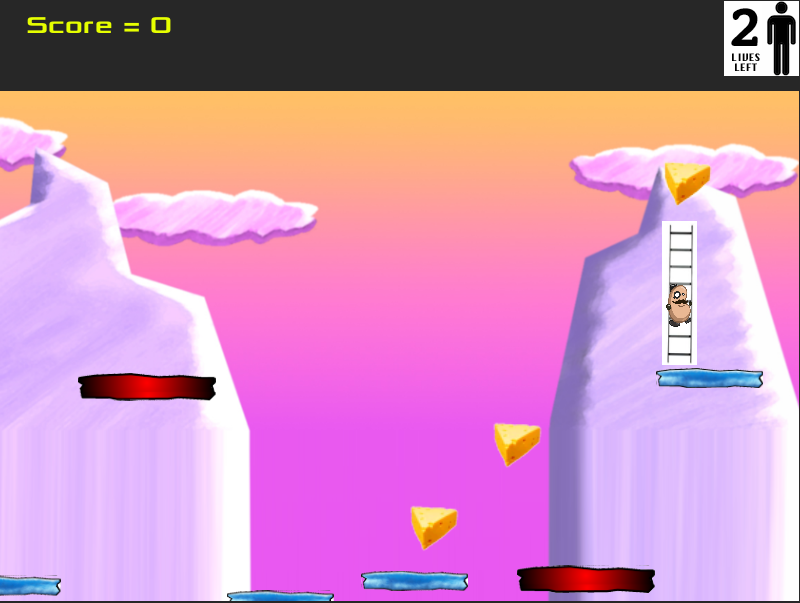
|  |  |
| --- | --- |
| ITB logo portrait B&W | INSTITUTE OF TECHNOLOGY BLANCHARDSTOWN |

A Taster of Computing

[[VERSION – Unity 2D – C# language]]

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

Part 8 – respawning / ladders / lives images and more…



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 Let the player WIN the game when all cheese eaten! 3

3 Stop guy jumping out of screen 5

4 Icons instead of numbers – for lives left … 6

5 Use a ‘Respawn’ object to determine restart position when lose a life 9

6 Choose from one of several different ‘Respawn’ game objects 11

7 Remove all left-right/up-down movement after respawning 12

8 Add ladders to your game 13

9 FULL LISTINGS 14

# 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:

* Game WON when all cheese items eaten
* Stop Hero guy being able to jump ‘off the screen’
* Display icons instead of text for lives
* Control where player is respawned through the use of invisible objects tagged ‘Respawn’
* Remove left/right/up/down movement after respawning
* Add ladders and ladder-climbing ability to our hero

# Let the player WIN the game when all cheese eaten!

## Count the number of pieces of cheese (if zero, game is won !)

To complete a level / game, we need to eat all the pieces tagged “Food”

## Add method to count objects in the scene with a given string ‘tag’

Add the following method to the **Player.cs** script:

private int CountObjectsWithTag(string tag)

{

GameObject[] foodObjects = GameObject.FindGameObjectsWithTag(tag);

return foodObjects.Length;

}

## Add method to load “gameWon” scene if no objects tagged “Food” remain

Add the following method to the **Player.cs** script:

**private void CheckGameWon()**

**{**

**int numFoodObjects = CountObjectsWithTag("Food");**

**print ("number of food objects left = " + numFoodObjects);**

**if(numFoodObjects < 1)**

**{**

*// make sure you spell this correctly!*

**Application.LoadLevel("scene3\_GameWon");**

**}**

**}**

## Create a gameWon scene, and add it to your game’s Build list

Create a new scene named **scene3\_GameWon**, with an appropriate message.

Just as you did for the game over scene .

In fact – just copy your game over scene, and change the text there to Game Won ☺

And ensure you **add this new scene to your game’s Build list**.

