Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #12

Event Filtering

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 6.0). Information online at: `http://dragonfly.wpi.edu/book/`

Copyright ©2012–2019 Mark Claypool and WPI. All rights reserved.

4.15 Filtering Events (optional)

Up to now, all game objects get every event when Manager `onEvent()` is called. For example, in the GameManager game loop, all Objects get a step event every loop iteration. This is useful for Objects that need to do something each step, like the Saucer Shoot Hero that uses step events to determine when it can shoot again (see Listing 3.5 on page 25). But many game objects do not need to update themselves each step event – such as walls, trees or rocks. The general idea, that not all objects want all step events, holds for other events, too. For example, keyboard events, generated when a player presses keys, are usually only handled by the object that the player controls (e.g., the Hero). The way the event handler works (see Listing 4.57), an event that is not acted upon can just be ignored, but that still means the Object `eventHandler()` method is invoked, which is inefficient at best, and can lead to unexpected errors if the game code does not ignore events properly, at worst.

The solution is to filter events, only passing specific events to Objects if they have indicated interested in events of that type. When an object does want a specific event, it registers with the manager in charge of that event. For example, an Object that wants to get step events registers interest in that event with the GameManager. When the event occurs, the manager invokes the `eventHandler()` method only for those Objects that are interested. Continuing the example, every game loop, the GameManager does not send an EventStep to all Objects (as it does currently), but only to those Objects that have registered interest. When an Object is no longer interested in an event, it explicitly un-registers interest. Note, the engine will do this un-registration automatically, too, when an Object is removed from the game world. Otherwise, an Object would receive an event even though it had been removed and deleted, a certain error.

From another vantage, providing for updates to Objects when events occur, and only providing the updates to interested objects is a form of an *observer* design pattern (sometimes called a *publish-subscribe* design pattern).

Figure 4.8 depicts the general idea. Objects, depicted by the grey boxes on the bottom, register their interest in an event with the appropriate manager, depicted by the ovals at the top. Objects may be interested in more than one event or in no events at all. For example, Object 1 is interested in only one event managed by Manager A, while Object 3 is interested in two events managed by Manager C and one event managed by Manager B, and Object n is interested in no events at all. Managers keep track of which Objects are interested in the events they manage, hence the two-way arrows. Managers may be responsible for no events (e.g., the LogManager), or many events. Even Managers that are responsible for one or more events may still have no Objects that are interested. For example, this may be the case for Manager X in Figure 4.8.

Figure 4.9 depicts a close-up view of data structures inside Object and Manager needed to support filtering events. The Object on the left, using Object 3 from Figure 4.8, keeps the names of the events in which it is interested in an array of strings, called `event_name[]`. It also has an attribute, `event_count`, for stores a count of the number of events in which it is interested.

The Manager on the right, Manager C from Figure 4.8, has an array, `event_name[]`, of the names of interested events as strings which aligns via a parallel array with `obj_list[]` which stores the Objects that are interested in that event. The attribute `event_count`

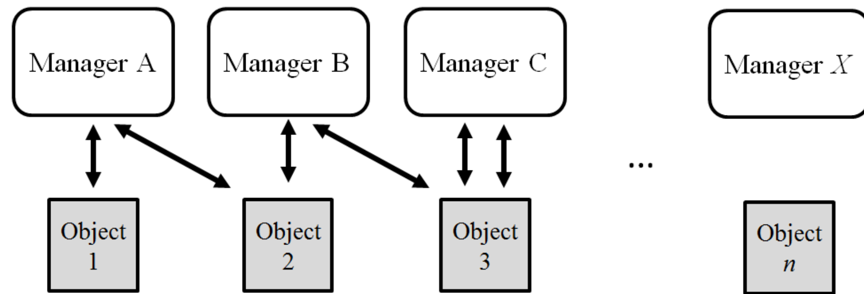

Figure 4.8: Event interest

stores the count of the number of events in which it is interested.

