Unreal Engine Terminology

Covers the most commonly used terms when working with Unreal Engine.

This page describes the most commonly used terms when working with **Unreal Engine**. If you find yourself wondering, "What is an **Actor**?" or "What is a **Component**?" this page will answer your questions, and more.

Once you understand a term, check the topics linked at the end of the section to learn more.

This page uses some programming concepts, notably **classes** and **subclasses**. In C++, a **class** is a code template that contains variables and behavior and can be extended. A **subclass** is a class that inherits some or all code and functionality from a parent class.

All of the C++ classes mentioned on this page are specific to Unreal Engine.

Project

An **Unreal Engine 5 Project** holds all the contents of your game. It contains a number of folders on your disk, such as `Blueprints` and `Materials`. You can name and organize folders inside a Project however you wish. The **Content Browser** panel inside the **Unreal Editor** shows the same directory structure found inside the `Project` folder on your disk.

Every project has a `.uproject` file associated with it. The `.uproject` file is how you create, open, or save a project. You can create any number of different projects and work on them in parallel.

For more information, see Projects and Templates.

Blueprint

The **Blueprint Visual Scripting** system is a complete gameplay scripting system that uses a node-based interface to create gameplay elements from within Unreal Editor. As with many common scripting languages, it is used to define object-oriented (OO) classes or objects in the engine. As you use Unreal Engine, you'll often find that objects defined using Blueprint are colloquially referred to as "Blueprints."

For more information, see Blueprints Visual Scripting.

Object

Objects are the most basic class in Unreal Engine - in other words, they act like building blocks and contain a lot of the essential functionality for your Assets. Almost everything in Unreal Engine inherits (or gets some functionality) from an Object.

In C++, `UObject` is the base class of all objects; it implements features such as garbage collections, metadata (`UProperty`) support for exposing variables to the Unreal Editor, and serialization for loading and saving.

For more information, see:

- Unreal Architecture

Class

A **Class** defines the behaviors and properties of a particular Actor or Object in Unreal Engine. Classes are hierarchical, meaning a Class inherits information from its parent Class (that is, the Class it was derived or "sub-classed" from) and passes that information to its children. Classes can be created in C++ code or in Blueprints.

For more information, see:

- Blueprint Class
- Gameplay Classes
- Class Creation Basics

Actor

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.

For more information, see:

- Actors
- Actors and Geometry

Casting

Casting is an action that takes an Actor of a specific class and tries to treat it as if it were of a different class. Casting can succeed or fail. If casting succeeds, you can then access class-specific functionality on the Actor you cast to.

For example, let's say you're making a game where you have multiple types of Volumes that can affect the player character in different ways. One of these volumes is **Fire**, which decreases player health over time. When the player overlaps with any Volume in the Level, you can **cast** that Volume to **Fire** to try to access its ""damage player health"" functionality.

- If the cast succeeds — that is, if the player is standing in the fire — the player's health starts to decrease.
- If the cast fails — that is, if the player is standing in any other kind of Volume — their health is not affected.

Casting is different from simply checking whether an Actor is of a given class, which would return a binary (yes or no) answer, but wouldn't allow you to interact with any specific functionality of that class.

Component

A **Component** is a piece of functionality that can be added to an Actor.

When you add a Component to an Actor, the Actor can use the functionality that the Component provides. For example:

- A Spot Light Component will make your Actor emit light like a spot light.
- A Rotating Movement Component will make your Actor spin around.
- An Audio Component will give your Actor the ability to play sounds.

Components must be attached to an Actor and can't exist by themselves.

For more information, see:

- Basic Components
- Components Window
- Components

Pawn

Pawns are a subclass of Actor and serve as an in-game avatar or persona (for example, the characters in a game). Pawns can be controlled by a player or by the game's AI, as non-player characters (NPCs).

When a Pawn is controlled by a human or AI player, it is considered to be *Possessed*. Conversely, when a Pawn is not controlled by a human or AI player, it is considered to be *Unpossessed*.

