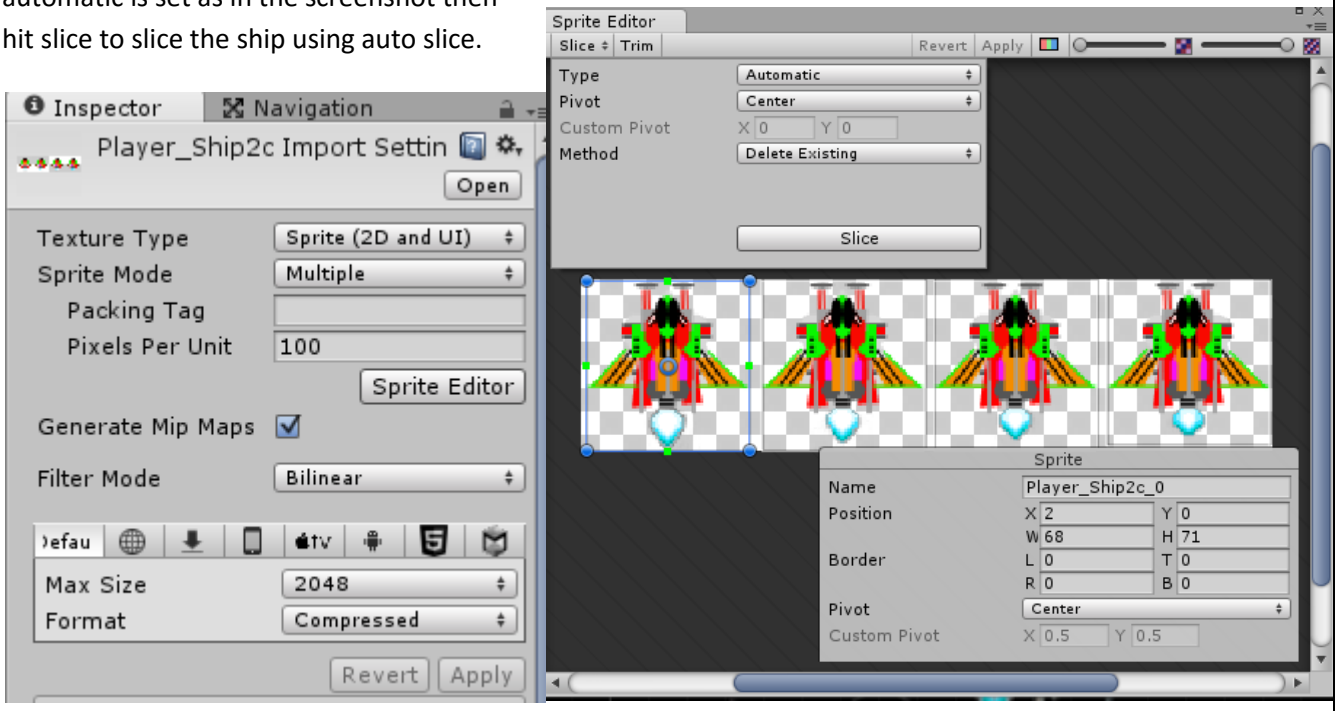
# Space Invadersesque 2D shooter

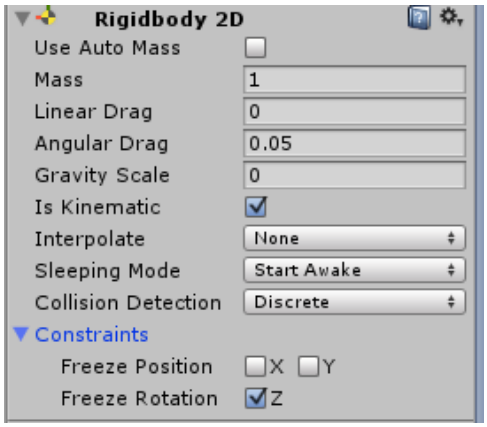
So, we're going to create another classic game here, one of space invaders, this assumes some basic 2D knowledge and is one in a beginning 2D game series of shorts. All in all, it should take a beginner no more than a few hours.

Start by creating a new 2D project and importing the assets from the site. Drag onto the scene the background image (Space texture curtesy of np4gamer, all images available on opengameart).