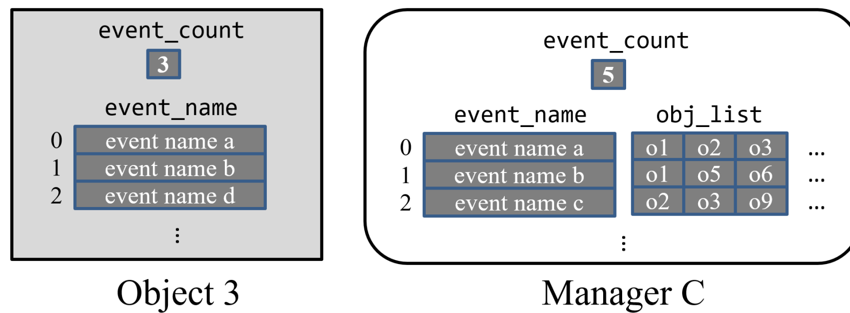

Figure 4.9: Event interest (zoom)

In order to build this functionality, the Manager class needs to be extended to support event interest management. To start, the Manager stores a list of all the events in which Objects have registered interest and, for each event, a list of the Objects that have registered interest. Listing 4.163 shows code fragments that provide attributes and methods for the Manager class to support interest management.

Listing 4.163: Manager extensions to support interest management

```
0 const int MAX_EVENTS = 100   // Maximum number of different events.
1
2  private:
3   int event_count;                          // Number of events.
4   std::string event[MAX_EVENTS];            // List of events.
5   ObjectList obj_list[MAX_EVENTS];          // Objects interested in event.
6
7   // Check if event is handled by this Manager.
8   // If handled, return true else false.
9   // (Base Manager always returns false.)
10  virtual bool isValid(std::string event_name) const;
11
12 public:
13  // Indicate interest in event.
14  // Return 0 if ok, else -1.
```

```
15   // (Note, doesn't check to see if Object is already registered.)
16   int registerInterest(Object *p_o, std::string event_type);
17
18   // Indicate no more interest in event.
19   // Return 0 if ok, else -1.
20   int unregisterInterest(Object *p_o, std::string event_type);
21
22   // Send event to all interested Objects.
23   // Return count of number of events sent.
24   int onEvent(const Event *p_event) const;
```

MAX_EVENTS is defined to be 100, which is large enough for most games. In fact, most games have far fewer than 100 different types of events – typically no more than 10 – but any extra, unused capacity has little overhead.

The private attributes provide data structures to store the events. The variable event_list_count keeps track of how many unique events this manager has been asked to register for (initialized to 0 in the constructor). The events themselves are stored as strings (the event type, as specified in Listing 4.48) in the array event[], and the objects registered for the corresponding event are held in obj_list[]. Note that event[] and obj_list[] are parallel arrays, so that the ith element of event[] corresponds to the ith element of obj_list[].

When an Object is interested in an event handled by a particular Manager, it invokes the method registerInterest(), providing a pointer to the Object itself (i.e., this) and the event name.

When an Object is no longer interested in an event (and when it is being deleted), it invokes the method unregisterInterest(), providing a pointer to itself and the event name.

In order to protect a game programmer from registering for interest in an event with a manager that does not handle that event, the Manager checks the isValid() function before accepting the registration. This method is virtual so that derived managers can specify which game events, if any, they manage. When an event occurs, the onEvent() method iterates through the list of all Objects that had registered for interest in the event and sends each of them the event by invoking their eventHandler() methods.

The method registerInterest() is provided in Listing 4.164. The first block of code, lines 4 to 9, checks to see if there is already an Object that has registered interest in this event. If so, line 6 adds the indicated Object to the list.

The second block of code, lines 12 to 18, is triggered when the event being registered for has no other Objects in the list. In this case, the arrays are first checked to see if the maximum number of events has been reached. If so, the routine needs to return an error – it will be up to the game code to figure out how to proceed.[20] If there is room, the event and Object are added to the end the lists and the number of events (event_count) is incremented.

