Implementation – Actions

## Introduction

As we already went through what actions can be used this section will keep that to a minimum and instead focus on our idea behind, and how we implemented them. It will furthermore cover the entire life span of an action object.

## Explanation

An action or as called in our engine XmasAction is a class which provides an API for performing state changes inside the engine, while also ensuring that only one action at a time is being executed.



As we can see it starts with the XmasModel running an endless loop that tells the ActionManager to execute all newly queued actions. The ActionManager then takes all the actions from a threadsafe list and places them in a local list. After which each action is executed individually, putting the action that is being executed in a running state, the state will not change before the actions “completed” method is called. Once an action has been properly executed it will be changed to a completed state will be properly disposed. When the last action has been executed by the ActionManager, the call to the ActionManager is returned and XmasModel will put the thread in a waiting state. The XmasModel will remain in wait state until a new action has been placed on the queue; this is preventing it from busy waiting when no actions are to be executed.

## Considerations

The way that action completion is designed might seem tedious in that it has to call a special method “Completed” on each action however it is quite necessary as the completion of the execute method call does not guarantee that a method is completed.

The reason why the completion of the method Execute does not guarantee that Action has actually completed is that in cases of actions not being instantaneous the rule would not apply. To give an example of such action, take the action of moving from one place to another. In this case the move action would need to create a timer to give the idea that the move action had a speed. As we can’t halt other actions during this time it is paramount that the Execute method is released so that other actions can be executed during this period.

This is also how the move action is designed in our reference implementation, the algorithm is as follows

1. Move action put on queue
2. Move action sets up a timer on a different thread and finishes its execution
3. The timer is fired after a given time, and places a new action on the queue
4. The new action performs the actual move, and calls the Completed method of its parent Action(The Move Action)

As one can see the problems in this design is the redundancy created by having to call a method on each execute, this might not seem like a problem however it is problematic in a few ways. First and foremost it adds complexity in usage of the engine, a person with no knowledge of using the engine would not have a way in realizing the correct way to make and use actions. Thus it creates a second problem there is no way to determine if an Action is correctly constructed during compile time, this means bugs will naturally accumulate during extended use, even if a user has experience and foreknowledge forgetting even for a single action can be crucial. This is because running actions use resources and if never completed the resources of the actions are never released. For instance let’s assume the “Move Action“ Completed method is never called, the result of this is that it is stored in the “ActionManager” as Running. Now let’s assume that this move action is continuously being executed by hundreds if not thousands of agents. As each action is never released the memory stored for each action is never released and an unintentional memory leak is thus created.

Another way we could have chosen to implement the action completion process, is the usage of child action. Imagine if an action could generate new actions that were linked with it, thus the completion of an action would be tied to the fact that all its child actions had been executed and not the arbitrary call of a “Complete” method. This could undoubtedly provide new problems to overcome and as such we have not fully followed this path, however given more time to study the consequences of this design would reveal whether or not this is a better design.

# Summery

A lot of the considerations when designing the action all comes down to the reliance on user to clean up the Action, which is generally not good from a design perspective; it is always preferable that used data is cleaned up automatically when it is out of scope. However it is not all bad as this design does guarantee a flexible usage of the actions; it provides more control to the user which might give the user abilities to do certain things which would otherwise be denied within the engine. This is also why this design method was chosen as our philosophy in the engine design was to minimize limitations as much as possible while still providing the features we thought necessary to fulfill the engines goal.