Game Architecture

Alien Blaster

# Game State

// No singletons!

// Levels are scriptable objects that the game manager reads and executes.

# Entities

The word “entity” is a very common and often ambiguous term. To make things clear, an “entity” as used in this document is defined as: A Unity Game Object with behaviors and state attached to it. For example, an alien enemy is an entity, the player ship is an entity, the lasers are entities, etc. By definition, all entities are also Game Objects, but a lone Game Object with no behaviors attached to it is not an entity.

From this definition of an entity, the different aspects that define the different types of entities can be outlined. Entities mirror real-life as closely as possible, which means (at the most baseline level) there are three aspects that can completely define any entity:

* Attributes
  + Define how entities can be interacted with.
* Data
  + Define the state of the entity.
* Behaviors
  + Define ways that the entity interacts with the world.

Any entity can be created by composing some combination of these three Component types.

## Attributes

All Attribute class names shall be prefixed with a capital ‘A’.

Entities can only be interacted with via their Attributes. Attributes create a sort of public-facing interface through which the state of the entity can be altered. For instance, damage can only be dealt to entities with an ADamageable Attribute. The damage source would deal damage to the entity through its ADamageable Component, and the ADamageable would then apply that damage to the health of itself.

Outside influences shall never directly manipulate the Data of an entity. This is key because all Components have public method signatures that can technically be invoked by any caller. Data Components have exposed methods for manipulating the internal state, but only the entity itself should ever manipulate its own data. This is the reason for the existence of Attributes. Attributes invert the control for handling changes in state which keeps things decoupled as much as possible.

At this point, there is a very important distinction to be made about this design. The presence of Attributes does not and *should* not enforce the presence of any other Component in the Entity. Attributes are implicitly coupled with the Data they manipulate, but nothing about the implementation should assume that the Data is present. If the corresponding Data Component is missing from the entity, then the Attribute does nothing when an external influencer interacts with it.

Continuing with the ADamageable example, an entity can be damageable and simultaneously have no health. An example of a damageable entity without any health would be an indestructible obstacle such as passing asteroids. Weapons would still need to be blocked by obstacles like these, but with no health, nothing happens to the asteroid when a laser hits it.

## Data

All Data class names shall be prefixed with a capital ‘D’.

Data holds all the state of an entity. Health, position, a list of enemies to target, etc. Any stateful aspect to the entity is considered Data. Data shall not make any business decisions about the entity. Its only purpose exists so that other entity Components can make the decisions. Because of this, Data should only expose methods for manipulating its internal state, and nothing more.

It is highly recommended (though not required) that all Data Component implement an event system for changes in state. When the Data’s state changes, any component that needs to know about that should be notified. This event-based design ensures that polling for state is not needed. If nothing needs to react to changes in the Data, then omitting the event system is perfectly fine.

NOTE! Do not use Unity’s Component.BroadcastMessage() for this purpose. The UnityEvent framework (or something equivalent) should be used to ensure that only the entities that truly care about changes in state are notified. State will change constantly throughout the game, and with possibly thousands of entities all updating state every frame, using BroadcastMessage would be disastrous.

One last thing to remember about Data is that they should not depend on any other Component, either implicitly or explicitly. The purpose of Data is to be a single piece of state for an entity, and if a piece of Data depends on something else, it can most likely be broken out into separate Data. This is not to say that object inheritance cannot be used. Abstract classes and inheriting from them are perfectly acceptable. Instead, the restriction is that no member variables can reference any other Component.

## Behaviors

All Behavior class names shall be prefixed with a capital ‘B’.

Behaviors are the Components that drive entity actions. Anything that the entity does to interact with the world or with other entities is done through Behaviors. These are encapsulated chunks of business logic that are self-contained within the entity and cannot be interacted with. They have no public-facing methods.

Behaviors often depend on and react to changes in Data. For instance, if an entity should be destroyed on zero health, this is a Behavior. The Behavior BKillOnZeroHealth would subscribe to the DHealth Component’s Health Change Event, and when DHealth reports zero health, the Behavior will destroy itself and the entity it is attached to.

It is important to realize that, while Behaviors are Components attached to an entity, Behaviors are still disallowed from directly manipulating Data. Only Attributes can manipulate Data, and so Behaviors that wish to act upon their own entities must do so through their entity’s Attributes. This is an effort to enforce business logic rules on how Data is allowed to be manipulated.

