INSTITUTE OF TECHNOLOGY
BLANCHARDSTOWN

Institute of Technology
Blanchardstown
Institiúid Teicneolaíochta
Baile Bhlainséir

**A Taster of Computing**
**[[VERSION – Unity 2D – C# language]]**

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

## Part 4 – ideas for improving the game …

## CONTENTS

# 1 Solves the multiple deaths when hits spikes objects problem

An issue with the previous method of dying when you hit 'spikes' is that you may hit multiple spike objects, and therefore lose more than one life, before the character is repositioned back to its start position.

The solution – only have a single gameObject tagged 'Spikes' (containing lots of untagged images):

- So it looks like lots of spikes, but there will only ever be one collision

## 1.1 Follow these steps

Preparation:

- remove all spikes / spikes group gameObjects from the scene
- delete your Spikes prefab (so all you have is the spikes image in the Sprites folder

Setup:

- position lots of Spikes sprite images along the bottom of the background
  - these are just 2D images – no collider, not tag
- create a new empty gameObject (named 'spikes-group')
- child all your spikes into that game object
  - i.e. drag each spikes image gameObject into spikes-group, so it becomes a child object of spikes-group
- tag your spikes-group 'Spikes'
- add to spikes-group a Box Collider 2D and tick its 'Is Trigger'
- size and position the collider in spikes-group, so that it stretches the whole width of the background, and that its top is in line with the spikes images

That's it – now you should only ever have a single collision with something tagged 'Spikes'

## 1.2 Alternative solution

An alternative would be to just have lots of untagged spikes sprites, and use the previous DEATH_Y y-position test to make the player lose a live and be repositioned when that Y position is hit.

There is no need for the empty gameObject containing the collider in this case, but it means the DEATH_Y value is 'hidden' in code, rather than an on-screen object that visually shows you where the action happens …

## 1.3 If you don't like 'invisible' game objects …

If you don't like invisible game objects with colliders, just use rectangular image as a sprite.

- Drag the sprite onto the scene, so it becomes a gameObject
- Rename it appropriately
- Resize it as required
- Add the collider
- Then un-tick it's renderer component, so the user cannot see the image

# 2 Stop guy jumping out of screen

## 2.1 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.

## 2.2 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.

Do the following

- create a new empty gameObject (e.g. named 'boundary')
    - or use any rectangular sprite image as the basis for your new gameObject
- add to the object a Box Collider 2D
    - do NOT tick its 'Is Trigger'
- size and position the gameObject / collider, so that the collider touches the edge of the game background
    - so the player can no longer move off screen because they'll hit the collider of this gameObject

# 3 Add ladders to your game

## 3.1 Do the following

Add ladders to your game as follows:

- replace the hero **PlayerControl** scripted component with the **PlayerControlLadder.cs**
- drag the ladder sprite onto the scene (and position/size)
- tag the ladder gameObject 'Ladder' (you'll need to add this as a new tag)
- add a Box Collider 2D and tick its ‘Is Trigger’ property

Your character should now be able to climb up and down the ladder.

## 3.2 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 ...

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

## 4.1 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 dynamicnally, 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.

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

## 4.3 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;
}
```

## 4.4 Playtest your game

That’s it!

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.

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

## 5.1 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 parameters, 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.

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

Either create some new empty objects and tag them “Respawn”, or just duplicate and move your previous respawn object.

## 5.3 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;
}
```

## 5.4 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.

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

## 6.1 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”

## 6.2 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;
}
```

## 6.3 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)
	{
		Application.LoadLevel("gameWon");
	}
}
```

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

Create a new scene named gameWon, with an appropriate message, and add it to your game’s Build list.
Just as you did for the gameOver scene .