## Edit Update() method, so each frame our first action is to see if game won or deathY reached

Edit the Update() method of the **Player.cs** script, so that its first statement is to call method CheckGameWon(), and its second statement calls a new method to check if death Y-value has been reached. Since we are now attempted to do 2 tasks in Update(), we need to encapsulate the death-Y logic into its own method to keep things tidy:

**void Update()**

**{**

**CheckGameWon();**

**CheckDeathYReached();**

**}**

**private void CheckDeathYReached()**

**{**

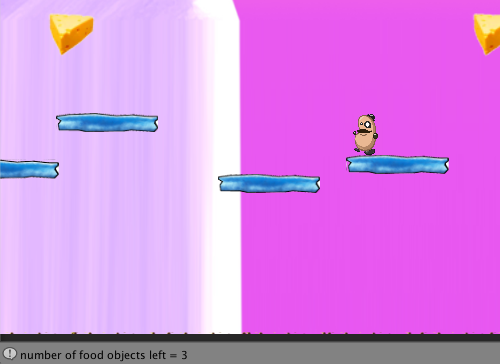
**float y = transform.position.y;**

**if(y < deathY){**

**LoseLife();**

**}**

**}**

****

## Playtest your game

Run your game. You should see a text message showing number of “Feed” objects remaining in the status bar at the bottom of the Unity application widow.



# Stop guy jumping out of screen

## The problem

Although the camera never moves to look past the edge of the background image, the hero-guy can still jump up/left/right off-screen.

## The solution: objects with Box Colliders preventing move off screen

An object with a Box Collider 2D (that does NOT have ‘**Is Trigger’** ticked), becomes a rectangle of the screen that the hero-guy is not allowed to enter.

So all we have to do is create some rectangles just off-screen (LEFT, RIGHT, and if necessary TOP), so the hero-guy cannot move off screen.

The simplest way to position items with box colliders is if they have some visual image associated with them, so either add a new image into your sprites folder to represent ‘border’ areas our hero cannot enter, or just select an image you already have – I used the blue platform image. Do the following:

* Drag from **Project/Sprites** the image **platform\_blue** into the scene
* Rename this gameObject in the **Hierarchy** to ‘**boundary’**
* Add a collider 2D (do NOT tick ‘**Is Trigger’**)
* Stretch this to be a tall thin shape, and position at the LEFT of the screen, touching the left edge of the background image
* Duplicate this gameObject and move its clone to the RIGHT side of the screen
* Duplicate this gameObject, stretch to be wide and short and move to the TOP of the screen

Our hero character should now be prevented from ‘jumping off screen’ by the colliders in these objects. Since objects created like have a default **Sorting Layer** setting of ‘**Default’** in their **Sprite Renderers**, and in our **Sorting Layers** this is first in the list – it means that these boundary objects will be BEHIND Sorting Layer **Background**, so the user should never see them if we have set out camera move limits correctly.



# Icons instead of numbers – for lives left …

## Wherever possible, display graphical icons to user, instead of numbers

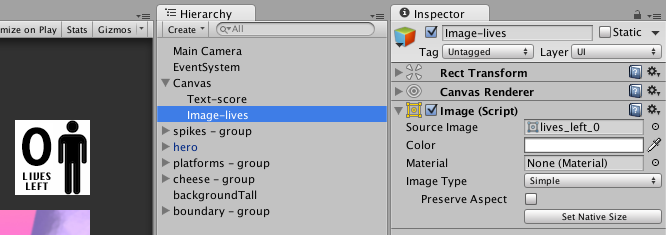
The number of lives left can be easily displayed to the user with graphics, e.g.:

   and so on.

## Edit your PlayerDisplay.cs script class, to display images for lives left

Do the following:

* Copy/create your lives left images into your **Project/Sprites** folder
  + Ensure you have images corresponding to EVERY POSSIBLE value of the livesLeft variable
  + (a good rule of thumb is to already create 1 or 2 more than you think you need, since it’s very quick to create 6 instead of 4 at the time, but if you create 4 and need 6, then you’ll have to move out of game mode and startup Photoshop/GIMP to create the extra 2 images – all very annoying …)
* In the **Canvas** in the **Hierarchy** delete gameObject **Text-lives**
* In its place create a new UI Image, named **Image-lives**
  + Drag Sprite lives\_left\_0 into the Source Image property of Image-lives Image (Script) component – and then click Set Native Size, so the UI Image on screen is resized to match the pixel-size of this Spriate
  + And position this top-right



* in class **PlayerDisplay** replace the **public Text livesText** variable for a public Image livesImage variable – i.e. we create a reference to our Image-lives UI Image object, so that our methods can change the Sprite image displayed to match the number of lives left

public class PlayerDisplay : MonoBehaviour {

public Text scoreText;

public Image livesImage;

* in class **PlayerDisplay** you need to declare **public Texture2D** variables for each image you want to display
  + by being public they can be assigned by drag-and-drop in the Inspector
  + name and order them carefully to avoid making mistakes later
  + e.g.

