TI GAME TELEMETRY PLUGIN SETUP

SETUP & UPROJECT

Locate the plugin in your launcher and install it to the engine instance that you are using for your game.

Now start your project and in the editor, go to Edit - Plugins. Locate the TIGameTelemetry Plugin, enable it and then restart your editor for that change to take effect. This will enable the plugin in your game's .uproject file.

QUICKSTART - BLUEPRINTS

NOTE:

This guide will assume that you are fairly experienced with Unreal Blueprints and know what the various nodes and events do, as well as how to set up various Blueprints that are used.

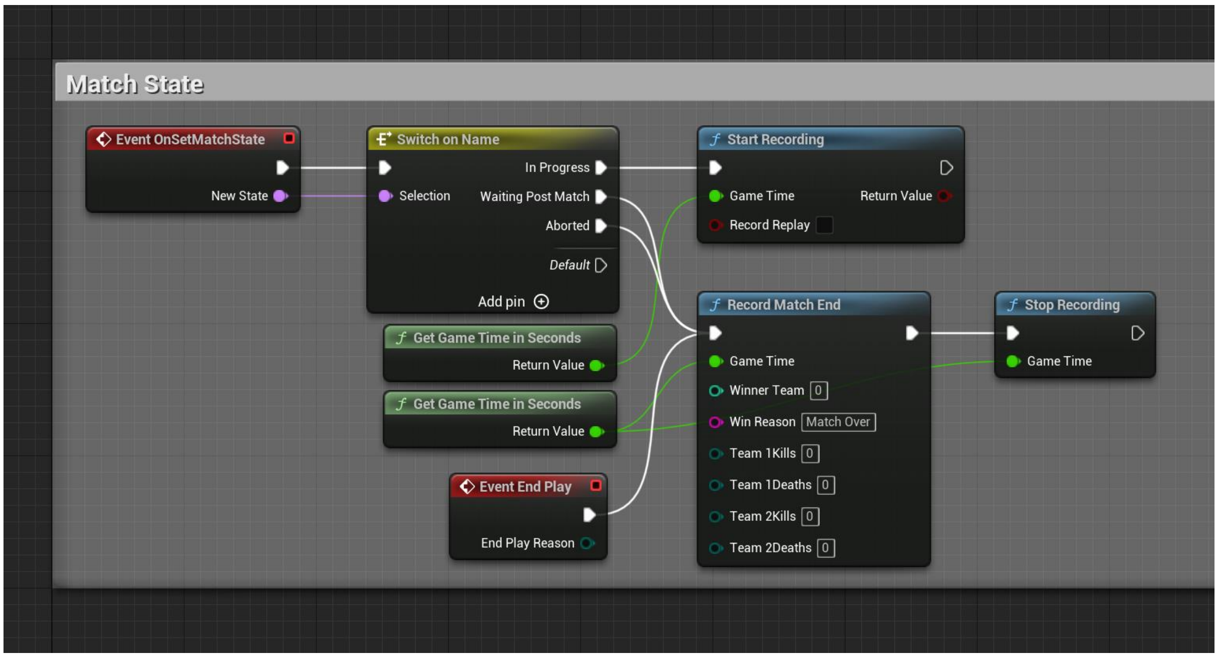
For Reference, check the Sample project's setup.

BASIC EVENT RECORDING

This quick start guide will cover how to set up the start, stop, player spawn and player death recording events in Blueprints.

