Game Object Dependencies and Intercommunication

Jeff Wilson

Member Variables

- Just assign member variables that reference other objects
- Unity: If public, can be wired up in the Inspector
  - View private vars by setting Unity's Inspector to *Debug*
- Initially easy to implement
- Declarative aspects
- Downsides:
  - Confusing interdependencies as game grows
  - Painful to rewire (perhaps if a prefab used in a new scene)

Managers/Singletons

- **Unity's recommended design pattern**
- Singleton – only one
- Singletons can be accessed globally, often via static class variable.
  - GameManager.instance
  - (No "wiring" needed)
- Watch out for abusing singleton pattern. Not always desirable to force yourself to only have one!

https://unity3d.com/learn/tutorials/projects/2d-roguelike-tutorial/writing-game-manager

C# Singleton – Singleton.Instance

See: http://wiki.unity3d.com/index.php/Singleton

```
public class EventManager : MonoBehaviour {
 private static EventManager eventManager;
 public static EventManager instance {
  get {
   if (!eventManager) {
    eventManager = FindObjectOfType (typeof(EventManager)) as EventManager;
    if (!eventManager) {
     Debug.LogError ("There needs to be one active EventManger script on a GameObject in your scene.");
    } else {eventManager.Init (); }
   } return eventManager;
}}
```

From EventManager:

```
public class EventManager : MonoBehaviour {
 private static EventManager eventManager;
 public static EventManager instance {
  get {
   if (!eventManager) {
    eventManager = FindObjectOfType (typeof(EventManager)) as EventManager;
    if (!eventManager) {
     Debug.LogError ("There needs to be one active EventManger script on a
GameObject in your scene.");
    } else {eventManager.Init (); }
   } return eventManager;
}}
```

Dependency Injection (DI)

- "Inversion of Control" – A class should not configure its own external dependencies. Instead dependencies should be configured externally
- Dependency Resolver (DR) wires up references to Classes/Interfaces
- Can couple DI with Manager/Singleton Pattern
- DR can even instantiate dependencies as needed (possibly lazily)
- DR may need to:
  - Resolve ambiguity (or notify developer to provide specificity in config)
  - Resolve circular dependencies
- Custom constructors or Attributes/Annotations used to denote member variables that need injection
  - C#: "[Inject]" or Java: "@Inject"

https://github.com/modesttree/Zenject

DI Benefits

- Can reduce spaghetti dependency complexity
- Centralized declarative configuration
- Can easily duplicate configuration or make multiple related configs
- Great for Unit and Integration testing! Test framework injects mocks instead of “real” interface providers
- Avoid limitations of singleton pattern

Unity – Dependency Injection

- Zenject - https://github.com/modesttree/Zenject
- Lightweight dependency injection framework for Unity

1 - Constructor Injection

```
public class Foo
{
    IBar _bar;

    public Foo(IBar bar)
    {
        _bar = bar;
    }
}
```

Declarative Specification of Dependencies in Script:

```
Container.Bind<Foo>().AsSingle();
Container.Bind<IBar>().To<Bar>().AsSingle();
```

This tells Zenject that every class that requires a dependency of type Foo should use the same instance, which it will automatically create when needed. And similarly, any class that requires the IBar interface (like Foo) will be given the same instance of type Bar.

Factory Pattern

- How to inject dependencies on dynamically created objects?
- Factory pattern ideal for DI of dynamically created game objects
- Factory is wired up by DR upon initialization
- Factory instantiate method used instead of conventional creation
- Factory wires up objects as needed
- Memory pool / Object recycling potential

DR – Dependency Resolver

Zenject Factory Example

```
class FooFactory : IFactory<Foo>
{
    public Foo Create()
    {
        // ...
        return new Foo();
    }
}

Container.Bind<Foo>().FromFactory<FooFactory>()
```

Observer Pattern

- Subject/Observer
- Subject maintains observer list
- Subject automatically notifies observers
- Must watch out for memory leaks (observer list refs block garbage collection/cleanup)
- Unity: Reflection (introspection) based implementation

Reflection: analyze class/object structure at runtime and change structure on the fly

Unity typically does this at startup/instantiation. Move towards avoiding inefficient reflection during Update()s.

