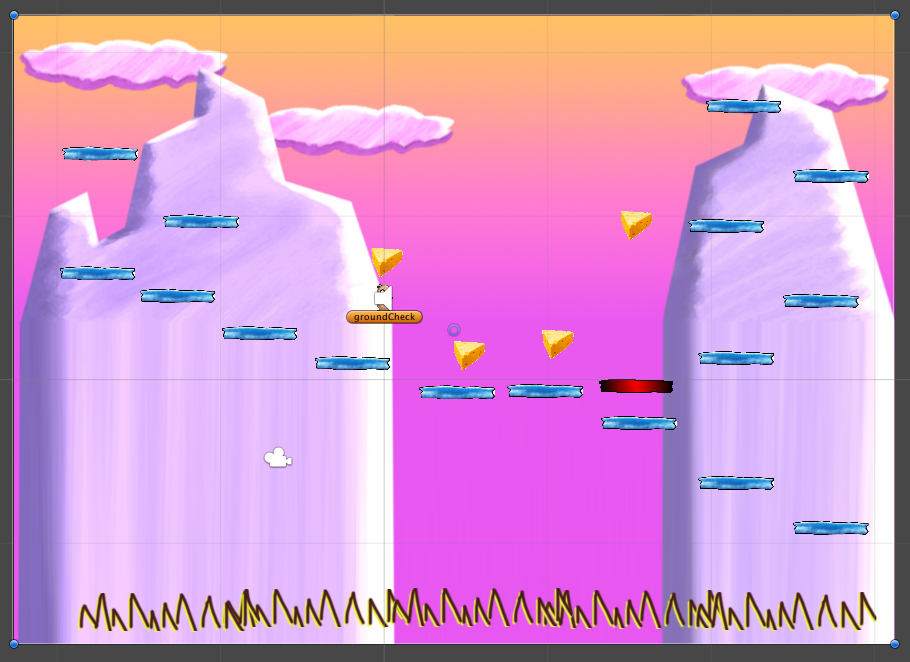
|  |  |
| --- | --- |
| ITB logo portrait B&W | INSTITUTE OF TECHNOLOGY BLANCHARDSTOWN |

A Taster of Computing

[[VERSION – Unity 2D – C# language]]

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

Part 7 – background & moving camera

****

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 2

2 If not done so already, make a ‘cheese’ prefab 2

3 Organise all those ‘sorting layers’ 3

4 Add background and move behind contents of scene 5

5 EXTRA – a first visit to 3D 8

6 Add ‘CameraFollow’ script to the camera, and set X/Y limits 9

7 FULL LISTINGS 12

# 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:

* Tidy up hierarchy by parenting child items to empty container objects
* Set the ‘resolution’ of the application build to 800 x 600 pixels
* Different sounds for different collisions

# If not done so already, make a ‘cheese’ prefab

## Turn your ‘cheese’ object in the scene into a prefab

Do the following if you do not already have a ‘cheese’ prefab:

* In your Project/Prefabs folder create a new empty prefab named ‘cheese’
* Drag ‘cheese’ from your Hierarchy into your new prefab, so the prefab turns blue and stores all the properties of the cheese gameObject in the scene
* Drag a few more instances of cheese to different parts of the screen

# Organise all those ‘sorting layers’

## Remove all unneeded sorting layers (since we seem to have inherited some)

It seems that as well as stealing the ‘hero’ potato man from the Unity 2D Platformer, we also inherited some ‘sorting layers’.

Since we are now going to be using sorting layers, so ensure our new background is BEHIND our player and platforms, then we’re going to tidy things up …

First, delete all sorting layers except ‘’Character”:

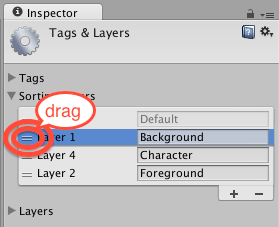
* Open the Tags and Layers lists in the Inspector
  + Choose menu: Edit | Project Settings | Tags and Layers



Now create a layer ‘Background’ (onto which we’ll place our background), and a layer ‘Foreground’.

Drag these (using the = ‘equals’ sign drag handle on the left) so that they are listed in the following top-down sequence:

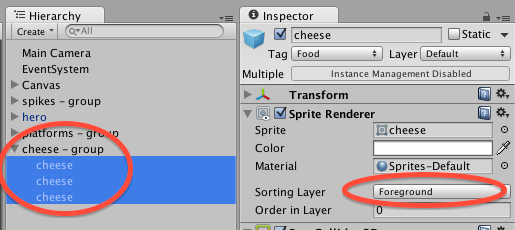
* Default
* Background
* Character
* Foreground



Note, Unity uses these Sorting Layers to visually decide what is displayed on top of what when 2D items overlap. Items at the bottom of the list are drawn last, and so appear ON TOP of items at the top of the list. So our **Foreground** items will be topmost, above items on sorting layer **Character**, and all above items on sorting layer **Background**.