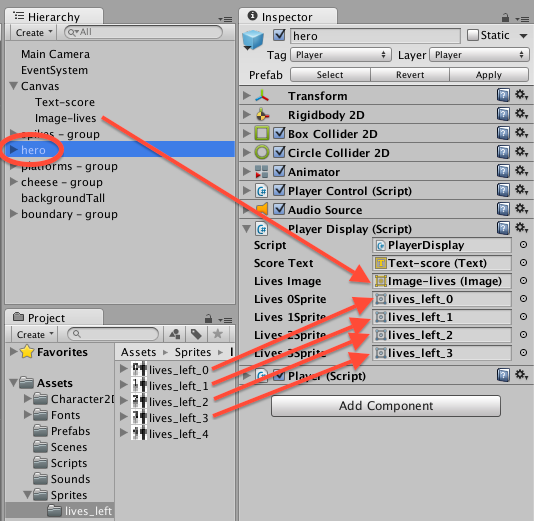
public Sprite lives0Sprite;

public Sprite lives1Sprite;

public Sprite lives2Sprite;

public Sprite lives3Sprite;

* + save your script, and drag the images over these variables of the Main Camera’s GameGUI script component in the Inspector:



Now in class PlayerDisplay replace method UpdateLivesText(…) with this new UpdateLivesImage(…) method:

**public void UpdateLivesImage(int newLives){**

**switch(newLives)**

**{**

**case 3:**

**livesImage.sprite = lives3Sprite;**

**break;**

**case 2:**

**livesImage.sprite = lives2Sprite;**

**break;**

**case 1:**

**livesImage.sprite = lives1Sprite;**

**break;**

**case 0:**

**default:**

**livesImage.sprite = lives0Sprite;**

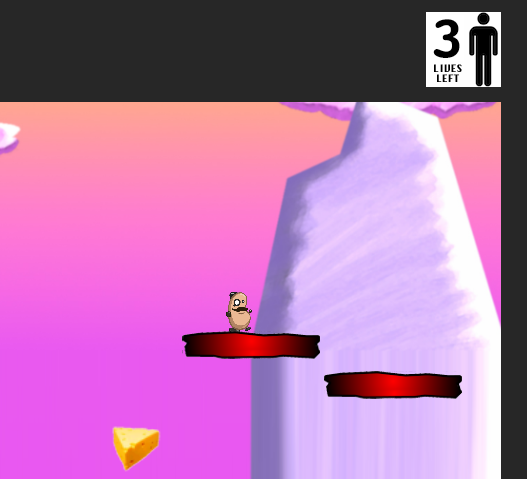
**break;**

**}**

**}**

Finally, in class Player you need to replace the 2 calls to **UpdateLivesText**(…) with calls to **UpdateLivesImage**(…).

When you play the game you should see the lives icons displayed, instead of the String text messages …



# Use a ‘Respawn’ object to determine restart position when lose a life

## Avoid ‘hard coding’ values into your game

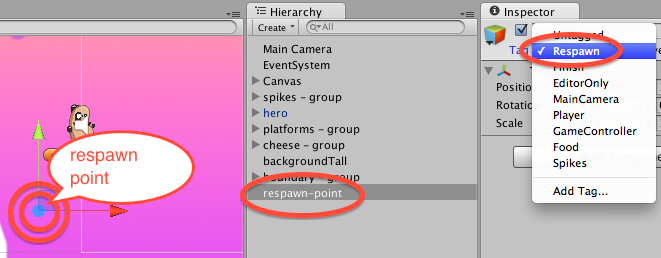
At present the **Player.cs** method MoveToStartPosition() uses a ‘hard coded’ start position of (0,5,0).

This is bad practice – we should dynamically, at run time, use the position of an object tagged ‘Respawn’ to decide where to move the player to when a life is lost. And perhaps we’ll choose from a selection, or if the player has passed some ‘checkpoint’ in the game, the set of respawn points is half-way through the game when a life is lost etc.