Messaging/Event Driven Architecture

- Often a specialized case of Observer Pattern
- Message object implementation (event)
- Emitter/Consumer + Event Channel (EC) + Event Type
- EC interface can be interchangeable (e.g. local memory OR network socket)
- EC can defer or filter events
- Asynch programming can impose complications on EC (e.g. overflow, timeouts, priorities, peek(), etc.)
- Often event payload should be immutable
- Serializable events for networking or interprocess communication (IPC)
  - Complex data structures can be difficult, especially with pointers/references

Unity EventManager Singleton Example

- Decouple emitter and consumers
- Based on C# Delegate Events
- Broadcasts but not async
- EventManager example could be modified to not be a Singleton and then you could have different specialized EventManagers

Example: Custom Event Type

```
//Specifies parameter of Vector3 for collision position
public class BoxAudioEvent: UnityEvent<Vector3>{}

//No logic necessary. We just need an event type declared and define
what the payload is
```

Example: Listener

```
//Declare listener (must match param payload of event)
private UnityAction<Vector3> boxAudioEventListener;

//Instantiate with ref to callback function
boxAudioEventListener = new UnityAction<Vector3> (boxAudioEventHandler);

//The callback func
void boxAudioEventHandler(Vector3 worldPos) {AudioSource.PlayClipAtPoint (boxAudioClip, worldPos);}

//Start Listening
EventManager.StartListening<BoxAudioEvent, Vector3> (boxAudioEventListener);

//Stop Listening – Must stop listening if the listener is destroyed!!!
EventManager.StopListening<BoxAudioEvent, Vector3> (boxAudioEventListener);
```

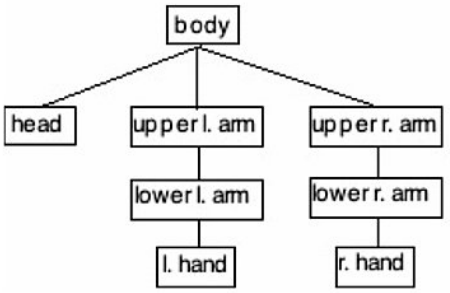
Example: Emitter

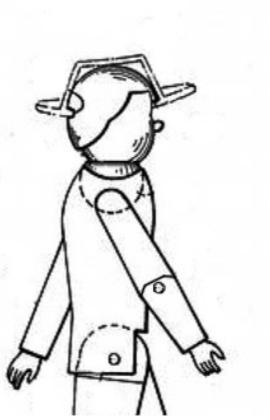
```
//The call
EventManager.TriggerEvent<AudioEventManager.BoxAudioEvent, Vector3> (contact.point);

//Example tied to physics collision:
void OnCollisionEnter(Collision c) {
  foreach (ContactPoint contact in c.contacts) {
    if (c.impulse.magnitude > THRESHOLD)
      EventManager.TriggerEvent<AudioEventManager.BoxAudioEvent, Vector3>(contact.point);
  }
}

//Emitter announces collision but doesn't know or care what any other game logic does with
  it!
```

Event Propagation

- Where in a hierarchy should events be handled? (e.g. a GUI with nested widgets/containers or scene graph)
- Sometimes useful to propagate events in a directed fashion for efficiency (e.g. UI events, raycasts, collisions, etc.)
- Unity: EventSystems.ExecuteEvents.Execute()/ExecuteHierarchy()
- Unity: ExecuteHierarchy() stops at first component to *implement* handler or in parent (no explicit handled/not handled return). If you don't want to stop propagation, you must call ExecuteHierarchy() again in the handler
- Unity: Don't use SendMessage()/BroadcastMessage() DEPRECATED (reflection implementation is inefficient)
- There is no ExecuteDownTheHierarchy(). Would be expensive!

Event Propagation in Hierarchy





Body might handle event for health loss on a collision, but hand might receive the initial collision event

Unity Recommendations

- GameManager singleton pattern to consolidate services
- Identify what functionality should be loosely coupled
  - E.g. You might want to manually wire up self-contained prefab hierarchies!
- EventManager approach when you want to notify any objects in a scene that are interested but can be loosely coupled
- EventSystems.ExecuteEvents for when events should be directed at specific game objects (e.g. user click events) or menu widgets
- Use Unity's conventional observer pattern when it makes sense (like physics collisions – OnCollisionXXX()). Generally for intra-object communication between components
- Mix if needed (e.g. OnCollisionEnter() calls EventSystems.ExecuteEvents.ExecuteHierarchy() to propagate up a complex multi-part GameObject with multiple rigid bodies/joints)
- Consider Dependency Injection Framework such as Zenject