Listing 4.164: Manager registerInterest()

```
0 // Indicate interest in event.
1 int Manager::registerInterest(Object *p_o, std::string event_type)
```

[20] Another reason that all system and engine calls should be error checked!

```
2
3   // Check if previously added this event.
4   for i = 0 to event_count-1
5     if event_name[i] is event_type then
6       insert object into obj_list[i]
7       return // Ok.
8     end if
9   end for
10
11  // Otherwise, this is new event.
12  if event_count >= MAX_EVENTS then
13    return // Error, list is full.
14  end if
15  event_name[event_count] = event_type
16  clear obj_list[event_count]  // In case ''re-using'' scooted list.
17  insert object into obj_list[event_count]
18  increment event_count
19
20  return // Ok.
```

`unregisterInterest()` is shown in Listing 4.165. The first block of code, lines 4 to 8, looks for the event that is being unregistered for. When found, the corresponding Object is removed in line 6. Note, if event is not found, the method returns an error (e.g., -1) – this is *not* shown in the pseudo code. Also, if the Object to be removed is not found in `obj_list`, an error should be returned.

The second block of code, lines 11 to 15, checks if the Object list at the spot of the event is empty. If so, it "scoots" the items in `event` and `obj_list` over and reduces the list count by one. Line 31 of Listing 4.31 shows an example of how this is done for an array of integers.

Listing 4.165: Manager unregisterInterest()

```
0 // Indicate no more interest in event.
1 int Manager::unregisterInterest(Object *p_o, std::string event_type)
2
3   // Check for event.
4   for i = 0 to event_count-1
5     if event_name[i] is event_type then
6       remove object from obj_list[i]
7     end if
8   end for
9
10  // Is list now empty?
11  if obj_list[i] is empty then
12    scoot over all items in object_list[]
13    scoot over all items in event_name[]
14    decrement event_count
15  end if
16
17  return // Ok.
```

With the Manager storing events and objects registered for them, the `onEvent()` method can be refactored, as shown in Listing 4.166. Line 5 iterates through all the events that have been registered for, and line 7 compares the type of the event (via the `getType()`

method of Event) to the event list items. If the event is found, the code on lines 7 to 12 iterates through all Objects, invoking the eventHandler() method for each Object, passing in the event (p_event). The count of events sent is also incremented, and returned when the method ends.

Listing 4.166: Manager onEvent() refactored to support filtering events

```
0  // Send event to all interested Objects.
1  // Return count of number of events sent.
2  int Manager::onEvent(const Event *p_event) const
3    count = 0
4
5    for i = 0 to event_count-1
6      if event_name[i] is p_event type then
7        create ObjectListIterator li on obj_list[i]
8        while not li.isDone() do
9          invoke li.currentObject() -> eventHandler() with p_event
10         increment count
11         li.next()
12       end while
13   end for
14
15   return count
```

With the above code in place, all Objects no longer receive a step event. Instead, only those that have done a registerInterest(STEP_EVENT) with the GameManager get the step event. Similarly for the keyboard and mouse events.

Not all managers handle all events. For example, it makes no sense for a game object to register for interest in a step event with the WorldManager (or LogManager!). In order to help the game programmer from making the mistake of registering for interest with the wrong manager, each manager defines an isValid() function. The base class Manager isValid() method is declared as in Listing 4.167. The method is declared virtual so that derived classes, such as the GameManager, can define their own isValid() methods, as appropriate. The base Manager isValid() always returns false – there are no real instances of the base Manager class and, if there were, it would not handle any events.

Listing 4.167: Manager isValid()

```
0    // Check if event is handled by this Manager.
1    // If handled, return true else false.
2    virtual bool isValid(std::string event_type) const;
```

The pseudo code for each derived manager class isValid() is shown in Listing 4.168. Note, the "DerivedManager" is not the real name of the class – rather, the name is replaced with the actual manager name (e.g., GameManager) when defined. The body of the method checks if the event is a valid event (e.g., a STEP_EVENT in the GameManager). If so, it returns true. Otherwise, it returns false.

Listing 4.168: Derived manager isValid()

```
0  // Check if event is allowed by this Manager.
1  // If allowed, return true else false.
2  bool DerivedManager::isValid(std::string event_type) const
```

```
3
4   if event_type is VALID EVENT1 then
5     return true
6   end if
7
8   if event_type is VALID EVENT2 then
9     return true
10  end if
11
12  ...
13
14  return false
```

As a specific example, pseudo code for the InputManager `isValid()` method is shown in Listing 4.169.

