Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #6

Input Manager

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 11.0). Information online at: `http://dragonfly.wpi.edu/book/`

4.9 Input Management

In order to get input from the player, a game could poll an input device directly. For example, for a platformer game on a PC, the code could check if the space bar was pressed and, if so, perform a "jump" action. The advantage of such a polling method is simplicity – the game code checks the input device right when it needs it and knows exactly what device to check. However, there are also significant disadvantages. First off, code to poll hardware devices is typically device dependent. If the device was swapped out, such as changing the keyboard for a joystick, the game would not work (at least, not without changing the game code and recompiling). Even with the same device, if the key was remapped, such as making the 'j' key execute a "jump" operation instead, then, again, the game code would need to be changed. Also, if there were supposed to be duplicate mappings for a single event, such as the left mouse button also being a "jump", then the code to do the polling (and maybe the jump action) would need to be duplicated.

The primary role of the game engine is to avoid such drawbacks by generalizing input from a variety of hardware-specific devices and code into general game code. The input flow generally goes as follows:

1. The player provides input via a specific device (e.g., a button press).
2. The game engine detects that input has occurred. The engine determines whether to process the input or ignore it (e.g., player input may be ignored during a cut-scene).
3. If input is to be processed, the data is decoded from the device. This may mean dealing with device-specific details (e.g., the degrees of rotation on an analog joystick).
4. The device-specific input is encoded into a more abstract, device-independent form suitable for the game.

After the above steps, all game objects that are interested in the input are notified – in Dragonfly, this means passing an input event to an Object using Manager `onEvent()`. The information in the input event depends upon the type. For example, a keyboard input needs the value of the key pressed while a mouse input needs the button type (left, right or middle) and the mouse (x,y) location.

4.9.1 Simple and Fast Multimedia Library – Input

While there are several options for getting user input, for Dragonfly, since the Simple and Fast Multimedia Library (SFML) is already used for graphical output (see Section 4.8.1 on page 111), it is also used for input. Specifically, SFML supports the ability to get keyboard input without waiting/blocking. In traditional keyboard input (e.g., `cin` or `scanf()`), a program waiting for input is blocked/suspended until the user presses the "enter" key. With SFML, the program can either be notified of a keypress event and/or the program can check if a particular key is being held down. SFML also supports mouse actions, tracking the current position of the mouse cursor and notifying when mouse buttons are pressed and released.

In order to use SFML input suitable for games, there are only two initial actions that need to be taken. SFML input events are provided for an SFML window, so such a window needs to be created (in Dragonfly, the window needed is created by the DisplayManager upon startup (see Listing 4.69 on page 111)). Also, by default, when a user holds down a key, after a small delay, the built-in "repeat" functionality of the keyboard will generate a log of keypress events. Since many games allow the user to hold down a key as an action (e.g., hold down arrow keys to move an avatar), it is useful to disable the repeat functionality, and can be done in SFML as shown in Listing 4.89.

Listing 4.89: SFML disable keyboard repeat

```
0 // Disable keyboard repeat.
1 p_window -> setKeyRepeatEnabled(false)
```

When disabled, a program only gets a single event when the key is pressed. To re-enable key repeat, `true` is passed into to `setKeyRepeatEnabled()`, enabling repeated KeyPressed events while keeping a key pressed.

Once initialized, SFML for game-type input proceeds first by using window input events, as shown in Listing 4.90. SFML provides event attributes through the `sf::Event` object, which is populated with the `pollEvent()` call. Each call to `pollEvent()` provides exactly one event, so to check all such events, it is placed inside a `while()` loop, as on line 1. The type of each event is checked through the `type` field. While SFML provides many window events, only some of them are useful for input – specifically, `sf::Event::KeyPressed`, `sf::Event::KeyReleased`, `sf::Event::MouseMoved`, and `sf::Event::MouseClicked`. When there is a `MouseClicked` event, the button can be checked for which mouse button is clicked, shown for the right mouse button on line 45. For the keyboard, the key that is pressed/released is in the SFML event `code`, and for the mouse, the button that pressed/released is in the SFML event `button`.

