INPUT DEVICES FOR 2D GRAPHICS

Extension of the 2D Graphics Proposal

Brett Searles < brett.searles@attobotics.net >
Michael McLaughlin
Jason Zink

Introduction

- Historical perspective
  - Started with C++
  - GPU, GPCPU
- General methodology to developing 2D Graphics
- Need to include Input from external devices
- Two efforts covered in this proposal
  - Create a framework to support event handling
  - Create a framework to support interrupt handling

Motivation

- To provide a process that C++ can be developed to handle Input-Output (IO) events similar to other UI based languages.
- To provide developers ways to dynamically add, remove, invalidate and step through both synchronous and asynchronous events without the encumbrance of writing lengthy code.
- To construct a framework for present and future development of IO devices, its data and its usage of that data.

Framework handle Input-Output (IO) events

- Other RAD languages have strong support
- To be used for both Multi-tasking and Real-Time Oses

Flexibility

- In development
- In operation

Extensibility

- Easily create new event and interrupt handlers
- Easily override existing event handlers
- Easily chain and sequence events

Scope

Framework handle Input-Output (IO) events

Flexible framework to create events

Current Practice

Boost

Signals

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

- Connection

```
boost::signals2::signal<void ()> sig;
```

- Add Slots

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

- Fire

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

Flexible framework to create events

Current Practice

Boost

```
// 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";
}
```

Flexible framework to create events

Current Practice

QT

- Observer Pattern
  - Virtual Base Class
  - Derived Classes register their event
  - Background Application notifies
  - Incorporated some Signals Library
  - Mainly around menus items

Flexible framework to create events

Current Practice

jQuery

- Ease it is to create a sequence of events without using object inheritance
- Include animation in the event with speed and easing parameters

Flexible framework to create events

Proposed

Allow

Inheritance

Methods to be overwritten

Lambda

Callback functions

To support polling or real-time event-handling

To support animation

- Events can be overridden by changing the callback method
  - Example of overriding a “Mouse Event”

```
mouse_event((*newhandler)(e_args, 10, 100));

mouse_event([](e, 10, 100) { … }));

(mouse_event, (*newhandler)(e_args, 10, 100));

(mouse_event, ([](e, 10, 100) { … }));
```

- Add special effect animation to the events
  - Can define the time it takes to show the complete control (speed)
- Can define the time it takes to render the control (easing)

Provide a collection that can sequence events based on a device interrupt

Sequence events

Current Practice

QT

Signal Mapper

```
signalMapper = new QSignalMapper(this);
signalMapper->setMapping(taxFileButton, QString("taxfile.txt"));
signalMapper->setMapping(accountFileButton, QString("accountsfile.txt"));
signalMapper->setMapping(reportFileButton, QString("reportfile.txt"));

connect(taxFileButton, SIGNAL(clicked()), signalMapper, SLOT (map()));
connect(accountFileButton, SIGNAL(clicked()), signalMapper, SLOT (map()));
connect(reportFileButton, SIGNAL(clicked()), signalMapper, SLOT (map())); [QT3]
```

Sequence events

Current Practice

Allegro

Event_Queue

```
while(1)
            {
    ALLEGRO_EVENT ev;
    ALLEGRO_TIMEOUT timeout;
    al_init_timeout(&timeout, 0.06);

    bool get_event = al_wait_for_event_until(event_queue,  &ev, &timeout);

    if(get_event && ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
      break;
    }

    al_clear_to_color(al_map_rgb(0,0,0));
    al_flip_display();
}
```

Sequence events

Current Practice

jQuery

Each loop

Trigger

```
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);

        });
```

Sequence events
Proposed

- Library to contain multiple containers
- To allow easy
  - Addition
  - Insertion
  - Removal
  - Invalidation
  - Operation

Portable Interrupt Framework

- Currently no specification in the C++ Standard
- Framework for firmware developers to store device data in objects easily accessible by event handlers
- To encapsulate event handling so that when an interrupt is set that it will trigger these events

Technical Specification

- For events, there are two(2) base classes and one container
  - event_base
  - event_args_base
  - event_baseContainer
- Interface between the events and the 2D Graphics object
  - io_surface
- Base class for describing the device and interaction with events
  - device_base

event_base

```
class event_base
{
        bool inset = true; // used by the container to turn off events for certain sequences
        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;
        const device_base* _device;


public:
        event_base(std::string name) : _name(name);
        event_base(std::string name, device_base *device)
        : _name(name), _device<event_base>(device, this);
        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 setInSet(bool);


};
```

event_args_base

```
class event_args_base
{
        virtual void SetData(void) = 0;
public:

        event_args_base ();
        virtual ~ event_args_base ();

        virtual event_args_base GetData() = 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_baseContainer

