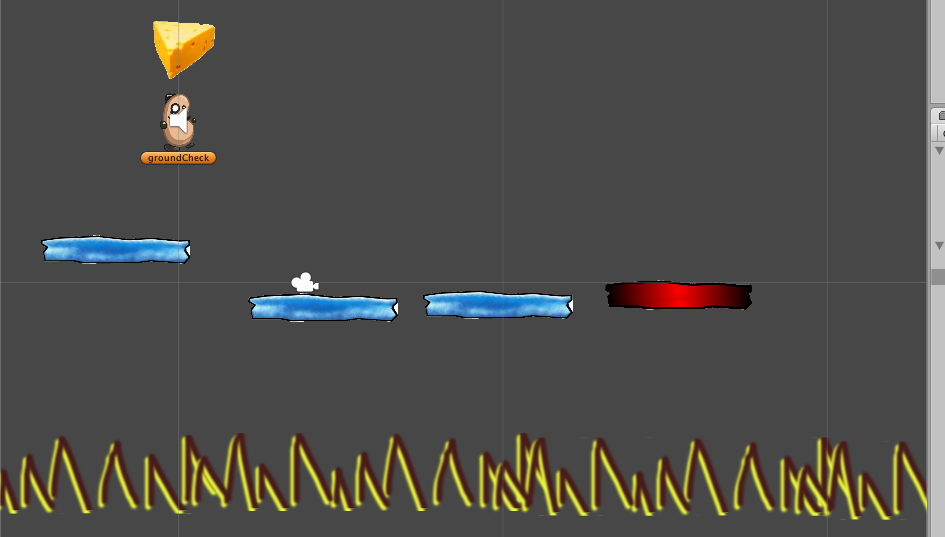
Rather than our player falling into some invisible minimum Y position, let’s actually display some ‘spikes’, whereby the player loses a life when they fall onto them.

Add some copies of the ‘spikes’ prefab as gameObjects to our scene, in a row BELOW the platforms:

* In the **Project** panel select the **Prefabs** folder
* Drag copies of the **spikes** prefab onto the scene, in a row below the platforms:

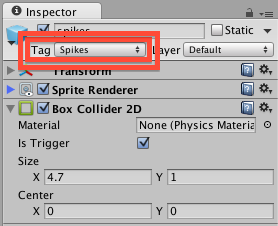


Your scene should now look something like the following:



­­­ ­­

If you select a ‘spikes’ gameObject, you’ll see that they have their collider ‘trigger’ ticked, and have the tag ‘Spikes’:



## Edit Player code, to decrement lives when hit something tagged ‘spikes’

We now need to edit our Player script class, so that when it collides with something tagged ‘spikes’, we decrement the lives, and move it back to the start position.

NOTE – we’ll also remove the code that repositions the character if the Y value is too low … so the onus is on the LEVEL DESIGNER to ensure all falls lead to ‘spikes’ …

Let’s edit our collision trigger method, and turn the code to move the object to the start position into a separate method:

* In the **Project** panel select the **Scripts** folder
* Double click the **Player** script class file to load it into the **Monodevelop** editor
* Edit the **code as follows:**

*using UnityEngine;  
using System.Collections;  
  
public class Player : MonoBehaviour {  
    private int score = 0;  
    private int lives = 3;  
      
    public int GetScore(){  
        return score;  
    }  
      
    public int GetLives(){  
        return lives;  
    }  
  
    private float DEATH\_Y = -5f;  
  
    private void OnTriggerEnter2D(Collider2D c){  
        string tag = c.tag;  
  
        if("Food" == tag){  
            score++;  
            audio.Play();  
        }* **if("Spikes" == tag){  
            lives--;  
            MoveToStartPosition();  
        }  
    }  
  
    private void MoveToStartPosition(){  
        Vector3 startPosition = new Vector3(0,5,0);  
        transform.position = startPosition;  
    }***}*

# Congratulations …

****

**Congratulations**

**You have now created part 2 of the tutorial !**