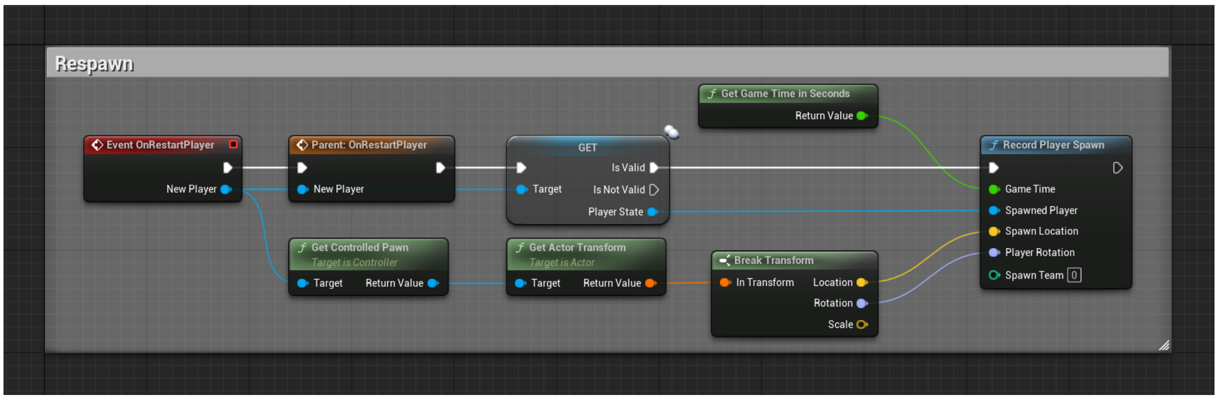
Start Recording

This is easiest achieved by using the Game Mode's OnSetMatchState Event. Create this node in your Blueprint and create a switch to determine what new match state you are working with. For InProgress, create a "Start Recording" Telemetry node, for WaitingPostMatch, create a RecordMatchEnd and Stop Recording node.