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

Edit the Update() method of the **Player.cs** script, so that its first statement is to call method CheckGameWon():

```
private void Update()
{
	CheckGameWon();

	// other actions we want to do each frame e.g.:
	CheckGameOver();
	CheckFireKey();
}
```

## 6.6 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.

# 7 Animate a sprite with multiple same-sized images

## 7.1 Simulate an animation effected by looping through images for a sprite

Preparation:

- have your set of equally sized 2D image sprites in your Project Sprites folder
    - make sure they are all Sprites rather than Textures

Do the following:

- drag first sprite from Project-Sprites into scene, to make a gameObject
- add an instance of the **AnimatedSprite.cs** script as a component of the gameObject
- set size of Sprite Array to 2 or 3 or whatever
- drag your sprites from the Project panel in the array Elements (0, 1, 2) in sequence to animate
- set the Frame Interval to speed things up / slow them down

# 8 Icons instead of numbers – for lives left …

## 8.1 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.

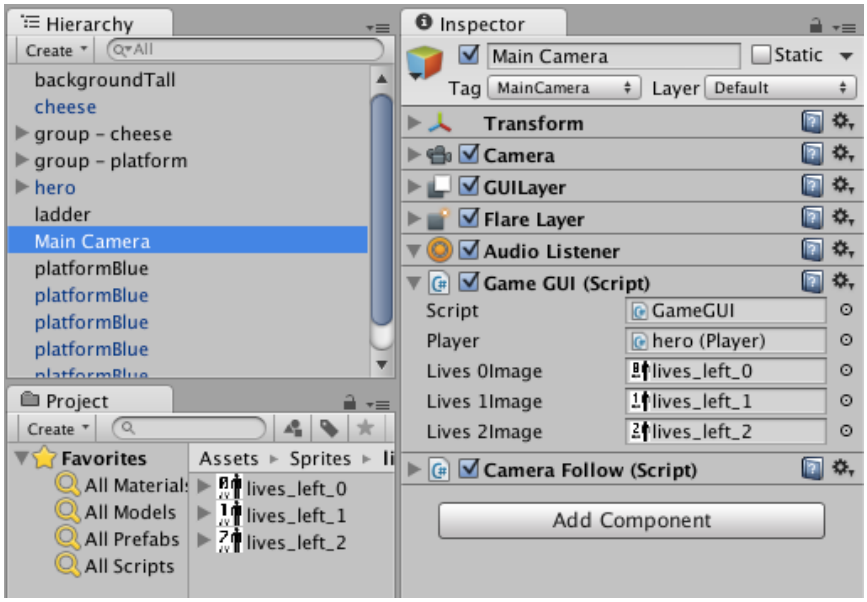
## 8.2 Edit your GameGUI.cs script, to display images for lives left

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 your **GameGUI.cs** script 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 Texture2D lives0Image;
public Texture2D lives1Image;
public Texture2D lives2Image;
```

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



Finally, change the code in your GameGUI DisplayLives() method to the following:

```
	private void DisplayLives()
	{
		int playerLives = player.GetLives();
		if(0==playerLives)
		{
			GUILayout.Label(lives0Image);
		}

