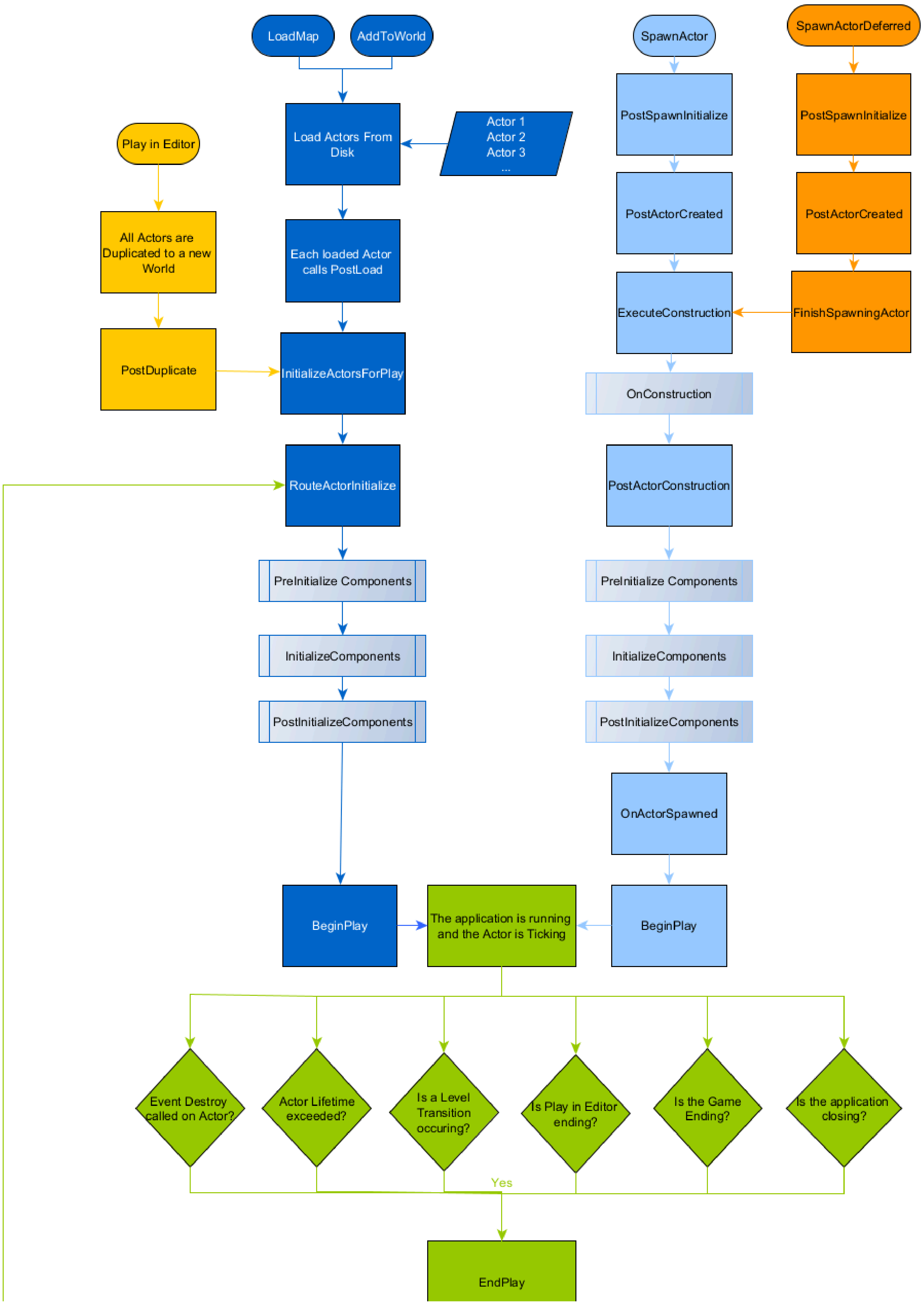
Actor Lifecycle

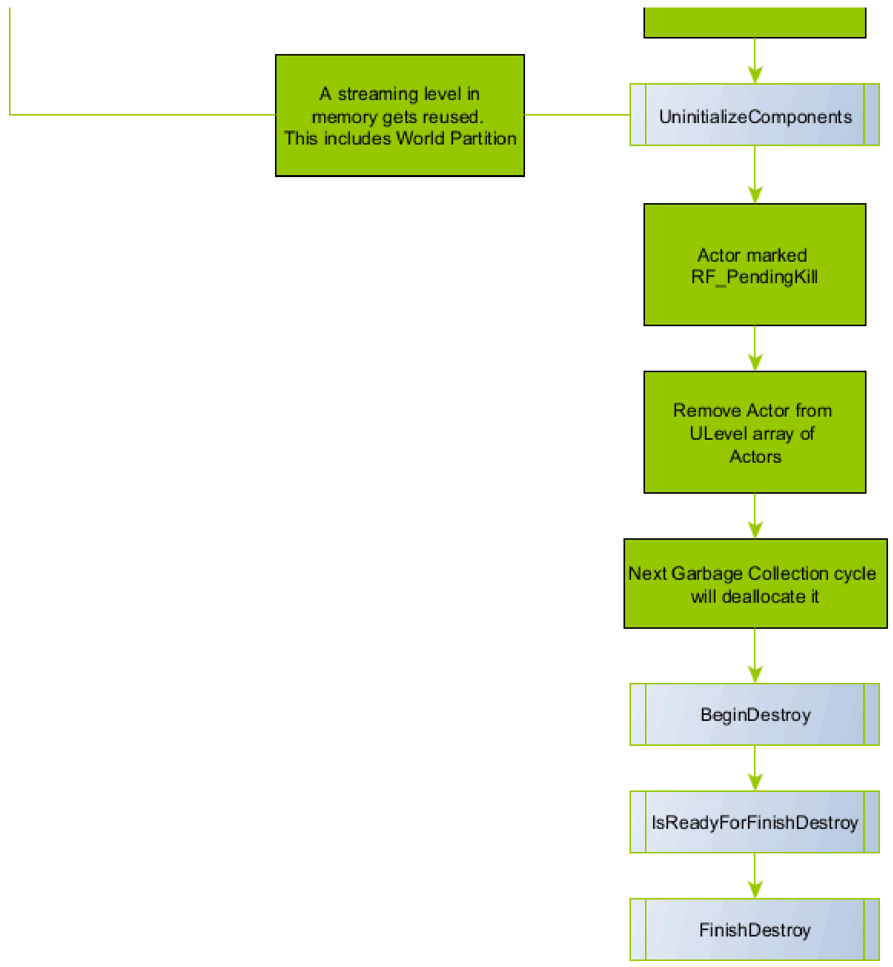
What actually happens when an Actor is loaded or spawned, and eventually dies.

This document is a high-level overview of the lifecycle of an Actor, which includes:

- How an Actor is instantiated or spawned into the level, including how the Actor is initialized.
- How an Actor is marked PendingKill and then removed or destroyed through Garbage Collection.
- The flow chart below shows the primary paths for how an Actor is instanced. No matter how an Actor is created, they all follow the same path to their destruction.

Lifecycle Break Down


LoadMap
AddToWorld
SpawnActor
SpawnActorDeferred
Load Actors From Disk
Actor 1
Actor 2
Actor 3
...
Play in Editor
PostSpawnInitialize
PostSpawnInitialize
PostActorCreated
PostActorCreated
All Actors are Duplicated to a new World
Each loaded Actor calls PostLoad
ExecuteConstruction
FinishSpawningActor
PostDuplicate
InitializeActorsForPlay
OnConstruction
RouteActorInitialize
PostActorConstruction
PreInitialize Components
PreInitialize Components
InitializeComponents
InitializeComponents
PostInitializeComponents
PostInitializeComponents
OnActorSpawned
BeginPlay
The application is running and the Actor is Ticking
BeginPlay
Event Destroy called on Actor?
Actor Lifetime exceeded?
Is a Level Transition occuring?
Is Play in Editor ending?
Is the Game Ending?
Is the application closing?
Yes
EndPlay