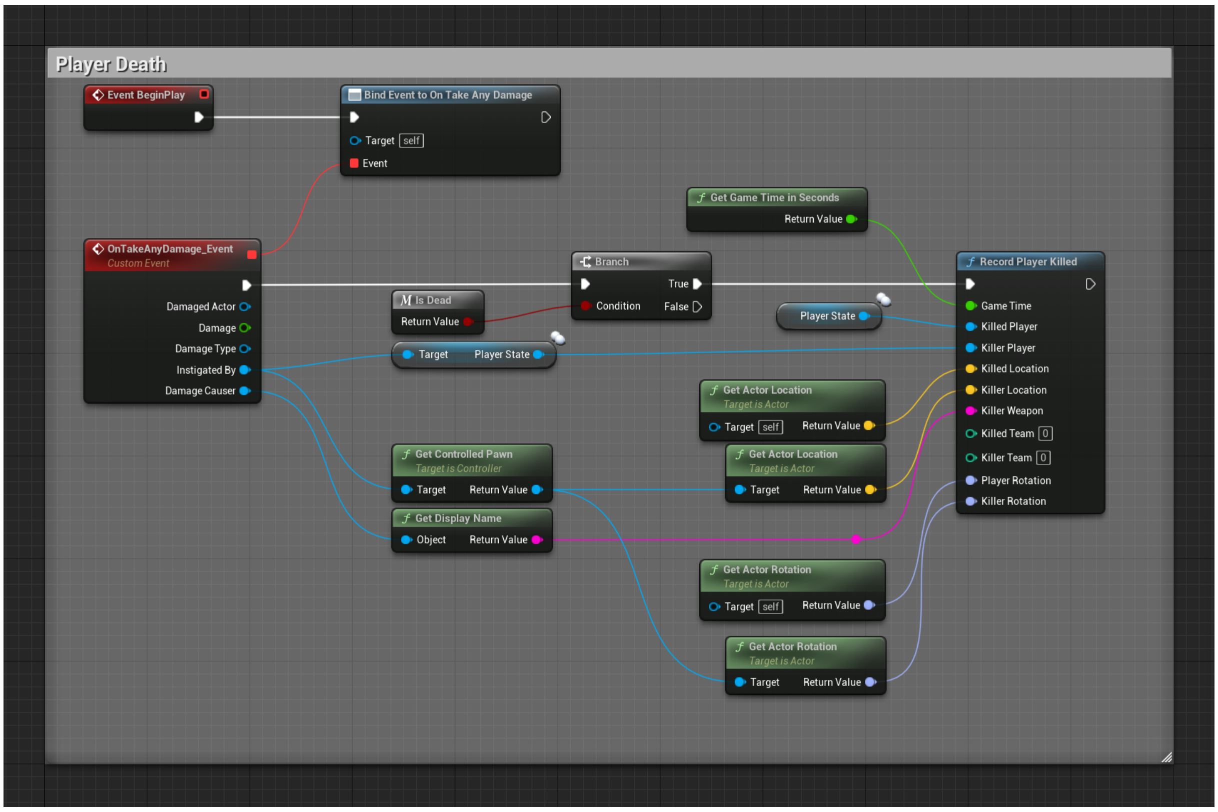
Player Respawn

The player Respawn Event is covered using the Game Mode's OnRestartPlayer Event. Create the node and hook it up to the RecordPlayerSpawn node as illustrated below.



Player Death

To record a player Death Event, we're going to use the Pawn Blueprint class. It comes with an Event listener that is called when the any kind of damage is dealt to the pawn. We need to register that listener first, using the Pawn's OnBeginPlay Event and whenever that listener's event is called, check if the pawn has taken too much damage to still live. Hook it up as illustrated below and you're good to go:



Note, you can copy – paste this logic into the bot pawn as well.

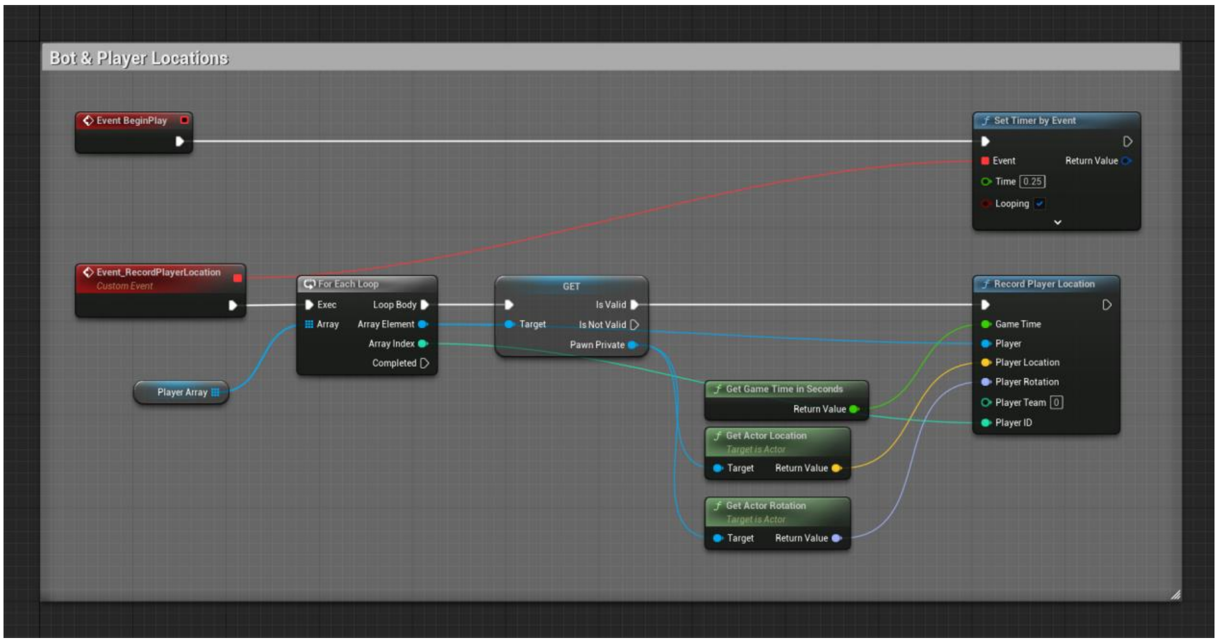
Player Location

There are two convenient ways to record a player location. One is to go through the Player Pawn, the second is to use the GameState Blueprint class, accessing the player array in there. There are pros and cons to both of these approaches but they'll both get the job done, regardless of which you chose.

