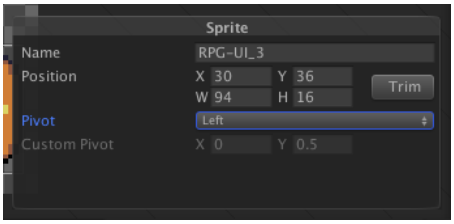
Unity Tutorial : Battle RPG

Open up the starting project and open the scene ‘Game Scene’. You will see that most of the assets are already set out but just require logic to be attached to them.

1. We will start by creating a progress bar. First create an Empty Game Object (Menu Bar > GameObject > Create Empty) and rename it to ‘Progress Bar’. In the project panel select the RPG-UI from the Sprites folder and drag in 2 sprites (both named RPG-UI_3). Name these clips ‘Bar’ and ‘BarBG’ and nest them under the ProgressBar object. To make the progress bar easier to use we want to set the pivot point of the bar to be on the leftmost of the sprite.

To set a sprites pivot, select the RPG-UI image in the Sprites folder and click the ‘Sprite Editor’ button in the Inspector. In the Sprite Editor select the Bar image and you will see some options appear. One of the options is called ‘Pivot’ and will be defaulted to ‘Center’, change this to ‘Left’ to move the pivot point. Next click ‘Apply’ to save these changes.



Position the sprites so that they look like so:

Set the Sprite Renderers ‘Sorting Layer’ of both sprites to be UI and set the ‘Order in layer’ value of the Bar sprite to be 1 so that they remain visible above the scene.

We are going to control the progress bar by manipulating the x scale of the bar sprite. When it is 0 the bar will be empty and when it is 1, the bar will be full.

To the parent Progress Bar object add the ‘ProgressBar’ script. This script has two parameters that can be edited in the Inspector: Bar which needs to be set to the white bar sprite and Progress specifies the current position of the progress bar from 0 - 1, set this to 1 for now so the bar starts full.

You can also change the colour of the progress bar by selecting the Bar sprite and changing the colour property of the Sprite Renderer, try choosing any colour and see the result.

We are going to need two progress bars for this game so duplicate the Progress bar, and set them up in the scene as follows:

Name the topmost progress bar ‘Health Progress Bar’ and the bottom one ‘Action Progress Bar’.

2. It’s time to set up some interactions for the game. For this game we are going to place most of the game logic in a single script and use something called ‘Delegates’ to deal with interacting with buttons and selecting targets.

A delegate allows us to pass a function to another function and treat it like any other variable. Multi-select ‘AttackBtn’,’MagicBtn’ and ‘HealBtn’ and add a ‘BoxCollider 2D’ component and an ‘ActionButton’ script.

The ActionButton script has no public variables but one public function, this function accepts another function as a parameter which it then stores a reference to and calls when the button gets clicked. A delegate is used to describe a function here we describe the function as follows:

```
public delegate void ButtonDelegate();
```

This describes a function that returns void/nothing and takes no parameters (empty () brackets) and we’ve called this type of function ‘ButtonDelegate’. We can now treat the word ButtonDelegate the same way we treat any other variable such as int, GameObject etc so we can make it into a class field or a local variable etc and each time it is assigned it must be assigned to a function that meets the criteria in it’s description. This is a very useful feature that for this game allows us to keep all of the attacking logic in a single class instead of creating it on each button. These buttons are now set up but another script needs to tell them what to do when they are clicked.

Create a new Empty Game Object (MenuBar > GameObject > Create Empty) and name it “Action Controller”. To this object add the script ‘ActionController1’. This script has a few parameters that need to be set:

AttackBtn, MagicBtn,HealBtn and InfoBar should all be set to the objects of the same name under the HUD game object. The Progress Bar parameter should be set to the ‘Action Progress Bar’.

