INPUT DEVICES FOR 2D GRAPHICS

Document Number:	P0249R2
Designated SG	13
Date:	10/18/2016
Reply-To:	brett.searles@attobotics.net

Contributing Authors:	Michael McLaughlin < mikebmcl@gmail.com > Jason Zink < jzink_1@yahoo.com >

I. Table of Contents

II. Introduction

The proposal describes the library that will support message handling. The main issue to be addressed is to make message handling in C++ to be comparable to other event based languages and to support Rapid Application Development techniques in C++. The proposal is composed of two (2) efforts. The first effort is to provide a framework for generic event handling for all types of events. The second effort is to employ the callback technique using old and current C++ methodologies for quickly creating events.

III. Motivation

The purpose of this work is two-fold.

1) To provide a process that C++ can be developed to handle Input-Output (IO) events like other event-based languages.
2) To provide developers ways to dynamically add, remove, invalidate and step through both synchronous and asynchronous events without the encumbrance of writing lengthy code.

Other languages strongly support Rapid Application Development of User Interfaces (UI) with canvas or surface objects connected to IO devices to handle events that are translated to the UI. These other languages also provide a consistent framework for creating and destroying events for multiple scenarios. One example is the way C# handles message handling between objects. Therefore, the effort of this standard was to eliminate macros or multiple layers of derivation from the base class, which is currently supported in existing libraries and design a framework that supports a dynamic and customizable approach to describing events. It also outlines a framework to create a sequence of events that are stored in a container.

This library can be applied to Multi-tasking, Real-Time Operating Systems and even operate in a bare metal environment. Therefore, the container that contains the sequence of events can be any type, including a custom type, based on the constraints of the system in which the events will be executed.

Some of the principles defined in the container that will hold the events have already been discussed in Boost using the Signals library[Boost]. The main difference here is that these libraries mainly treat their containers as queues, whereas the proposed library supports any type of container including the containers that have the flexibility of a **std::vector** container.

The framework is defined around a couple of base classes. They are all abstract classes because the actual event handling is dependent upon the developer's intent for the data or usage of the data.

A. Motivation Scope

The library that is being proposed covers two main problems to solve, which is the ability to

1) Provide a flexible framework to create Events and retrieve information from either hardware or software initiated devices,
2) Provide a collection that can sequence events

1. Scope: Provide a flexible framework to create Events, retrieve information

a. *Current practice*

OpenGL and COM-based languages use the callback methodology to create events[OpenGL1]. There are also libraries that use class types to create events and sometimes need to have multiple layers of inheritance to support some specialized events[Qt].

```
glutReshapeFunc( change_viewport );      // when window is resized
glutDisplayFunc( render );               // when window needs to be drawn
glutIdleFunc( animate );                 // when there is nothing else to do
glutMouseFunc( callback_mouse_button );  // when mouse is clicked
glutKeyboardFunc( keyboard_down );       // when a key is down
glutKeyboardUpFunc( keyboard_up );       // when the key goes up
glutPassiveMotionFunc( look );           // when mouse moves
glutMotionFunc( look );                  // when mouse drags around
```

b. What is being proposed

The library is designed to support inheritance, lambda functions and callback functions. The proposed classes are all pure virtual base classes to give developers and library writers the framework to build event handlers. This library is designed to support both polling and real-time event handling. There are two distinct differences between this library and previous implementations of other event handling libraries.
They are

1) Events can be overridden by changing the callback method
    1) Example of overriding a "Mouse Event"
        i. mouse_event((*newhandler)(e_args, 10, 100)); or
        ii. mouse_event([](e, 10, 100) { … }); or
        iii. (mouse_event, (*newhandler)(e_args, 10, 100)); or
        iv. (mouse_event, ([](e, 10, 100) { … }));
2) Support the option to allow for special effect animation to be added to event handlers
    1) Can define the time it takes to show the complete control (speed)
    2) Can define the time it takes to render the control (easing)

The event_args_base, which is the data retrieved from a device, the speed and easing variables are all optional. One item to note here is that the event_args_base class provides encapsulation of data so that it is harder to tamper with variables. While with other libraries, they use a naked callback function pointer, which could be more easily maligned. Therefore, the library has an inherent security elements already in place

2. Scope: Provide a collection that can sequence events

a. Current Practice

Many current event-based libraries written in C++ use the Observer Pattern. This pattern allows a developer to inherit from an observer object. The event-based objects subscribe to an Observer class. The Observer class has a container that contains the collection of subscribers. When the Observer object receives a notification, it iterates through the list of subscribers to inform them of a change. There are issues with this pattern. One is latency and information lost due to race conditions if there are too many subscribers to only one observer class. [OP]