A streaming level in memory gets reused. This includes World Partition
UninitializeComponents
Actor marked RF_PendingKill
Remove Actor from ULevel array of Actors
Next Garbage Collection cycle will deallocate it
BeginDestroy
IsReadyForFinishDestroy
FinishDestroy

Load from Disk

The Load From Disk path occurs for any Actor that is already in the level, like when **UEngine::LoadMap** occurs, or when level streaming calls **UWorld::AddToWorld**.

1. Actors in a package/level are loaded from disk.
2. The serialized Actor calls **PostLoad** after it has finished loading from disk. Any custom versioning and fixup behavior should be implemented here. PostLoad is mutually exclusive with **AActor::PostActorCreated**.
3. The world calls **UAISystemBase::InitializeActorsForPlay** to prepare Actors to start gameplay.
4. The level calls **ULevel::RouteActorInitialize** for any non-initialized Actors and any seamless travel carry-over.
    a. **AActor::PreInitializeComponents** is called before InitializeComponent is called on the Actor's components.
    b. **UActorComponent::InitializeComponent** is a helper function for the creation of each component defined on the Actor.
    c. **AActor::PostInitializeComponents** is called after the Actor's components have been initialized.
5. **AActor::BeginPlay** is called when the level is started.

Play in Editor

In the Play in Editor path, Actors are copied from the Editor instead of being loaded from the disk. Then, the copied Actors initialize similarly to the flow described in the Load From Disk path.

1. Actors in the Editor are duplicated into a new World.
2. **UObject::PostDuplicate** is called.
3. **UAISystemBase::InitializeActorsForPlay**
4. **ULevel::RouteActorInitialize** for any non-initialized Actors and covers any seamless travel carry over.
    a. **AActor::PreInitializeComponents** is called before InitializeComponent is called on the Actor's components.
    b. **UActorComponent::InitializeComponent** is a helper function for the creation of each component defined on the Actor.

   c. **AActor::PostInitializeComponents** is called after the Actor's components have been initialized.
5. **AActor::BeginPlay** is called when the level is started.

Spawning

When you spawn an instance of an Actor, this is the path that will be followed:

1. **UWorld::SpawnActor** is called.
2. **AActor::PostSpawnInitialize** is called after the Actor is spawned in the world.
3. **AActor::PostActorCreated** is called for spawned Actors after its creation, any constructor implementation behavior should go here. PostActorCreated is mutually exclusive with PostLoad.
4. **AActor::ExecuteConstruction**:
5. **AActor::OnConstruction** - The construction of the Actor, this is where Blueprint Actors have their components created and Blueprint variables are initialized.
6. **AActor::PostActorConstruction**:
   a. **AActor::PreInitializeComponents** Called before InitializeComponent is called on the Actor's components.
   b. **UActorComponent::InitializeComponent** is a Helper function for the creation of each component defined on the Actor.
   c. **AActor::PostInitializeComponents** is called after the Actor's components have been initialized.
7. **UWorld::OnActorSpawned** is broadcast on UWorld.
8. **AActor::BeginPlay** is called.

Deferred Spawn

An Actor can be Deferred Spawned by having any properties set to "Expose on Spawn."

1. **UWorld::SpawnActorDeferred** is meant to spawn procedural Actors, allows additional setup before Blueprint construction script.
2. Everything in SpawnActor occurs, but after **AActor::PostActorCreated** the following occurs:

   a. Do setup and call various "initialization functions" with a valid but incomplete Actor instance.
   b. **AActor::FinishSpawning** is called to Finalize the Actor, picks up at **AActor::ExecuteConstruction** in the Spawn Actor line.