In the scripts start function, each of the Action buttons is assigned a function (via delegates for when it gets clicked) like :

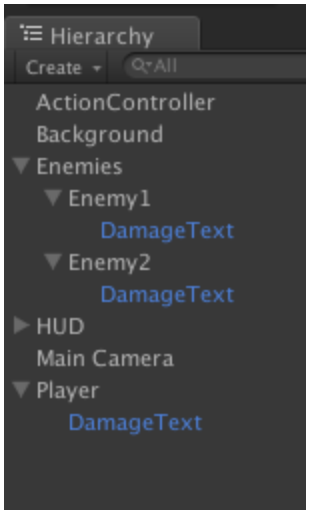
```
attackBtn.GetComponent<ActionButton>().addClickHandler(onAttackClicked);
```

This is pointing to the ActionButton script on that object and storing a reference to the function onAttackClicked for whenever the button gets clicked.

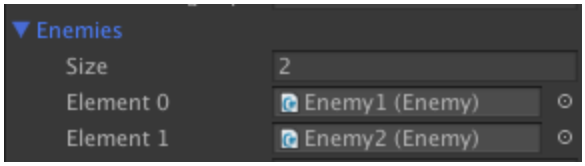
If you test the game you will be able to choose an action, get text feedback in the info bar as to which action was chosen and then need to wait for the Action Progress Bar to fill before another action can be chosen.

3. It's time to set the enemies up so that we can damage them. Select both Enemy1 and Enemy2 in the scene and to them add a 'BoxCollider 2D' a 'Damageable' script and an 'Enemy' script. The enemy script has 2 parameters Attack Speed and Attack Power. Try leaving these default for one enemy and for the other try raising them to around 5 and 0.05 to make it a bit tougher.

The enemies are almost set up we just need to add a text object to them so that we can see how many points of damage we deal to them. In the prefabs folder you will find a 'GameText' prefab. This is simply a 3D text object that has already been configured. Create 3 of these objects and position them in the scene above the player and enemy characters and in the Hierarchy nest them under the corresponding characters:



Select the Action Controller object and modify the ActionController script so that it now points to ActionController2. Doing this will reveal a new parameter 'Enemies' which is an array. Set the length to 2 and assign the Enemy values to the ones in the scene.



In the ActionController2 script we now set up the enemies during the Start function. During this setup process, the Array of enemies gets read and each one has a delegate (function) set to it for when that enemy gets clicked. The previous Action functions for Attack and Fire have also

been modified so that when clicked the action gets stored temporarily and the player is prompted to select an enemy (which calls our delegate function) which then applies the corresponding damage for that action to the chosen enemy.

Now if you test the game, when you select ‘Attack’ or ‘Fire’ you will be prompted to select a target, and then when you click on an enemy you will see damaging Hit Points appear above them. Repeating this will eventually result in the enemy being destroyed.

4. We can now hurt and even defeat the enemies! But what’s a game without challenge!? So let’s give the enemies a little bit of logic so that they can attack the player.

In the enemy script uncomment the code from the Start function to start the cycle of enemy actions.

The Enemy script now calls a special unity method called ‘Invoke’ this allows you to call another function after a set period of time has elapsed and it’s how we make the enemy only attack every few seconds as opposed to constantly.

Change the Action Controller script to be ‘ActionController3’ . This will reveal the Player parameter. Set this to the player. Also add the ‘Damageable’ script to the player.

The ActionController3 now passes a second function through via the delegate to each enemy, which is what function to call when the enemy attacks the player. This function works the same as the enemy equivalent and adjusts the health value of the players Damageable component. The Heal function of this script was also modified to increase the players health when used.

Now when you test the game the enemies will periodically attack the player which is visible by the Hit Points appearing over the Player who now may also be destroyed.

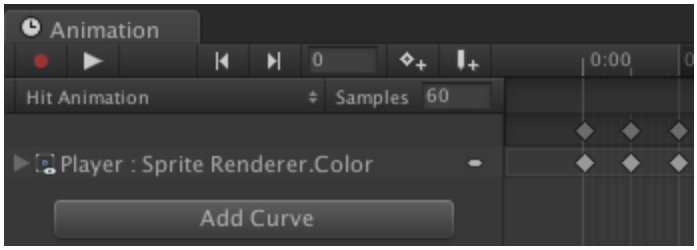
To make it easier to see the players health, let’s hook up the top Health Bar that we created at the start of this tutorial. Select the ‘Health Progress Bar’ and to it add the script also named ‘HealthProgressBar’ this script will link a progress bar to a damageable component to represent it. So assign it’s two values to the ‘ProgressBar’ script that is also attached to this object as well as the ‘Damageable’ script attached to the player.