Note – you can further arrange items in the same Sorting Layer by changing the Order in Layer number, but we don’t need to go that far for Gravity Guy.

You can add as many of these are you need, so for example you might have ‘lampposts’ some of which a character moves behind and some of which in front of etc. All the parts of the **hero** potato man should already have their Sprite Rendered Sorting Layer set to Character. Choose Foreground for the Sorting Layer for all platforms, spikes and cheese gameObjects:

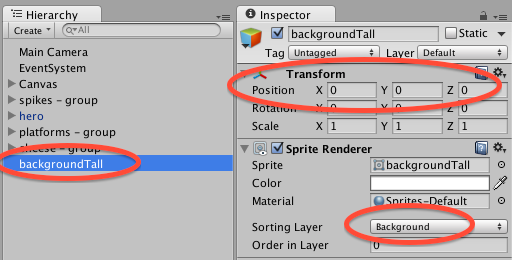


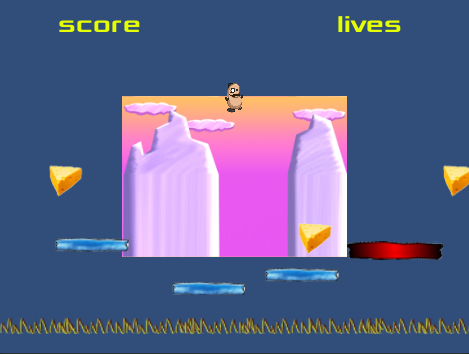
# Add background and move behind contents of scene

## Add background image sprite to scene

First drag a copy of the backgroundTall image sprite into the scene, and position at (0,0,0)

* Drag image **backgroundTall** from Project ‘**Sprites’** folder into the scene
* In the **Inspector** set the **Position** of this gameObject’s **Transform** property to (0,0,0)
* Set the **Sorting Layer** of the **Sprite Renderer** component to **Background**

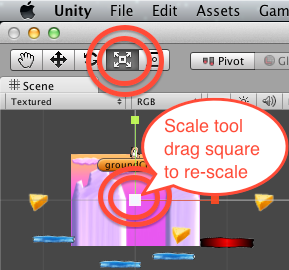
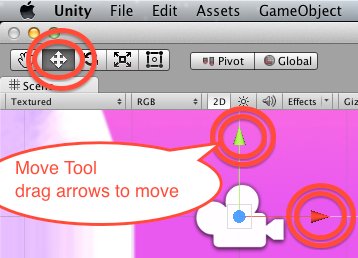




## Scale the backgroundTall sprite to (3,3,3)

As can be seen, the image is too small at present, so scale this image to (3,3,3) – or manually as you prefer.

* In the **Inspector** set the **Scale** of this gameObject’s **Transform** property to (3,3,3)
* Alternatively, you may wish to use the Move tool (Shortcut key ‘W’) and the Scale tool (Shortcut key ‘R’) to move and resize the image to fit your layout
  + In the Scene panel, click and drag the white rectangle when using the Scale tool to then be able to drag the mouse left/right to rescale the image manually



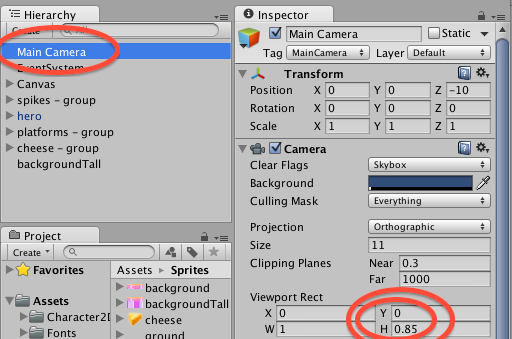
## Reduce the height of the camera’s ViewPort rectangle

What the user sees in the final Game window when the game is player comes from:

* Rectangles showing what each active camera can ‘see’
* UI gameObjects

Each active camera has its output mapped to a rectangle ‘Viewport’ in the final Game window. While this defaults to the whole game window (0,0) to (1,1), we can change which parts of the Game window show contents from which camera.Let’s make the Main Camera only display the bottom 85% of the Game window, leaving no background to be displayed behind our UI score and lives text items.