End of Actor Lifecycle

You can destroy Actors in several ways, but the way they are removed from the world follows the same method. During Gameplay, you can call the functions below, however, they are completely optional as many Actors are not actually destroyed during play (See Garbage Collection):

- **AActor::Destroy** is called manually by a game any time an Actor is meant to be removed, but gameplay is still occurring. The Actor is marked pending kill and removed from Level's array of Actors.
- **AActor::EndPlay** is called in several places to guarantee the life of the Actor is coming to an end. During play, Destroy calls this method and Level Transitions if a streaming level containing the Actor is unloaded.
- EndPlay is called from:
  - An explicit call to Destroy.
  - When Play in Editor has Ended.
  - Level Transitions (seamless travel or load map).
  - A streaming level containing the Actor is unloaded.
  - The lifetime of the Actor has expired.
  - Application shut down (All Actors are Destroyed).

Regardless of how this happens, the Actor will be marked `RF_PendingKill` so that UE de-allocates it from memory during the next garbage collection cycle. Also, rather than checking for pending kill manually, consider using an `FWeakObjectPtr<AActor>` as it is cleaner.

Actors may not necessarily be destroyed when their EndPlay is called. For example, if `s.ForceGCAfterLevelStreamedOut` is `false` and a sublevel is quickly reloaded then an Actor's EndPlay would be called, but the actor may be "resurrected" and would be the exact same actor that previously existed along with its local variables that were not re-initialized to its defaults

- **AActor::OnDestroyed** - This is a legacy response to Destroy. We recommend you move any logic here to EndPlay as it is called by level transition and other game cleanup functions.

Garbage Collection

Some time after an object has been marked for destruction, Garbage Collection will remove it from memory, freeing any resources it was using.

The following functions are called on the object during its destruction:

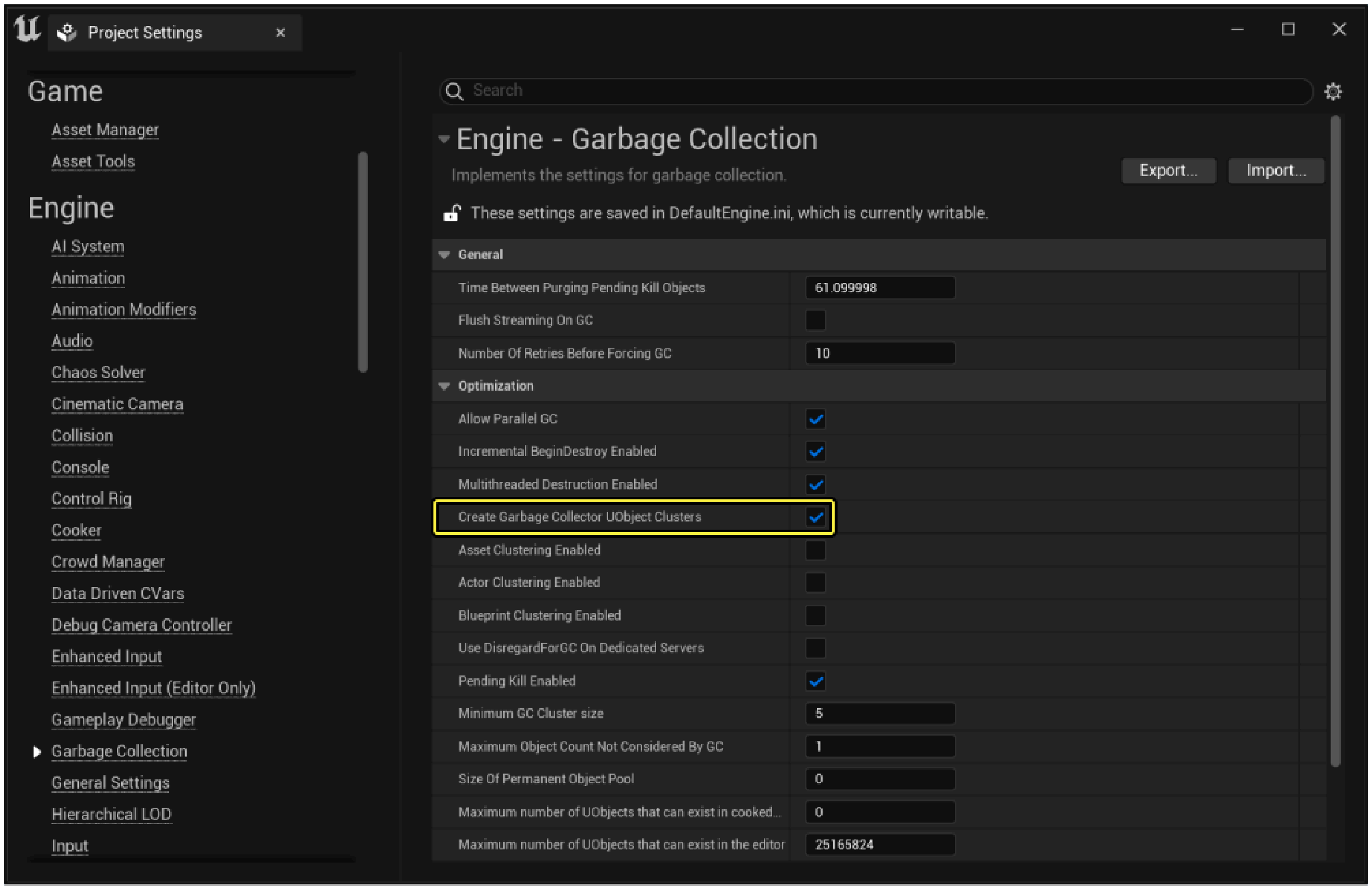
1. **UObject::BeginDestroy** - This is the object's chance to free up memory and handle other multithreaded resources (ie: graphics thread proxy objects). Most gameplay functionality related to being destroyed should have been handled earlier, in EndPlay.
2. **UObject::IsReadyForFinishDestroy** - The garbage collection process will call this function to determine whether or not the object is ready to be deallocated permanently. By returning false, this function can defer actual destruction of the object until the next garbage collection pass.
3. **UObject::FinishDestroy** - Finally, the object is really going to be destroyed, and this is another chance to free up internal data structures. This is the last call before memory is freed.