Now when the player takes damage it will be reflected in the progress bar!

5. Create damage animation

Open the animation panel and select Enemy1 in the scene. Click the record button on the Animation panel and choose to name the animation “Hit Animation”. Click ‘Add Curve’ and choose Sprite Renderer > Color. Two sets of frames will be added to the timeline around 0:00 and 1:00. Click at around 0:30 to move the red progress bar to that point and then in the

Inspector select the Color value of the Sprite Renderer and change the color to be red. You should see that an animation key is automatically added for you in the Animation window at your current frame. If you choose to play the animation you will see the sprite flash red and return to normal.



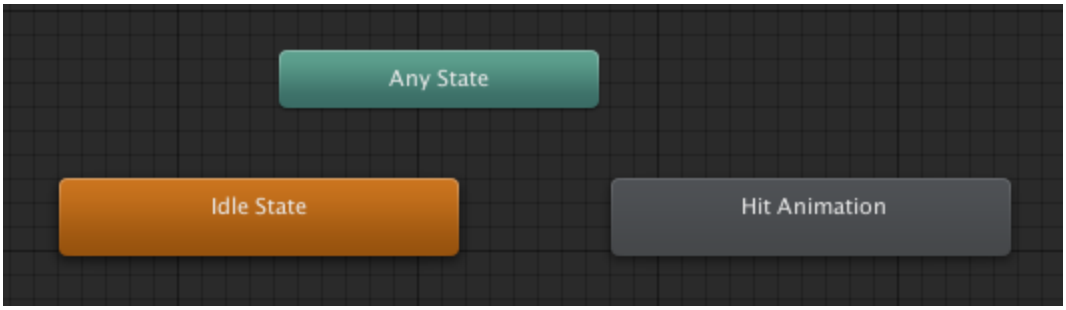
It's a bit slow at the moment however so let's move the frames so that they are closer together so it plays faster. Simply click and drag the diamond shape of the frames and move them so they are at 0, 0:05 and 0:10 respectively. You can then hit the Record button again to stop editing the animation.

If you test the game now you will see that the animation is constantly playing which isn't very nice!

First thing to solve this is to select the 'Hit Animation' in the Project panel and to uncheck the 'Loop Time' value in the Inspector so that the animation does not repeat.

Next up, In the Inspector (with Enemy1 selected) double click the Controller value of the Animator component to open up the Animator window.

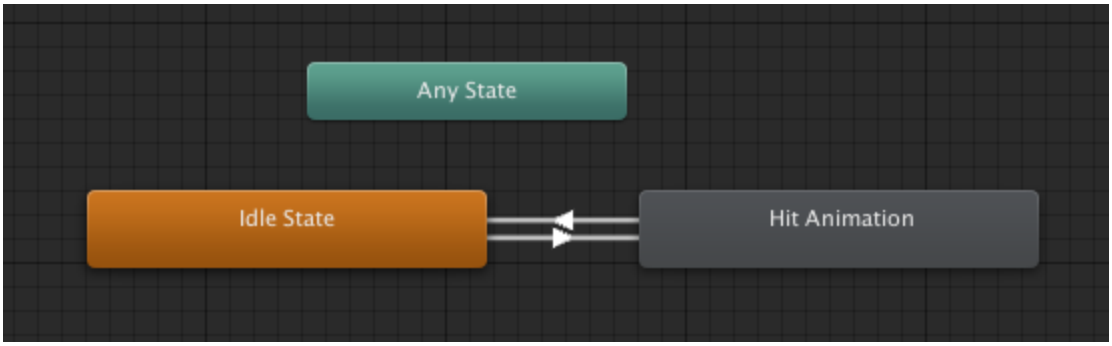
You will see only one animation in here which is the Hit Animation. Let's add a second empty animation that we can use by default. Right click anywhere in the Animator and choose 'Create State > Empty' this will create a 'New State' box. In the Inspector you can rename this, So I suggest calling it 'Idle State'. Right click the Idle State and choose 'Set as default'. You will see that it will become the orange block. This indicates that this is the animation that will be playing when the game starts. And if you test the game now, you will see that the Hit animation no longer plays!