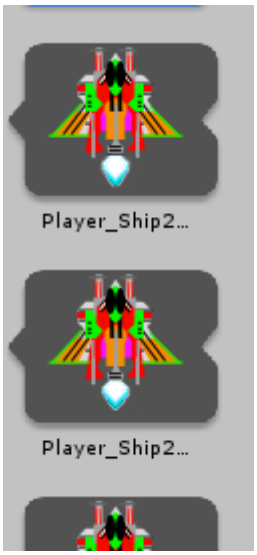
Now, find the ship, click on it in project view, this is a sprite sheet so we need to cut it up into several images; click and change sprite Mode to multiple, then, click sprite editor, slice and ensure that automatic is set as in the screenshot then hit slice to slice the ship using auto slice.




If we select all the images that have now been created (you may need to press the little arrow to expand) by ctrl clicking each, then drag them as one onto the unity hierarchy Unity should prompt you to save. It's creating a simple animation that just flicks between them, giving us a nice simple little effect, save it and continue.




You'll now have a gameObject on the hierarchy that's your player ship, add a Rigidbody2D component and tick the is kinematic and freeze Z constraint. A kinematic object will not be affected by collisions, forces (like gravity) or any other bit of physics, they are "non physical" objects. That's fine for our simple game and simple ship, we can directly influence the position of the ship using controls.



Helpfully, whilst colliders don't work, triggers do. We need something to check if we've been hit so we'll use triggers to check for them. Add a BoxCollider2D component and tick, "isTrigger".

We now create a new C# script, shipControl that we can use to, surprise surprise, control the ship, fill the following in.

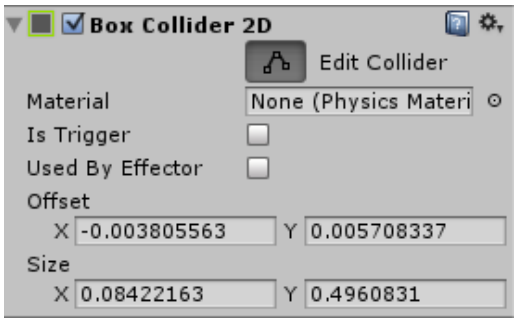
```
// Update is called once per frame
void Update () {
    GetComponent<Rigidbody2D>().velocity = new Vector2(Input.GetAxis("Horizontal") * 10, Input.GetAxis("Vertical") * 10);

}
```

It's a lazy all in one line solution, but effective- essentially we're getting the velocity component of the rigidbody (this holds speed and direction of travel) then setting it to a new Vector 2 representing the value of the button pressed for the in built horizontal or vertical axis (arrow keys by default). We times it by ten to get a faster movement, you might want to vary this.

Test the game. Testing is good.

Now for some pew pew, find and slice the beams image in the assets, in a way similar to above. This one we're not going to use as an animation, instead, choose the laser you want. I went with beams_51 for the enemy and beams_43 for the player's beams. Once you've chosen, drag the image (singular) you want onto the scene and then into a folder in your project called "Prefabs". You don't strictly need the folder but it helps- everything you do now to the objects in this folder will be applied to all others. You can see if they are prefabs, their name will be blue in the hierarchy. The Laser pack is by Rawdanitsu.

Add a BoxCollider2D to the laser beam, clicking edit collider to adjust it to be smaller better fitting to the laser shape and click IsTrigger. Now add a Rigidbody2D and once again make kinematic.




## Player beams

Create a new script in C# called shotScript, setup as below.

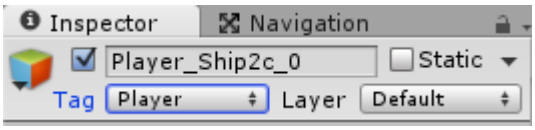
```
public int speed=5;
public int damage;

// Use this for initialization
void Start () {
    GetComponent<Rigidbody2D>().velocity = new Vector2(0, speed);
}

void OnBecameInvisible()
{
    Destroy(gameObject);
}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag != "Player")
    {
        Destroy(other.gameObject);
        Destroy(gameObject);
    }
}
```

Let's go through this, the first two lines setup global variables, these are values that are available across the whole script. In Start (remember, this is run whenever the script is first loaded) we set the velocity of the beam to 0 in the horizontal direction and "speed" in the vertical. This lets us easily change the speed of the beam later.

voidOnBecameInvisible() is a built in Unity function, it triggers whenever the player is no longer visible by the Camera (technically when it's no longer being rendered, there are occassions when this is isn't the same). When it's no longer visible, we destroy it (requires it to have been visible at one point).

OnTriggerEnter2D(Collider2D other) is another built in, it triggers whenever a collider enters a trigger. A message is sent to both the object with the trigger and the one triggering, a trigger is only triggered is one of the colliders also have a rigidbody attached to it. Here, we check if we've hit a player, if we haven't, we destroy the object we hit and the beam, no mercy here.

