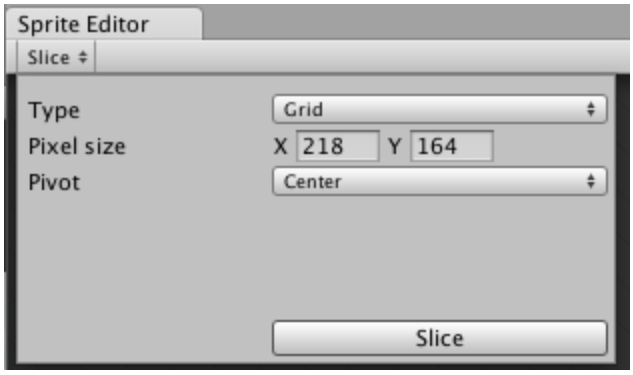
Download the project from https://github.com/mindcandy/lu-angry-scrawls-2 - you can either clone using git, or simply download a zip at https://github.com/mindcandy/lu-angry-scrawls-2/archive/master.zip

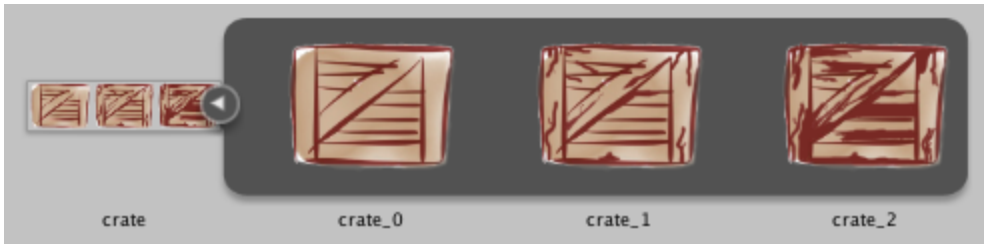
In Unity, open the project and then open the GameScene.scene (if it doesn't open automatically.

I've split up the sprites from Angry Scrawls 1 and made 'damaged' versions by duplicating them in Photoshop and drawing cracks on them! Duplicating the layer and using the 'Offset' filter is a great way of doing this. Alternatively, you could simply copy the file using Finder/Explorer and rename it so you have crate1.png, crate2.png, etc and simply draw over the damaged versions in your favourite image editor. There's no requirement to have all the sprites in the same file, it just makes them easier to manage.

I kept a note of the original size of each item, and then used the 'grid' method of importing the sprites. I've already split up the imported sprites in the starter project, but if you ever need to do this yourself, open the Sprite Editor and then select Slice > Type = Grid and enter the width/height of the grid. Unity will chop up your sprite sheet automatically.



If you take a look in the Asset/Sprites folder you can click on the little arrow to the right of each sprite to see how it is split up:



I've also created an enemy, which has two sprites - a normal and 'damaged' one.
I've made a score sprite too! We'll get back to these later...

In the starter project, I've swapped over form the old to the new sprites and I've also set up sprite render Sorting Layers - for more info on these see the "Maze Game" tutorial at https://github.com/mindcandy/lu-maze-game

So that's some housekeeping done - what we want to do in this lesson is add the ability to break stuff!
To do this, we'll be adding the Destroyable script to our breakable objects. Let's take a look at it:

```
public class Destroyable : MonoBehaviour {

    // how strong is this object? larger values mean it will be harder to break
    public float toughness = 10.0f;

    // ignore force less than this
    const float threshold = 0.1f;

    // health of object
    float health;

    // Use this for initialization
    void Start () {
        health = toughness;
    }

    // when a collision occurs
    void OnCollisionEnter2D (Collision2D collision) {

        if (rigidbody2D != null) {

            // calculate force of collision
            float force = collision.relativeVelocity.magnitude * (massOf(rigidbody2D) +
massOf(collision.rigidbody));

            if (force > threshold) {
```

```
            health -= force;

            if (health < 0.0f) {
                // send message saying this is destroyed
                SendMessage("OnDestroyed", SendMessageOptions.DontRequireReceiver);
                Destroy(this.gameObject);
            } else {
                // send a message saying how damaged this object is (1= no damage, 0= dead)
                SendMessage("OnDamaged", health/toughness,
SendMessageOptions.DontRequireReceiver);
            }
        }
    }
}

// get the mass of the body or 0 if there is no body
float massOf(Rigidbody2D body) {
    if (body == null)
        return 0;
    else
        return body.mass;
}
}
```