Listing 4.90: SFML input for games

```
0  // Loop, handling all events in window.
1  while (const std::optional<sf::Event> p_event = p_window -> pollEvent()) do
2
3    // Get event.
4    sf::Event e = p_event.value()
5
6    // Window closed?
7    if p_event -> is<sf::Event::Closed>() then
8      // Do close stuff.  e.g., set game over
9    end if
10
11   // Key was pressed?
12   if p_event -> is<sf::Event::KeyPressed>() then
13
14     // Setup as KeyPressed event.
15     sf::Event::KeyPressed *p_kb_event =
16         reinterpret_cast<sf::Event::KeyPressed *> (&e)
17
18     // Get SFML keyboard code.
19     sf::Keyboard::Key key
20     key = p_kb_event -> code
```

```
21
22      // Do other keypress stuff.
23    end if
24
25    // Key was released?
26    (similar to key pressed)
27    ...
28
29    // Mouse moved?
30    if p_event -> is<sf::Event::MouseMoved>() then
31
32      // Setup as MouseMoved event.
33      sf::Event::MouseMoved *p_mse_event =
34          reinterpret_cast<sf::Event::MouseMoved *> (&e)
35
36      // Get pixel location.
37      sf::Vector2i pixel_pos = p_mse_event -> position
38
39      // Do other mouse moved stuff.
40
41    fi
42
43    // Mouse clicked?
44    (similar to mouse moved, but also check buttons ...)
45      if p_mse_event -> button == sf::Mouse::Button::Right then
46        // Do mouse button stuff
47    ...
48
49  end while
```

4.9.1.1 SFML – Polled Input (optional)

While the code in Listing 4.90 provides all window based events, trying to move an avatar by pressing and holding a key using `sf::Event::KeyPressed` will not work since only one such event is provided – when the key is first pressed. Instead, SFML provides methods to directly check if a key or mouse button is currently pressed by polling it. This is illustrated in Listing 4.91. Note, this code is *outside* of the `while()` loop in Listing 4.90. Here, a specific key can be polled (e.g., `sf::Keyboard::Key::Left`) to see if it is currently being held down. Similarly, a specific mouse button can be polled (e.g., `sf::Mouse::Button:Left`) to see if it is currently being held down

Listing 4.91: SFML input – polling key/mouse pressed

```
0  // Key is pressed.
1  if sf::Keyboard::isKeyPressed(keycode) then
2    // Do key is pressed stuff.
3  end if
4
5  // Mouse button is pressed.
6  if sf::Mouse::isButtonPressed(button) then
7    // Do mouse is pressed stuff.
8  end if
```

4.9.2 The InputManager

The InputManager is a singleton derived from the Manager class, with private constructors and a `getInstance()` method to return the one and only instance (see Section 4.2.1 on page 55). The header file, including class definition, is provided in Listing 4.92.

The InputManager constructor should set the type of the Manager to "InputManager" (i.e., `setType("InputManager")` and initialize all attributes.

The `startUp()` method gets the display ready for input using SFML, as per Section 4.9.1. Similarly, the `shutDown()` method reverts to normal use. Finally, the method `getInput()` uses SFML to obtain keyboard and mouse input and is called by the GameManager once per game loop.

Listing 4.92: InputManager.h

```
#include "Manager.h"

class InputManager : public Manager {

 private:
  InputManager();                          // Private (a singleton).
  InputManager (InputManager const&);      // Don't allow copy.
  void operator=(InputManager const&);     // Don't allow assignment

 public:
  // Get the one and only instance of the InputManager.
  static InputManager &getInstance();

  // Get window ready to capture input.
  // Return 0 if ok, else return -1.
  int startUp();

  // Revert back to normal window mode.
  void shutDown();

  // Get input from the keyboard and mouse.
  // Pass event along to all Objects.
  void getInput() const;
};
```

For starting up, an SFML window needs to be created first. However, the InputManager does not do this – rather, the InputManager assumes this was done already by the DisplayManager. Thus, there is now a starting order dependency for the Dragonfly managers in that the DisplayManager must be started before the InputManager. The InputManager checks that the DisplayManager has successfully been started via a check to `isStarted()` – if it has not, then the InputManager does not start up successfully, either.