For the purpose of this guide, we used the GameState's Blueprint. To get started, use the Event BeginPlay and register a Timer By Event.

Give it a *Time* setting that fits your needs, we used .25s which equals four location records per player per second. Lower values record the paths a player takes more accurately but tend to create tons of data.

Whenever this timer fires, access the player array that contains all player states. These player states have a Pawn Private property, that you can read to determine the location and rotation of each player and bot. Use the Record Player Location node to record this data.



Note: If you struggle with the amount of data or notice significant performance impacts when recording or evaluating the data, we recommend to use higher *Time* values for the timer. This is especially important for multiplayer games where you have dozens of clients connected at any point in time and can easily create several megabytes worth of telemetry data over the course of a few minutes. This doesn't sound like much, but depending on your server or client's hardware and the amount of data to be written, writing the telemetry queue to disk can cause noticeable interference.

That concludes the basic recording set up. From here you can add additional events that matter to your game loop and are worth evaluating.

C++ START GUIDE

SETUP

To use the plugin with C++, there are additional steps required to reference the plugin headers in your game code.

1. In your game's build file, reference the plugin as "TIGameTelemetry" in the PublicDependencyModuleNames, PrivateDependencyModuleNames and PrivateIncludePathModuleNames.
2. In any class that you want to call the plugin functions from, add an include to the plugin's header: `#include "TIGameTelemetry.h"`

EVENT RECORDING

Once the plugin is properly referenced, you can call the available functions. A few examples:

To determine whether the telemetry is running:

- `FTIGameTelemetryModule::GetIsRecordingTelemetry();`

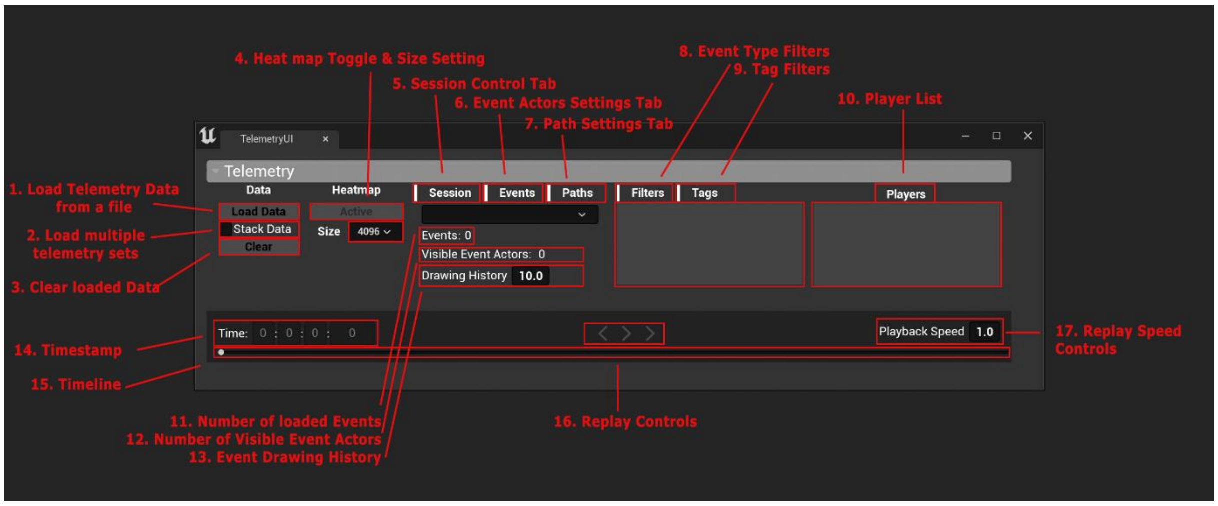
To start and stop recording:

- `FTIGameTelemetryModule::TelemetryStopRecording(GetWorld()->TimeSeconds);` and
- `FTIGameTelemetryModule::TelemetryStartRecording(GetWorld()->TimeSeconds, false);`

Check TIGameTelemetry\Public\TIGameTelemetry.h for a full list of existing functions.

