Performance Report

Modular AI Unity package

Benjamin Wharton

It was found that during the creation of a test application, there were many things that had to be accomplished with annoying workaround due to some limitations with delegates. This goes against the whole purpose of the system which is to be as user-friendly as possible. The regular delegate container class has a limit on the number of parameters due to delegates requiring a set number of types at compile time. This led to problems mainly encountered with Interrupt Conditions - methods like comparing a behaviours score against another value would have to be hardcoded (e.g., checking whether a behaviour score is <= 0.5 or == 10, these would both need separate methods).

The solution to this was creating a separate type of delegate container class which completely removed the use of delegates and instead stores parameters in an object array and uses dynamic MethodInfo invocation instead. This means any method can be supplied and any length of parameters can be passed through that method. A separate inspector drawer class was also made to display fields for all method parameters. However, because Unity doesn’t support object type serialisation, a new “TypeContainer” class was created which contains a set of basic data types. If a methods parameters are one of the basic types defined in the TypeContainer, the inspector can take values and pass them to the delegate container, where an object array is made. This system allows for a much wider support for methods that the user may want.

With this new system, there are a few performance concerns. The biggest concern in regard to performance was the high invocation times that dynamic MethodInfo calls may have. Due to the need for unknown parameter inputs at runtime, the class cannot store the method inside a delegate, which means it must use reflection on each call to find a given method inside an instance. It’s obvious that having to find the method on each call is going to take longer to execute. To see the full extent of this, some testing was done.

Over the period of 5 minutes in the example simulation, the following results were found between the 3 types of invocation techniques (Delegate, MethodInfo, and UnityEvent):

|  |
| --- |
| The containers utilising **delegates** (Behaviour selectors and Evaluators) ran a total of 4754 invocations with an average of **0.00028ms** execution time.  The containers utilising **method info** (Interrupt Conditions) ran a total of 369 invocations with an average of **0.0058ms** execution time.  The **Unity events** (used for Start, Active, and End) ran a total of 10926 times with an average of **0.0026ms** execution time. |

The results show that delegate containers ran the fastest, Unity Events second, and MethodInfo the slowest. Delegates ran over 9x faster than Unity Events, and Unity Events ran over 2x faster than MethodInfo. This result was expected, as delegates do most calculation on initialisation, allowing invocations to be called without much overhead, and both Unity Events and the MethodInfo technique use reflection. The full extent of Unity Events reflection use is not clear; however, it is built into the engine in C++ and likely utilises the higher freedom of generic class types that C++ provides. Despite the higher performance cost, the MethodInfo technique does have functionality for any number of pre-defined parameters from both the inspector and code which makes it a powerful user-friendly tool, and worth the costs in most cases.

Besides the performance of the invocations themselves, the general performance of an AI instance is highly dependant on what the user is doing with their AI. If the user is utilising very costly methods in their AI, such as pathfinding, the performance will be worse. This is why many parameters are available to the user which allow them to control the frequency and cooldowns of each part of the system. The user can decrease or increase these to meet their needs or compensate for any performance problems.