One consequence of this architecture is the possibility of circular behavior. An example would be if a Behavior deals damage to itself in response to taking damage. This would cause an endless loop of dealing damage to itself. Care must be taken to ensure that any Behavior does not interact with Attributes that modify Data the Behavior itself depends on.

## C# Interfaces

All C# Interface names shall be prefixed with a capital ‘I’.

So far, everything mentioned above has been derived from MonoBehavior because Attributes, Data, and Behaviors are attached to Game Objects to create entities. It becomes necessary at times, however, to utilize other C# language features to our advantage. An Interface is an example of using the language features to describe the Components themselves, rather than the entities the Components create.

Where an Attribute itself can be thought of as a “public interface” through which something can interact with an entity, a C# Interface is used to guarantee functionality exists within a given Component. It is an error to use Interfaces in place of an Attribute; rather, Interfaces are used to denote logical groups of Components and have no restriction for whether the Component is an Attribute, Data, or Behavior.

Putting it another way: Attributes describe how you can interact with entities, and Interfaces describe how you can interact with the Components themselves. One example is the IResettable Interface. This guarantees implementation for resetting the state of a Component to some starting point. An *entity* is not resettable by having Components that implement IResettable. An entity is resettable if it has an AResettable Attribute. Implementation for AResettable would most likely look for all IResettable Components and call their Reset() methods.

The same rules about interacting with entity state apply. Just because a Data Component implements IResettable, it does not mean that any outside entity should ever invoke the Reset() method. Leave it to the Attributes to decide how and when IResettable Components are reset. If a particular entity does not implement an AResettable Attribute, it cannot be reset.

This example focused on IResettable, but all of this holds true for any Components that implement a C# Interface.

# Object Pooling

Object pooling is a concept used to reduce the performance hit when the engine must instantiate large numbers of entity copies. A great example of this performance issue is attack entities. Traditionally, if a group of enemies are all shooting at the player, each projectile entity must be instantiated by the engine and then subsequently destroyed when the laser collides with the player. With dozens of enemies all firing several times per second, the sheer amount of work the engine must do to handle all the instantiation and garbage collection is hugely inefficient.

An Object Pool solves this problem by instantiating all the copies of these entities before the level starts. Then, whenever a copy of that entity is needed, the instigator simply requests the Object Pool for the next available entity in the pool. With this, all the expensive operations are done outside of gameplay. The only overhead that is required is to activate/deactivate the entity, plus a small amount of queuing logic, which is vastly superior to the traditional approach.

## Implementation

Because all entities in the Scene need access to the Object Pools, there must be an instance of each Pool in the Scene as well. Instead of creating an Object Pool for each type of entity and dragging them all into the scene, there is a singular ObjectPoolMgr script that handles creation of the Object Pools. This script’s job is to give out references to the appropriate Object Pool to anything that needs it.

The ObjectPoolMgr script exposes two things: a list of Object Pool Prefabs to instantiate, and a public method for getting a reference to the appropriate Object Pool. Each Object Pool Prefab is then added to the singular Object Pool Manager Prefab. The Object Pool Manager Prefab, is what is dragged into the scene. All entities that need access to an Object Pool simply look for the instance of the Manager and request the correct one.

The reason why this approach is taken is simple. If each Object Pool had its own instance in the scene, then entities would be bound to the specific Object Pool instance it needed. This is completely unmaintainable because as soon as anyone changes an Object Pool instance, the references to every entity that uses it would need to be re-bound. The Object Pool Manager solves this by having a single thing in the scene for every entity to look for.

Of course, how does each entity know which Object Pool it wants then? Using Scriptable Objects, each Object Pool has a “type” associated with it. An empty Scriptable Object Asset is created for each kind of Object Pool, and the Object Pool Prefabs are bound to these types. When an entity needs a specific Object Pool, it asks the Manager for the type of Object Pool via these Assets. This decouples Object Pool instances from the entities that need them. The entities rely on a *type* of Object Pool rather than the Object Pool instance itself.

# Combat

## Damage

## Attacks

## Weapons

# Movement

## Scripted

## AI

# Player

// Discuss player state here

## Upgrades

# Naming Conventions