Unreal Object Handling

Overview of the features of the UObject system.

Marking classes, properties, and functions with the appropriate macros turns them into `UClasses`, `UProperties`, and `UFunctions`. This gives Unreal Engine access to them, which allows for a number of under-the-hood handling features to be implemented.

Automatic Property Initialization

`UObjects` are automatically zeroed on initialization, before the constructor is called. This happens for the whole class, `UProperties` and native members alike. Members can subsequently be initialized with custom values in the class constructor.

Automatic Updating of References

When an `AActor` or `UActorComponent` is destroyed or otherwise removed from play, all references to it that are visible to the reflection system (`UProperty` pointers and pointers stored in Unreal Engine container classes such as `TArray`) are automatically nulled. This is beneficial in that it prevents dangling pointers from persisting and causing trouble down the road, but it also means that `AActor` and `UActorComponent` pointers can become null if some other piece of code destroys them. The ultimate advantage of this is that null-checking is more reliable, as it detects both standard-case null pointers and cases where a non-null pointer would have been pointing at deleted memory.

It is important to realize that this feature applies only to `UActorComponent` or `AActor` references marked with `UPROPERTY` or stored in an Unreal Engine container class. An Object reference stored in a raw pointer will be unknown to the Unreal Engine, and will not be automatically nulled, nor will it prevent garbage collection. Note this does not mean that all `UObject*` variables must be `UProperties`. If you want an Object pointer that is not a `UProperty`, consider using `TWeakObjectPtr`. This is a "weak" pointer, meaning it will not prevent garbage collection, but it can be queried for validity before being accessed and will be set to null if the Object it points to is destroyed.

Another case where a referenced UObject UProperty will be automatically null'ed is when using 'Force Delete' on an asset in the editor. As a result, all code operating on UObjects which are assets must handle these pointers becoming null.

Serialization

When a `UObject` is serialized, all `UProperty` values are automatically written or read unless explicitly marked as "transient" or unchanged from the post-constructor default value. For example, you could place an `AEnemy` instance in a level, set its Health to 500, save it and successfully reload it without writing a single line of code beyond the `UClass` definition.

When UProperties are added or removed, loading pre-existing content is handled seamlessly. New properties get default values copied from the new CDO. Removed properties are silently ignored.

If custom behavior is required, the `UObject::Serialize` function can be overridden. This can be useful to detect data errors, check version numbers, or perform automatic conversions or updates if the data format has changed.

Updating of Property Values

When the **Class Default Object** (or CDO) of a `UClass` has changed, the engine will try to apply those changes to all instances of the class when they are loaded. For a given Object instance, if the updated variable's value matches the value in the old CDO, it will be updated to the value it holds in the new CDO. If the variable has any other value, the assumption is that the value was set intentionally, and those changes will be preserved.

As an example, let us say you saved a level with several of your `AEnemy` Objects placed in it, and you had set the default Health value in the `AEnemy` constructor to 100. Let us also assume you set the health for Enemy_3 to 500, because they are particularly tough. Now imagine you changed your mind and increased the default value of Health to 150. When you next load your level, Unreal will realize you have changed the CDO and will update all instances of `AEnemy` with the old default Health value (100) to have a Health value of 150. Enemy_3's Health will remain at 500 because it wasn't using the old default value.

Editor Integration

`UObjects` and `UProperties` are understood by the Editor, and the Editor can expose these values automatically for editing without the need to write special code. This can optionally include integration into the Blueprint visual scripting system. There are many options to control the accessibility and exposure of variables and functions.

Run-Time Type Information and Casting

Because `UObjects` are part of the Unreal Engine's reflection system, they always know what `UClass` they are, and type-related decisions and casts can be made at runtime.

In native code, every `UObject` class has a custom `Super` typedef set to its parent class, which allows easy control of overriding behavior. As an example:

```
class AEnemy : public ACharacter
{
virtual void Speak()
{
Say("Time to fight!");
}
};

class AMegaBoss : public AEnemy
{
virtual void Speak()
{
Say("Powering up! ");
Super::Speak();
}
};

```

Copy full snippet

As you can see, calling `Speak` will result in the MegaBoss saying "Powering up! Time to fight!".