In general, the startup order for the Managers defined thus far should be:

1. LogManager
2. DisplayManager
3. InputManager

Remember, as described in Section 4.4.4, the game programmer instantiates (via `getInstance()`) and starts up (via `startUp()`) the GameManager, and the GameManager in its `startUp()` instantiates and starts up the other managers in the proper order. Only if they all start up successfully should the game manager report a successful startup.

In more detail, the InputManager `startUp()` method does the steps shown in Listing 4.93. First, the DisplayManager is checked to see if it has been started. If so, the SFML window (of type `sf::RenderWindow` is obtained from it. The window is used to disable key repeat. If everything succeeds, `Manager::startUp()` is called to indicate successful startup.

Listing 4.93: InputManager startUp()

```
// Get window ready to capture input.
// Return 0 if ok, else return -1.
int InputManager::startUp()

  if DisplayManager is not started then
    return error
  end if

  sf::RenderWindow window = DisplayManager getWindow()

  disable key repeat in window

  call Manager::startUp()
```

The InputManager `shutDown()` method re-enables key repeat and invokes Manager `shutDown()` to indicate the InputManager is no longer started.

Steps in the InputManager's `getInput()` method are provided in Listing 4.94 and are similar to those in Listing 4.90 (on page 129).

In the while loop, SFML window events are checked. If there are respective keyboard and/or mouse actions, a corresponding Dragonfly keyboard or mouse event is generated. To "send" the event to Objects, the `onEvent()` method is used. See Listing 4.67 on page 110 for a refresher on what it does. The keyboard and mouse events themselves are described in upcoming Section 4.9.2.1 and Section 4.9.2.2, respectively.

Listing 4.94: InputManager getInput()

```
// Get input from the keyboard and mouse.
// Pass event along to all Objects.
void InputManager::getInput() const

  // Check past window events.
  while event do

    if key press then

      create EventKeyboard (key and action)
      send EventKeyboard to all Objects

    else if key release then

      create EventKeyboard (key and action)
```

```
15        send EventKeyboard to all Objects
16
17      else if mouse moved then
18
19        create EventMouse (x, y and action)
20        send EventMouse to all Objects
21
22      else if mouse clicked then
23
24        create EventMouse (x, y and action)
25        send EventMouse to all Objects
26
27      end if
28
29    end while // Window events.
```

To use the InputManager, the GameManager adds a call to `getInput()` in the game loop (inside GameManager `run()`):

```
0   ...
1   // (Inside GameManager run())
2   // Get input.
3   InputManager getInput()
4   ...
```

4.9.2.1 Keyboard Event

Listing 4.95 provides the header file for the EventKeyboard class.

The top part of the header file defines two `enum` types.

The first, `enum EventKeyboardAction`, specifies the types of keyboard actions Dragonfly recognizes, namely: `KEY_PRESSED`, and `KEY_RELEASED`. The `UNDEFINED_KEYBOARD_ACTION` action is used for the default.

The second, `Keyboard::Key`, specifies the keys Dragonfly recognizes. It is placed inside its own namespace, `Keyboard`, for clarity. All major keys are recognized, with `UNDEFINED_KEY` used for the default. Note, the key types here are all Dragonfly attributes and not SFML (i.e., not `sf::Keyboard::Key`) in order to encapsulate the SFML code inside the engine. This way, game code that examines what keys are pressed is not dependent (nor even aware) of the underlying SFML. This would allow, say, a change in the input layer, say by replacing SFML with something else, without changing the game code.[15]

For the class body, as for all Dragonfly events, EventKeyboard is derived from the Event base class. It stores the keystroke in `key_val` and the keyboard action in `keyboard_action`. Each attribute has a pair of methods to get and set it. For example, the method `setKey()` takes on the value of the key based on what is pressed (typically only done by the InputManager), and the method `getKey()` is used by game code for retrieving the key value. The constructor sets `event_type` to `KEYBOARD_EVENT`, defined in the top of the header file.

