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

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

Part 3 – improving the ‘view’ (UI display of score) …



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

CONTENTS

1 Aims of this part of the tutorial 1

2 Create a UI Text field on screen to display our score 2

3 Separate the GUI ‘view’ from the Player ‘model’ 3

# Aims of this part of the tutorial

## 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:

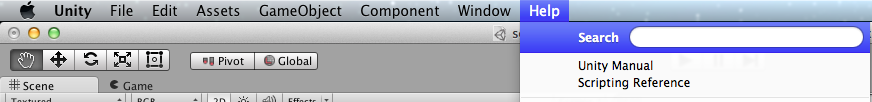
* Learn how to lookup things in the Unity manual
* Create and size/position a UI “Text” object in the scene
* Change the text displayed through code in your Player class
* Move UI updates from outside of method Update(), into a separate “View” class
  + Improving efficiency and also refactoring our code towards the “model-view-controller” pattern software architecture, which separates different responsibilities into different classes
  + (de-coupling code about the game “state” from code that displays the game state to the user via the GUI)

# Learn how to lookup things in the Unity manual

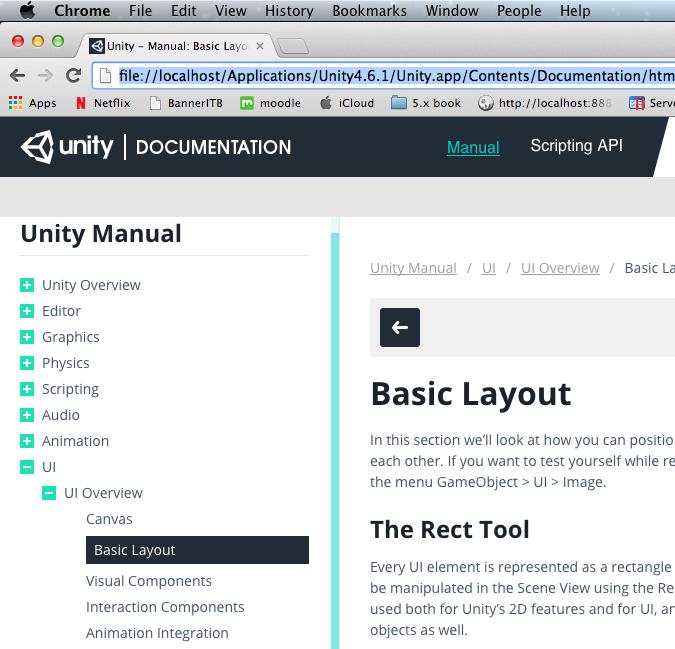
## Manual and Scripting reference installed as part of Unity application

As well as all the documentation on the Unity website, as part of the Unity application installation you already have the Unity Manual and the Unity Scripting Reference installed on the computer you are using.

Open these web-page based manuals from the Help menu in the Unity application:



For this part of the Gravity Guy tutorial look up and learn about the Unity UI system:

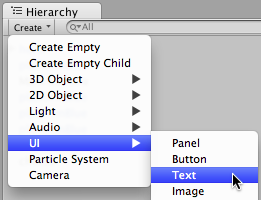


# Create a UI Text field on screen to display our score

## Adding UI objects to the scene

Since Unity 4.6.1 a new improved UI (User Interface) system has been created. It continues the Unity approach of allowing as much as possible to created via drag-and-drop & Inspector properties as possible, and then adding small code components to respond to events and update values at run-time.

Create a new UI Text object. In the Hierarchy panel choose menu: Create | UI | Text.



