Actors

The basic gameplay elements, Actors and Objects

An **Actor** is any object that can be placed into a level, such as a Camera, static mesh, or player start location. Actors support 3D transformations such as translation, rotation, and scaling. They can be created (spawned) and destroyed through gameplay code (C++ or Blueprints).

In C++, `AActor` is the base class of all Actors.

Note that actors do not directly store Transform (Location, Rotation, and Scale) data; the Transform data of the Actor's Root Component, if one exists, is used instead.

Creating Actors

Creating new instances of `AActor` classes is called **spawning**. This can be done using the generic `SpawnActor()` function or one of its specialized templated versions.

See Spawning and Destroying an Actor for detailed info on the various methods of creating instances of `AActor` classes for gameplay.

Components

Actors can be thought of, in one sense, as containers that hold special types of **Objects** called Components. Different types of Components can be used to control how Actors move, how they are rendered, etc. The other main function of Actors is the replication of properties and function calls across the network during play.

Components are associated with their containing Actor when they are created.

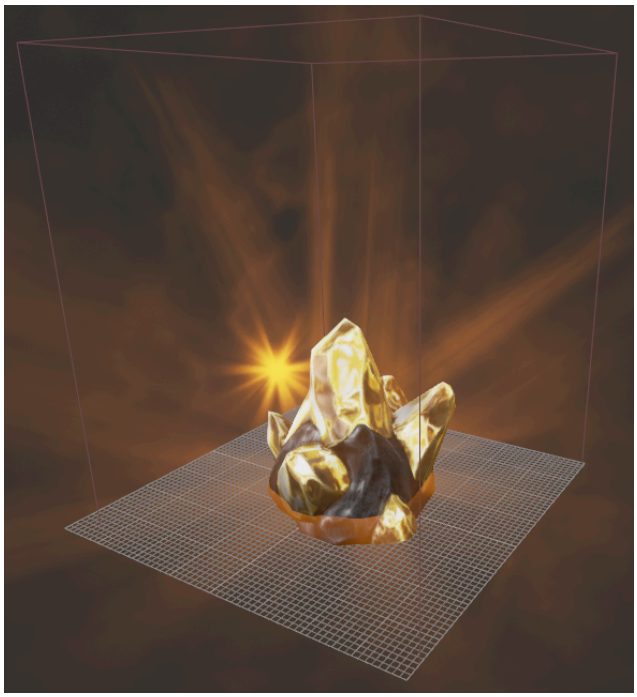
A few of the key types of Components are:

- **UActorComponent** : This is the base Component. It can be included as part of an Actor. It can Tick if you want it to. ActorComponents are associated with a specific Actor, but do not exist at any specific place in the world. They are generally used for conceptual functionality, like AI or interpreting player input.
- **USceneComponent** : SceneComponents are ActorComponents that have transforms. A transform is a position in the world, defined by location, rotation, and scale. SceneComponents can be attached to each other in a hierarchical fashion. An Actor's location, rotation, and scale are taken from the SceneComponent that is at the root of the hierarchy.
- **UPrimitiveComponent** : PrimitiveComponents are SceneComponents that have a graphical representation of some kind (e.g. a mesh or a particle system). Many of the interesting physics and collision settings are here.

Actors support having a hierarchy of SceneComponents. Each Actor also has a `RootComponent` property that designates which Component acts as the root for the Actor. Actors themselves do not have transforms, and thus do not have locations, rotations, or scales. Instead, they rely on the transforms of their Components; more specifically, their root Component. If this Component is a **SceneComponent**, it provides the transformation information for the Actor. Otherwise, the Actor will have no transform. Other attached Components have a transform relative to the Component they are attached to.

An example Actor and hierarchy might look something like this:

GoldPickup Actor	Hierarchy

	• **Root - SceneComponent**: Basic scene Component to set the Actor's base location in the world.
	  ◦ **StaticMeshComponent**: Mesh representing gold ore.
	  ▪ **ParticleSystemComponent**: Sparkling particle emitter attached to the gold ore.
	  ▪ **AudioComponent**: Looping metallic chiming audio emitter attached to the gold ore.
	  ▪ **BoxComponent**: Collision box to use as trigger for overlap event for picking up the gold.

Ticking

Ticking refers to how Actors are updated in Unreal Engine. All Actors have the ability to be ticked each frame, or at a minimum, user-defined interval, allowing you to perform any update calculations or actions that are necessary.

Actors all have the ability to be ticked by default via the `Tick()` function.

ActorComponents also have the ability to be updated by default, though they use the `TickComponent()` function to do so. See the Updating section of the Components page for more information.

Lifecycle

See the Actor Lifecycle documentation for more information on how an Actor is created and removed from the game.

Replication

Replication is used to keep the Actors within the world in sync when dealing with networked multiplayer games. Property values and function calls can both be replicated, allowing for complete control over the state of the game on all clients.

Destroying Actors

Actors are not generally garbage collected, as the World Object holds a list of Actor references. Actors can be explicitly destroyed by calling `Destroy()`. This removes them from the level and marks them as "pending kill", which means they will hang around until they are cleaned up on the next garbage collection.