[15] In fact, an earlier version of Dragonfly used Curses, a set of library functions that enable controlling text output in terminal windows.

Listing 4.95: EventKeyboard.h

```
#include "Event.h"

const std::string KEYBOARD_EVENT = "df::keyboard";

// Types of keyboard actions Dragonfly recognizes.
enum EventKeyboardAction {
  UNDEFINED_KEYBOARD_ACTION = -1, // Undefined.
  KEY_PRESSED,                    // Was down.
  KEY_RELEASED,                   // Was released.
};

// Keys Dragonfly recognizes.
namespace Keyboard {
enum Key {
  UNDEFINED_KEY = -1,
  SPACE, RETURN, ESCAPE, TAB, LEFTARROW, RIGHTARROW, UPARROW, DOWNARROW,
  PAUSE, MINUS, PLUS, TILDE, PERIOD, COMMA, SLASH, LEFTCONTROL,
  RIGHTCONTROL, LEFTSHIFT, RIGHTSHIFT, F1, F2, F3, F4, F5, F6, F7, F8,
  F9, F10, F11, F12, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q,
  R, S, T, U, V, W, X, Y, Z, NUM1, NUM2, NUM3, NUM4, NUM5, NUM6, NUM7,
  NUM8, NUM9, NUM0,
};
} // end of namespace Keyboard

class EventKeyboard : public Event {

 private:
  Keyboard::Key m_key_val;                // Key value.
  EventKeyboardAction m_keyboard_action;  // Key action.

 public:
  EventKeyboard();

  // Set key in event.
  void setKey(Keyboard::Key new_key);

  // Get key from event.
  Keyboard::Key getKey() const;

  // Set keyboard event action.
  void setKeyboardAction(EventKeyboardAction new_action);

  // Get keyboard event action.
  EventKeyboardAction getKeyboardAction() const;
};
```

4.9.2.2 Mouse Event

Listing 4.96 provides the header file for the EventMouse class, also derived from the Event base class. Dragonfly only recognizes a fixed set of mouse actions and buttons, defined by `MouseAction` defined by `MouseButton`, respectively. Mouse actions recognized are `CLICKED` and `MOVED`, with `UNDEFINED_MOUSE_ACTION` being the default. Mouse buttons recognized are

LEFT, RIGHT, MIDDLE, with UNDEFINED_MOUSE_BUTTON being the default. As for EventKeyboard, the mouse buttons are Dragonfly attributes to encapsulate the SFML code inside the engine.

The mouse action is stored in the attribute mouse_action, the mouse button in mouse_button, and the (x,y) location in mouse_xy.

Methods are provided to get and set each attribute. The "set" methods are typically only used by the InputManager, while the "get" methods are used in the game code to retrieve values and act appropriately for the game. The EventMouse constructor sets event_type to MSE_EVENT.[16]

Listing 4.96: EventMouse.h

```
#include "Event.h"

const std::string MSE_EVENT = "df::mouse";

// Set of mouse actions recognized by Dragonfly.
enum EventMouseAction {
  UNDEFINED_MOUSE_ACTION = -1,
  CLICKED,
  MOVED,
};

// Set of mouse buttons recognized by Dragonfly.
namespace Mouse {
enum Button {
  UNDEFINED_MOUSE_BUTTON = -1,
  LEFT,
  RIGHT,
  MIDDLE,
};
} // end of namespace Mouse

class EventMouse : public Event {

 private:
  EventMouseAction m_mouse_action;  // Mouse action.
  Mouse::Button m_mouse_button;     // Mouse button.
  Vector m_mouse_xy;                // Mouse (x,y) coordinates.

 public:
  EventMouse();

  // Load mouse event's action.
  void setMouseAction(EventMouseAction new_mouse_action);

  // Get mouse event's action.
  EventMouseAction getMouseAction() const;

  // Set mouse event's button.
  void setMouseButton(Mouse::Button new_mouse_button);
```