## Create new empty GameObject and tag it ‘Respawn’

Do the following:

* Create a new empty GameObject named ‘respawn-point’
  + menu: GameObject | Create Empty
* Move it to where you want the hero-guy to be moved to when the lose a life
* Tag this gameObject “Respawn”
  + It’s one of the pre-written tags Unity provides, since its use is so common



NOTE – if you don't like invisible objects, then choose a Sprite image and drag it onto the Scene, naming and tagging it appropriately – and remember to un-tick its Sprite Renderer component before saving the scene (otherwise this respawn point will be visible to the user when playing the game …

## Refactor (improve) method MoveToStartPosition() to use of the respawn GO

Replace method MoveToStartPosition() in **Player.cs** with the following code:

**private void MoveToStartPosition()**

**{**

**GameObject respawnGO = GameObject.FindWithTag("Respawn");**

**Vector3 startPosition = respawnGO.transform.position;**

**transform.position = startPosition;**

**}**

NOTE – the abbreviated acronym “GO” is often used in Unity documentation to refer to a GameObject (GO).

## Playtest your game

When you lose a life your hero-guy should be moved to wherever you place your respawn GO.

Try moving the respawn GO to another part of the screen, and run it again – when the player loses a life they should be ‘transported’ to the location of your **respawn-point** gameObject.

# Choose from one of several different ‘Respawn’ game objects

## Write method to choose a random tagged gameObject

Add the following method to **Player.cs**:

**private GameObject ChooseRandomObjectWithTag(string tag)**

**{**

**GameObject[] taggedObjects =**

**GameObject.FindGameObjectsWithTag(tag);**

**int numTaggedObjects = taggedObjects.Length;**

**int randomIndex = Random.Range(0, numTaggedObjects);**

**return taggedObjects[randomIndex];**

**}**

GameObject.FindWithTag() works when we know there is only going to be **one** gameObject in the scene with that tag. When there may be more, we need to use FindGameObjectsWithTag() which returns an ARRAY of GameObjects.

The method above accepts a string parameter, gets an array of all gameObjects in the scene with that tag, and then uses the Random.Range(<min>,<max>) to choose a random location within the array.

The object corresponding to that location in the array is then returned by the method.

NOTE: Range() will never return <max> as long as <min> and <max> are different. So this method is PERFECT for choosing items from an array using 0 and **array.Length** as parameters.

## Add 1 or 2 new “Respawn” empty game objects, at other locations in your scene

Duplicate and move 2 or 3 move copies of your respawn-point gameObject.

## Refactor method MoveToStartPosition() to use the new method

Change **Player.cs** to use our new method, so a random start position is chosen:

**private void MoveToStartPosition()**

**{**

**GameObject respawnGO = ChooseRandomObjectWithTag("Respawn");**

**Vector3 startPosition = respawnGO.transform.position;**

**transform.position = startPosition;**

**}**

## Playtest your game

You may want to increase the number of lives for testing. Play the game, and keep losing lives on purpose. After losing a life the restart position should randomly be one of your respawn GOs.

# Remove all left-right/up-down movement after respawning

## Set velocity to zero (0,0) to prevent movement after respawning

When a player’s hero character is respawned we do NOT want him to be still moving left / right, or falling DOWN very fast due to gravity. We want the hero character to ‘appear’ at a respawn point, not moving at all. After a very short delay gravity will slowly kick in, and he will start falling straight DOWN from that respawn point.

The way to achieve this is very simple, we need to zero the velocity vector of the physics component Rigid Body 2D. Luckily we are given access to a public property ‘velocity’ in this component that we can set with a single line of code. So add an extra statement to the end of our MoveToStartPosition() method as follows:

Add the following statement to the end of method to **Player.cs**:

*private void MoveToStartPosition()*

*{*

*GameObject respawnGO = ChooseRandomObjectWithTag("Respawn");*

*Vector3 startPosition = respawnGO.transform.position;*

*transform.position = startPosition;*

**// remove all horizontal/vertical movement when respawned**

**rigidbody2D.velocity = Vector2.zero;**

*}*

## Playtest your game

Now after respawning the hero character should ‘float’ stationary for a fraction of a second, and then start falling straight down from the point at which they were respawned.