Also, you can safely cast an Object from a base class to a more derived class using the templated Cast function, or query if an Object is of a particular class using `IsA`. A quick example follows:

```
class ALegendaryWeapon : public AWeapon
{
void SlayMegaBoss()
{
TArray<AEnemy> EnemyList = GetEnemyListFromSomewhere();

// The legendary weapon is only effective against the MegaBoss
for (AEnemy Enemy : EnemyList)
{
AMegaBoss* MegaBoss = Cast<AMegaBoss>(Enemy);
if (MegaBoss)
{
Incinerate(MegaBoss);
}
}
}
};
```

Copy full snippet

Here we have used `Cast` to attempt to cast the `AEnemy` to an `AMegaBoss`. If the Object in question is not actually an `AMegaBoss` (or a child class thereof), the cast will return a null pointer and we can react appropriately. In the code above, the `Incinerate` function will only be called against the MegaBoss.

Garbage Collection

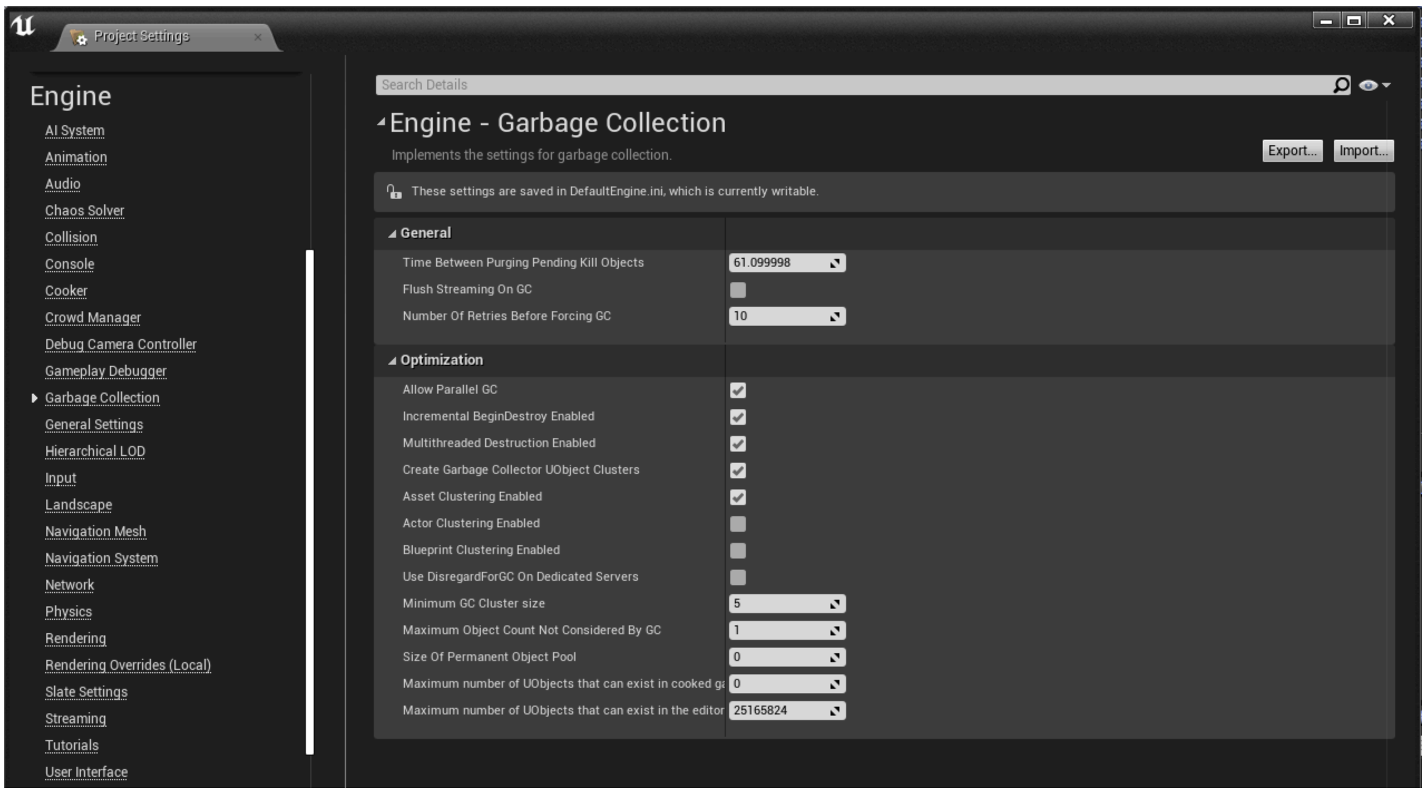
Unreal implements a garbage collection scheme whereby `UObjects` that are no longer referenced or have been explicitly flagged for destruction will be cleaned up at regular intervals. The engine builds a reference graph to determine which `UObjects` are still in use and which ones are orphaned. At the root of this graph is a set of `UObjects` designated as the "root set". Any `UObject` can be added to the root set. When garbage collection occurs, the engine can track all referenced `UObjects` by searching the tree of known `UObject` references, starting from the root set. Any unreferenced `UObjects`, meaning those which are not found in the tree search, will be assumed to be unneeded, and will be removed.

