Game Mode and Game State

Overview of the Game Mode and Game State

There are two main classes which handle information about the game being played: **Game Mode** and **Game State**.

Even the most open-ended game has an underpinning of rules, and these rules make up a **Game Mode**. On the most basic level, these rules include:

- The number of players and spectators present, as well as the maximum number of players and spectators allowed.
- How players enter the game, which can include rules for selecting spawn locations and other spawn/respawn behavior.
- Whether or not the game can be paused, and how pausing the game is handled.
- Transitions between levels, including whether or not the game should start in cinematic mode.

When rule-related events in the game happen and need to be tracked and shared with all players, that information is stored and synced through the **Game State**. This information includes:

- How long the game has been running (including running time before the local player joined).
- When each individual player joined the game, and the current state of that player.

- The base class of the current Game Mode.
- Whether or not the game has begun.

Game Modes

While certain fundamentals, like the number of players required to play, or the method by which those players join the game, are common to many types of games, limitless rule variations are possible depending on the specific game you are developing. Regardless of what those rules are, Game Modes are designed to define and implement them. There are currently two commonly-used base classes for Game Modes.

Engine version 4.14 introduces `AGameModeBase`, which is the base class for all Game Modes and is a simplified and streamlined version of the classic `AGameMode`. `AGameMode`, the Game Mode base class before version 4.14, still exists and functions as it did before, but is now a child of `AGameModeBase`. `AGameMode` is more suited to standard game types like multiplayer shooters due to its implementation of the concept of match state. `AGameModeBase` is the new default game mode included in new code projects due to its simplicity and efficiency.

AGameModeBase

All Game Modes are subclasses of `AGameModeBase`, which contains considerable base functionality that can be overridden. Some of the common functions include:

Function/Event	Purpose
`InitGame`	The `InitGame` event is called before any other scripts (including `PreInitializeComponents`), and is used by `AGameModeBase` to initialize parameters and spawn its helper classes. This is called before any Actor runs`PreInitializeComponents`, including the Game Mode instance itself.

Function/Event	Purpose
`PreLogin`	Accepts or rejects a player who is attempting to join the server. Causes the `Login` function to fail if it sets `ErrorMessage` to a non-empty string. `PreLogin` is called before `Login`, and a significant amount of time may pass before Login is called, especially if the joining player needs to download game content.
`PostLogin`	Called after a successful login. This is the first place it is safe to call replicated functions on the `PlayerController`. `OnPostLogin` can be implemented in Blueprint to add extra logic.
`HandleStartingNewPlayer`	Called after `PostLogin` or after seamless travel, this can be overridden in Blueprint to change what happens to a new player. By default, it will create a pawn for the player.
`RestartPlayer`	This is called to start spawning a player's Pawn. There are also `RestartPlayerAtPlayerStart` and `RestartPlayerAtTransform` functions available if you want to dictate where the Pawn will be spawned. `OnRestartPlayer` can be implemented in Blueprint to add logic after this function is finished.
`SpawnDefaultPawnAtTransform`	This is what actually spawns the player's Pawn, and can be overridden in Blueprints.
`Logout`	Called when a player leaves the game or is destroyed. `OnLogout` can be implemented to do Blueprint logic.

A subclass of the `AGameModeBase` class may be created for each match format, mission type, or special zone that the game offers. A game may have any number of Game Modes, and thus subclasses of the `AGameModeBase` class; however, only one Game Mode may be in use at any given time. A Game Mode Actor is instantiated each time a level is initialized for play via the `UGameEngine::LoadMap()` function.

The Game Mode is not replicated to any remote clients that join in a multiplayer game; it exists only on the server, so local clients can see the stock Game Mode class (or Blueprint) that was used, but cannot access the actual instance and check its variables to see what has changed as the game progresses. If players do need updated information relating to the current Game Mode, that information is easily kept in sync by being stored on an `AGameStateBase` Actor, one of which will be created along with the Game Mode and then replicated out to all remote clients.

AGameMode

`AGameMode` is a subclass of `AGameModeBase` that has some extra functionality to support multiplayer matches and legacy behavior. All newly created projects use `AGameModeBase` by default, but you can switch to inheriting from `AGameMode` if you need this extra behavior. If you inherit from `AGameMode`, you should also inherit your game state from `AGameState`, which also supports the match state machine.