We're giving each game object a 'toughness' that we can edit, and an internal 'health' variable that tracks how much of that toughness we've used up. The OnCollisionEnter2D() method is called any time we have a collision between this object and another RigidBody2D. Unity makes it fairly easy to work out roughly the force of that collision, because it gives us the relativeVelocity and we can ask each RigidBody what its mass is. (I know this isn't really the force for people who actually know physics, but it works well enough for this situation).

So we get the 'force' via this line:

```
        float force = collision.relativeVelocity.magnitude * (massOf(rigidbody2D) +
massOf(collision.rigidbody));
```

We simply subtract this from the health (if its above a simple threshold). If we're out of health then we Destroy() the game object and send an 'OnDestroyed' message to tell other behaviours on this game object that we're destroyed. If we have some health remaining we instead send an 'OnDamaged' message that includes a value of how 'healthy' we still are.

This is the first time we've used messages so it's worth taking a moment to talk about why we use them. Essentially, sending little 'messages' between behaviours allows those behaviour to pass information and co-ordinate, without them having to be very strongly bound together. So in

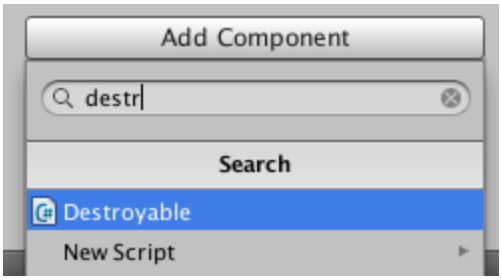
this case, the Destroyable script doesn't know who might listen out for the 'OnDestroyed' message - and it doesn't care!

Messages allow us to have really generic scripts/behaviours that we can use on a variety of game objects. It saves us from writing more code than we would otherwise :)

In this game, we'll be using the messages from Destroyable to show damage and to know when an enemy has been killed -- but more on that later!

For now, let's add Destroyable to our blocks. Go to the Assets/Prefabs folder and select all the prefabs except Player - you can use Cmd + click to select multiple things.

You can then click on Add Component in the Inspector and it will add the component to all of the prefabs at once. Add the 'Destroyable' component:



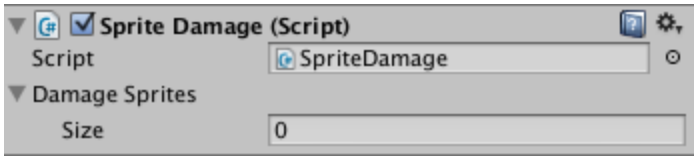
Now play the game and you should see that objects disappear after a while. Toughness of 10 seems a little low, so edit it to be 30 and try again. You can even set different prefabs to have different toughness - e.g. set the yellow Block to have a toughness of 50.