```
class event_baseContainer
{
        // ************************
        //  operators listed below
        // ************************

        const device_base*_device;

        std::vector<event_base> events;
public:

        event_baseContainer(const device_base *device)
        : _device<event_baseContainer> (device, this)
        { }

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

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

Add a single event

a. control.event (void (*event)(event_args_base eventargs = NULL, int speed = 0, int effect = 0))

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

Add multiple chained events

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

Add a single event to end of list

a. control.add(void (*event) (event_args_base eventargs = NULL, int speed = 0, int effect = 0));

b. control.add(std::function<T>(event_args_base eventargs = NULL, int speed = 0, int effect = 0));

Insert an event chained event

Definition of newevent_base

a. newevent_base = ((*fptr)(event_args_base eventargs = NULL, int speed = 0, int effect = 0))

b. newevent_base = (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

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

Or the index of the event before the new event

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

Remove an event

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

Remove multiple events

a. Use the **wipe** method
   - control.wipe(event)
   - control.wipe (index)

b. To remove certain events in a list use **remove**
   - control.remove(event)[.remove(event)] **
   - control.remove(index)[.remove(index)] **

Add asynchronous event

use **async** method

async(void (*fptr)()))())[.async (void (*fptr2)())]**

async(std::function<void>()) [.async (std::function<void>())]**

Use **random** method

random(void *fptr())[.(void (*fptr)())]**

random(std::function<void>())[.(std::function<void>())]**

Remove asynchronous event

use the **stop** method

stop(event)

Prevent an event from firing
-- overrule

```
control.overrule(event);
```

Filter Events

```
control | event[x0] | event[x1] … | event[xn];
```

Remove restriction
-- continue

```
control.continue();
```

Change start
-- startAt

```
control.startAt(eventn);
control.startAt(index);
```

Reversing Events

reverse

```
control.reverse();
```

reverseAt

```
control.reverseAt(event_n);
control.reverseAt(index);
```

io_surface

Interface between the event handlers and the 2D Graphics **surface** class

```
class io_surface
{
        std::vector<event_base> detached_events;
        std::vector<event_baseContainer> sequenced_events;
        std::vector<device_base> devices;
        template <class T> std:multimap<device_base*, T> map;
public:
/*
        Possible methods that could be used for the surface (2D) object to use to add devices or events:
*/

        void AddDevice(const device_base* device);
        void DeleteDevice(device_base *device);
        void AddEventContainer(const event_baseContainer* baseContainer);
        void DeleteEventContainer(event_baseContainer* baseContainer);
        void AddDetachedEvent(const event_base* event);
        void DeleteDetachedEvent(event_base* event);
}
```

device_base

Portable Interrupt Framework

```
class device_base
{
const int[32, 64, 128]* int_handler;
const int[32, 64, 128]* timer_handler;

/// triggerFlag is to give the state of the device. If the device is busy or suspended, the timer
/// interrupt is enabled to monitor the trigger until the ready state is set
/// once set, the timer interrupt will call the Trigger method, which in turns fires the events
/// listed in the container, event_baseContainer.
/// Otherwise, the interrupt handler directly will call the Trigger method.

                volatile int triggerFlag;                       // will be updated by Interrupt Service Routine (ISR) for a particular
// device like mouse, keyboard
// 00001 means      in-use
// 00010 means      ready
// 00100            “                       busy
// 01000            “                       suspend
// 10000            “                                               device error

const int[32, 64, 128] deviceId;

void (*fptr)();

virtual void Trigger() = 0; // trigger would execute a callback in the container of events

event_args_base& _args;

                public:

                device_base();

                template<typename U>
// used for copy constructor in initialization list of
// event_base or event_baseContainer
                // set the function pointer so when an interrupt is
                // triggered, it will call the right function
                device_base(const device_base<U>& device, U&& u)
                {
                                fptr = u->[Execute|Fire]();
                                std::forward<device_base>(device);
                }

                virtual ~device_base();

                event_args_base getArgs(void);

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

}
```

Examples of events to be handled

- 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
  - BeforeShow
  - AfterShow
- Hide
  - BeforeHide
  - AfterHide
- OnDraw
  - Before
  - After

Future Work to Consider

1. Completion of Interface between the 2D Surface object and the event handling objects
2. Messaging
    a. Synchronous
    b. Asynchronous
3. Multithreading
    - Need to consider synchronization of events for messaging
4. Security
5. Support more input devices
6. Parallel Hardware handling of events
7. Simpler coding of the event sequencing defined in the event_baseContainer class.

Acknowledgements

- Michael Mclaughlin
- Jason Zink
- Herb Sutter