`AGameMode` contains a state machine that tracks the state of the match or the general gameplay flow. To query the current state, you can use `GetMatchState`, or wrappers like `HasMatchStarted`, `IsMatchInProgress`, and `HasMatchEnded`. Here are the possible match states:

- `EnteringMap` is the initial state. Actors are not yet ticking and the world has not been fully initialized. It will transition to the next state when things are fully loaded.
- `WaitingToStart` is the next state, and `HandleMatchIsWaitingToStart` is called when entering. Actors are ticking, but players have not yet spawned in. It transitions to the next state if `ReadyToStartMatch` returns *true*, or if `StartMatch` is called.
- `InProgress` is the state where the main part of the game will take place. `HandleMatchHasStarted` is called when entering it, which then calls `BeginPlay` on all Actors. At this point, normal gameplay is in progress. The match will transition to the next state when `ReadyToEndMatch` returns *true* or `EndMatch` is called.
- `WaitingPostMatch` is the second-to-last state, and `HandleMatchHasEnded` is called when entering it. Actors are still ticking, but new players cannot join. It transitions to the next state when a map transfer starts.
- `LeavingMap` is the last state in a normal flow, and calls `HandleLeavingMap` upon entry. The match stays in this state while transferring to a new map, where it will transition back to `EnteringMap`.

- `Aborted` is the failure state, and can be started by calling `AbortMatch`. This is set when there is an unrecoverable error.

The match state will almost always be `InProgress` since this is the state where `BeginPlay` is called and actors begin ticking. However, individual games can override the behavior of these states to build a multiplayer game with more complicated rules, such as permitting players to fly around the level freely while waiting for other players to join in a multiplayer shooter.

Game Mode Blueprints

It is possible to create Blueprints derived from Game Mode classes, and use these as the default Game Mode for your project or level.

Blueprints derived from Game Modes can set the following defaults:

- Default Pawn Class
- HUD Class
- Player Controllers Class
- Spectator Class
- Game State Class
- Player State Class

In addition, Blueprints of Game Modes are very useful because they enable adjustment of variables without altering code, and can therefore be used to adapt a single Game Mode to multiple different levels without using hard-coded asset references or requiring engineering support and code changes for every tweak.

Setting the Game Mode

There are several methods to set the Game Mode for a level, ordered here from lowest priority to highest priority:

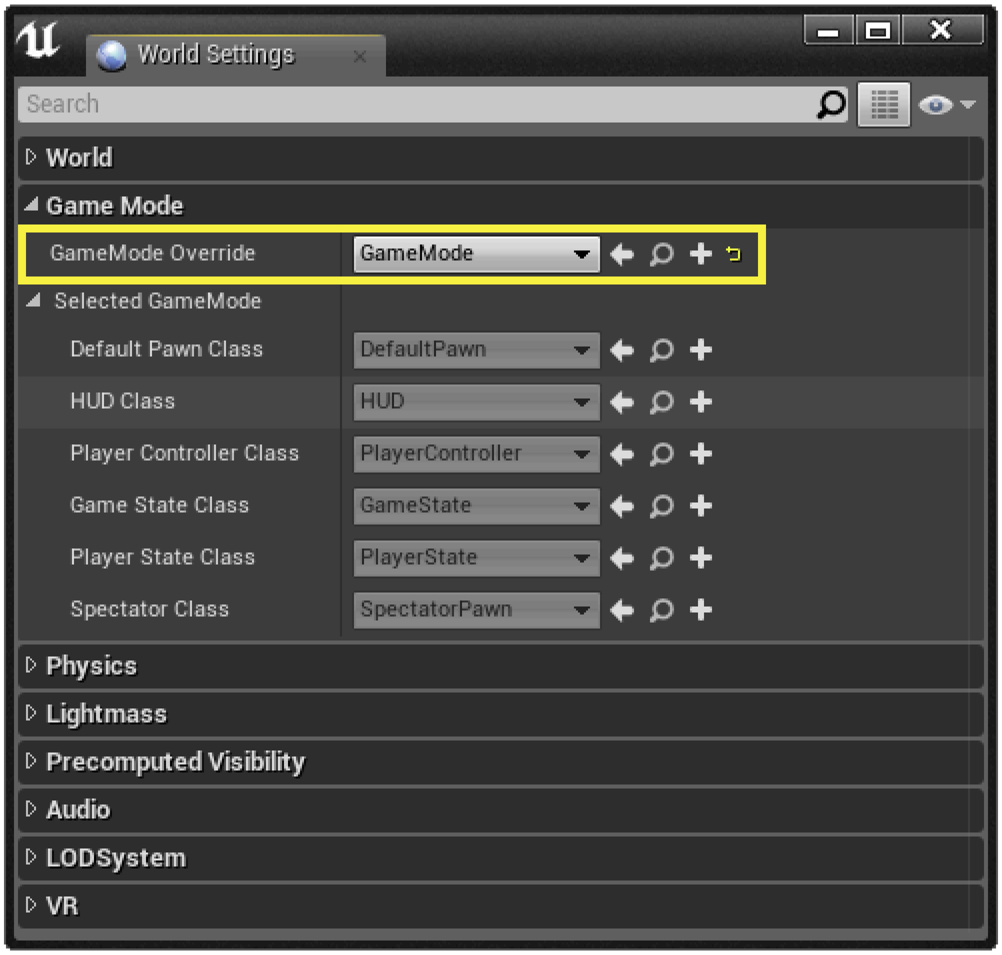
- Setting the `GlobalDefaultGameMode` entry in the `/Script/EngineSettings.GameMapsSettings` section of the `DefaultEngine.ini` file will set the default game mode for all maps in the project.

```
[/Script/EngineSettings.GameMapsSettings]
GlobalDefaultGameMode="/Script/MyGame.MyGameGameMode"
GlobalDefaultServerGameMode="/Script/MyGame.MyGameGameMode"