It's hard to see what's going on at the moment, because the blocks don't change appearance before disappearing. We can solve that by adding the 'Sprite Damage' component which is another script we've written:

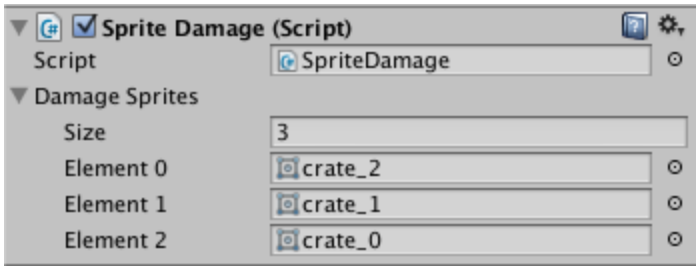
```
public class SpriteDamage : MonoBehaviour {

    // array of sprites to show at different health levels
    // first is most damaged, last is 100% healthy
    public Sprite[] damageSprites;

     SpriteRenderer spriteRenderer;

    // Use this for initialization
    void Start () {
        spriteRenderer = GetComponent<SpriteRenderer>();
    }

    // sprite is damaged
    void OnDamaged(float normalisedHealth) {
        int spriteIndex = Mathf.FloorToInt(normalisedHealth * (damageSprites.Length - 0.01f));
        spriteRenderer.sprite = damageSprites[spriteIndex];
    }
}
```

We have an array of sprites, which is then used such that he first sprite is the *least* healthy state and the last sprite is 'full health'. When we get an 'OnDamaged' message (from the Destroyable script) we can then take the supplied health value and use it to decide which sprite to show (by setting spriteRenderer.sprite). Note that name 'normalisedHealth' -- this is a term from mathematics, generally when a variable has a range from 0.0 to 1.0 we say it is 'normalised'. The advantage of passing around these normalised values is that they help to make behaviours more general. For example, we can give each block a different toughness without having to change the the sprite damage code at all.

Select the crate prefab and add the Sprite Damage component. When you do this the Damage Sprites array will have size 0.



Set this to 3, because we have three sprites for the crate. When you do this, new slots will appear in the Inspector. You can then drag in the crate sprites from the Assets/Sprites window, or find them by clicking on the circle icon to the right and browsing within the Assets. Remember to put the *most* damaged crate in slot 0 and the *least* damaged crate in slot 2. It should look like this:

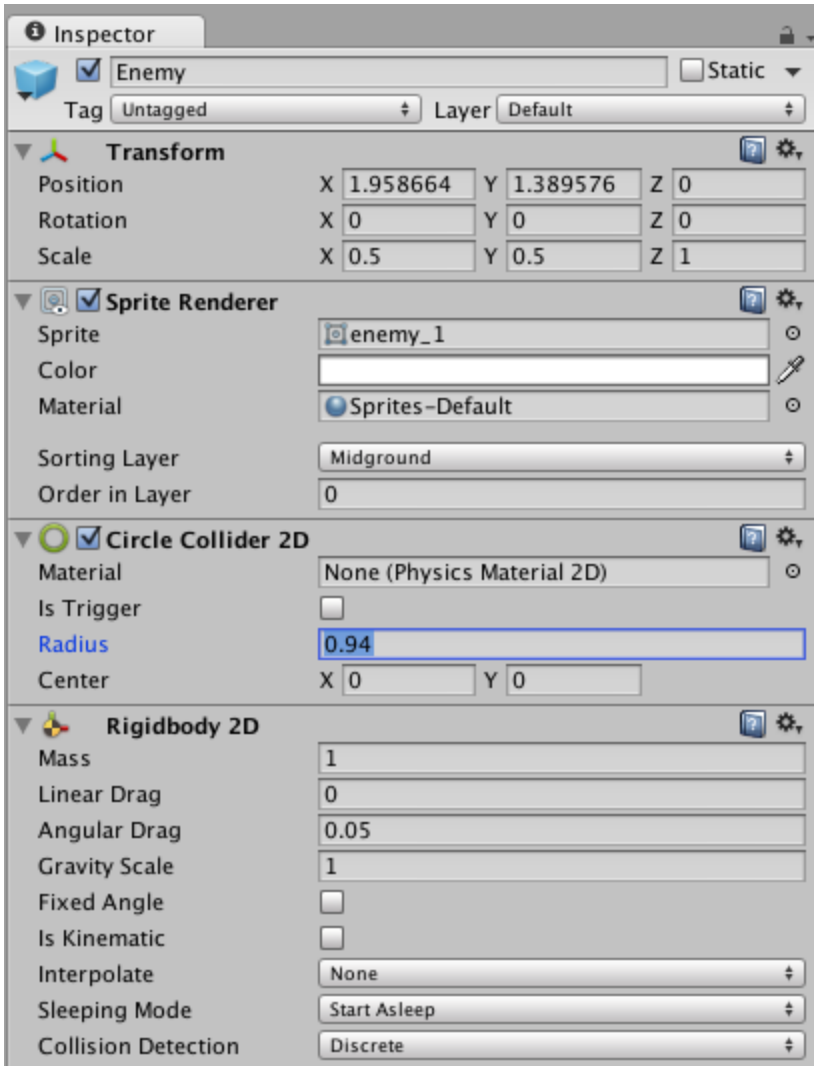


Now if you run the game you should be able to see the crate getting damaged before it disappears. Much better!

Now repeat adding Sprite Damage and sprites to all the other prefabs (except for the Player - we don't want that to be destroyed!).

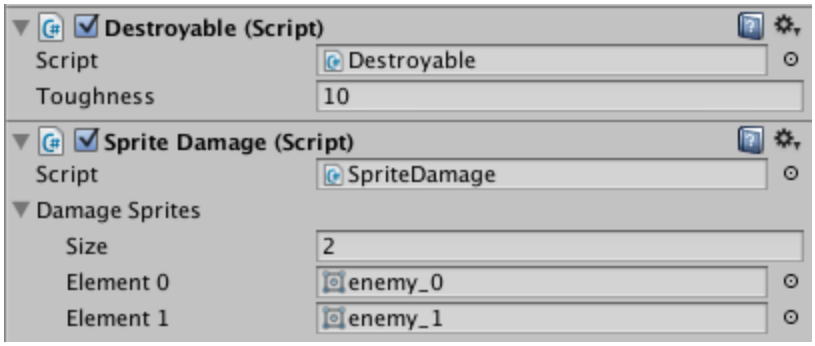
We want to add an enemy now, so find the enemy_1 sprite in Asstes/sprites and drag it on to the Scene. Initially it might not appear until you set Sorting Layer = Middleground. Rename the object to 'Enemy'.

We want the enemy to behave under physics so add a Circle Collider 2D and a Rigid Body 2D to it using the 'add components' button. Set the radius in the Circle Collider to 0.94 or whatever appears correct to you.



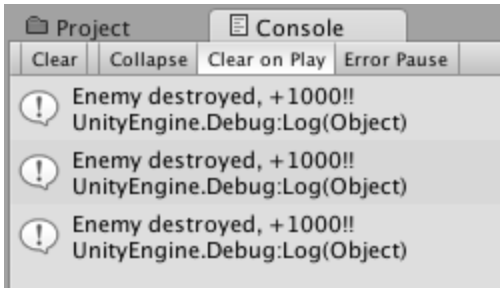
Test the game and tweak radius (and optionally mass) until you are happy.

Now add the Destroyable and Sprite Damage components and test again! You should be able to damage and then destroy the enemies. You might want to make the enemies tougher if they die too easily!