Listing 4.169: InputManager isValid()

```
0 // Input manager only accepts keyboard and mouse events.
1 // Return false if not one of them.
2 bool isValid(std::string event_name) const
3
4   if event_name is keyboard event
5     return true
6   else if event_name is mouse event then
7     return true
8   else
9     return false
10  end if
```

The `isValid()` method needs to be defined for the GameManager, Input Manager and WorldManager (which accepts all events the other two Managers do not).

With `isValid()` defined, the Manager method `registerInterest()` is extended to call `isValid()` before adding the game object to the list of interested objects. If `isValid()` returns `false`, the Object (and event) are not added.

The last bit of bookkeeping that needs to be done is to extend the Object class so each object keeps track of the events it has in which it has registered interest. That way, if an Object goes out of scope (is deleted) it can automatically unregister for interest in all events. Not doing this automatic unregistration would mean if the event occurred, the manager would try to send the event to the object, but since the object was deleted and the memory no longer allocated, a segfault (a spurious memory error) would occur.

Methods `registerInterest()` and `unregisterInterest()` are declared as in Listing 4.170. Like the Manager's methods, the Object's interest management methods both return 0 if successful, and -1 if there is an error. Unlike in the Manager, the Object's methods only take the event string they are interested in. The array on line 2 and associated integer on line 1, are to keep track of the events this Object has registered in. Only the string is needed here, since each event type matches up uniquely with a specific manager. For example, if the Object is interested in a step event, that is registered only with the GameManager.

Listing 4.170: Object class extensions for registerInterest()

```
private:
 int event_count;
 std::string event_name[MAX_OBJ_EVENTS];

public:
 // Register for interest in event.
 // Keeps track of manager and event.
 // Return 0 if ok, else -1
 int registerInterest(std::string event_type);

 // Unregister for interest in event.
 // Return 0 if ok, else -1
 int unregisterInterest(std::string event_type);
```

The Object's `registerInterest()` method is provided in Listing 4.171. The first block of code checks to see if there is room in the array of events by comparing `event_count` to the maximum (`MAX_OBJ_EVENTS`, defined in `Object.h` to be some reasonable maximum say, 100).

The next block of code checks to see if the event is a step event. If so, it registers for interest with the GameManager by calling `registerInterest()`, passing in the pointer to the current Object (`this`) as well as the event string. As more managers are defined (e.g., InputManager), more cases can be handled similarly to the step event (in the "..." region in line 14). By default, all remaining events are handled by the WorldManager – that way, user defined events (such as the "nuke" in Saucer Shoot, see Section 3.3.8 on page 33) can be accommodated. Note, the `registerInterest()` method call to each individual manager can fail, depending upon their definition of `isValid()` and the number of Objects already registered, so the calls should be error checked.

The last block of code starting at line 20 keeps track of the event name that has been registered by adding it to the array and incrementing the count of events.

Listing 4.171: Object registerInterest()

```
// Register for interest in event.
// Keeps track of manager and event.
// Return 0 if ok, else -1
int Object::registerInterest(std::string event_type)

  // Check if room.
  if event_count is MAX_OBJ_EVENTS then
    return -1
  end if

  // Register with appropriate manager.
  if event_type is STEP_EVENT then
    GameManager registerInterest(this, event_type)
  else if
    ...
  else
    WorldManager registerInterest(this, event_type)
  end if

  // Keep track of added event.
  event_name[event_count] = event
```

```
21   increment event_count
22
23   // All is well.
24   return ok
```

The Object's `unregisterInterest()` method is provided in Listing 4.172. The first block of code checks to see if the event was previously registered – if it was not, an error (-1) is returned. Similar to `registerInterest()`, the next block of code checks first to see if the event is a step event and, if so, unregistering for interest with the GameManager. Other Managers are handled similarly, in the '`...`' region. By default, any remaining event is unregistered with the WorldManager. The last block of code starting at line 25 removes the event from the `event_name` array (at spot `index`), scooting over the following items. See line 31 in Listing 4.31 on page 77 for an example of doing "scooting" for an array of integers.