This script will be used for the player's beams, in order to recognise the player (so we don't accidentally hit ourselves) we use something called a tag, which is just a name attached to the object. We need to find the player object and, after clicking on it, tag it as Player by clicking the drop down next to Tag (Player is a pre existing tag).



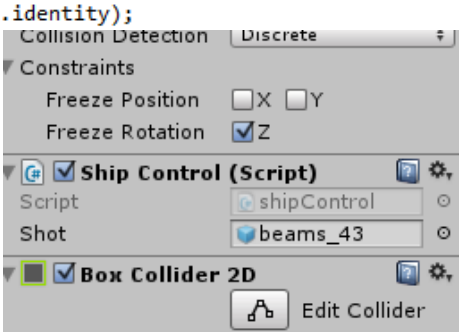

Attach your new beam script to the player beam prefab you created earlier. Drag the beam prefab onto the scene and test, they should fly forwards automatically now.

## Shooting back

Open shipControl script and create new variable "shot" which will be used to hold the GameObject that represents the beams. Below the GetComponent line from previous and in Update() we add a condition, "if the fire1 button is pressed, create a beam, at this objects position and straight on".

```
public GameObject shot;

// Use this for initialization
void Start () {

}

// Update is called once per frame
void Update () {
    GetComponent<Rigidbody2D>().velocity = new Vector2(Input.GetAxis("Ho

    if (Input.GetButtonDown("Fire1"))
    {
        Instantiate(shot, transform.position, Quaternion.identity);
    }

}
```

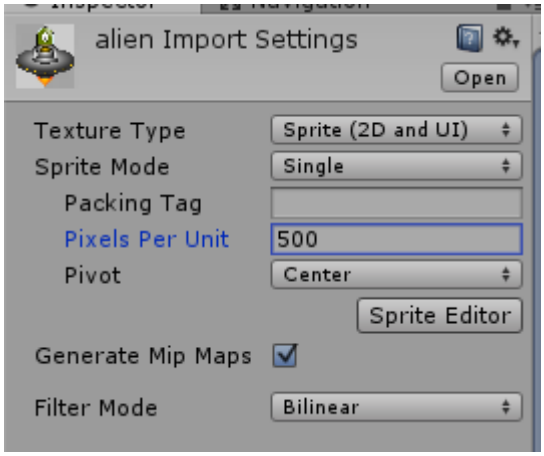
Drag the prefab you made for the beam onto the script in the inspector, assigning it's value to the variable shot above.



Test once again, you should be able to shoot beams now.

## Enemies

We're using another free art asset, an alien by Ctoy- http://c-toy.blogspot.pt/ configure it using the settings below. This ones not a sprite sheet, so just a single, but the pixels per unit (how many pixels in the image refer to one unit in Unity) does need setting (it'll be too big otherwise).



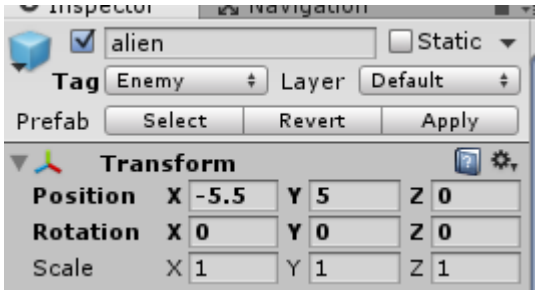

Drag alien onto the scene, then from the scene into the prefab folder. To the prefab, as before, add a Rigidbody2d, make kinematic, add a BoxCollider2D, make Trigger and test if you can shoot and kill the enemy.

Add a new script enemyScript, this controls your enemies, "public int speed=2;" is a variable to holw the speed of the enemy.

```
using UnityEngine;
using System.Collections;

public class enemyScript : MonoBehaviour {

    public int speed=2;

    // Use this for initialization
    void Start () {
        GetComponent<Rigidbody2D>().velocity = new Vector2(speed, 0);
    }

// Update is called once per frame
void Update () {
    if (transform.position.x >= 6)
    {
        transform.position = new Vector2(transform.position.x-1, transform.position.y - 1);
        speed = -speed;
        GetComponent<Rigidbody2D>().velocity = new Vector2(speed, 0);
    }
    else if (transform.position.x <= -6)
    {
        transform.position = new Vector2(transform.position.x + 1, transform.position.y - 1);
        speed = -speed;
        GetComponent<Rigidbody2D>().velocity = new Vector2(speed, 0);
    }
}
```

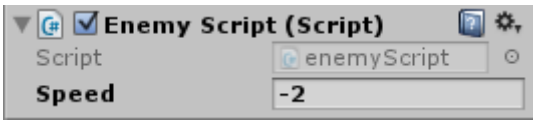
The by now familiar velocity code in start, then the update code which may be slightly confusing.