```

Copy full snippet

- To override the project settings for an individual map, set the **GameMode Override** in the **World Settings** tab in the editor.



GameMode Override: GameMode

Selected GameMode

Default Pawn Class: DefaultPawn

HUD Class: HUD

Player Controller Class: PlayerController

Game State Class: GameState

Player State Class: PlayerState

Spectator Class: SpectatorPawn

- URLs can be passed to the executable to force the game to load with specific options. Use the `game` option to set the game mode. See Command-Line Arguments for more information.

```
UE4Editor.exe /Game/Maps/MyMap?game=MyGameMode -game

```

Copy full snippet

- Finally, map prefixes (and aliases for the URL method) can be set in the `/Script/Engine.WorldSettings/` section of the `DefaultEngine.ini` file. These prefixes set the default game mode for all maps that have a given prefix.

```
[/Script/EngineSettings.GameMapsSettings]
+GameModeMapPrefixes=(Name="DM",GameMode="/Script/UnrealTournament.UTDMGameMode")
+GameModeClassAliases=(Name="DM",GameMode="/Script/UnrealTournament.UTDMGameMode")
```

Copy full snippet

For an example of setting up a Game Mode, refer to the Setting Up a Game Mode documentation.

Game State

The **Game State** is responsible for enabling the clients to monitor the state of the game. Conceptually, the Game State should manage information that is meant to be known to all connected clients and is specific to the Game Mode but is not specific to any individual player. It can keep track of game-wide properties such as the list of connected players, team score in Capture The Flag, missions that have been completed in an open world game, and so on.

Game State is not the best place to keep track of player-specific things like how many points one specific player has scored for the team in a Capture The Flag match because that can be handled more cleanly by **Player State**. In general, the GameState should track properties that change during gameplay and are relevant and visible to everyone. While the Game mode exists only on the server, the Game State exists on the server and is replicated to all clients, keeping all connected machines up to date as the game progresses.

`AGameStateBase` is the base implementation, and some of its default functionality includes:

Function or Variable	Use
`GetServerWorldTimeSeconds`	This is the server's version of the `UWorld` function `GetTimeSeconds` and will be synchronized on both the client and server, so it is a reliable time to use for replication.
`PlayerArray`	This is the array of all `APlayerState` objects, which is useful when doing something to all players in a game.
`HasBegunPlay`	Returns true if the `BeginPlay` function has been called on actors in the game.

`AGameStateBase` is very commonly extended in C++ or Blueprints to contain additional variables and functions that are needed to keep players informed of what's going on in the game. The specific modifications made are generally based on a paired Game Mode for which the Game State is made. The Game Mode itself can also override its default Game State type to be any C++ class or Blueprint derived from `AGameStateBase`.