Performance Report

Modular AI Unity package

Benjamin Wharton

**Issues encountered when integrating the modular complex system into the test application.**

* Implementing delegate systems had some limitations
  + There was a limit on the number of parameters which can be used, which often meant workarounds had to be designed in the test application. This goes against the whole purpose of the system which is to make it as user-friendly as possible.
    - Originally, for the delegation of methods to work correctly, the container class had to use generic types to specify a set number of parameters that a method must have in order to be used.
    - This meant that when dealing with them in the unity inspector, I did not have the option to enter pre-defined parameters. This is often important for interrupt conditions, for example, the timer condition would ideally be able to take a pre-defined float number that represents a time in seconds.
    - A solution to this was creating a separate type of delegate container class which completely removed the use of delegates and instead store inspector defined parameters in an array and use MethodInfo invocation instead.
    - To receive parameters from the inspector, a new type container had to be created which contains a set of basic data types. Through the inspector, a methods parameter types are checked, and corresponding input fields are then created for these parameters as long as they are one of the basic data types. This system basically allows for any number of parameters for a methods.

**Be sure to outline any changes required when implementing your system that were different to the details included in your initial project brief.**

**The performance of the system.  
How you measure performance will be dependent on the type of complex system implemented. However, you should aim to benchmark your system against similar implementations.**  
The biggest concern in regards to performance was the invocation time for the delegate container that utilised dynamic MethodInfo calls instead of Delegates. Due to the need for unknown parameter inputs at runtime (Editor runtime), the class cannot store the method inside a delegate, which means it must use reflection on each call to find the 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 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 of AI behaviours) ran a total of 10926 times with an average execution time 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 expected, delegates do most calculation on initialisation allowing calls to be called without overhead, and both Unity Events and the MethodInfo technique use reflection. The full extent of Unity Events use of Reflection 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.

**Any required changes for the system to function as intended.**