The “if” bit, checks the objects x position, if it’s greater than or equal to 6, the position is equal to it’s current position -1 and reverse it’s speed before passing this to velocity. A similar thing happens if the position is less than -6. This means if it’s partly off the camera on either side, we reverse it’s speed, or make it go the same speed, in the other direction. We also bring it down a line.




Drag a prefab onto the scene, position it at -5.5,5 then duplicate them (right click it, duplicate), move this one to -3.5, 5, then again and -1.5,5 repeating up to 4.5,5.

Highlight these and duplicate again, drag them down and find their speed in EnemyScript attached to these ones and set it to -2 for this entire row. Now, with the first one starting at -4.5,5 and going up to 5.5,4 (x going up by 2 each time, y staying the same) position them evenly. Test.




## Enemies shooting

Create a new script, enemyShoot, this will be the script that controls our enemy’s movement. We setup three variables, two float (decimal) numbers for the time between shots (how fast we can shoot) and the time the next shot takes place and a GameObject that will hold our bullet prefab.

```
public class enemyShoot : MonoBehaviour {
    public float timeBetweenShots;
    public float nextShot = -1;
    public GameObject bullet;


    // Use this for initialization
    void Start () {
        nextShot = Random.Range(1, 3.0f);
        timeBetweenShots = Random.Range(3, 6.5f);
    }

    // Update is called once per frame
    void Update () {
        if (Time.time > nextShot)
        {
            Instantiate(bullet, transform.position, Quaternion.identity);
            nextShot = Time.time + timeBetweenShots;
        }
    }
}
```

When the script starts, the nextShot variable is given a random time between 1 and 3 seconds, this is so we don’t just start with a wall of beams. The timeBetweenShots is also randomised to give it a bit more variety.

In Update, every frame we check if the time has gotten to the time in which the next shot can be fired, instantiate (bring in to being) a bullet at the enemy’s location and then set the next time it can fire to be the current time plus the time allowed between shots.

If you try this now using the prefab you made for the player, you'll find that it works but blows the enemy up too. This is because the beam collides with the alien's body. What we need is to check if the thing we hit is the target of the shooter.

Create a new shot prefab by duplicating it and changing (dragging a different one in) the sprite.

Modify the shotscript, open it up and make the following changes;

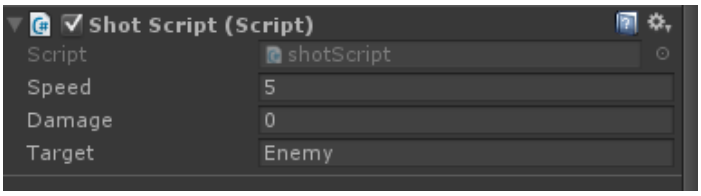
```
public int speed=5;
public int damage;
public string target;
```

Only the bottom line is new, we add a new field that lets up set the target of the shot.

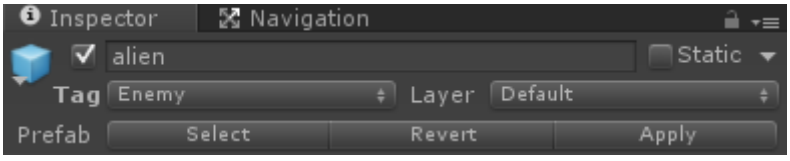
```
void OnBecameInvisible()
{
    Destroy(gameObject);
}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == target)
    {
        Destroy(other.gameObject);
        Destroy(gameObject);
    }
}
```

Here we modify OnTriggerEnter2D so that instead of directly checking for the player tag, we compare with the tag of the expected target.

If you see the shot script on the beam now, you should see a space for a tag to be entered, the one below is the Enemy tag for the player beam, the enemy's beam would have the player's tag.




Note: you will need to find the enemy prefab created earlier, under Tag, you'll need to add Enemy (Tag->Add tag-> Enemy) don't forget, once you create a new tag you've still got to assign it.




