Three ways to architect your game with ScriptableObjects

unity.com/how-to/architect-game-code-scriptable-objects

What you will get from this page: Tips for how to keep your game code easy to change and debug by architecting it with Scriptable Objects.

These tips come from Ryan Hipple, principal engineer at Schell Games, who has advanced experience using Scriptable Objects to architect games. You can watch Ryan's Unite talk on Scriptable Objects here; we also recommend you see Unity engineer Richard Fine's session for a great introduction to Scriptable Objects. Thank you Ryan!

What are ScriptableObjects?

ScriptableObject is a serializable Unity class that allows you to store large quantities of shared data independent from script instances. Using ScriptableObjects makes it easier to manage changes and debugging. You can build in a level of flexible communication between the different systems in your game, so that it's more manageable to change and adapt them throughout production, as well as reuse components.

Three pillars of game engineering

Use modular design:

- Avoid creating systems that are directly dependent on each other. For example, an inventory system should be able to communicate with other systems in your game, but you don't want to create a hard reference between them, because it makes it difficult to re-assemble systems into different configurations and relationships.
- Create scenes as clean slates: avoid having transient data existing between your scenes. Every time you hit a scene, it should be a clean break and load. This allows you to have scenes that have unique behavior that was not present in other scenes, without having to do a hack.
- Set up Prefabs so that they work on their own. As much as possible, every single prefab that you drag into a scene should have its functionality contained inside it. This helps a lot with source control with bigger teams, wherein scenes are a list of prefabs and your prefabs contain the individual functionality. That way, most of your check-ins are at the prefab level, which results in fewer conflicts in the scene.
- Focus each component on solving a single problem. This makes it easier to piece multiple components together to build something new.

Make it easy to change and edit parts:

- Make as much of your game as data-driven as possible. When you design your game systems to be like machines that process data as instructions, you can make changes to the game efficiently, even when it's running.
- If your systems are set up to be as modular and component-based as possible, it makes it easier to edit them, including for your artists and designers. If designers are able to piece things together in the game without having to ask for an explicit feature – largely thanks to implementing tiny components that each do one thing only – they can potentially combine such components in different ways to find new gameplay/mechanics, Ryan says that some of the coolest features his team has worked on in their games come from this process, what he calls "emergent design'".
- It's crucial that your team can make changes to the game at runtime. The more you can change your game at runtime, the more you can find balance and values, and, if you're able to save your runtime state back out like Scriptable Objects do, you're in a great place.

Make it easy to debug:

This one is really a sub-pillar to the first two. The more modular your game is, the easier it is to test any single piece of it. The more editable your game is – the more features in it that have their own Inspector view – the easier it is to debug. Make sure you can view debug state in the Inspector and never consider a feature complete until you have some plan for how you're going to debug it.

Architect for Variables

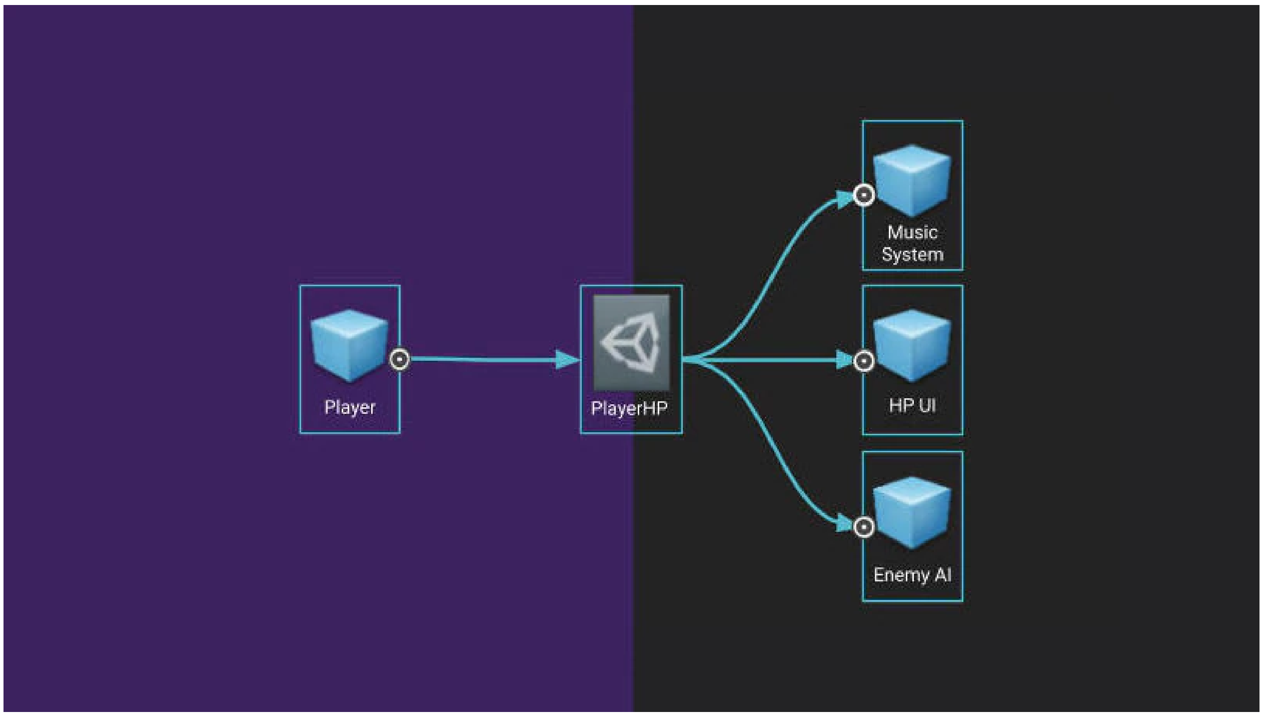
One of the simplest things you can build with ScriptableObjects is a self-contained, asset-based variable. Below is an example for a FloatVariable but this expands to any other serializable type as well.