Three (yes 3!) items will be added to the Hierarchy of game objects in the current scene:

1. Your new UI Text object

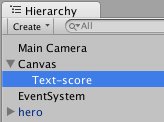
* It will start with the default name “Text” – append to this something meaningful, e.g. “Text-score”

1. A Canvas object

* all UI objects are ‘children’ of a Canvas – so one is created if none exist when a new UI object is added)

1. An EventSystem

* This allows us to link methods in script classes with UI events such as button clicks and mouse over/out etc.
* Since our Score display is not interactive we’ll not worry about UI events until another day ☺

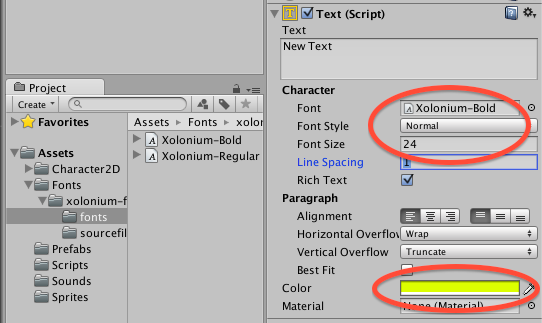


## Changing properties for a UI Text object

Import the provided Fonts folder into your project (just drag it from a folder window into the right-hand side of your Project panel).

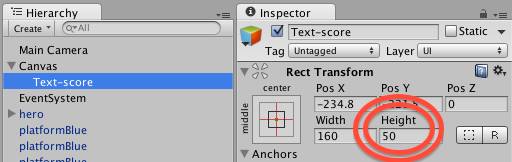
Now set the colour of the Text-score to yellow and the font to Xolonium-Bold.

Ensure first you select Text-score in the Hierarchy panel:



Note:

* If you wish to make the text larger than 24, you will also need to increase the Height property of the UI Text object:



## Position and stretch the UI Text object – the Rect Transform

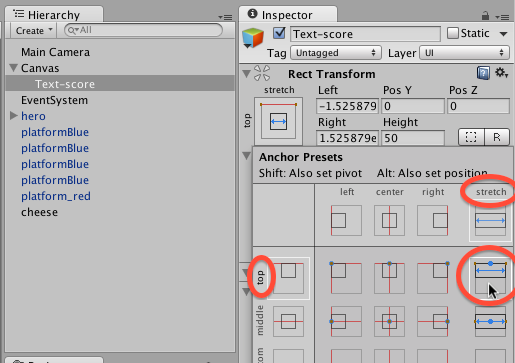
Part of the new UI system is a powerful approach to RELATIVE positioning, the gracefully adapts to different sized screens. The position and size system for UI objects is based around two concepts:

* Pivot point
  + The pivot point is the point about which scaling and rotation, and re-sizing is based
* Anchors
  + Anchors are points in the PARENT of a UI element, and control how the position of the 4 corners of the rectangle for a UI element will change if the size/shape of the parent of the UI element changes
  + In most cases one of the pre-set positions of the anchores will meet your needs

For our Score text, we just want it to be at the center top of the game panel, and the simplest pre-set for this is to choose **top-stretch** for the **anchors** of our Text-score gameObject.

Do the following:

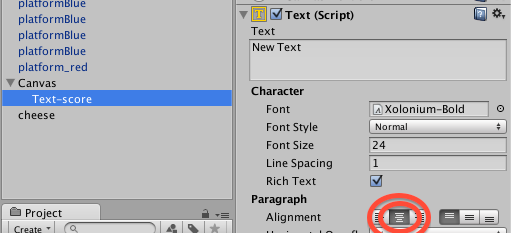
1. Ensure **Text-score** is selected in the **Hierarchy**
2. In the **Inspector** open the anchor preset popup, which is achieved by clicking inside the rectangle with ‘cross-hairs’ to the left of the Pos X property of the **Rect Transform** component
3. While holding down the SHIFT and ALT keys click the **top-stretch** box



Check how things look in the Game panel:



If you want the text cented in its stretched rectangle, just choose the center tool in the **Paragraph** properties in the **Text (Script)** property in the Inspector:



# Change the text displayed through code in your Player class

## Edit your Player class

Rather than displaying the score via the Console status bar, let’s edit our Player code to change the ‘text’ property of our new UI Text game object.