Finally, one last change to an existing script, in enemy script we make a couple changes that can be used for quality of life/ game.

```
void OnBecameVisible()
{
    GetComponent<enemyShoot>().enabled=true;
}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Player")
    {
        Destroy(other.gameObject);
    }
}
```

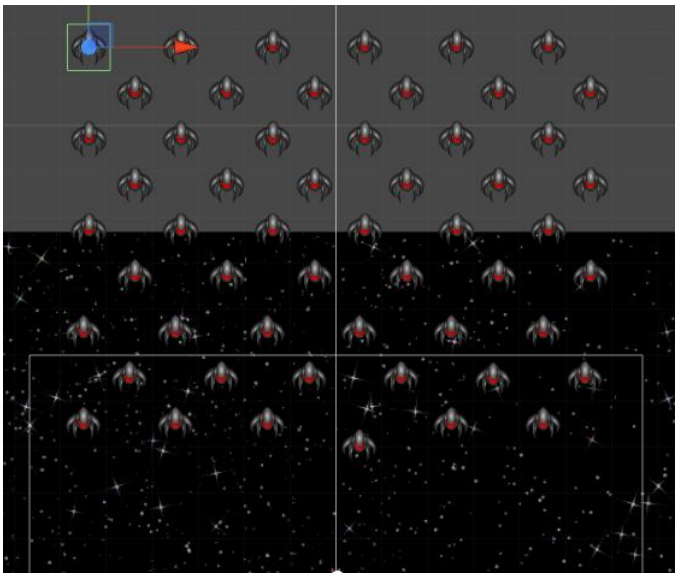
This is added just above the last } the first is a built in function that checks if the object is visible, if it is we enable the shoot script, meaning we don't get waves shooting from off screen. Do ensure you disable the enemy shoot script manually by unchecking it in Unity.

The second just says if the player collides with an alien, kill it.

Select all the aliens, duplicate off screen, in waves of two, each moved up by 2 in the Y axis.

You can make it harder by increasing the speed of some of the ones above if you wish but timing may vary.

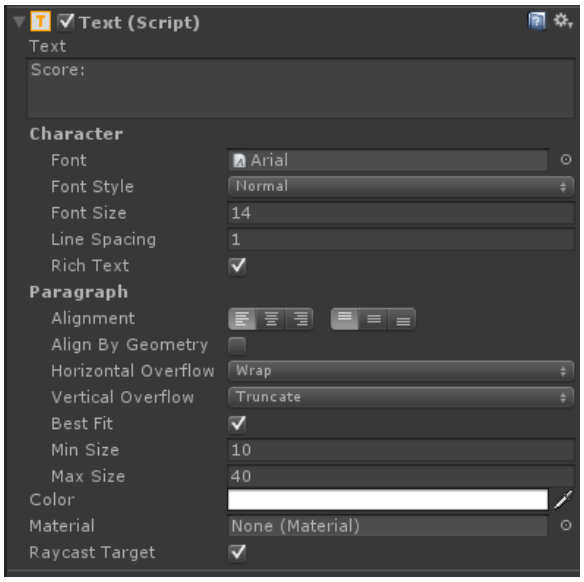
You should have something that looks a bit like the below by now.



## Keep score

You'll want a way of keeping score, add a canvas and text (UI -> Text) then double click canvas to focus on it.

Click text object and make sure, best fit is selected and colour changed to white, whilst changing the text in the box to "Score:" like the below.




Move to reposition text to the bottom left of the screen.

Create a new script score, this will be have static variables and methods. Static variables and methods aren't actually created or attached to things that are created, they sit on their own so there is only ever one copy of them. They can be called directly from the script.

```
public class Score : MonoBehaviour {

    public static int score;

    public static void updateScore()
    {
        score++;
    }

}
```

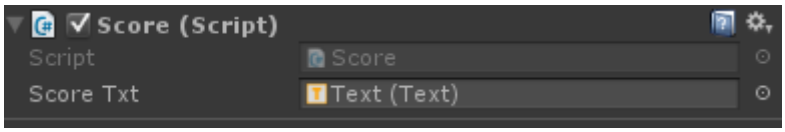
In ShotScript change the OnTriggerEnter2D to call the score script after you've destroyed an enemy and before you've destroyed the gameObject. This isn't a perfect implementation but it works.

```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == target)
    {
        Destroy(other.gameObject);
        Score.updateScore();
        Destroy(gameObject);
    }
}
```

Back to score, add the text "**Using unityengine.ui**" to the top of the file to tell Unity we're using the new(er) UI system. Then, in a function called LateUpdate() which is a built in that runs at the end of each frame, you update the scoreTxt's text to be equal to the score.

```
public static int score;
public Text scoreTxt;

public static void updateScore()
{
    score++;
}

void LateUpdate()
{
    scoreTxt.text = "Score: "+score;
}
```

Attach Score to a new  gameObject called score and drag text from UI to the public field.