HOW TO LOAD AND READ THE DATA

After recording a telemetry session, it's time to evaluate what you recorded. Let's look at the UI itself first.



1. Load recorded telemetry data from a file
2. Load multiple telemetry files. Regular behavior clears a previously opened telemetry set if you load a new one. Ticking this checkbox will keep a previous set loaded while loading a new one on top.
3. Clear loaded data and destroy all event actors
4. Toggle the heatmap on and off, increase or decrease the size of the heatmap decal used to project it into the world.
5. Open the session controls tab. It allows you to modify the drawing history of events. This feature allows you to draw actors for a determined amount of seconds after they happened.
6. Event Actors Settings Tab: This tab allows you to increase or decrease the drawing distance and pool size of the event actors, that represent recorded events in your map.
7. Paths Settings Tab: This tab serves the same purpose as the event actors tab, but for paths.
8. Event Filters Tab: Here you can toggle events on and off, useful to only focus on a certain event type, disable others and is generally useful to determine what's drawn in in the event heatmap.
9. Tag Filters: As above, but instead of filtering Event Types, it allows you to filter by recorded data
10. Player List: A list of all players in a recorded session. Can be used to filter events by player, or exclude certain player records
11. Total number of loaded events
12. Number of currently visible event actors
13. The event drawing history: Events will be rendered as long as the currently set timestamp is within the timeframe of the events' time stamps + the drawing history
14. The currently displayed timestamp
15. The recorded timeline
16. Replay controls: Rewind, Play and Forward
17. Replay Speed: Slow down or increase the speed at which the record is played back

The sample project comes with a sample recording for you to get the hang of it.

MISC & TROUBLESHOOTING

I AM RECORDING EVENTS IN A MULTIPLAYER/COOP GAME AND SOME DON'T END UP IN THE SERVER'S OR HOST'S TELEMETRY DATA

Make sure to record all events in your authoritative game instance, i.e., your dedicated server or your coop host. Recording events client side will put them in a separate telemetry file. If this happened and you still need the data, you can load multiple telemetry data sets on top of each other to get the full picture, but we recommend you record them all in one place to begin with.

MY EVENTS DON'T UPDATE WHEN RERUNNING THE RECORDING VIA THE EDITOR UI

Make sure that the Telemetry UI widget is in focus. If you put it in a tab group, it must be the active tab and cannot be hidden anywhere. The update functionality relies on the widget's Tick funtion which won't run if the UI is hidden.

I RAN THE SAMPLE GAME FOR A MOMENT TO TEST THE PLUGIN AND NOTHING GOT RECORDED

The plugin collects the events in a queue and then writes that to a file every 60 seconds. Make sure to run the test session for at least a minute to allow the plugin to write that queue to the telemetry file. This was done to minimize the impact of writing large queues to the harddisk.

You can modify the frequency at which this happens in the plugin's code, by changing the value of `QueueWriteFrequency`. Note: You'll need to recompile the plugin to make this work.

Alternatively, if your game sessions are too short to get recorded properly within that 60 second window, you can call the *Stop Recording* node to force the queue to be written to disk.

WHAT'S THE DIFFERENCE BETWEEN THE TWO NODES OBJECTIVE STATE CHANGE AND OBJECTIVE TAKEN?

One denotes that something is happening with your objective and the other signals the final state of that change. The implementation depends on your game's needs, of course, and you might choose to not use one or both in the end. Just for our game, they made sense.

WHAT DO THE //COUNT UMETA(HIDDEN) LINES IN YOUR ENUMS MEAN?

They were planned to be used to count the length of the enums but broke blueprint accessibility of these enums. We chose to keep them there, commented out, for later revision.

THE PLUGIN SPAWNS A TON OF NEW ACTORS INTO MY LEVEL. DO THEY STAY AFTER SAVING AND RELOADING A LEVEL?

The event actors are marked as transient. They're only temporary and won't save as part of your level.