[16] MSE_EVENT is used instead of MOUSE_EVENT since the latter can conflict with a macro if developing in Windows.

```
39
40    // Get mouse event's button.
41    Mouse::Button getMouseButton() const;
42
43    // Set mouse event's position.
44    void setMousePosition(Vector new_mouse_xy);
45
46    // Get mouse event's position.
47    Vector getMousePosition() const;
48 };
```

At this point, a view of complete version of the game loop is warranted, shown in Listing 4.97. Unlike in early versions of the game loop shown, code can be constructed for each game loop element based, as indicated in the comments.

Listing 4.97: The complete game loop

```
0  Clock clock                          // Section 4.4.3 on page 69.
1  while (game not over) do             // Line 11 of Listing 4.25 on page 73.
2    clock.delta()                      // Line 14 of Listing 4.20 on page 69.
3    GameManager onEvent(EventStep)     // Listing 4.67 on page 110.
4    InputManager getInput()            // Listing 4.94 on page 132.
5    WorldManager update()              // Listing 4.64 on page 104.
6    WorldManager draw()                // Listing 4.81 on page 121.
7    DisplayManager swapBuffers()       // Listing 4.79 on page 120.
8    loop_time = clock.split()          // Line 19 of Listing 4.20 on page 69.
9    sleep(TARGET_TIME - loop_time)     // Listings 4.21 and 4.22 on page 70+.
10 end while
```

4.9.3 Development Checkpoint #6!

Continue with Dragonfly development by adding functionality for managing input from Section 4.9. Steps:

1. Create an InputManager derived class, inheriting from the Manager class. Implement InputManager as a singleton, described in Section 4.2.1 on page 55. Add `InputManager.cpp` to the project and include stubs for all methods in Listing 4.92. Make sure the class, with stubs, compiles.

2. Write `startUp()` and `shutDown()` methods for the InputManager, referring to Listing 4.93 as needed. Write a small test program (with only an InputManager and DisplayManager) that verifies the InputManager can only start successfully when the DisplayManager is started first.

3. Create EventKeyboard and EventMouse classes, referring to Sections 4.9.2.1 and 4.9.2.2, as needed. Add `EventKeyboard.cpp` and `EventMouse.cpp` to the project and stub out each method so it compiles. Verify both classes can get and set all values in a stand alone program (running outside of the other engine components).

4. Implement InputManager `getInput()`, referring to Listing 4.94 for the structure and Listing 4.90 for the SFML code, as appropriate. First, get `getInput()` implemented

and tested with one key (e.g., the letter '`A`') and then one mouse button. Once that is working properly, continue implementation for all keys and all mouse buttons.

5. Test the InputManager `getInput()` outside of a running game loop creating a program that starts the DisplayManager and the InputManager, then repeatedly calls `getInput()`, writing the return values to the logfile. This can be tested extensively with different mouse and keyboard inputs.

6. Integrate the InputManager with the GameManager by having the GameManager start up the InputManager in the proper order. Write a game object that takes input from the keyboard, responds to input by changing position and have that change in position be visible on the screen.

Tip 17! Reticle for testing mouse input. The Reticle from the Saucer Shoot tutorial (Chapter 3) is a good game object to use as a start for testing mouse input. You can copy the `Reticle.cpp` and `Reticle.h` code from that project and test your newly-implemented Input Manager once integrated with the engine. (Note, you'll need to remove the `registerInterest()` call from the Reticle constructor unless and until you implement filtering of events, Section 4.15 on page 204.) The "game" can just be a `new Reticle` and the player should be see a `+` character moving around the screen with the mouse. Add handling of mouse button presses in the Reticle `eventHandler()`, writing a message to the LogManager or changing the Reticle color.

At this point, the engine should now be able to get input from a player and have game objects respond to the input! This means, coupled with the DisplayManager, the engine supports game objects a player can move (e.g., via the arrow keys or the mouse) around the screen. This closes the interaction loop, taking user input, updating the game world, and displaying the resulting output – basic interaction!