Listing 4.172: Object unregisterInterest()

```
0  // Unregister for interest in event.
1  // Return 0 if ok, else 1.
2  int Object::unregisterInterest(std::string event_type)
3
4    // Check if previously registered.
5    found = false
6    for index = 0 to event_count-1
7      if event_name[index] is event
8         found = true
9      end if
10   end for
11   if not found then
12     return -1
13   end if
14
15   // Unregister with appropriate manager.
16   if event is STEP_EVENT then
17     invoke GameManager unregisterInterest with "this" and "event"
18   else if
19   ...
20   else
21     invoke WorldManager unregisterInterest with "this" and "event"
22   end if
23
24   // Keep track.
25   scoot over all items in event_name
26   decrement event_count
27
28   // All is well.
29   return ok
```

When an Object is deleted, it is important to remove registration for all events it was interested in – failure to do so will result in the Object getting the event (by having its `eventHandler()` method called) even if it has been deleted. In fact, while game objects can certainly unregister for events they are no longer interested in, in many cases unregistration *only* happens when the Object is deleted. Thus, the destructor of the Object is extended to

automatically unregister for all events the Object had registered for interest in, as shown in Listing 4.173. Thus defined, game objects will typically register for interest in events, but not explicitly unregister since unregistration for all events happens when the life of the Object is over.

Listing 4.173: Unregistering from all registered events

```
0 for index = event_count-1 to 0
1   invoke unregisterInterest with event_name[index]
2 end for
```

In order to use the new methods, game objects that are interested in input, say, from the keyboard, would register for interest:

```
0   // Inside a game object's constructor.
1   registerInterest(KEYBOARD_EVENT)
```

Registering for interest in a mouse event is similar, except that `MOUSE_EVENT` is used instead of `KEYBOARD_EVENT`.

4.15.1 Program Flow for Events

This section provides a summary of the program flow in Dragonfly for events.

Consider an event that needs to be sent to all interested Objects – for example, a step event or even a user defined event such as "nuke".

1. The event is created, say `EventNuke e`.
2. WorldManager `onEvent()` is called, invoked with the address of the event (i.e., `&e`).
3. Since WorldManager `onEvent()` is only defined in the Manager base class, Manager `onEvent()` is invoked.
4. Manager `onEvent()` iterates through the `event_name[]` array until a match for the event type (via `p_event->getType()`) is found at index `i`.
5. Manager `onEvent()` then iterates through the `obj_list[]` ObjectList at index `i`.
6. For each Object (via `currentObject()`) in the list, its `eventHandler()` is invoked with the event (`*p_event`).
7. The derived Object `eventHandler()` (e.g., Saucer `eventHandler()`) inspects (via `p_e->getType()`) and handles the event, as appropriate.

4.15.2 Development Checkpoint #12!

Continue with your Dragonfly development. Specifically, develop functionality for filtering events from Section 4.15. Steps:

1. Refactor the Manager to support registration for interest in events. See Listing 4.163 for the methods needed. Add the necessary attributes, then write `registerInterest()` and `unregisterInterest()` based on Listing 4.164 and Listing 4.165, respectively. Test that objects can successfully register and unregister successfully, verifying working code with logfile output.

2. Refactor the Manager `onEvent()` method, referring to Listing 4.166 as needed. Test that Objects can register for step events and receive step events each game loop while the game is running.

3. Implement Manager `isValid()` to accept no events, but define `isValid()` for the GameManager, InputManager and WorldManager to handle step, keyboard and mouse and every other event, respectively. Refer to Listing 4.168 as needed. Verify that a game object cannot explicitly register for interest in events with inappropriate Manager (e.g, a step event with the InputManager), but can register for interest in events with an appropriate Manager (e.g., a keyboard and a mouse event with the InputManager). Create a user-defined event (e.g., a "nuke" event) and verify that this event can only be registered with the WorldManager.

4. Add attributes and methods supporting the Object's ability to register and unregister for events based on the Listing 4.170. Implement the `registerInterest()` and `unregisterInterest()` methods, referring to Listing 4.171 and Listing 4.172, respectively. Verify code functionality by having game code register and unregister for an event, both the step event and user-defined event, successfully.

5. Add code to the Object's destructor to unregister from all registered events, referring to Listing 4.173 as needed. Verify working code by having an object register for a step event, then destroying the object. Repeat with multiple events, such as a step event and a user-defined event.