Implementation – Events and Triggers

# Introduction

This section will cover the inner workings of how events and triggers are connected, furthermore it will detail why we designed it as it is. We will also cover exact producer when an event is raised, to defuse any confusion their might be as to what happens inside the engine.

# Explanation

By themselves events are not particularly complicated since they are essentially just date structures that are transferred to all its listeners upon triggering. As such we shall do a close examination of how exactly the EventManager works.

The EventManager is tethered through the engine to all entities that are inside, whenever an entity has an event raised on it; it is copied to the event manager which also raises the event. The intent is to minimalize the number of events needed to cover a single case. To be clear on how exactly this transpires we have drawn a sequence diagram.

[SHOW Sequence diagram of an event being raised on an entity and then transferred to the event manager]

As shown on this sequence diagram, an action in this case a move action raises an “EntityMoved” Event on a given entity, the entity then calls all triggers registered to it, where the trigger also contain the event being raised. After this the entity informs the EventManager that an event has been raised on this, this causes the EventManager to also call all its registered Triggers with the given Event. Once all relevant triggers have been informed of the Event being raised, the procedure is complete and the EventManager returns to its dormant state.

In the case of events are not linkable to a specific entity such as an “Engine Close Event” the raise of the event to done purely on the EventManager itself. The process is otherwise the exact same as above except that no particular entity is involved.

# Considerations

As there were many considerations that went through our design process we will take each component of this area and break down why exactly why we designed it as is.

## Events

### Problems of C# events and why we chose to design our own events

The language which our engine is written in is C#, one of the fantastic things about C# is that events is built into the language, as such it may come as a surprise that we have chosen to re-implement events ourselves. However while the name be the same the intent between C# events and our events is so different that it is impossible to compare the two. The intent behind C# events is to keep maintenance on single objects, so that changes to a given object can affect its linked objects without having to be designed specifically to do so. This allows for great decoupled projects and is what makes object maintenance in C# quite easy. However our events are not meant for such low-level tasks instead they are meant to allow reactions to occur in responds to other actions. Furthermore C# actions are bound to a specific class, and can only be fired inside methods of an instantiation of the specific class. The events we have designed are meant to be raised by all types of class that wish to signal such an event has occurred.

To give an idea of what sort of problems that would arise from using C# events one need only look at how global events would have to be implemented. Since Events using C# are linked to a specific class, which would essentially mean that the EventManager class would need to be setup for every single event the engine is capable of running. What this basically has accomplished is to couple a single class into the entire workings of an engine, this makes for poor programming design in more than one way and greatly reduces productivity. It also makes the engine difficult to extend and modify at a later time since the design would be practically hardcoded into it.

### Improvmens of events

As of now our events are currently not tied to being an Entity Event or environment Event like actions are, however this might have been a wrong move on our part. The problem is that the user of the engine might be unclear as to which is what, currently the difference lies in the name convention used for events. For instance an event such as “EntityMoved” it is clear that this is an event that can be tied to a specific entity. However for an event such as “AddedEntityToEngine” this might be confusion because the event is clearly speaking about a single entity but as the entity is only just added it would have been impossible for any trigger to be registered to it. If one was to make improvements to the event design this would be one change that was worth looking into.

## Triggers

The trigger design came about as a necessity for providing a way for the user to easily design reactions to a given event. The trigger design is very minimalistic except for the fact that it has a condition. We designed it with the condition because we wanted it to be obvious how unwanted events should be handled. Furthermore it also helps to split up the code containing if the action should be executed away from the action itself allowing for more readable code.

Another way the triggers could have been designed would be if the user simply registered lambda functions(anonymous function), this would help reduce the amount of class a user should know and understand. However we preferred to encapsulate this into what we call the Trigger, since we wanted to have the ability to expand the capability of the trigger at a later time.

In short triggers are a simple design that gives the engine logic a lot of flexibility.

# Summery

Events and triggers might be a hassle to setup and design however what they return in value is tenfold to that, they provide the engine with much flexibility. Without Events the engine would suffer greatly and all actions would be required to be bogged down with a lot of extra logic. This would not only remove the modularity of the engine but also make using engine more error-prone.