To get the animation playing correctly we need to do a few things. Firstly we will create a parameter in the Animator to handle this. At the bottom of the animator, click the + button next to Parameters, choose 'Trigger' and rename it to 'Hit'. A trigger parameter is a more useful version of a boolean, one that is always assumed to be false by default and when set will trigger an animation but then as soon as the animation starts it will set itself back to default allowing for the animation to be easily triggered with minimal code.

To link this parameter we need to create a transition from the Idle State animation to the Hit animation. To do this, right click the Idle animation, select 'Make transition' and then click on the Hit animation to draw an arrow between the two. Now select the newly created transition arrow and in the Inspector change the 'Exit Time' condition to be our newly created 'Hit' trigger. Note: Exit Time is only useful for ensuring that the current animation has played a certain percent of it's current animation however our Idle animation is empty so it's not important here and we can just transition whenever.

We need to make sure we don't get stuck playing the Hit animation forever so we have to add a second transition back to the Idle animation. Right click the Hit animation, choose 'Make transition' and select the Idle animation to create a second transition back. You will want to change the condition of this new transition so that the Exit Time value is at 1:00 so that it plays all of the animation before returning.



The animator is now set up, but we need to activate that trigger from code! The best place to trigger a damage animation is from our Damageable component! So uncomment the following line from the scripts 'showDamage' function:

```
GetComponent<Animator>().SetTrigger("Hit");
```

Now if you test the scene and attack Enemy1 you will see the animation play!

The best thing about this animation we've created is that it isn't tied specifically to Enemy1! We can use it for the other enemies and the player character too! In the inspector click the settings cog, next to the Animator component and choose 'Copy component'. Now select both Enemy2

and Player in the Hierarchy and then in the Inspector click the Settings cog next to the transform values and choose ‘Paste Component As New’. And now whenever any of the characters in the scene receives damage they will flash red to show this!

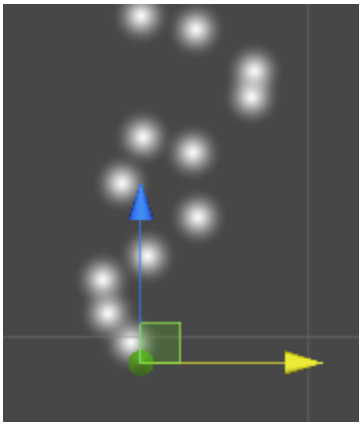
6. Set- up magic, add particle effects.

Unity has some built in demos of particle systems that are great for creating realistic effects such as fire, bubbles, water and smoke. To Import these examples select from the Menu Bar -> Assets > Import Package > Particles and choose Import on the pop-up that appears. This will result in a folder being added to the Project View called ‘Standard Assets’. In this folder you will find various prefabs showcasing pre made particle systems, Take a look and see!

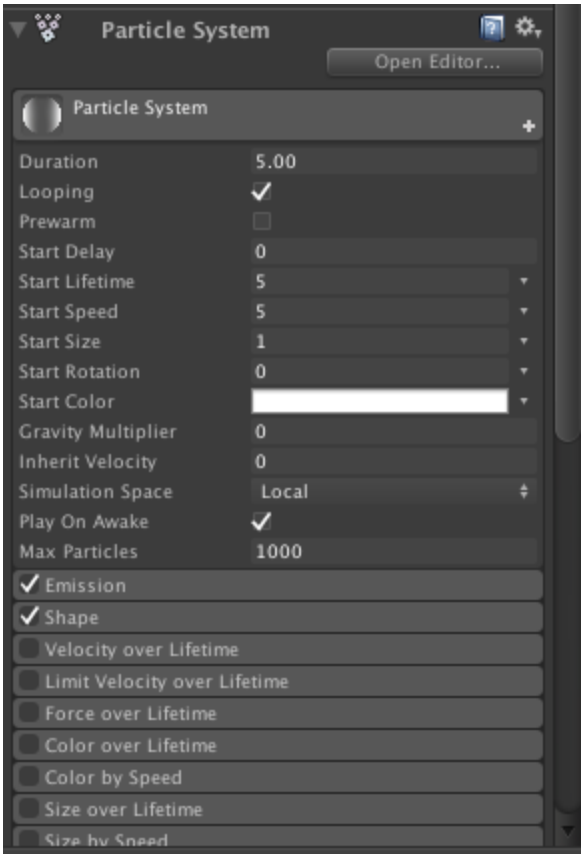
Now there are 2 downsides to these particles! Firstly the particles have been created using a legacy particle system of Unitys and so if you try to create a new particle system, it will work quite differently! Also the particle systems are designed to work in 3D systems and so currently don’t render in front of our 2D assets!! But fear not we are gonna use the new Shuriken Particle System for this tutorial!

