A simple templated delegate event system

Second Draft

by Christian Sagel

For most of us familiar with the Zero Engine, or perhaps Unity we have grown fond of constructing events within components and dispatching them to any entities that the component can a grab reference to. These events are in themselves objects that carry data and that can be created at will to fit the needs of the developer to create and extend gameplay logic through the Observer pattern.

Overview

For those of you on the fence regarding the usefulness of using event objects rather than passing enumerations through a virtual method or strings with variadic templates, I will try to convert through giving a brief overview on the how the event system works. First, I will give a sample case of the event system being used, then second I will explain its implementation.

Case: We have started working on our engine and our physics programmer needs graphical and input tools to test his collision code. As a developer we have been benevolent and given our entity composition object a *'BoxCollider'* component in order to detect collisions and resolve them. We know that our physics system is sending *'CollisionStarted'* and *'CollisionEnded'* events whenever an object with the 'BoxCollider' component enters and exits collision with another object. In order to give the physics programmer a way to debug the component without polluting its code with traces and other checks, we will do the following:

1. Create a component, aptly named "*DebugCollision*".
2. We will create and attach this component to an object that already has the *'BoxCollider'* component attached to them.
3. In this component's *DebugCollision::Initialize()* function, we will connect to its owner's *'CollisionStarted'* and *'CollisionEnded'* events. (Which the GameObject, currently having a BoxCollider component, will be receiving)
4. When the *'DebugCollision'* component receives these events, it will call a function and perform some logic operation. It may flip the color of the component when it's colliding, then return it to normal when it's not. It could decide to play sounds by dispatching a request to play a sound to the *'SoundSpace'* component on the space, etc.
5. What if the programmer wants to be able to press a key at any time to be able to print to screen the object's position on screen and other relevant data? All she need do is connect to the *'KeyDown'* event and print whatever data he want based off any input he want.

The event system pipeline works on two ends:

1. A component subscribes to the events that a particular entity is receiving. That entity may be its owner object, another object, its space or even its GameSession. By connecting to an event, the component calls an engine-class function that registers a member function of the component to a specific event class type on the entity's registry of subscribers. The technical details of this will be elaborated on shortly.

Code example:

```
// This is a macro to simplify the syntax for the Engine::Connect templatized function call.
// The first argument is the entity that the component wants to subscribe to.
// The second argument is the event class.
// The third argument is a reference to the member function that will receive the event.
Connect(gameObj, Events::CollisionStarted, DebugCollider::OnCollisionStartedEvent);
```

2. A component or a system (Really, any object with access to any entity's public interface) dispatches a specific event to a specific entity. This is done by dynamically allocating an event object of a specific event class type, then passing the pointer to it as an argument alongside the event class to the entity's dispatch function.

Code example:

```
// 1. Construct the update event and assign it the engine's delta time
auto logicUpdateEvent = new Events::LogicUpdate();
logicUpdateEvent->Dt = dt;
// 2. Dispatch a pointer to this LogicUpdate event to the space
space_->Dispatch<Events::LogicUpdate>(logicUpdateEvent);
```

By now the reader may have noticed that the arguments being passed into these functions are class types and member functions, not enumerations. If the implications of this haven't dawned on the reader, I will spook them now. This means that when components register member functions to specific event classes, as well as when an entity decides who to forward an event object to when an event is received are all decided at runtime by the compiler. (!!!)

This is done through the magic of std::type_index and std::type_id. And the use of templated delegates to package member functions into. I will now proceed to the technical details.

Technical Overview

There's two logistical problems that need to be solved before the event system works as one intends:

- First, a component needs to connect one of its member functions to the events of any entity it can grab a reference to, thus subscribing itself to that entity's events.
- Second, we need to be able to create a pointer to an event object and dispatch it to any entity we can grab a reference to, then have the entity forward that pointer to every component that is subscribed to that event.