# Add ladders to your game

## Do the following

Add ladder climbing ability to our hero charcter as follows:

* Copy script **PlayerControlLadder.cs** into your **Scripts** folder
* Select hero in the Hierarchy, delete the **PlayerControl** scripted component, and drag an instance of **PlayerControlLadder** into the Inspector

Now let’s add a ladder into the scene:

* Copy **ladder.png** into your **Sprites** folder
* drag the ladder sprite onto the scene (and position/size) next to a platform
* tag the ladder gameObject 'Ladder' (you'll need to add this as a new tag)
* set the Sorting Layer of this ladder gameObject to ‘Ladder’ (you’ll need to add this as a new Sorting Layer BEFORE the Character layer, so the ladders will be BEHIND the hero character)
* set the Layer of the ladder object to ‘Ground’ (so the player can JUMP off the top of ladders)
* add a Box Collider 2D and tick its ‘Is Trigger’ property

Your character should now be able to climb up and down the ladder, and jump off the top !

## Create a reusable prefab so adding more ladders is easy …

* create an empty prefab named 'ladder'
* drag the ladder gameObject into it
* and now you can add more ladders to your game very easily ...

### Playtest your game

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

You should now respawn to one of several respawn locations, have lives displayed by images, and have ladders that the character can climb!

**Congratulations**

**You have now created part 8 of the tutorial !**

# FULL LISTINGS

## Player

using UnityEngine;

using System.Collections;

public class Player : MonoBehaviour {

public AudioClip yumSound;

public AudioClip dieSound;

private PlayerDisplay playerDisplay;

private int lives = 3;

private int score = 0;

private float deathY = -15;

//-----------------------------

void Start()

{

playerDisplay = GetComponent<PlayerDisplay>();

playerDisplay.UpdateScoreText(score);

playerDisplay.UpdateLivesImage(lives);

}

//-----------------------------

void Update()

{

CheckGameWon();

CheckDeathYReached();

}

//-----------------------------

private void CheckDeathYReached()

{

float y = transform.position.y;

if(y < deathY){

LoseLife();

}

}

//-----------------------------

private void LoseLife()

{

lives--;

if(lives < 0)

{

Application.LoadLevel("scene1\_GameOver");

}

playerDisplay.UpdateLivesImage(lives);

MoveToStartPosition();

audio.PlayOneShot(dieSound);

}

//-----------------------------

private void MoveToStartPosition()

{

GameObject respawnGO = ChooseRandomObjectWithTag("Respawn");

Vector3 startPosition = respawnGO.transform.position;

transform.position = startPosition;

// remove all horizontal/vertical movement when respawned

rigidbody2D.velocity = Vector2.zero;

}

//-----------------------------

private GameObject ChooseRandomObjectWithTag(string tag)

{

GameObject[] taggedObjects = GameObject.FindGameObjectsWithTag(tag);

int numTaggedObjects = taggedObjects.Length;

int randomIndex = Random.Range(0, numTaggedObjects);

return taggedObjects[randomIndex];

}

//-----------------------------

void OnTriggerEnter2D(Collider2D hit)

{

if(hit.CompareTag("Food"))

{

score++;

playerDisplay.UpdateScoreText(score);

Destroy (hit.gameObject);

audio.PlayOneShot(yumSound);

}

if(hit.CompareTag("Spikes"))

{

LoseLife();

}

}

//------------------------------

private int CountObjectsWithTag(string tag)

{

GameObject[] foodObjects = GameObject.FindGameObjectsWithTag(tag);

return foodObjects.Length;

}

//-------------------------------

private void CheckGameWon()

{

print ("hello");

int numFoodObjects = CountObjectsWithTag("Food");

print ("number of food objects left = " + numFoodObjects);

if(numFoodObjects < 1)

{

Application.LoadLevel("scene3\_GameWon");

}

}

}

## PlayerDisplay

using UnityEngine;

using System.Collections;

using UnityEngine.UI;

public class PlayerDisplay : MonoBehaviour {

public Text scoreText;

public Image livesImage;

public Sprite lives0Sprite;

public Sprite lives1Sprite;

public Sprite lives2Sprite;

public Sprite lives3Sprite;

//-------------------------------

public void UpdateScoreText(int newScore){

string scoreMessage = "Score = " + newScore;

scoreText.text = scoreMessage;

}

//-------------------------------

public void UpdateLivesImage(int newLives){

switch(newLives)

{

case 3:

livesImage.sprite = lives3Sprite;

break;

case 2:

livesImage.sprite = lives2Sprite;

break;

case 1:

livesImage.sprite = lives1Sprite;

break;

case 0:

default:

livesImage.sprite = lives0Sprite;

break;

}

}

}

## PlayerControlLadder

using UnityEngine;

using System.Collections;

public class PlayerControlLadder : MonoBehaviour

{

[HideInInspector]

public bool facingRight = true; // For determining which way the player is currently facing.