Boost Signal Library allows users to create a collection of functions pointers and fires them in the order described by the developer. At first, the connection needs to be declared boost::signals2::signal<void ()> sig; Then the slots are filled

```
sig.connect(&print_args);
sig.connect(&print_sum);
sig.connect(&print_product);
sig.connect(&print_difference);
sig.connect(&print_quotient);
```

or they can be added in a sequence defined by the developer as

```
sig.connect(1, World());  // connect with group 1
```

```
sig.connect(0, Hello());  // connect with group 0
```

Then will be called when the signal object is initiated with the arguments that were described in the declaration

```
sig(5., 3.);
```

In the article about signals, the authors also showed how it could be used in event handling

```
// a pretend GUI button  class
Button
{
  typedef boost::signals2::signal<void (int x, int y)> OnClick;
public:    typedef OnClick::slot_type OnClickSlotType;
  // forward slots through Button interface to its private signal
boost::signals2::connection doOnClick(const OnClickSlotType & slot);

  // simulate user clicking on GUI button at coordinates 52, 38
void simulateClick();  private:
  OnClick onClick;
};

boost::signals2::connection Button::doOnClick(const OnClickSlotType & slot)
{
  return onClick.connect(slot);
}
void Button::simulateClick()
{
  onClick(52, 38);
}
void printCoordinates(long x, long y)
{
  std::cout << "(" << x << ", " << y << ")\n";
} [Boost]
```

There are two other aspects that the Boost library discusses that are employed in this proposal. The features provide techniques to scope and block slots.

jQuery, however, is the inspiration for this proposal because of the ease it is to create a sequence of events without using object inheritance.

```
   $('#toggleButton').bind('click',function(){
      $dataBox[$dataBox.is(':visible')?'hide':'show'](100,'swing',function(){ alert('end of animation');});
return false;
   })
```

Also jQuery events support animation in its inclusion of an easing and speed parameters. [jQuery1]

JQuery is a library that can better explain what the proposal is trying to accomplish. Basically, the developer can bind event handlers to a method call or an event handler. In order to execute each of the members of a sequence, the developer uses the trigger method to execute a specific event when iterating through the list of events as shown below:

```
return this.each(function () {
   var obj = $(this),
      oldCallback = args[args.length-1],
```

```
                newCallback = function () {
if ($.isFunction(oldCallback)){
                    oldCallback.apply(obj);
                }
                obj.trigger('after'+m);
            };

            obj.trigger('before'+m);
            args[args.length-1]=newCallback;

            //alert(args);
            F.apply(obj,args);

        });
```
[Rahen]

The only caveat is that the developer must explicitly trigger each event defined in a sequence using a **for** (**foreach**) loop.

b. What is being proposed

The library is designed to support multiple container objects filled with sequencing events. The difference between this library and JQuery is that each event will be fired implicitly in the order described by the developer and that the developer can order or filter how the events will be triggered.

The containers will also have features like functionality described in Boost Signals, where a slot can be blocked, scoped or inserted. However, the new library would expand available operations to allow for events to be removed, sequenced, and operated in reverse.

IV. Impact On C++

The proposal is only a framework to standardize event handling and does not add any new functionality, operators or keywords beyond what C++ 11 supports.

V. Design Decisions

The main focal point of the proposal was to design a framework and library to support flexibility and extensibility.

Use of Callbacks

Callback functions are useful to execute in specified ways depending upon context state. This allows for flexibility in code generation so that the developer can switch functionality easily based ion various inputs. In the graphics world, multiple libraries support event handling via the **callback** technique.

Native C++ [MS01], COM C++ [MS02], OpenGL, C#, and jQuery utilize callback functions to allow for one entry point to support multiple functionality. As in the case of Native C++, the developer would use the _hook method that would attach a receiver object to the source object as follows: __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);

In OpenGL, as the receiver of keyboard events, the developer can write **void glutSpecialFunc(void (*func) (int key, int x, int y));**[OpenGL2] This method uses the function pointer as a parameter.

In this proposal, the callback technique is quite heavily relied on to allow developers to override base class functionality.

Sequencing of Events

There is precedence in the ability to fire more than one event within a sequence.