Both ends of what I call the event-system pipeline are handled through the use of templates and run-time information (RTTI), providing introspection through the use of:

std::type_index: The type_index class is a wrapper class around a std::type_info object, that can be used as index in associative and unordered associative containers. The relationship with type_info object is maintained through a pointer, therefore type_index is CopyConstructible and CopyAssignable.

std::typeid: The typeid keyword is used to determine the class of an object at run time. It returns a reference to std::type_info object, which exists until the end of the program.

By combining both, we are able the class types of two objects to check for matches. This will prove invaluable for this implementation of the event system as well as providing infinitely useful for other templated functions (such as returning a pointer to a derived system type to the client!) With that said, we will now start with subscribing a component to an entity.

Connecting to an event

Because the engine itself runs off a super 'Engine' object, I decided to have the engine handle subscribing components to entities. My implementation of subscribing to an entity's events is done in the following way:

1. First, inside of any one component's member functions, we call a function on the engine object with the following signature:

```
template <typename EventClass, typename ComponentClass, typename MemberFunction>
void Connect(Entity* entity, MemberFunction fn, ComponentClass* comp);
```

Of course, this is simplified for our general use behind a macro. Inside a component we would use a macro call like this:

```
void DebugCamera::Initialize()
{
	Connect(Daisy->getKeyboard(), Events::KeyDown, DebugCamera::OnKeyDownEvent);
```

The 'Connect' function does the following things:

1. First, it constructs a templated member function delegate of the input event class and input component class.
2. Second, since the delegate has a pointer to the component and to the member function that needs to be called on the event, it assigns both to the delegate.
3. Third, it creates a base delegate pointer. Why? An entity needs to have a container of delegates in order to know what member functions to call on receiving an event, but it cannot possibly have containers for every templated delegate possible. Instead, it will contain a container of the base delegate ones, which all have a virtual functions

for Call(), the method that actually dispatches the event to the component's member function.

4. It stores this base delegate inside a container for delegates that every entity has. This container is not a vector, but a map of std::type_index and a list of delegates. This allows multiple delegates for each event type to be subscribed. The container is declared as follows:

```
std::map<std::type_index, std::list<DCEngine::Delegate*>> ObserverRegistry;
```

5. The map uses 'typeid(EventClass)' as the key for the map, meaning that the delegate is subscribing specifically only to events of that particular class type.
The delegate is then inserted into a list of delegates for that particular event class type.

Now our entity has added this delegate to its container of delegates for different kinds of events. Next time this entity receives an event object, it will look through this container and forward the event object to the component member function that has connected to it by invoking the delegate's 'Call' method, which takes an event object.

Simple, right? It *could* be, if you were already familiar with delegates and the implementation of templated delegates. On the unlikely chance that you aren't, I will explain what they are, and how I implemented them for use in the event system. I define a delegate as a class that wraps a pointer or reference to an object instance, a member method of that object's class to be called on that object instance, and provides a method to trigger that call.

The concept of a templated delegate hinges on the fact that we will have delegates for both different kinds of components and different kinds of events. By having a templated delegate class that derives from a virtual base one, we are able to store containers of these base delegates then invoke a virtual method such as Call(), that will call the component's member function given an event object.

For implementation purposes, let's have our base delegate class:

```
class Delegate {
public:
        Delegate() {}
        virtual ~Delegate() {};
        virtual void Call(Event* event) = 0;
};
```

Note the virtual method 'Call' which takes an 'Event' pointer. This method must be implemented in any class that derives from it, which the will!