We need our Player object have a reference to the UI Text object, so we need a variable for this. We’ll make this a PUBLIC variable – which will allow us to simply DRAG a reference from the Text component inside the Text-score gameObject in the Hierarchy, to the public variable revealed in the Inspector for the instance of our Player class in the hero gameObject ! Once you get the hang of this, you’ll see why Unity is so popuar, and quick, to develop game with…

We need to write/change 3 lines:

1. The Text UI class is part of the UnityEngine.UI package, so we must add a ‘using’ statement at the beginning of our class so we can declare objects of type ‘Text’
2. We need to declare our public Text variable inside our class
3. We replace the print() statement inside Update() with a statement to set the ‘text’ property of the Text property in our Text-score object to our scoreMessage string

*using UnityEngine;*

*using System.Collections;*

// 1. we need this package to use the ‘Text’ class

using UnityEngine.UI;

*public class Player : MonoBehaviour {*

// 2. public variable, reference to UI Text-score

public Text scoreText;

*private int score = 0;*

*private float deathY = -15;*

*void Update(){*

*string scoreMessage = "Score = " + score;*

// 3. update the ‘text’ property of UI Text-score

scoreText.text = scoreMessage;

*float y = transform.position.y;*

*if(y < deathY){*

*MoveToStartPosition();*

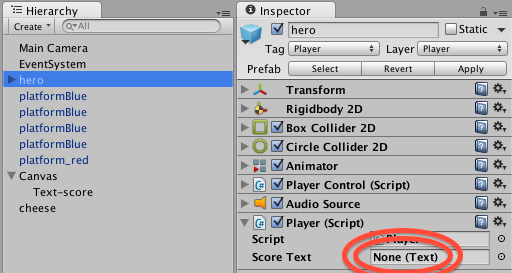
*}*

*}*

*…as before*

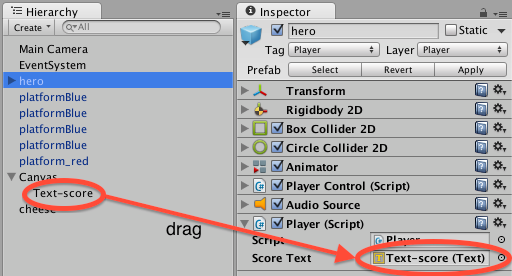
## Drag reference from Text-score to the public variable in Player instance

If you view the properties of our hero gameObject in the Inspector, you’ll see that we have a NULL (in the Unity Inspector we see “None (Empty)”) value for the scoreText variable in the instance of our Player class that is in our hero gameObject:



NOTE: Unfortunately Unity’s Inspector capitalises the each word of the variable name, and adds spaces between them! So our public variable ‘**scoreText’** becomes ‘**Score Text’** in the Inspector – it’s just something you have to learn to live with I’m afraid …

Unity let’s us drag a reference from a gameObject in the Hierarchy to a public variable in a component in another gameObject. Since our public variable is of type **Text** Unity will notice that there is a **Text** component inside our **Text-score** gameObject, and when dragged in at design-time, Unity will automatically assign this object reference when the scene starts running.



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

## Get code out of Update() & sepate View update to separate class

We don't to update the score EVERY FRAME (50 – 100 times per second!). So Update() is the **wrong** location for us to update the score text display.

Also, our **Player** class should be about the players ‘state’ (lives / score / inventory etc.) – how we indicate this state to the user is a different task, which a different team member of our game company might have. So let’s refactor (improve our code by rewriting – even thought the final user experience might be exactly the same) to separate out UI text update into a PlayerDisplay class. While it will grow bigger, initially our PlayerDisplay class needs only one method, which we'll name UpdateScore():

using UnityEngine;

using System.Collections;

using UnityEngine.UI;

public class PlayerDisplay : MonoBehaviour {

public Text scoreText;