Advanced Garbage Collection

The Garbage Collection process in **Unreal Engine** builds clusters of objects that are all destroyed together. **Clustering** reduces the total time and overall memory churn associated with garbage collection compared to deleting objects indivudally. As an object loads, it may create subobjects. By combining the object and its subobjects into a single cluster for the garbage collector, the engine can delay freeing the resources used by the cluster until the entire object is ready to be freed, and can then free all of the resources at once.

Garbage collection does not need to be configured or modified at all for most projects, but there are some specific cases in which the "clustering" behavior of the garbage collector can be altered to improve efficiency in the following ways:

- **Clustering** - Turn clustering off. In **Project Settings**, under the **Garbage Collection** section, the **Create Garbage Collector UObject Clusters** option can be set to false. For most projects, this will result in less efficient garbage collection, so it is recommended in cases where performance testing reveals that it is beneficial.



Engine - Garbage Collection

Implements the settings for garbage collection.

These settings are saved in DefaultEngine.ini, which is currently writable.

General	
Time Between Purging Pending Kill Objects	61.099998
Flush Streaming On GC	☐
Number Of Retries Before Forcing GC	10
Optimization	
Allow Parallel GC	✓
Incremental BeginDestroy Enabled	✓
Multithreaded Destruction Enabled	✓
Create Garbage Collector UObject Clusters	✓
Asset Clustering Enabled	☐
Actor Clustering Enabled	☐
Blueprint Clustering Enabled	☐
Use DisregardForGC On Dedicated Servers	☐
Pending Kill Enabled	✓
Minimum GC Cluster size	5
Maximum Object Count Not Considered By GC	1
Size Of Permanent Object Pool	0
Maximum number of UObjects that can exist in cooked...	0
Maximum number of UObjects that can exist in the editor	25165824

Clustering options for Garbage Collection within the Project Settings menu.