Test your work, it should play and give you points every time you kill an enemy, great, now we need some refinements.

## Refining and a few changes

At the moment when we get hit, we just disappear, we want to play a nice animation or effect and then the game to end. We could do this using particle systems, there's plenty of ways of doing it, but we'll play an animation instead, nice and simple again.

We're using smoke and explosion sprites by Kenney, ensure they're all set as sprites, select them all and drag them onto Hierarchy like before. Again save the explosion animation it wants to create. You can shrink the explosion animation too, scale it down to 0.3. Make the explosion a prefab and we'll get on with modifying the shot Script to actually create an explosion.

Find the shot prefab and open the Shot Script, we are going to change a few things.

```
public class shotScript : MonoBehaviour {

    public int speed=5;
    public int damage;
    public string target;
    public GameObject explosion;
```

We add a new public variable explosion, to drag our exploision prefab onto.

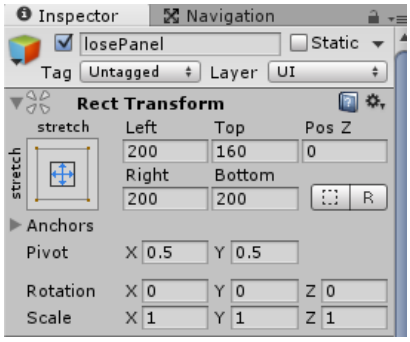
```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == target)
    {
        Destroy(other.gameObject);
        GameObject fire =(GameObject) Instantiate(explosion,other.gameObject.transform.position,Quaternion.identity);
        Destroy(fire, 1.0f);
        Score.updateScore();
        Destroy(gameObject);
    }
}
```

After the first Destroy we add this new line, we create a new variable “fire” of type GameObject, the (GameObject) on the right of the equals means that we are forcing it to be a GameObject (not a generic object as is created). Then we instantiate our explosion prefab, at the position of the object the shot collided with, with no rotation. We then call Destroy on the explosion with a time til destroy of 1 second.

Drag the explosion prefab onto the shot prefab’s shotscript explosion field and give it a try. Note, you need to drag it onto both, I’ve included other explosion sprites into the asset pack so you can have different colours / types for enemy and player.

## Win and lose state

The lose state is easy, the player dies, let's deal with that first. What do we want to happen when we lose? Well, we want a message to the player and the option to restart.




So, when the player dies, we'll pop up a message and button. Add a new panel to the canvas we used to display the score. Resize the panel as below and add an image component then drag the you lose message image in. You may also want to select the panel and tweak the colour too, I used a semi transparent black background.

Create a new script, call it "gameManager" we'll use this to check on a number of things, firstly, if the player is dead.

```
static bool isPlayerDead;
public GameObject loseScreen;

// Use this for initialization
void Start () {
    isPlayerDead = false;
    loseScreen.SetActive(false);
}
```

First, we'll need a couple variables, the first is a static Boolean to signify if the player is dead. Static variables are accessible without a direct reference to a copy of the script. In other words, the variable will be the same for all copies of gameManagers, we can use it in a global sense to keep track of if the player is dead. The next variable will hold a reference to the lose screen panel we just created. Finally, in start() we set the isPlayerDead variable to false (no, he's alive), then deactivate the loseScreen.

Then, in update, we check if the player is dead, if he is we activate the loseScreen. The "playerDead" method is static, and public, allowing other scripts to call it to set the player as Dead.

```
// Update is called once per frame
void Update () {
    if (isPlayerDead)
    {
        loseScreen.SetActive(true);
    }
}

public static void playerDead()
{
    isPlayerDead = true;
}
```

Which means we need to make some modifications to the other scripts; in enemy script where we kill the player if he collides with an enemy we need to update the OnTriggerEnter2D like so. Calling the static function we created earlier.

```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Player")
    {
        Destroy(other.gameObject);
        gameManager.playerDead();

    }
}
```

```
void OnTriggerEnter2D(Collider2D other)
{

    if (other.tag == target)
    {
        if (other.tag == "Player")
        {
            gameManager.playerDead();
        }
        else
        {
            Score.updateScore();
        }

        Destroy(other.gameObject);
        GameObject fire = (GameObject)Instantiate(explosion,
            other.gameObject.transform.position, Quaternion.identity);
        Destroy(fire, 1.0f);
        Destroy(gameObject);
    }
}
```

We also need to do something similar in shotScript where it checks if it hits the target. We made some more major changes here, checking if the thing we hit is the Player and if it is setting player to dead so the screen launches, if it's not updating the score. Then destroying shot and thing we hit.