		if(1==playerLives)
		{
			GUILayout.Label(lives1Image);
		}

		if(2==playerLives)
		{
			GUILayout.Label(lives2Image);
		}
	}
```

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

# 9 Take over part of the screen, just for HUD (Head Up Display)

## 9.1 Default is gameplay covers all of Game window

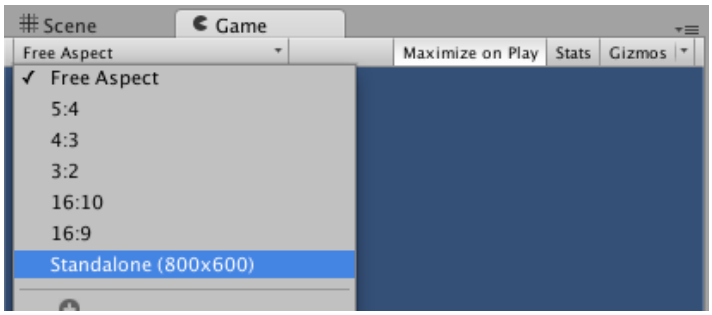
The default setting for the Main Camera, is that is should display the game player over the whole area of the Game window when the game is running.

But – why not have a dedicated part of the Game window just for HUD messages for the user, such as score, lives left, inventory, health bars, time remaining etc.

3 steps:

- Know the size of your game window
  - You should have set / chosen a Standalone game resolution by now
    - The gravity guy defaults are 800 x 600 pixels (width x height)
    - You can change them via menu:
      - File | Build Settings … | button “Player Settings …” | Inspector **Resolution**
- Know how much screen ‘real estate’ you want for your HUD – and where
  - E.g. if want 60 pixels across the top of the screen for your HUD, then you need to reduce the size of the Main Camera coverage by 10%
- Use GUI rather than GUILayout for display of strings and images
  - And define the coordinate rectangles appropriately for size/position of GUI elements

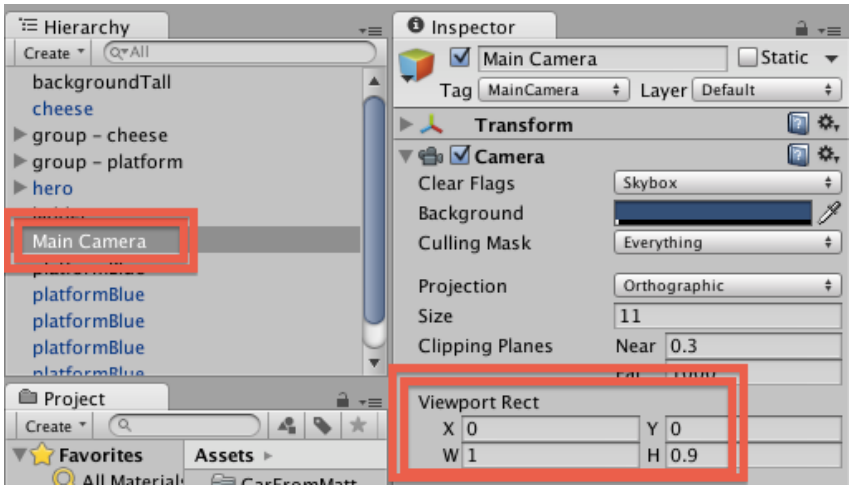
## 9.2 Ensure game window resolution set to 800 x 600 (or your own dimensions)




## 9.3 Reduce Main Camera coverage of Game window

Reduce the camera height coverage to 90%:

- In the **Hierarchy** select the **Main Camera**
- In the **Inspector** for the **Camera** component, set the H (height) property of Viewport Rect to 0.9




What does this do:

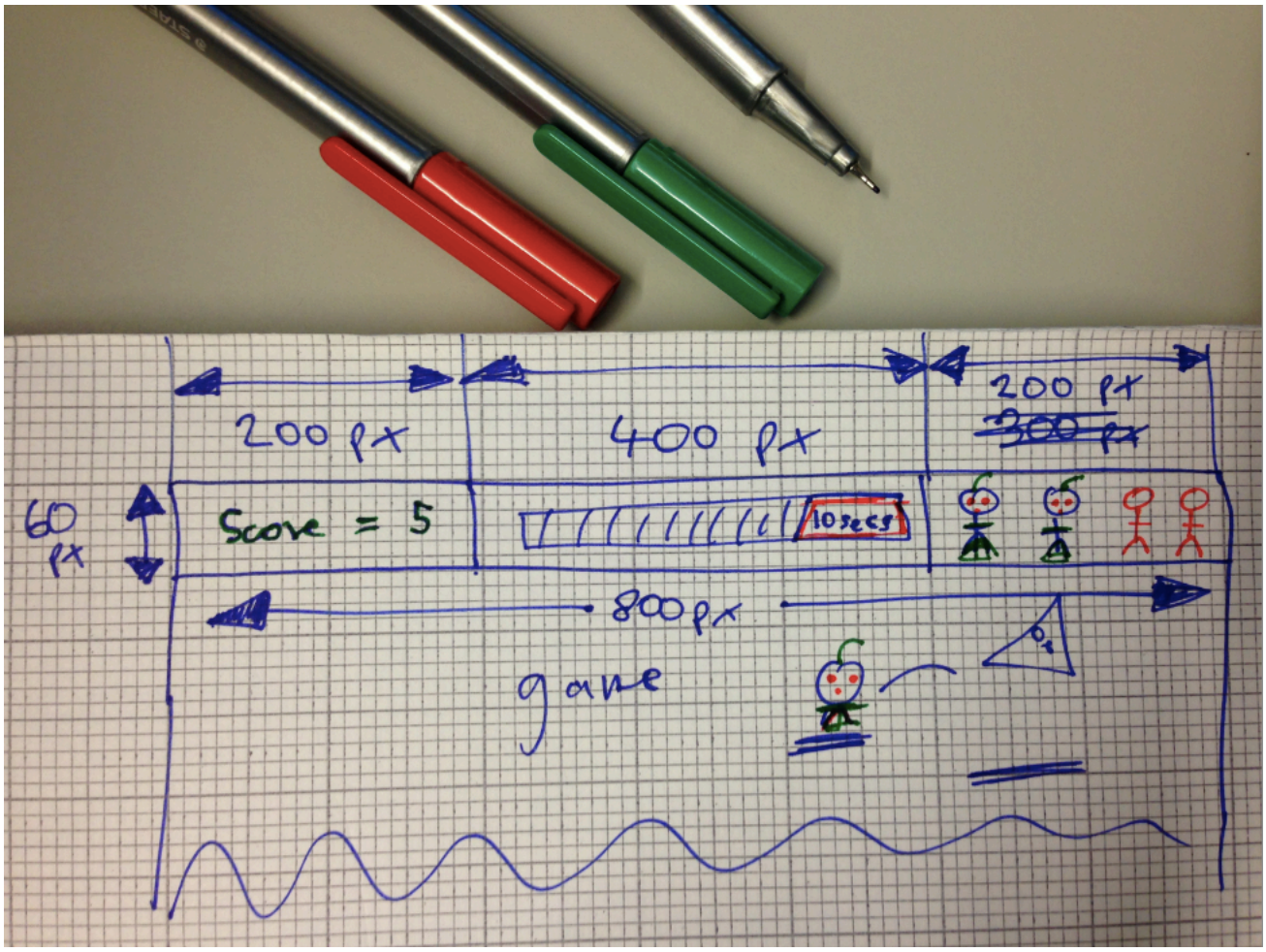
- The ‘viewport’ is how much of the game window the camera contents take up
- The X and Y are where its rectangle starts
  - (0, 0) means bottom left
- The W and H are the width and height, in terms of values between 0.00 and 1.00
  - So height 0.9 means 0.9 * width of 600 pixels = 540 pixels
- So there will be a ‘gap’ of 60 pixels across the top of the Game window when running
  - This is where we can locate our game HUD display of score, lives etc.
- NOTE: to have our HUD at the bottom of the game window, just set Y = 0.1, and the 60 pixel gap will be at the bottom …

## 9.4 Move away from reliance on GUILayout to decide where to put GUI elements

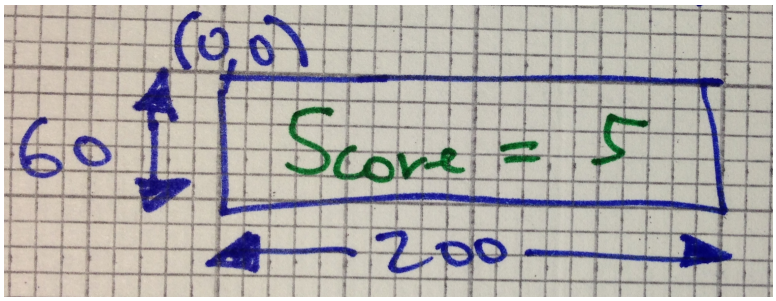
GUILayout uses a similar layout strategy to a Web browser

- Start at top left
- For each new item, start a new line / row

While it is possible to override these settings (e.g. to display thing left to right, rather than top-down), for the sake of a few minutes with a pencil and paper you can have EXACT control of where to place your GUI elements:

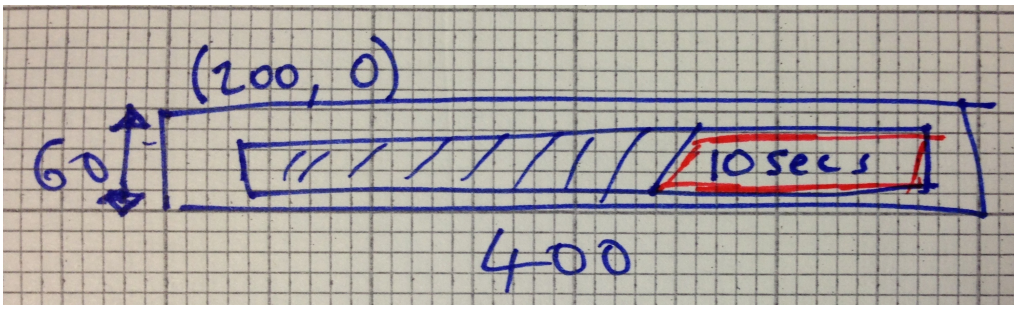



So we get the following for score:



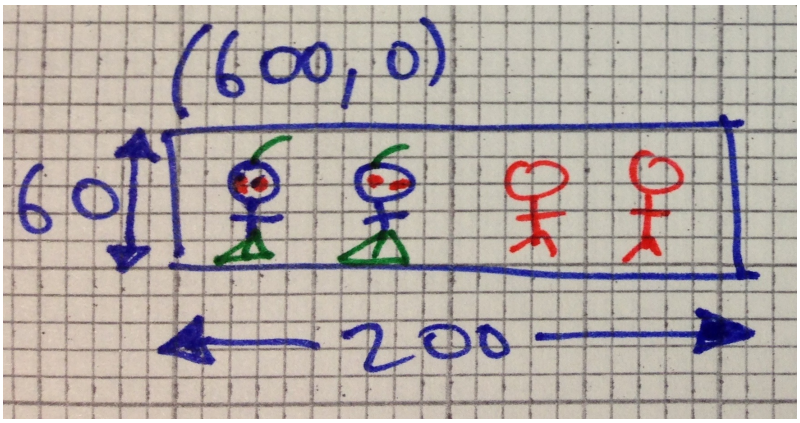