In Native C++, there is the support to create hooks to multiple events and point to different callback functions. Then when a Source event function is acted upon can call all of the events in the Receiver that have been hooked. The same holds true for COM related event handling. jQuery also supports multiple event handlers being fired in sequence.

In this proposal, sequencing of events is also supported. The main difference is that sequence can be filtered dynamically by the developer.

Removing

A majority of graphic libraries support the removing of events. In Native and COM related C++, the developer would unhook an event handler. In C# [MS03], the developer can remove an event using the -= operator. This is important because maybe the developer only wants the user to use functionality one time. Therefore, it is necessary to allow developers the capability to remove events.

Blocking

Few libraries allow for blocking callbacks. However, in Boost Signals, there is allowance to let developers block slots in a collection as shown below

```
boost::signals2::connection c = sig.connect(HelloWorld());
  std::cout << "c is not blocked.\n";
sig(); // Prints "Hello, World!"

{
    boost::signals2::shared_connection_block block(c); // block the slot
std::cout << "c is blocked.\n";
    sig(); // No output: the slot is blocked
  } // shared_connection_block going out of scope unblocks the slot
std::cout << "c is not blocked.\n";   sig(); // Prints "Hello,
World!"}
```

In this proposal, the event container will support this process. The main difference is that Signals makes this blocking temporary, while this proposal will make it last until the developer suggests otherwise.

Animation

The proposed extension of allowing animation to be supported in this document is that these variables are optional parameters to be supplied. The only supporting precedence is found in jQuery and that library also makes support for animation optional as well.

VI. Technical Specification

For this standard, there will be two **base** classes defined that will provide the framework for developers to utilize data received from low-level device handlers to affect the User Interface(UI) Surface object and a collection object to sequence events around a particular device trigger. These objects are defined as follows:

1) event_base
2) event_args_base
3) event_base_container

event_base will define how the UI will be manipulated once an interrupt is fired by the device. event_base can be inherited in order to customize the object in order to fit the software requirements. There is a listing of potential events that can be handled at the end of the document. It also has the feature that the callback function can be overwritten using the **( )** operator. The callback function may either be a specific function that has a signature or a lambda method.

event_args_base is the data that is sent to or received from a device. This will be one of the arguments used in the event_base implementation of the Fire method. Since this is a framework class, it will need to be implemented by the device manufacturer to provide the data context and memory location that is to be used by the events when fired.

event_base_container is the container that may contain a sequence of events that will execute due to an external device's input. It is designed to give developers the framework and flexibility to add, insert, remove, or invalidate events within an existing collection around a specific input device source.

A. event_base

event_base is a pure virtual base class that can be inherited for each type of event that the developer would like to create. Since a control object, event_base is designed so that it cannot be copied or moved.

event_base_args is the data received from a device, which in the event allows for the manipulation of the UI.

The basic structure of the class event_base will be as follows:

```
class event_base
{
        bool is_set = true; // used by the container to turn off events for certain sequences

        template <class U> void write_isr(uint_t int_handler,   U* this);

        void (*event)(event_base_args& eventargs = NULL, int speed = 0, int effect = 0)); std::function<void>
        fevent_base;

        virtual void Fire(void  (*event)(event_base_args& eventargs = NULL, int speed = 0, int effect = 0)) ;
        virtual void Fire(void) = 0; // maybe the class that inherits event_base will have its own
        implementation

        const std::string _name;

    public:
        event_base(std::string name) : _name(name);
        virtual ~event_base();
```

```
        event_base& operator()(void (*event)( event_base_args& eventargs = NULL, int speed = 0, int effect
                                                                                                = 0) );
        event_base& operator()(std::function<void>(( event_base_args& eventargs = NULL, int speed = 0,
                                                                                        int effect = 0));
        event_base& operator()(const event_base* evt, void (*event)( event_base_args& eventargs = NULL,
                                                                        int speed = 0, int effect = 0) );
        event_base& operator()(const event_base* evt, std::function<void>((event_base_args& eventargs =
                                                                        NULL, int speed = 0, int effect = 0));

        event_base(const event_base& obj) = delete;
        event_base(event_base&& obj) = delete; event_base&
        operator=(const event_base& obj) = delete;
        event_base& operator=(event_base&& obj) = delete;

        void set_in_set(bool);

};
```

Therefore, events can use callbacks to describe what the event will do when fired. The callbacks can be external functions or lambda statements using **std::function<T>**.

