Actor Ticking

Explanation of the ticking system used to update Actors each frame.

Ticking

"Ticking" refers to running a piece of code or Blueprint script on an actor or component at regular intervals, usually once per frame. Understanding the order in which your game's actors and components tick relative to each other and other per-frame tasks performed by the engine is essential to avoid off-by-one frame issues, as well as to ensure consistency in the way your game runs. Actors and components can be set up to tick each frame, at set minimum time intervals, or not at all. In addition, they can be grouped together at different phases in the engine's per-frame update loop, and can be individually instructed to wait for specific ticks to complete before beginning.

Tick Groups

Actors and components are ticked once per frame, unless a minimum ticking interval is specified. Ticking happens according to tick groups, which can be assigned in code or Blueprints. An actor or component's tick group is used to determine when in the frame it should tick, relative to other in-engine frame processes, mainly physics simulation. Each tick

group will finish ticking every actor and component assigned to it before the next tick group begins. In addition to tick groups, actors or components may set tick dependency, which means that they will not tick until the specified other actor's or component's tick function is finished. Using tick groups and tick dependency can be vital to ensuring that your game works properly with regard to physics-dependent behaviors, or sequential gameplay behaviors that involve multiple actors or components.

The following are the tick groups available for gameplay use, in the order in which they run during the frame, as well as the specific meaning and context of each group:

Tick Group Order

Tick Group	Engine Activity
TG_PrePhysics	Beginning of the frame.
TG_DuringPhysics	Physics simulation has begun by the time this step is reached. Simulation may finish and update the engine's physics data at any time while ticking this group, or after all group members have ticked.
TG_PostPhysics	Physics simulation is complete and the engine is using the current frame's data by the time this step begins.
n/a	Process latent actions, tick the world timer manager, update cameras, update level streaming volumes and streaming operations.
TG_PostUpdateWork	n/a
n/a	Handle deferred spawning of actors created earlier in the frame. Finish the frame and render.

TG_PrePhysics

- This is the tick group to use if your actor is intended to interact with physics objects, including physics-based attachments. This way, the actor's movement is complete and

can be factored into physics simulation.
- Physics simulation data during this tick will be one frame old - i.e. the data that was rendered to the screen last frame.

TG_DuringPhysics

- Since this runs at the same time as physics simulation, it is unknown whether physics data during this tick is from the previous frame or the current frame. Physics simulation can finish at any time during this tick group and will not present any information to indicate this fact.
- Because physics simulation data could be current or one frame out of date, this tick group is recommended only for logic that doesn't care about physics data or that can afford to be one frame off. Common cases might be updating inventory screens or minimap displays, where physics data is either entirely irrelevant, or displayed coarsely enough that the potential to have a single frame of lag does not matter.

TG_PostPhysics

- Results from this frame's physics simulation are complete by the time this tick group runs.
- A good use of this group might be for weapon or movement traces, so that all physics objects are known to be in their final positions, as they will be when this frame is rendered. This is especially useful for things like laser sights in shooting games, where the laser beam must appear to come from the player's gun at its final position, and even a single frame of lag will be very noticeable.

TG_PostUpdateWork

- This runs after TG_PostPhysics. Historically, its primary function has been to feed last-possible-moment information into particle systems.
- TG_PostUpdateWork happens after cameras are updated. If you have any effects that rely on knowing exactly where the camera is pointed, this is a good place to put the actors that control those effects.

- This can also be used for any game logic that is intended to run after absolutely everything else in the frame, such as resolving two characters trying to grab each other on the same frame in a fighting game.

Tick Dependency

The `AddTickPrerequisiteActor` and `AddTickPrerequisiteComponent` functions, which exist on both actors and components, will set the actor or component on which the function is called to wait to tick until the specified other actor or component has finished ticking. This can be particularly useful for things that happen at roughly the same time in a frame, but where one actor or component sets up data that the other will need. The reason to use this over a tick group is that many actors can be updated in parallel if they're in the same group. Making one group of actors move to an entirely new group might not be needed if those actors are only individually dependent on one or two other actors, rather than needing to wait for the whole group to finish before before ticking.

Examples

Tick Group/Tick Dependency Usage Example

As an example of how to use each of the tick groups listed above, imagine a game where the player controls an animated actor who aims a laser, which places a special targeting reticule actor at the point of impact. A special meter fills up as long as the laser stays pointed at a certain type of target object, and a HUD actor displays this meter on the screen.