Give it a try, play the game and test getting shot and colliding with the enemy, both times it should pop up the "you lose" screen. You might notice though that there's no way to get back or dismiss this, let's fix that.

Select the losePanel you created and right click it to add a UI Button. We'll use this button to allow the player to continue. Rename it continue, position it how you want and then find the text component then delete it. Right click it and an image UI component, we're going to use the restart image as a button image. Resize it as appropriate.

Open up your gameManager script, add the following code;

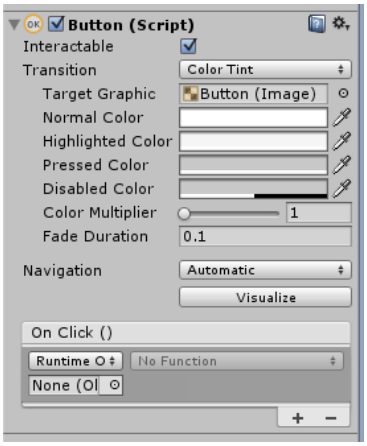
At the top, tell Unity we're using scene management, (the second line is new)

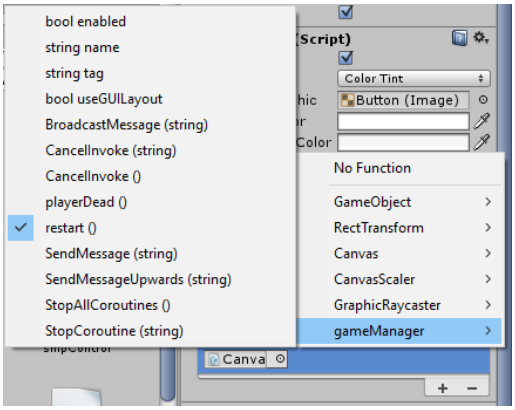
```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections;
```

Then, underneath update, add the restart code, LoadScene(0) just means load the first scene.

```
public void restart()
{
    SceneManager.LoadScene(0);
}
```

Now over to buttons, buttons use the built in Unity UI Event system, looking at the button object there is an "On Click()" section, this defines what happens when the button is pressed. We drag our Canvas with our gameManager script onto there and then select the function we want within gameManager as below. You can see all the scripts attached to the canvas, even the ones hidden from view normally. We want the gameManager, then the restart() method as seen from below.