Now the member function delegate class, which we will be actually constructing to store the member function pointers:

```
template <typename ComponentClass, typename EventClass>
class MemberFunctionDelegate : public Delegate {
public:
        typedef void(ComponentClass::*EventFn)(EventClass* event);
        EventFn FuncPtr;
        ComponentClass* CompInst;

        /*****************************************************************************/
        /*!
        \brief  Calls the member function given an event.
        \param  A pointer to the event object.
        */
        /*****************************************************************************/
        virtual void Call(Event* event) {
        EventClass* eventObj = dynamic_cast<EventClass*>(event);
        (CompInst->*FuncPtr)(eventObj);
        }
};
```

Let's go over a few things:

- The MemberFunctionDelegate class derives from Delegate, inheriting its virtual 'Call' method as well as making it possible to be stored through pointers to its base class in memory.
- A member function delegate takes two arguments, the class of the component and the class of the event.
- It uses a typedef for the member function pointer. Note the 'void(ComponentClass::*'. This is required syntax for member function pointers.
- It stores a pointer to the instance of that component's class that we need to call the member function on.
- The 'Call' method is virtual, allowing it to be called through the base class.

The Call method is invoked by an entity's Dispatch() method, which passes it the base event class pointer. The Call method receives this base event pointer, casts this pointer to the derived event pointer then using its pointer to the instance of the component, calls the component's member function with this derived event object as its argument.

And we are done connecting our components to specific events on our entities! Now that we have set up the connecting part of the event system pipeline, let's move on to wrting the mechanism with which to dispatch these events.

Dispatching an event

Since our engine uses events to pass data between components and systems, all events derived from a base event class. This allows us to have access the derived event classes through the base one.

The process of dispatching an event is done by the following steps:

1. First, we create a pointer to the event object that we are dynamically allocating:
   ```
   auto logicUpdateEvent = new Events::LogicUpdate();
   ```

2. Second, we set the values of any member variables of the object as we see fit.
   ```
   logicUpdateEvent->Dt = dt;
   ```

3. Third, using a reference to the entity we want to dispatch the event to, we call the Dispatch() method with the event class as the template argument and the event object itself as the other non-templated argument.
   ```
   gamesession_->Dispatch<Events::LogicUpdate>(logicUpdateEvent);
   ```

4. Fourth, we now delete the event object now that we are done with it. (Unless you w-wanna leak memory?)
   ```
   delete logicUpdateEvent;
   ```

The signature of the Dispatch() method for the entity class is as follows:

```
template <typename EventClass>
void Entity::Dispatch(Event * eventObj);
```

The Dispatch method takes two arguments: the class of the event as a template argument, and a pointer to the event object. Note that we are allowed to pass the derived event class object instead of the base one (because the derived one contains the base class).

The method does the following:

1. First, it constructs an std::type_index object with the event class as the argument:
   ```
   auto eventTypeID = std::type_index(typeid(EventClass));
   ```
   This type_index object will be used as the key into the map.

2. Second, it starts iterating through its map of <event class, delegate list> to look for a match for event classes. Since the map stores the event classes as type_index objects, we are able to compare them for matches.
   ```
   for (auto& eventKey : ObserverRegistry) {
   if (eventTypeID == eventKey.first) {
   ```

3. Third, once it finds a match, it then iterates through the list of delegates and invokes each delegate's 'Call' method by passing it the event object as the argument.

```
for (auto deleg : eventKey.second) {
deleg->Call(eventObj);
```

Once the Dispatch method has been implemented, the event system pipeline can be considered to be implemented! There are still situations to take into consideration, but for now the reader should be able to get it up and running and start iterating with it.
The reader should note that he need not maintain a list of enumerations as you add more events or components to the engine, as class type-matching will be done at runtime by the compiler through comparing std::type_index objects. Note that this is C++11 feature so make sure you are doing this on a compiler that can support it. If you got any questions about implementation, feel free to come talk to me.

Special thanks to

Arend Danielek: He who explained to me the architecture behind writing the Connect/Dispatch functions as well as that of events)
Gab Chenier: He who explained to me how delegates worked… with much patience.)
Trevor Sundberg: He who took time off his schedule to explain to me how templated delegates work and to dispatch events in a type-safe way.
Allan Deutsch: He who helped debug several revisions of the system. Without his keen powers of observation I would have been surely doomed to banging my head against the proverbial fragile laptop keyboard.