To create a particle system choose GameObject > Create Other > Particle System

Position it off to the side of the game window now so that you can see your changes to it:



In the inspector you will now see many options for changing how the particle system behaves:



For the new particle system we're only going to look at a few of the properties as it would take many tutorials to cover them all! Try changing the following properties in the Inspector to create a cool looking Fire spell effect!

Renderer, Change the material to 'Fire Add'.

Shape, Change from Cone to Box, controls the location that particles are created from.

Start Lifetime, Change to 0.5 (how long they stay on screen)

Start Speed, Change to 0 so they don't move.

Clicking Stop and Simulate (In the scene view) will give you a good indication of how long the particles will last.

In order to view these particles with the 2D components of our game we need to add the Render3Don2DLayers script to the object so that it renders in front of the 2D assets.

We need to add a Timed Object Destructor script to the object to ensure that the particle system gets destroyed too. The default value for this is 1 second which matches the duration of the

system so it will be destroyed as soon as it finishes. (This script was included in the Standard Assets > Particles package that you imported.)

Now drag the particle system into the Project folder to turn it into a prefab.

Now let's duplicate the particle system and name it 'Sparkles', this will be for a heal animation.

Adjust the following properties:

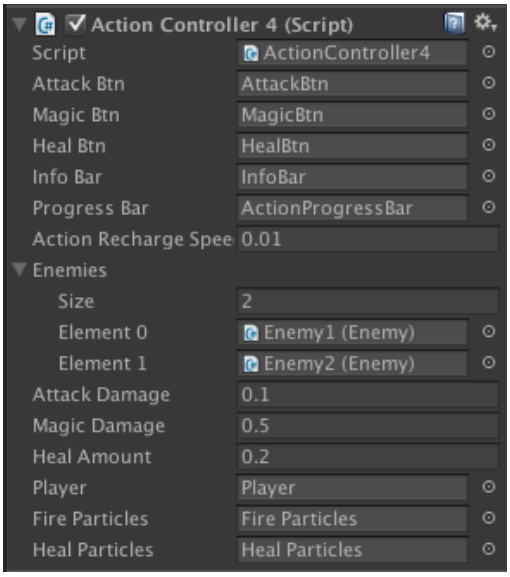
Renderer, Change the material to 'Sparkles 1'
Start Speed, Change to 0 .8

Shape, Scale the box to make it bigger

Start Colour, click the arrow next to colour and choose 'Random between two colours' and choose two colours to see the effect!

Make sure that you drag this Heal particle system into the Project folder to also turn it into a prefab and the n make sure that the Fire and Heal objects are deleted from the current scene.

To hook them up in our code change the ActionController to use the final ActionController4 and specify the new prefabs in the appropriate new fields:



Now when you test the game and choose Fire or Heal, you will see the particle system appear for the spell!

The code for this is really simple! The corresponding prefab just gets Instantiated when the action is chosen and positioned at the target that was clicked/player. The object then destroys itself after a second from the script we added to it earlier!

7. Finally just for some extra polish select the Main Camera and add the sound ‘battleThemeA’ in Assets/Sounds to it, giving us that authentic RPG battle feeling!