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 projectile 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 projectile 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 gameplay 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. Instead, the only overhead 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 of 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 added to the Object Pool Manager Prefab. A singular instance of 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 and Health

In many games, damage and health are represented by floating point values. This is not the case here. Damage and health are all unsigned integers. The decision to use unsigned integers over floats is simple: There is a built-in way to ensure non-negative values. It makes no sense for damage and health to be negative. What do “negative damage” even mean? Some might argue that negative damage counts as health gain, but I argue that it is easy to implement health gain as a positive number as well. Enforcing this arbitrary idea that “negative damage is just health gain” is not something I agree with.

There is another argument for the use of floating point values over integers: the range of these values. The maximum value for an unsigned 32-bit integer is 4.29 billion, which is significantly less than the max value for floating point values (which have ranges up to the hundreds of digits). I argue that there is never a time when 4.29 billion health and/or damage is not enough. Furthermore, I also argue that, while the range of floating point values is larger than integers, the loss of precision negates this advantage.

For these reasons, unsigned 32-bit integers are used for damage and health.

## Attacks

Attacks are the damage-dealing entities that exist in the Scene. Not all Attacks are projectiles, but many of them are. An example of a non-projectile Attack is the super laser. It is an Area Of Effect (AOE), Damage Over Time (DOT) attack where any entity that crosses its path is dealt damage continuously.

There are three different fundamental ways of dealing damage: Collision damage, AOE damage, and DOT damage. Collision damage sources deal damage to an entity when they collide (obviously). Examples include lasers, missiles, debris, etc. AOE damage is where damage is dealt to all entities within a range. The entities do not necessarily have to collide with the damage source for them to be affected. An example of this is a bomb attack where the detonation deals damage to all entities in a radius. Lastly, DOT damage is one where the damage is dealt to the targets on a repeating timer. This timer can fire every frame, or it can be more spread-out.

AOE and DOT damage is actually combined into one script called BAOEDmgOnTimer. This script exposes a list of targets and a timer, and it can be customized to fit both strict AOE or DOT, or it can be some combination of the two. For example, a proximity bomb can look for a change in the entities in range, and when it increases, start the timer with zero seconds to immediately detonate.

## Weapons

Weapons are a bit confusing. There is no such thing as a “Weapon” script. A Weapon is a Prefab with an arrangement of empty child Game Objects and a AFireable script. The empty child Game Objects serve as the spawn transforms for each attack that the Weapon will spawn. When the Fire() method is invoked on the Weapon, for each spawn transform, the Weapon gets a copy of the Attack entity from the associated Object Pool and sets its initial transform to that of the spawn.

Weapons do not have any meshes or other details in their Prefabs. The only things a Weapon has are the child spawn transforms and a AFireable script which keeps track of the spawn transforms and which Attack to use. Care must be taken to ensure that the Weapon’s spawn transforms do not overlap with the collider of the entity that owns the Weapon. If there is overlap, the Attacks could damage the instigator.

A Weapon can only have one Attack type to use. If different Attacks must be spawned with one press of the ‘fire’ button, there must be a different Weapon for each of those Attacks.

# Movement

## Player

Player movement is straightforward. The input (either keyboard or on-screen joystick) translates the ship using Unity’s Rigidbody physics handling. Instead of using physics forces, the MovePosition() method for Rigidbody entities is used. This means there is infinite acceleration, or in other words, the responsiveness of the controls is instantaneous. It gives the game the snappy, reactive feeling appropriate for weaving in and out of enemy shots.

## Scripted Behavior

Enemy movement is quite complicated. Every action an enemy takes (aside from mind-controlled enemies) is governed by what we call “Scripted Behavior.” In a sense, all actions are driven by a list of instructions akin to a movie script (hence the name). The result is perfectly repeatable movement patterns, attacks, etc. As per the GDD, every shot and every enemy must be in the same spot every playthrough with zero variation.

Additionally, since time can be “slowed down” for the enemy (again, per the GDD), there is a mechanism for playback through these action instructions. To facilitate this, several special Components exist. DObjectElapsedTime, ASeekable, and BScriptedBehavior. These three Components allow the use of Scripted Behavior Assets to create a fast and intuitive way for designers to quickly design levels. The functionality is described below.

### DObjectElapsedTime

The purpose of this Data Component is to keep track of the amount of time an entity has been active for. It exposes functionality to rewind, fast-forward, slow down, or pause “time” for an entity. You can think of this like the “seek handle” on media playback (the little dot you can click and drag with your mouse to seek through things like video or audio). The idea is to create an unambiguous way to have an entity always be doing the exact same thing at the exact same place every time. This Data Component facilitates that.

### ASeekable

Speaking of seeking through media, manipulating the entity’s elapsed time is done through the ASeekable Attribute. This Attribute is the public facing interface for how outside clients can interact with the entity’s elapsed time. Specifically, ASeekable allows clients to pause, rewind, fast forward, or slow down elapsed time for its associated entity.

One critical piece of logic contained in ASeekable is how multiple time manipulation requests are handled. The exposed methods have safeguards to ensure that a given operation finishes before the next one is processed. Additionally, the methods expose ways to override the current seeking operation. Essentially, ASeekable gives fine-grained control over the seeking operations, but it shoulders the responsibility of dealing directly with the data so that clients do not have to worry about it.

### BScriptedBehavior

BScriptedBehavior is responsible for executing the scripted actions. It exposes an ordered list where each list item is a Scripted Behavior Asset (see below) and a duration that Behavior should run for. This way, BScriptedBehavior just runs down the list of Scripted Behavior Assets and feeds the current elapsed time to the appropriate Asset. The Assets themselves do the heavy lifting in terms of moving the entity, shooting, etc. BScriptedBehavior is the Component that coordinates the effort.

### Scripted Behavior Assets

Utilizing Unity’s ScriptableObject type, functionality that is defined in a C# Script can be turned into Assets. These Assets can expose parameters that govern how the Scriptable Behavior should behave, and it allows it to be customizable by the designers without having to know code.

As an example, consider making an enemy move in a circle pattern. There is a ScriptableObject file called SOCircleMovement (the “SO” prefix stands for “Scriptable Object”). This file implements a method called ApplyTransform() that takes two arguments: the entity’s elapsed time and the entity’s transform. The position on the circle is calculated using Sine, Cosine, and the elapsed time. The mathematical result is then applied to the transform. SOScriptedMovement also exposes several parameters that affect the circle pattern’s properties. Specifically, it exposes radius, angular frequency, offset, and clockwise/counter-clockwise.

Making an Asset out of this Scripted Behavior allows the designer to drag it onto whichever enemy Prefab they want to move in a circle, and the BScriptedBehavior takes care of the rest.

// TODO: Put in UML for showing how things interact here.

## AI

# Player

// Discuss player state here

## Upgrades