One practical implication here is that you typically need to maintain a `UPROPERTY` reference to any Object you wish to keep alive, whether it's a simple Object pointer or an Unreal Engine container class that contains Object pointer types, such as `TArray<UObject*>`. Actors and their Components are frequently an exception to this, since the Actors are usually referenced by an Object that links back to the root set, such as the Level to which they belong, and the Actor's Components are referenced by the Actor itself. Actors can be explicitly marked for destruction by calling their `Destroy` function, which is the standard way to remove an Actor from an in-progress game. Components can be destroyed explicitly with the `DestroyComponent` function, but they are usually destroyed when their owning Actor is removed from the game.

Garbage collection in Unreal Engine 4 is fast and efficient, and has a number of built-in features designed to minimize overhead, such as multithreaded reachability analysis to identify orphaned Objects, and unhashing code optimized to remove Actors from containers as quickly as possible. There are other features that can be adjusted to gain more precise control over how and when garbage collection is performed, most of which are found in **Project Settings** under **Engine - Garbage Collection**. The following settings are commonly used to tune garbage collector performance for a project:

Setting(s)	Feature Description
Create Garbage Collector UObject Clusters	This can be turned on or off (it is on by default) in the project's settings. If turned on, related Objects will be grouped together in a garbage collection cluster, so that only the cluster itself needs to be checked, rather than each individual Object. This means that reachability is performed faster because an entire cluster can be treated as one object, but it also means that individual items in that cluster will all be unhashed

Setting(s)	Feature Description
	and prepared for deletion in the same frame, potentially causing a hitch if the cluster is large enough. In the general case, cluster creation improves garbage collection performance and decreases time spent in reachability analysis.
Merge GC Clusters	Cluster merging can be enabled to cause clusters to join together when an object from one cluster references an object in another. Note that clearing the reference that caused the merge will not cause the newly-merged cluster to dissolve or break up in any way. **Create Garbage Collector UObject Clusters** must also be turned on for this feature to work. This will lead to the garbage collector unhashing and destroying objects less frequently, but larger numbers of objects will be unhashed and destroyed at once. In addition, there may be some cases where garbage collection will not happen with merged clusters when it would have happened without them, since any reference to any object in the cluster will keep the entire cluster from being garbage-collected.
Actor Clustering Enabled	Actors can be put into clusters by turning on this option in **Project Settings**, and by either setting the `bCanBeInCluster` variable to `true`, or overriding the `CanBeInCluster` function in code so that it returns `true`. By default, Actors and Components have this turned off, except for Static Mesh Actors and Reflection Capture Components. This feature is useful for grouping Actors that are expected to be destroyed all at once, usually static meshes placed in a level that cannot be destroyed except by unloading the sublevel that contains them.
Blueprint Clustering Enabled	A Blueprint's `UBlueprintGeneratedClass` and related data, such as the shared UPROPERTY and UFUNCTION data, can be clustered by turning this setting on. It is important to recognize that this clustering refers to the Blueprint Generated Class itself, not individual instances of the Blueprint.
Time Between Purging Pending Kill Objects	The frequency of garbage collection activity can be adjusted in the project's settings. This high-level control is especially useful in preventing hitches. By shortening the time between collections, you decrease the likely amount of unreachable objects that will be discovered on the next reachability analysis pass, and can avoid the hitches that can happen when a lot of Actors are cleaned up at the same time.



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	☐
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 g	0
Maximum number of UObjects that can exist in the editor	25165824

The Garbage Collection settings within Project Settings.

Network Replication

The `UObject` system includes a robust set of functionality to facilitate network communication and multiplayer games.

`UProperties` can be tagged to tell the Engine to replicate their data during network play. A common model here is that a variable gets changed on the server, and the Engine then detects this change and sends it reliably to all clients. Clients can optionally receive a callback function when the variable changes from replication.

`UFunctions` can also be tagged to execute on a remote machine. A "server" function, for example, when called on a client machine, will cause the function to execute on the server machine for the server's version of the Actor. A "client" function, on the other hand, can be called from the server and will run on the owning client's version of that Actor.