The player's animated actor would move and animate in TG_PrePhysics. It needs to animate before physics in order for physics-simulated objects to follow and interact with it correctly.

The HUD can update in any tick group, but TG_DuringPhysics is a good selection for two reasons. First, TG_DuringPhysics is acceptable because it doesn't directly interact with, or use data from, the game's physics simulation. Second, it is a good idea because there is no reason to force physics simulation to wait for the HUD to finish updating, nor is there a reason to force the HUD to wait for physics simulation to finish. Note that the HUD will be one frame behind the game, i.e. pointing at a target object this frame will not be reflected in the meter until next frame.

The reticule actor would update in TG_PostPhysics. This way, the reticule knows it is tracing against the scene as it will be rendered at the end of the frame, so it knows that it will appear exactly on the surface of the object as intended. It also knows that it will adjust the meter value based on the correct locations of target objects.

Finally, in TG_PostUpdateWork, the laser particle effect would be updated with the final locations of both the aiming actor and the reticule.

Tick dependency can be used to eliminate the need for TG_PostUpdateWork. The laser particle can be placed in TG_PostPhysics along with the reticule actor, using tick dependency to ensure that the laser is updated with the reticule's location only after the reticule has had a chance to tick. By setting the laser's tick dependency to the reticule, we can ensure that the laser isn't updated too early, but without waiting for other post-physics ticks that aren't relevant. This can be more efficient than moving the laser into a different tick group.

As an example of a case that would not benefit from tick dependency, the reticule itself does not need to be tick-dependent on the aiming actor, even though it needs the aiming actor to have finished ticking before it can do its own tick. The reason that tick dependency is unnecessary is that the reticule is in post-physics while the aiming actor is in pre-physics. Because they're in different tick groups, we can be assured that they will always run in the order of the groups themselves, since each tick group completes ticking all of its actors and components before the next tick group can begin.

Actor Spawning

In `BeginPlay`, an actor will register its primary tick function and the tick functions of its components with the engine. An actor's tick function can be set to run in a specific tick group, or disabled entirely, via the `PrimaryActorTick` member. This is generally done in the constructor to ensure that the data is set correctly before `BeginPlay` is called. Some commonly-used code follows:

```
1 PrimaryActorTick.bCanEverTick = true;
2 PrimaryActorTick.bTickEvenWhenPaused = true;
3 PrimaryActorTick.TickGroup = TG_PrePhysics;
4
```

Copy full snippet

Component Ticking

Just as Actors can be segregated into different ticking groups, so can Components. Previously, an Actor ticked all of its Components during the Actor's tick. This still happens, but Components that need to be in different groups are added to a list that manages when they will be ticked. Components should be assigned ticking groups using the same criteria for assigning an Actor to a tick group. The tick structure for Components is named differently from the one that Actors use, but it works the same way:

```
PrimaryComponentTick.bCanEverTick = true;
PrimaryComponentTick.bTickEvenWhenPaused = true;
PrimaryComponentTick.TickGroup = TG_PrePhysics;

```

Copy full snippet

> Remember that `PrimaryActorTick` uses the Actor's `Tick()` function, while PrimaryComponentTick uses the ActorComponent's `TickComponent()` function.

Advanced Ticking Functionality

The default tick function for an actor or component can be enabled or disabled during the game with the `AActor::SetActorTickEnabled` and `UActorComponent::SetComponentTickEnabled` functions, respectively. In addition, an actor or component can have multiple tick functions. This is accomplished by creating a struct that inherits from `FTickFunction` and overriding the `ExecuteTick` and `DiagnosticMessage` functions. The default actor and component tick function structures are good examples of how to build your own, and can be found in EngineBaseTypes.h under the names `FActorTickFunction` and `FComponentTickFunction`.

Once you have added your own tick function structure to your actor or component, it can be initialized, usually in the constructor of the owning class. To enable and register the tick function, the most common route is overriding `AActor::RegisterActorTickFunctions` and adding calls to the tick function structure's `SetTickFunctionEnable`, followed by

`RegisterTickFunction` with the owning actor's level as an argument. The end result of this process is that any actor or component that you create can tick multiple times, including ticking in different groups and with individual dependencies per tick function. To set a tick dependency manually, call `AddPrerequisite` on the tick function structure you want to make dependent on another and pass in the tick function structure that you wish to use as your dependency.