For more information, see:

- Pawn
- Possessing Pawns

Character

A **Character** is a subclass of a Pawn Actor that is intended to be used as a player character. The Character subclass includes a collision setup, input bindings for bipedal movement, and additional code for player-controlled movement.

For more information, see:

- Characters
- Setting Up a Character
- Setting Up Character Movement

Player Controller

A **Player Controller** takes player input and translates it into interactions in the game. Every game has at least one Player Controller in it. A Player Controller often possesses a Pawn or Character as a representation of the player in a game.

The Player Controller is also the primary network interaction point for multiplayer games. During multiplayer play, the server has one instance of a Player Controller for every player in the game since it must be able to make network function calls to each player. Each client only has the Player Controller that corresponds to their player and can only use their Player Controller to communicate with the server.

The associated C++ class is `PlayerController`.

For more information, see Player Controllers.

AI Controller

Just as the Player Controller possesses a Pawn as a representation of the player in a game, an **AI Controller** possesses a Pawn to represent a non-player character (NPC) in a game. By default, Pawns and Characters will end up with a base AI Controller unless they are specifically possessed by a Player Controller or told not to create an AI Controller for themselves.

The associated C++ class is `AIController`.

For more information, see AI Controllers.

Player State

A **Player State** is the state of a participant in the game, such as a human player or a bot that is simulating a player. Non-player AI that exists as part of the game world doesn't have a

Player State.

Some examples of player information that the Player State can contain include:

- Name
- Current level
- Health
- Score
- Whether they are currently carrying the flag in a Capture the Flag game.

For multiplayer games, Player States for all players exist on all machines and can replicate data from the server to the client to keep things in sync. This is different from a Player Controller, which will only exist on the machine of the player it represents.

The associated C++ class is `PlayerState`.

For more information, see Gameplay Framework Quick Reference.

Game Mode

The **Game Mode** sets the rules of the game that is being played. These rules can include:

- How players join the game.
- Whether or not a game can be paused.
- Any game-specific behavior such as win conditions.

You can set the default Game Mode in the Project Settings and override it for different Levels. Regardless of how you choose to implement it, you can only have one Game Mode for each Level.

In a multiplayer game, the Game Mode only exists on the server and the rules are replicated (sent) to each of the connected clients.

The associated C++ class is `GameMode`.

For more information, see:

- Game Mode and Game State.
- Setting Up a Game Mode

Game State

A **Game State** is a container that holds information you want replicated to every client in a game. In simpler terms, it is 'The State of the Game' for everyone connected.

Some examples of what the Game State can contain include:

- Information about the game score.
- Whether a match has started or not.
- How many AI characters to spawn based on the number of players in the world.

For multiplayer games, there is one local instance of the Game State on each player's machine. Local Game State instances get their updated information from the server's instance of the Game State.

The associated C++ class is `GameState`.

For more information, see Game Mode and Game State.

Brush

A **Brush** is an Actor that describes a 3D shape, such as a cube or a sphere. You can place brushes in a level to define level geometry (these are known as Binary Space Partition or BSP brushes). This is useful if you want to quickly block out a level, for example.

For more information, see:

- Geometry Brush Actors
- Content Examples

Volume

Volumes are bounded 3D spaces that have different uses based on the effects attached to them. For example:

- **Blocking Volumes** are invisible and used to prevent Actors from passing through them.
- **Pain Causing Volumes** cause damage over time to any Actor that overlaps them.
- **Trigger Volumes** are programmed to cause events when an Actor enters or exits them.

For more information, see Actors Reference.

Level

A **Level** is a gameplay area that you define. Levels contain everything a player can see and interact with, such as geometry, Pawns, and Actors.

Unreal Engine saves each level as a separate `.umap` file, which is why you will sometimes see them referred to as **Maps**.

For more information, see:

- Levels
- Level Editor

World

A **World** is a container for all the Levels that make up your game. It handles the streaming of Levels and the spawning (creation) of dynamic Actors.

For more information, see:

- World Settings
- Level Streaming