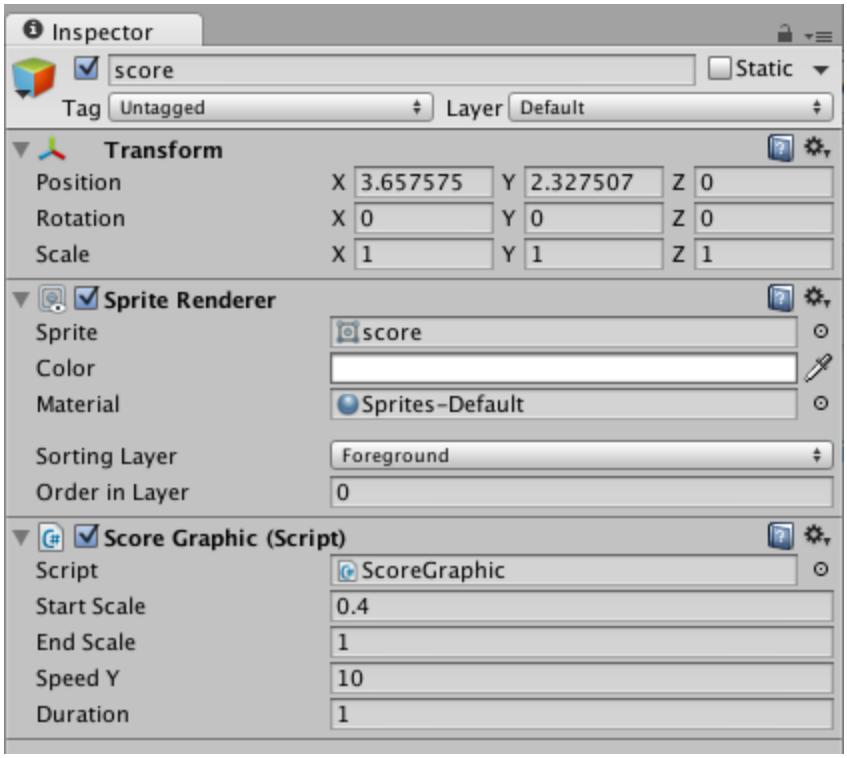


Now drag the Enemy from the Hierarchy to the assets/prefabs folder to create a prefab, then drag back out of the prefabs folder to place a couple more enemies in the Scene.

Now add the Enemy Script to the enemy prefab and when you destroy enemies you should see messages in the console afterwards (it might be best to turn off “Maximize on Play” in the Game window to see this).



Now we want to add a nice “1000!” graphic when you kill an enemy. We want to build a new prefab, but we have to do that in the Scene. So drag the score sprite from the Assets/sprites window into the Scene, and set the Sorting Layer to Foreground so it appears on top of everything.



Also add the Score Graphic component to the score. This provides some animation by scaling (in code) between two scales, whilst moving up and fading out. You can adjust these parameter to change the animation of the score. For example, change Speed Y to 4 and see how that changes it.

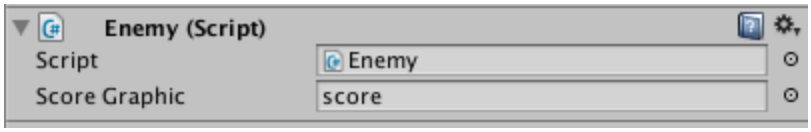
The code is here:

```
public class ScoreGraphic : MonoBehaviour {

    public float startScale = 0.4f;
    public float endScale = 1.0f;
    public float speedY = 10.0f;
    public float duration = 1.0f;

    float time;

    // Use this for initialization
    void Start () {
        time = 0.0f;
    }

    // Update is called once per frame
    void Update () {
```

```
time += Time.deltaTime;

float t = Mathf.Clamp01(time / duration);
float scale = Mathf.Lerp(startScale, endScale, t);
float alpha = 1.0f - t;

transform.localScale = new Vector3(scale, scale, 1.0f);
transform.position = transform.position + new Vector3(0.0f, speedY * Time.deltaTime, 0.0f);
renderer.material.color = new Color(1.0f, 1.0f, 1.0f, alpha);

if (time > duration) {
    Destroy(this.gameObject);
}
```

We could actually do this more elegantly with the Animator, but we'll cover that another time! For now, just drag the score object from the Hierarchy into assets/prefabs to make it into a Prefab. Then you can remove it from the scene.

Now select the Enemy prefab and find the Score Graphic property - drag in the newly created score prefab.



You should now be able to play the game and see enemies span a score effect when they die!