Everyone your team, no matter how technical, can define a new game variable by creating a new FloatVariable asset. Any MonoBehaviour or ScriptableObject can use a public FloatVariable rather than a public float in order to reference this new shared value.

Even better, if one MonoBehaviour changes the Value of a FloatVariable, other MonoBehaviours can see that change. This creates a messaging layer between systems that do not need references to each other.

FloatVariable.cs (C#)

```
[CreateAssetMenu]
public class FloatVariable : ScriptableObject
{
        public float Value;
}
```



Example: Player’s Health Points

An example use case for this is a player’s health points (HP). In a game with a single local player, the player’s HP can be a FloatVariable named PlayerHP. When the player takes damage it subtracts from PlayerHP and when the player heals it adds to PlayerHP.

Now imagine a health bar Prefab in the scene. The health bar monitors the PlayerHP variable to update its display. Without any code changes it could easily point to something different like a PlayerMP variable. The health bar does not know anything about the player in the scene, it just reads from the same variable that the player writes to.

Once we are set up like this it’s easy to add more things to watch the PlayerHP. The music system can change as the PlayerHP gets low, enemies can change their attack patterns when they know the player is weak, screen-space effects can emphasize the danger of the next attack, and so on. The key here is that the Player script does not send messages to these systems and these systems do not need to know about the player GameObject. You can also go in the Inspector when the game is running and change the value of PlayerHP to test things.

When editing the Value of a FloatVariable, it may be a good idea to copy your data into a runtime value to not change the value stored on disk for the ScriptableObject. If you do this, MonoBehaviours should access RuntimeValue to prevent editing the InitialValue that is saved to disk.

RuntimeValue.cs (C#)

```
[CreateAssetMenu]
public class FloatVariable : ScriptableObject, ISerializationCallbackReceiver
{
        public float InitialValue;

        [NonSerialized]
        public float RuntimeValue;

public void OnAfterDeserialize()
{
                RuntimeValue = InitialValue;
}

public void OnBeforeSerialize() { }
}
```

Architect for Events

One of Ryan's favorite features to build on top of ScriptableObjects is an Event system. Event architectures help modularize your code by sending messages between systems that do not directly know about each other. They allow things to respond to a change in state without constantly monitoring it in an update loop.

The following code examples come from an Event system that consists of two parts: a GameEvent ScriptableObject and a GameEventListener MonoBehaviour. Designers can create any number of GameEvents in the project to represent important messages that can be sent. A GameEventListener waits for a specific GameEvent to be raised and responds by invoking a UnityEvent (which is not a true event, but more of a serialized function call).

Code example: GameEvent ScriptableObject

GameEvent ScriptableObject:

GameEvent ScriptableObject.cs (C#)

```
[CreateAssetMenu]
public class GameEvent : ScriptableObject
{
        private List<GameEventListener> listeners =
                new List<GameEventListener>();

public void Raise()
{
        for(int i = listeners.Count -1; i >= 0; i--)
listeners[i].OnEventRaised();
}

public void RegisterListener(GameEventListener listener)
{ listeners.Add(listener); }

public void UnregisterListener(GameEventListener listener)
{ listeners.Remove(listener); }
}
```

Code example: GameEventListener

GameEventListener:

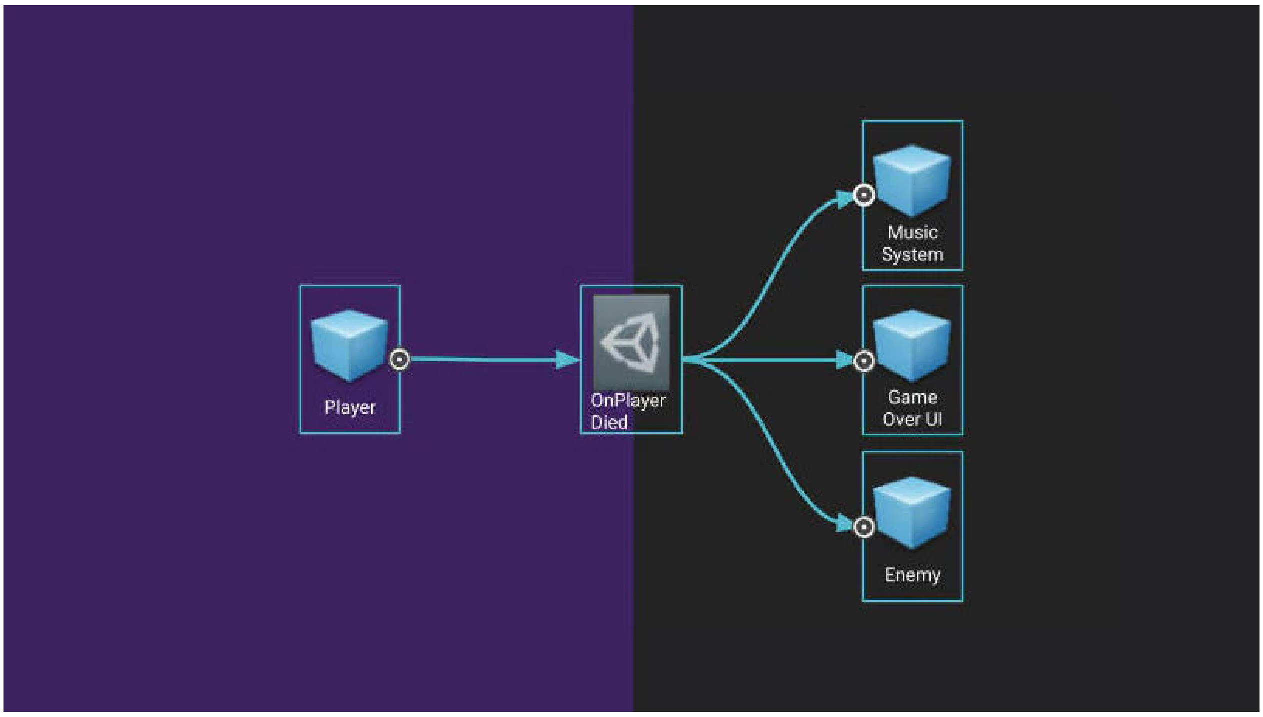
GameEventListener.cs (C#)

```
public class GameEventListener : MonoBehaviour
{
public GameEvent Event;
public UnityEvent Response;

private void OnEnable()
{ Event.RegisterListener(this); }

private void OnDisable()
{ Event.UnregisterListener(this); }

public void OnEventRaised()
{ Response.Invoke(); }
}
```



Event system that handles Player death

An example of this is handling player death in a game. This is a point where so much of the execution can change but it can be difficult to determine where to code all of the logic. Should the Player script trigger the Game Over UI or a music change? Should enemies check every frame if the player is still alive? An Event System lets us avoid problematic dependencies like these.

When the player dies, the Player script calls Raise on the OnPlayerDied event. The Player script does not need to know what systems care about it since it is just a broadcast. The Game Over UI is listening to the OnPlayerDied event and starts to animate in, a camera script can listen for it and start fading to black, and a music system can respond with a change in music. We can have each enemy listening for OnPlayerDied as well, triggering a taunt animation or a state change to go back to an idle behavior.

This pattern makes it incredibly easy to add new responses to player death. Additionally, it is easy to test the response to player death by calling Raise on the event from some testing code or a button in the Inspector.

The event system they built at Schell Games has grown into something much more complex and has features to allow for passing data and auto-generating types. This example was essentially the starting point for what they use today.

Architect for other Systems

Scriptable Objects don't have to be just data. Take any system you implement in a MonoBehaviour and see if you can move the implementation into a ScriptableObject instead. Instead of having an InventoryManager on a DontDestroyOnLoad

MonoBehaviour, try putting it on a ScriptableObject.

Since it is not tied to the scene, it does not have a Transform and does not get the Update functions but it will maintain state between scene loads without any special initialization. Instead of a singleton, use a public reference to your inventory system object when you need a script to access the inventory. This makes it easy to swap in a test inventory or a tutorial inventory than if you were using a singleton.

Here you can imagine a Player script taking a reference to the Inventory system. When the player spawns, it can ask the Inventory for all owned objects and spawn any equipment. The equip UI can also reference the Inventory and loop through the items to determine what to draw.