Defining Generic Events

Overview

Throughout development, we want to make life as easy for the designers as possible when it comes to defining actions that should occur in response to in-game events.

For example, instead of writing code to play a sound file whenever a certain action happens, we should just expose an event for that action, and then designers- through the editor- can hook in responses to that event.

This system- whatever form it takes- needs to be suitably applicable both through code AND the editor, so that programmers can hook into whatever events they like in the code without affecting or being affected by the designers.

How?

Exposing events to the Unity Editor is going to take a fair whack of GUI work, but it is definitely possible. Thankfully, hooking up via code is a much simpler affair that does not need any large setup on our end to use.

Creating Events

An event can be created directly through C# syntax:

```
public event System.EventHandler<GenericEventArguments> MyEvent;
```

In this instance, `System.EventHandler<GenericEventArguments>` can refer to any function that matches the `System.EventHandler` message signature.

To trigger the event, just call it like any old function:

```
MyEvent(this, new EmptyEventArguments());
```

Listening to Events via Code

To listen to this event, all you need to do is define an appropriate function:

```
private void HandleEvent(object sender, GenericEventArguments args)
{
    //Do something
}
```

And then hook your function into the exposed event, such as the following:

```
GetComponent<EventComponent>().MyEvent += HandleEvent;
```

It's really as simple as that!

Wait, What is GenericEventArguments?

GenericEventArguments refers to an abstract base-class that allows arguments to be specified and set for an event in a way such that they can

be utilised by the events system without it having to know about each and every event arguments class. A default implementation of this class exists, called `EmptyEventArguments`, which contains no settable parameters.

If you want to pass additional information in your event, you need to define a class that inherits off of it, specifies all of its arguments, and sets them as appropriate:

```
/// <summary>
/// Collision event arguements that exposes the BlobCollision object
/// </summary>
public class CollisionEventArguments : GenericEventArguments
{
    // Valid argument names
    public const string BlobCollisionName = "BlobCollision";
    // The blob collision object of the collision
    public BlobCollision BlobCollision { get{ return
GetArgument<BlobCollision>(BlobCollisionName); } }
    /// <summary>
    /// Returns a mapping of valid argument names to their valid types.
    /// </summary>
    /// <returns></returns>
    public override Dictionary<string,System.Type> GetValidArguments()
    {
      return new Dictionary<string,System.Type>()
      {
           {BlobCollisionName, typeof(BlobCollision)}
      };
    }
    /// <summary>
    /// Initilises a collision event with the collision data
    /// </summary>
    /// <param name="blobCollision">BlobCollision object of the collision</param>
    public CollisionEventArguments(BlobCollision blobCollision)
    {
        SetArgument(BlobCollisionName, blobCollision);
    }
}
```

And then of course include that info when you fire off the event:

```
if (MyEvent != null) MyEvent(this, new CollisionEventArguments (collision));
```

> It is important to null-check your events before calling them, because in C#, an event with no listeners has the value `null`. If you don't null-check it, you're very likely to get some unwanted exceptions in your code!

Here's the best bit: **Any listener function that accepts GenericEventArguments will be able to accept any subclasses of that, meaning functions that don't need the additional information can still hook into that event as if it gave them nothing**.

Listening to Events via the Editor

See here for more info.