[HideInInspector]

public bool jump = false; // Condition for whether the player should jump.

public float climbSpeed = 10;

private bool isClimbing = false;

public float moveForce = 365f; // Amount of force added to move the player left and right.

public float maxSpeed = 5f; // The fastest the player can travel in the x axis.

public AudioClip[] jumpClips; // Array of clips for when the player jumps.

public float jumpForce = 1000f; // Amount of force added when the player jumps.

public float tauntProbability = 50f; // Chance of a taunt happening.

public float tauntDelay = 1f; // Delay for when the taunt should happen.

private int tauntIndex; // The index of the taunts array indicating the most recent taunt.

private Transform groundCheck; // A position marking where to check if the player is grounded.

private bool grounded = false; // Whether or not the player is grounded.

private Animator anim; // Reference to the player's animator component.

private float ladderX;

void Awake()

{

// Setting up references.

groundCheck = transform.Find("groundCheck");

anim = GetComponent<Animator>();

}

void Update()

{

// The player is grounded if a linecast to the groundcheck position hits anything on the ground layer.

grounded = Physics2D.Linecast(transform.position, groundCheck.position, 1 << LayerMask.NameToLayer("Ground"));

// If the jump button is pressed and the player is grounded then the player should jump.

if(Input.GetButtonDown("Jump") && grounded)

jump = true;

}

void FixedUpdate ()

{

if (isClimbing) {

ClimbingActions ();

}

WalkingActions ();

}

private void ClimbingActions()

{

float v = Input.GetAxis ("Vertical");

Vector2 moveDirection = new Vector2 (0, v);

moveDirection = transform.TransformDirection (moveDirection);

moveDirection \*= climbSpeed;

rigidbody2D.velocity = (moveDirection \* Time.deltaTime \* climbSpeed);

// set X value to ladder X - so in middle of ladder (leave Y and Z alone)

// (only IF moving verically)

if( v != 0)

{

Vector3 newPosition = new Vector3(ladderX, transform.position.y, transform.position.z);

transform.position = newPosition;

}

}

private void WalkingActions()

{

// Cache the horizontal input.

float h = Input.GetAxis("Horizontal");

// The Speed animator parameter is set to the absolute value of the horizontal input.

anim.SetFloat("Speed", Mathf.Abs(h));

// If the player is changing direction (h has a different sign to velocity.x) or hasn't reached maxSpeed yet...

if(h \* rigidbody2D.velocity.x < maxSpeed)

// ... add a force to the player.

rigidbody2D.AddForce(Vector2.right \* h \* moveForce);

// If the player's horizontal velocity is greater than the maxSpeed...

if(Mathf.Abs(rigidbody2D.velocity.x) > maxSpeed)

// ... set the player's velocity to the maxSpeed in the x axis.

rigidbody2D.velocity = new Vector2(Mathf.Sign(rigidbody2D.velocity.x) \* maxSpeed, rigidbody2D.velocity.y);

// If the input is moving the player right and the player is facing left...

if(h > 0 && !facingRight)

// ... flip the player.

Flip();

// Otherwise if the input is moving the player left and the player is facing right...

else if(h < 0 && facingRight)

// ... flip the player.

Flip();

// If the player should jump...

if(jump)

{

// Set the Jump animator trigger parameter.

anim.SetTrigger("Jump");

// Add a vertical force to the player.

rigidbody2D.AddForce(new Vector2(0f, jumpForce));

// Make sure the player can't jump again until the jump conditions from Update are satisfied.

jump = false;

}

}

void Flip ()

{

// Switch the way the player is labelled as facing.

facingRight = !facingRight;

// Multiply the player's x local scale by -1.

Vector3 theScale = transform.localScale;

theScale.x \*= -1;

transform.localScale = theScale;

}

private void OnTriggerEnter2D(Collider2D c)

{

if (c.CompareTag ("Ladder"))

{

isClimbing = true;

ladderX = c.transform.position.x;

}

}

private void OnTriggerExit2D(Collider2D c)

{

if (c.CompareTag ("Ladder"))

{

isClimbing = false;

}

}

}