# Making a simple VFX system

The aim here is to produce a simple visual FX system that has the following features:

* It covers a range of visual effects, such as particles, color changes and so on, all in a data driven manner.
* Its fire-and-forget – you can say “Make this character flash blue”, “Show particles on their right hand for one second” and so on.

The VisualFX ScriptableObject handles all of this. When you create one you specify a prefab for it to instantiate and a color to tint it with. You can then add copies of this to a VFXHolder object, which will manage them and their lifetimes and expiry.

The VFXTest scene provided allows you to add Visual effects to a capsule using the keyboard. I, O, P, K, L and M will each add a different visual effect to the capsule via the throwaway component VFXTester, which is written to demonstrate the other classes.

Each VisualFX object instantiates a copy of its prefab when added to a VFXHolder. Its UpdateFX function is then called from the Update of the VFXHolder, and will eventually destroy it when its exceeded its duration.

We can manually stop an effect by setting its duration to one second after the current timer, giving it time to fade away.

The main trick here is that each VisualFX creates a GameObject. These GameObjects can have particle effects on them, but can also have anything else such as mesh geometry, projectors, or even a small custom component that changes the parent mesh’s color as a shader property.

## Why use VisualFX objects intead of just GameObjects?

It’s much easier to pick a VisualFX object in the editor than to pick a GameObject, as its screens out objects by type correctly.

It also means that two VisualFX can share a prefab with different customization data (here it’s the tint color).

## Making the Energy Sphere

Add a Unity Sphere, remove the collider, and give it a particles texture.

(Assets->Import Package ->Particles to get the textures.)

I added a small “Turner” script to rotate it

## Setting up Projectors

Setting up the projector effect was pretty straightforwards.

Assets->Import Package->Effects and import everything, and then add a Projector to the GameObject. The material selected has to use the special projector shaders. By default, the projector points along the z axis. In order to get it pointing down and from above I had to put it on a child of the main object and rotate it around X by 90 degrees and move it up.

To use a custom material, just change the texture in the Projector settings. Make sure the texture is set to Clamp rather than repeat for it to work properly.

If you don’t want the projector to affect the player, use layers to make it only project on the ground. Put the Player on a “Characters” layer and ignore that layer in the Projector.

## Toggle Effects vs Fixed Duration

You may want some effects to have a fixed duration ( a short burst of fire or a flash), and some to loop until you say otherwise (an aura, a shield etc)

VisualFX with a duration of zero do not count down and destroy themselves, but require an external piece of code to turn them off. In these cases, the client code has to store the VisualFX returned by VisualFXHolder.AddFX and call Stop on it when it’s time to deactivate.

## Object Pools

For performance, it may be best to set up Object pools and use them instead of instantiating and destroying GameObjects.

As always, profile before optimizing.

## Adding Audio

You can add Audio to the Visual FX system simply by placing AudioSource components on the prefabs used to create the visual effects.

This works OK as a first pass, but has a couple of quality issues.

### The sound stops abruptly when an effect wears off.

We can fix this by adding some fade out code for the sound to the VisualFX class

Store the Audio Source in the cloned visualfx object as a member variable

AudioSource soundInst;

Store it when we start an FX

soundInst = inst.GetComponent<AudioSource>();

And fade it out when we’re fading out particles and renderers

// fade out the sound

if (soundInst)

soundInst.volume = timeRemaining;

### The sound is the same every time

We can put together a simple sound randomiser script for doing this, and load it up with as many AudioClips as we like.

public class RandomSound : MonoBehaviour {

public AudioClip[] clips;

// Use this for initialization

void Start () {

PlayRandomClip();

}

public void PlayRandomClip()

{

if (clips.Length > 0)

{

int index = Random.Range(0, clips.Length);

AudioClip clip = clips[index];

AudioSource.PlayClipAtPoint(clip, transform.position);

}

}

}

Rather than bake the behavior into Start() I’ve put it into a separate function, so we can call it from events as well should we wish.