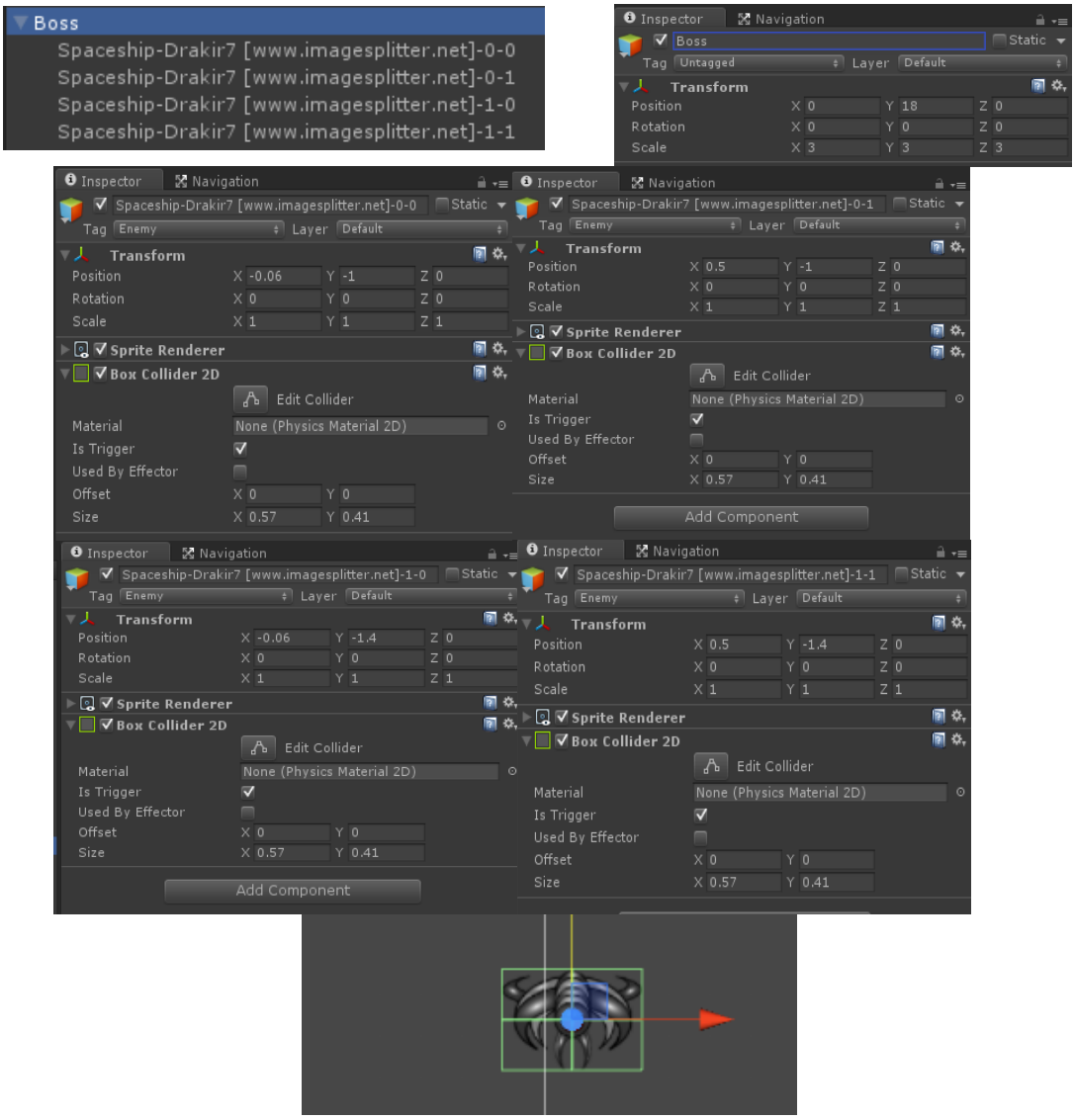




Give it a try, when you die you should get a message and the opportunity to restart and reload the game.

## Win condition

Let's add a win condition, we'll have a simple one, the continuous line of enemies will give way to one big guy, when you kill the big guy, you win. Once more we're using one of C-toys alien spaceships. This time though, I've split it into 4 images, create a new empty gameobject and rename it boss, this will be the container to hold the individual images. Scale the empty gameobject up by 3, 3, 3, and position it at 0,18,0. Now, select each of the individual images from the project view and drag them, one by one into the scene, parented to the empty gameobject.

The below screenshots show the placement of the images, note I've also added a box collider 2d set as triggers to each of the images and each of the separate images are tagged as enemy.




Now, you need to add some logic to the "boss" object, add a rigidbody2D, enemy script and enemy shoot script to the boss object. Try it out, don't forget to drag a bullet prefab to the shoot script.

You should find that if you get to the boss you can chip away at him, what we want is that when the boss is dead, aka all pieces are gone, you win. Let's set that up now.

Find the losePanel you set up before and duplicate it, change the image from lose to win text. Open the GameManager script, add a new variable winScreen under loseScreen and set the loseScreen to not be active at start, finally, we add a "hasWon" variable that signals whether we've won. That's the code to the left.

```
static bool isPlayerDead;
static bool hasWon;
public GameObject loseScreen;
public GameObject winScreen;

// Use this for initialization
void Start () {
    isPlayerDead = false;
    loseScreen.SetActive(false);
    winScreen.SetActive(false);
}
```

```
// Update is called once per frame
void Update () {
    if (isPlayerDead)
    {
        loseScreen.SetActive(true);
    }
    else if (hasWon)
    {
        winScreen.SetActive(true);
    }
}

public static void winGame()
{
    hasWon = true;
}
```

Now we modify the update code to say, “if the player has won, show the winScreen” and we also add a static method to set the hasWon variable to true, what we’ll use in our boss script.

Now, if we save this and create a new script on the Boss called “boss script” we can add the following into our update method. This just says if the object it’s attached to no longer has any children- so all the boss bits are dead, you win. Et Voila, your game is pretty much done, you can win, lose and score.

```
// Update is called once per frame
void Update () {
    if (transform.childCount <= 0)
    {
        gameManager.winGame();

    }
}
```

It’s up to you now, you can easily extend this, check out some other tutorials aimed at extending the knowledge. Building an in game menu for instance, transitioning between levels, by the same logic as above you can add stuff for the enemies to damage. It’s all up to you to try.