public void UpdateScoreText(int newScore){

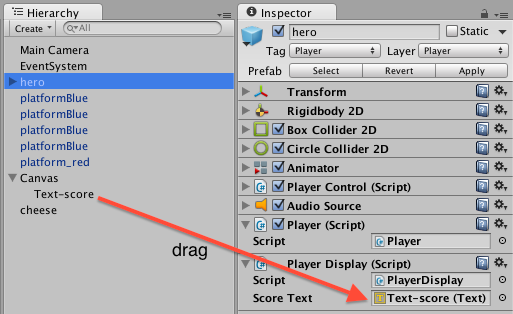
string scoreMessage = "Score = " + newScore;

scoreText.text = scoreMessage;

}

}

Create a new C# file in the Project panel named PlayerDisplay, containing this code. The drag an instance of this class into the hero gameObject in the Hierarchy. Finally, with hero selected in the Hierarchy, drag the **Text-score** gameObject from the Canvas into the **Score Text** public variable slot in the **PlayerDisplay (Script)** component of hero in the Inspector.



Now, we need to update our Player class, so that it doesn’t have to worry about the score display UI Text object. Remember, an instance of our Player class is a component in our hero gameObject. We also have an instance of our new PlayerDisplay class as a component in hero. What Player needs to do is get a reference to the PlayerDisplay instance, luckily Unity provides the very hand GetComponent<>() method allowing a scripted component to use reflection to find references to any other component inside the same parent gameObject. The pointy brackets <> are part of the ‘type’ C# language feature, whereby any class may be used by the method.

So in our Player class we need a variable that is a reference to an object of the class PlayerDisplay:

private PlayerDisplay playerDisplay;

We also want to assign this reference to the PlayerDisplay object that is inside the same hero gameObject that our Player instance is inside:

playerDisplay = GetComponent<PlayerDisplay>();

It is also a good idea to update the UI Text for the score when the scene first starts (so the initial score of 0 is seen by the user). We’ll put the statement above, and our first call to update the score display both into our Start() method – which is called in each object when the scene begins (or when an object is first created while a scene is running):

void Start(){

playerDisplay = GetComponent<PlayerDisplay>();

playerDisplay.UpdateScoreText(score);

}

Finally, just after we change the score (in out OnTriggerEnter2D() method) we’ll then immediately call our UpdateScoreText() method again – so as soon as the score changes the user can see the new score:

*void OnTriggerEnter2D(Collider2D hit){*

*if(hit.CompareTag("Food")){*

score++;

playerDisplay.UpdateScoreText(score);

*Destroy (hit.gameObject);*

*audio.Play();*

*}*

*}*

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.

## Playtest your game

Close the Build Settings, and reload **scene1**, then run your game. Then keep falling off platforms to make the number of lives less than zero

**Congratulations**

**You have now created part 3 of the tutorial !**

# FULL LISTINGS

## Player

using UnityEngine;

using System.Collections;

public class Player : MonoBehaviour

{

private PlayerDisplay playerDisplay;

private int score = 0;

private float deathY = -15;

void Start()

{

playerDisplay = GetComponent<PlayerDisplay>();

playerDisplay.UpdateScoreText(score);

}

void Update()

{

float y = transform.position.y;

if(y < deathY)

{

MoveToStartPosition();

}

}

private void MoveToStartPosition()

{

Vector3 startPosition = new Vector3(0,5,0);

transform.position = startPosition;

}

void OnTriggerEnter2D(Collider2D hit)

{

if(hit.CompareTag("Food"))

{

score++;

playerDisplay.UpdateScoreText(score);

Destroy (hit.gameObject);

audio.Play();

}

}

}

## PlayerDisplay

using UnityEngine;

using System.Collections;

using UnityEngine.UI;

public class PlayerDisplay : MonoBehaviour

{

public Text scoreText;

public void UpdateScoreText(int newScore)

{

string scoreMessage = "Score = " + newScore;

scoreText.text = scoreMessage;

}

}