```
Rect scoreRect = new Rect(0,0,200,60);
GUI.Label(scoreRect, scoreMessage);
```

For time left:




```
Rect timeRect = new Rect(200,0,400,60);
GUI.Label(timeRect, timeImage);
```

For lives left:




```
Rect livesRect = new Rect(600,0,200,60);
GUI.Label(livesRect, scoreMessage);
```

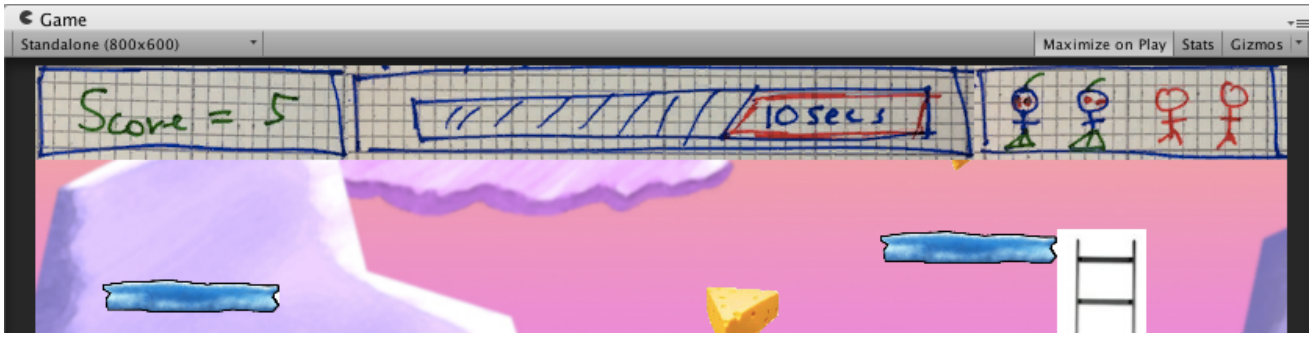
## 9.5 Stop press – need to give extra height – bug in Unity ?

When I tried to replicate the above by pixel-sizing my mockups with 200x60 and 400x60 images, they were getting scaled DOWN for some reason in Unity.

My solution was to make sure the WIDTH and HEIGHT were the same (whichever is smaller, make that equal to the larger value). Since there is clipping of the Label rectangle, if the image is larger than the rectangle then the overflow is clipped (not seen). But without changing these dimensions Unity kept on scaling down my images – so my final rectangles (that worked) were actually:

```
Rect scoreRect = new Rect(0,0,200, 200 ); // so square 200 x 200
Rect timeRect = new Rect(200,0,400, 400 ); // so square 400 x 400
Rect livesRect = new Rect(600,0,200, 200 ); // so square 200 x 200
```

Here is screenshot proof I got it working:



# 10 Improve death – temporary invulnerability !

## 10.1 Multiple lost life problem

One problem with the combination of **frame-based methods** (e.g. Update(), FixedUpdate()) and e**vent-based methods** (OnTriggerEnter2D(), OnGUI()), and the Unity **physics** system, is that somethings things happen in odd or unplanned combinations.

This can lead, for example, to losing multiple lives, because a character is not moved back to the start position before hitting a second object etc.

One robust solution, is to ensure certain actions cannot happen again within a certain 'safety' time period, e.g. 0.5 seconds. In some games this temporary invulnerability is visually shown to the user (e.g. when you get a new space ship in Asteroids, for a second or two you see a force field circle around the new ship, and you won't become vulnerable again until there is a clear area around you.

Many times in computer games, a simple solution is to compare the current time in the game (Time.time) with the next time something is allowed to happen (e.g. nextTimeAllowedToDie). When an event happens, e.g. we lose a life, we also reset the next time for that event to happen to be the current time plus our delay, e.g.:

```
nextTimeAllowedToDie = Time.time + delayBetweenDeaths;
```

So lets add this feature to our game, so we stop that annoying, losing 2 lives when you hit the spikes problem …

> **NOTE: Time.time**
>
> In Unity our game timer counts the number of seconds (float) since the application started running. So after 1-and-a-half seconds, Time.time will be 1.5 and so on ….

## 10.2 Declare variables for nextTimeAllowedToDie and delayBetweenDeaths

Add the following variables to your **Player.cs** class:

```
// public – so can change in Inspector
public float delayBetweenDeaths = 0.5f;

// for private method use only …
private float nextTimeAllowedToDie = 0;
```

Note – we initialise it to zero, so from as soon as the game starts (Time.time starts at zero) we are allowed to lose a life.

## 10.3 Refine our OnTriggerEnter2D method to only lose life is it's time to die …

Change yourOnTriggerEnter2D() method in **Player.cs** class to the following:

```
private void OnTriggerEnter2D(Collider2D c)
{
	string tag = c.tag;

	if("Food" == tag)
	{
		score++;
		audio.PlayOneShot(yumSound);
	}

	if("Spikes" == tag)
	{
		// only lose life if current time past next time to die
		if(Time.time > nextTimeAllowedToDie)
		{
			LoseLife();
		}
	}
}
```

And now write your LoseLife() method()

```
private void LoseLife()
{
	audio.PlayOneShot(dieSound);
	MoveToStartPosition();
	lives--;

	// update our next time allowed to die for a future time ...
	nextTimeAllowedToDie = Time.time + delayBetweenDeaths;
}
```