* Select the Main Camera in the Hierarchy, then in the **Inspector** set the **H (height)** property for the Camera – Viewport Rect component to 0.85



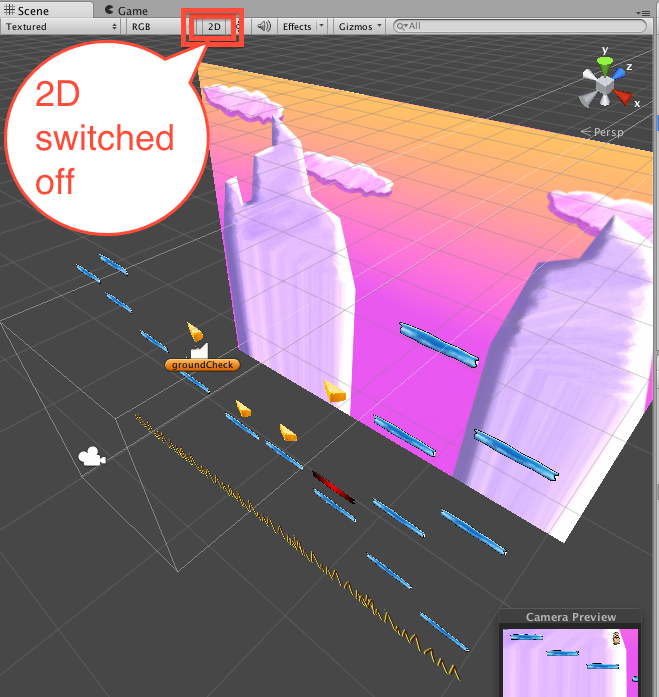
You should now easily see the Score and Lives UI text in their own little rectangle with no background image making them hard to read:



# EXTRA – a first visit to 3D

## Switch TO 3D view to SEE objects with z-values

If you turn off ‘2D’ mode in the **Scene panel**, you can see how the objects are positioned by the x,y and z values:



**NOTE – controlling what you see in the Scene panel in 3D mode takes a bit of practice, try this:**

* Select (double click) the ‘Main Camera’
* Use ALT-mouse-drag to rotate what is viewed
  + I find having camera on the lower left, ‘looking’ toward the upper right (positive Z) a good arrangement
* You may need to ZOOM-out to see most objects
  + Either use the mouse scroll wheel to zoom
  + OR you can go into HAND-tool mode (press Q), and then use CTRL-mouse-drag to zoom in/out

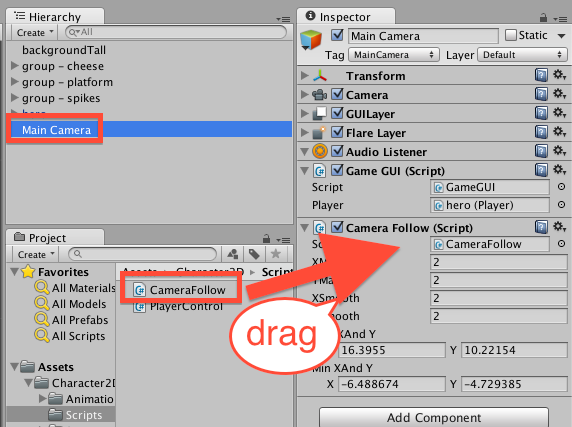
In the ‘olden days’ we used the Z-value to manage which items are infront / behind the camera, but Sorting Layers are a better solution for this for 2D games.

# Add ‘CameraFollow’ script to the camera, and set X/Y limits

## Add the basic CameraFollow script to the Main Camera

Let’s add a pre-written script to the camera, so that it moves when the player’s ’hero’ travels to one of the edges of the screen:

* In **Hierarchy** select Main Camera
* Drag ‘CameraFollow’ script from **Project panel** folder Assets – Character2D – Scripts into the **Inspector**
  + Or directly onto the Main Camera in the **Hierarchy**
  + Both methods add an instance of the script class as a component of the Main Camera GameObject



## Playtest your game

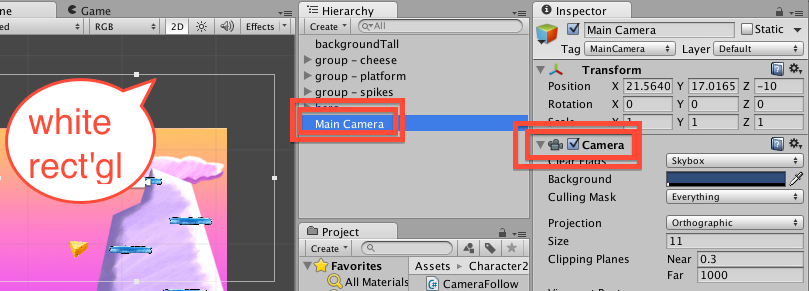
You should find the camera now moves to follow the camera, when the player moves some distance away from the center of the window.