As shown in the Scope statements, there is no need to inherit new events except the original implementation of the abstract class. If that implementation exists, functionality can be written in a callback function with access to the event_args_base data. This allows developer to change the original functionality in a manner that complies with specifications. This function will be executed when the event is fired. To demonstrate, let us assume that a developer wants to write an event handler for a mouse click. If the library has a MouseClick implementation written, the developer only needs to overwrite the pointer to the function that will be executed when the MouseClick event is activated. The developer can also create a lambda function to overwrite as well. However, for events that do not have an input associated with it, then the developer will need to create a new class object.

The class can be instantiated in two ways. If for example the event is an individual event based on an input device, the developer needs to supply the device that the event is attached. Then when the device executes the Trigger function, it will execute the events Fire method. However, if the event is part of a sequence of events, the class is instantiated with only the name parameter supplied. It would be nice to have that parameter be an **enum** for reliability, yet it would restrict the flexibility of the developer to provide custom events.

To register the event to be fired when data has been updated in the Interrupt Service Routine (ISR), the event will need to use the **write_isr** function when the event subscribes. This function will add the event to a container existing in the ISR.

The purpose of the **is_set** variable is that it will tell the sequence container object whether to fire the event or not. It is default equal to true. Once set to false, the developer will need to reset it back to true with one of the functions described in the event_base_container class.

B. event_args_base

The basic structure of the class event_args_base will be as follows:

class event_args_base

```
{
        virtual void set_data(void) = 0; // still need to determine the exact method

    public:

        event_args_base ();
        virtual ~ event_args_base ();

        virtual event_args_base get_data() = 0;

        event_args_base (const event_args_base & obj) = delete;
        event_args_base (event_args_base && obj) ) = delete;
        event_args_base & operator=(const event_args_base &obj) ) = delete;
        event_args_base && operator=( event_args_base &&obj) ) = delete;

}
```

event_args_base is a class that is designed to hold a memory location where the device can communicate its data. It provides a context for storage of device data to be used by the event.

Therefore, an event can use this location to send data to the device or receive data from the device. Since the data is specific to a device, the object's data is not transferrable to other event_base_args derived objects. However, there may be more than one event_base object that uses only one set of data from a device. Examples are that the mouse device firmware would provide certain data, yet there may be events like,

MouseClick,
MouseDown,
MouseUp,
MouseEnter

that would use the same data. Therefore, there is only need for one event_base_args to contain data that the mouse transmits, yet can be used by multiple events that use that data.

This class is the base class for all device data and needs to be configured specifically to the data that the developer wants to pull from or push to a device. So the developer would write derived classes for each device.

C. event_base_container

event_base_container is used to contain a sequence of events in a container. The class is shown below as follows:

```
class event_base_container
{
        template <class U> void write_isr(uint_t int_handler,   U* this);


        // ************************
        //  operators listed below
        // ************************
```

```
        void (*fptr)();
        template<class T = std::vector<*event_base>> T events;
    public:

        event_base_container() ;

        event_base_container() = delete;
        event_base_container (const event_args_base & obj) = delete;
        event_base_container (event_args_base && obj) ) = delete;
         event_base_container & operator=(const event_args_base &obj) ) = delete;
        event_base_container && operator=( event_args_base &&obj) ) = delete;

        void (*Execute)();
        void assign(void (*fptr)());
}
```

The Execute function is the action that will start the firing of events in the container. Based on the container used by the developer to create a sequence of events, the Execute method will iterate through the collection. So if the developer uses a **std::list** container, the iterator will be forward only. However, if the developer uses **std::vector**, the developer could use a Random Access Iterator. So some functionality described below will depend upon the container used to store the events.

With that in mind, the container can be even a custom container written by the developer. The function pointer would allow the developer to customize the execution of the events. If not set, the **Execute** function will just iterate through the events as sequenced by the developer. The **assign** function will allow the developer to set the function pointer.

Below is a list of the event_baseContainer functions/operators:

1. To add a single event

    a. control + (void *event(event_args_base eventargs = NULL, int speed = 0, int effect = 0))
    b. control + (std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0))

The + operator adds events to a “control” object.