BUT: it is likely that the camera limits are not correct – i.e. the script has a MAXIMUM and MINIUM X and Y setting, beyond which the camera will never be allowed to move …

## Calculate your scene’s MAX and MIN X and Y values

Here is my personal procedure to calculate the 4 values (maxX, maxY) (minX, minY) that will ensure the camera will allow your player to see all around your scene, but never beyond the scene contents. Do the following:

* Select the Main Camera in the **Hierarchy**
  + And ensure its ‘Camera’ component has its properties displayed in the **Inspector**
    - This ensures you can see the WHITE RECTANGLE showing the part of the scene the camera can ‘see’ and will be displayed in the **Game panel** when the game runs

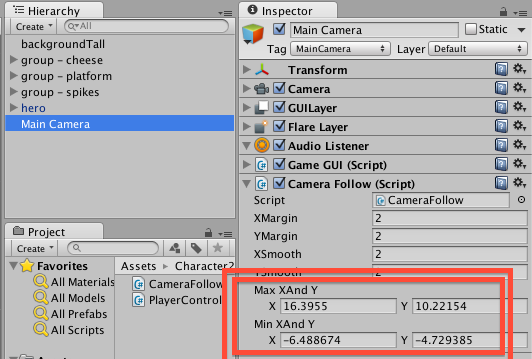


* Select the MOVE-tool (press W)
  + You should see the X and Y move arrows on the camera in the **Scene panel**
    - The colours of the move arrows can be remembered as follows:
      * X, Y, Z
      * R, G, B (from RGB colours – red, green, blue)
      * So X is RED and Y is GREEN
* Find the MAXIMUM X- and Y- values for our camera movement
  + Move the camera so that its TOP RIGHT white rectangle is at the TOP RIGHT of your background image
  + Make a note of the X and Y values of the camera’s transform-position
    - For me it was: X = 16.3955, Y = 10.22154
    - TIP: copy and paste the exact decimal values from the **Inspect** into a blank text editor window
* Find the MINIMUM X- and Y- values for our camera movement
  + Move the camera so that its BOTTOM LEFT white rectangle is at the BOTTOM LEFT of your background image
  + Make a note of the X and Y values of the camera’s transform-position
    - For me it was: X = -6.488674, Y = -4.729385

## Set your scene’s camera limits in the Inspector

Now we know the numbers to type in for the properties of our CameraFollow script component:

* Select the Main Camera in the **Hierarchy**
* In the **Inspector** for the Camera Follow component, type in (or copy/paste) the maximum and mimum X and Y values you noted from the previous step



Now when you run the game, you should find the camera can follow the hero guy all around your scene, but never moves too far left/right or up/down, using the limits we have entered

NOTE – you may have to add a few extra platforms / move the spikes, to fully test camera movement in your scene

You can also SCALE the background sprite (or add a copy left/right) to make a larger playing area, and adjust the camera limits appropriately

### Playtest your game

Start on the Welcome scene and click the button to play the game.

You should now have a nice background, and the camera should move around with the player, until the movement limits are reached

**Congratulations**

**You have now created part 7 of the tutorial !**

# FULL LISTINGS

## CameraFollow.cs

using UnityEngine;

using UnityEngine;

using System.Collections;

public class CameraFollow : MonoBehaviour

{

public float xMargin = 2f;

public float yMargin = 2f;

public float xSmooth = 2f;

public float ySmooth = 2f;

public Vector2 maxXAndY = new Vector2(5f, 5f);

public Vector2 minXAndY = new Vector2(-5f, -5f);

private Transform player;

void Awake ()

{

player = GameObject.FindGameObjectWithTag("Player").transform;

}

bool CheckXMargin()

{

// Returns true if the distance between the camera and the player in the x axis is greater than the x margin.

return Mathf.Abs(transform.position.x - player.position.x) > xMargin;

}

bool CheckYMargin()

{

// Returns true if the distance between the camera and the player in the y axis is greater than the y margin.

return Mathf.Abs(transform.position.y - player.position.y) > yMargin;

}

// FixedUpdate – fixed time interval (for physics)

void FixedUpdate ()

{

TrackPlayer();

}

void TrackPlayer ()

{

// By default the target x and y coordinates of the camera are it's current x and y coordinates.

float targetX = transform.position.x;

float targetY = transform.position.y;

// If the player has moved beyond the x margin...

if(CheckXMargin())

// ... the target x coordinate should be a Lerp between the camera's current x position and the player's current x position.

targetX = Mathf.Lerp(transform.position.x, player.position.x, xSmooth \* Time.deltaTime);

// If the player has moved beyond the y margin...

if(CheckYMargin())

// ... the target y coordinate should be a Lerp between the camera's current y position and the player's current y position.

targetY = Mathf.Lerp(transform.position.y, player.position.y, ySmooth \* Time.deltaTime);

// The target x and y coordinates should not be larger than the maximum or smaller than the minimum.

targetX = Mathf.Clamp(targetX, minXAndY.x, maxXAndY.x);

targetY = Mathf.Clamp(targetY, minXAndY.y, maxXAndY.y);

// Set the camera's position to the target position with the same z component.

transform.position = new Vector3(targetX, targetY, transform.position.z);

}

}