2. To add multiple chained events need to use **+ operator** in sequence with the events that are added

    a. control + (void* event (event_args_base eventargs = NULL, int speed = 0, int effect = 0)) [+ (void *event(event_args_base eventargs = NULL, int speed = 0, int effect = 0))]**
    b. control + (std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0))[+ (std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0))] **

3. To add a single event to the end of the list, use the postfix **++ operator**

    a. control(void* event (event_args_base eventargs = NULL, int speed = 0, int effect = 0))++;
    b. control(std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0))++;

4. To add a single event at the beginning of a list, use the prefix **++ operator**

a. ++control(void* event (event_args_base eventargs = NULL, int speed = 0, int effect = 0));
b. ++control(std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0));

5. To insert an event chained event, need to use the **insert** method

Definition of newevent_base

a. newevent_base = (event, (*fptr)(event_args_base eventargs = NULL, int speed = 0, int effect = 0))
b. newevent_base = (event, std::function<void>(event_args_base eventargs = NULL, int speed = 0, int effect = 0))

Can be added to the chain via the event before the new event to be inserted

a. control.insert(beforeevent_base, newevent_base)[.insert (beforeevent_base, newevent_base)] **

Or the index of the event before the new event

a. control.insert(index, newevent_base) [.insert(beforeevent_base, newevent_base)] **

5. To remove an event use **- operator**

a. control - (event);
b. control - (index);

6. To remove multiple events

a. To remove from an event to the end of the list use the postfix **-- operator** method

- control.(event)--
- control.(index)—

b. To remove from an event to the beginning of the list use the prefix **-- operator** method

- --control.(event)
- --control (index)

c. To remove certain events in a list use **remove**

- control.remove(event)[.remove(event)] **
- control.remove(index)[.remove(index)] **

7. To prevent an event from firing, use **overrule**

control.overrule(event);

8. To filter events that you want to allow for certain processes to fire without removing elements from container.

control | event[x_0] | event[x_1] … | event[x_n];

9. To remove the restriction of event filtering, use the **resume** method.

control.resume();

10. To indicate that the events start at a location within the container, need to use the **start_at** method

    control.start_at($event_n$);
    control.start_at(index);

    This will fire the events starting at the event named in the parameter

The following will work if the container has a backward iterator:

11. To start the event firing sequence from the last item in the container and iterate backwards, need to use the **reverse** method.

    control.reverse();

12. To start the event firing sequence at a certain index within the container and iterate backwards, need to use **reverse_at** method.

    control.reverse_at($event_n$);
    control.reverse_at(index);

The concept of "control" is a placeholder for the **surface** object that is described in the 2D graphics standard. It can also be a placeholder for a thread or networking object.

D. Examples of events to be handled by event_base

Timer

Keyboard
- OnKeyUp
- OnKeyDown
- OnKeyPress

Mouse
- OnClick
- OnMouseOver
- OnMouseDown
- OnMouseUp
- OnMouseMove
- OnContent

Touch
- OnTouch
- OnSwipe
  - SwipeUp
  - SwipeDown
  - SwipeLeft
  - SwipeRight
- OnTouchPosition
  - TouchStart

TouchEnd
TouchCancel
TouchMove
TouchHold

Show
Show
BeforeShow
AfterShow
Hide
BeforeHide
AfterHide

OnDraw

Before
After

VII. Future Work to Consider

1. Completion of Interface between the "control" objects and the event handling objects
2. Security
3. Simpler coding of the event sequencing defined in the event_base_container class.

6. Acknowledgements

Would like to acknowledge Michael McLaughlin for inspiring me to work on the standard. Without his efforts on developing the standard for 2D Graphics, this would not have been possible. Would also like to thank Jason Zink for his input about other work that is currently being done which allowed me to show that this standard emphasizes current work done already. Of course, without Herb Sutter's input, none of this work would have taken place. Would like to thank my son, Christopher, because he is the biggest motivation in my life.

7. References

[Boost] http://www.boost.org/doc/libs/1_59_0/doc/html/signals2/tutorial.html

[Qt] http://doc.qt.io/qt-5/eventsandfilters.html

[OP} http://stackoverflow.com/questions/11619680/why-should-the-observer-pattern-be-deprecated

[jQuery1] https://api.jquery.com/category/events/

[jQuery2] https://api.jquery.com/bind/

[Rahen] http://jsfiddle.net/rahen/eUaAw/5/

[OpenGL1] http://www.c-jump.com/bcc/common/Talk3/OpenGLlabs/c262_lab08/c262_lab08.html

[OpenGL2] http://cs.brynmawr.edu/Courses/cs312/fall2010/lectures/gl_02.pdf

[MS01] https://msdn.microsoft.com/en-us/library/ee2k0a7d.aspx

[MS02] https://msdn.microsoft.com/en-us/library/hdcxwbd5.aspx

[MS03] https://msdn.microsoft.com/en-us/library/ms366768.aspx

** [] means optional.