Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #13

View Objects

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/`

4.16 View Objects

Thus far, Dragonfly has been discussed in terms of game objects – objects that interact with each other in the game world. Examples from the Saucer Shoot tutorial (Chapter 3) include Saucers, Bullets and the Hero. Game objects are the basic building blocks for games and so are the primary types of objects that a game engine must support.

However, most games include other types of objects that do not interact with other objects in the game world. Such objects may display information or allow a player to control game settings. For example, an object that displays a player's score does not collide with the hero, spaceships, rocks or any other typical game objects. Buttons and other menu objects that let players choose settings, weapons or other game options do not interact with game objects in the world.

In Dragonfly, supporting such view-only objects is done through a new engine object type, a ViewObject. ViewObjects inherit from the base Object class. This allows the rest of the engine code, such as the WorldManager and all the utilities such as lists and iterators, handle ViewObjects as they would standard Objects without change. What ViewObjects do have that is different are additional attributes that make it more convenient for game programmers to create "heads-up display" (or HUD) types of interfaces.

While game objects are positioned in game world coordinates, ViewObjects are positioned relative to the screen coordinates. For example, the game programmer may want to display the points in the upper right corner of the screen, or the health in the bottom left corner of the screen. To support the abstraction of screen placement rather than game world position, ViewObjects use an `enum` named `ViewObjectLocation` (as defined on line 8 of Listing 4.174) with positions of top or bottom and left, center and right.

Beyond what is available for Objects, ViewObjects have additional attributes shown starting on line 24 of Listing 4.174. These include a string (`view_string`) that provides a text label for the ViewObject (e.g., "points"), an integer (`value`) to hold the ViewObject value (e.g., the player's points, say 150), a boolean (`draw_value`) that indicates if said value should be drawn or not, a boolean (`border`) that indicates if the ViewObject should be drawn with a decorative border, and an integer (`color`) that provides an optional color for the ViewObject (if different than the default color). Methods to get and set `view_string`, `value`, `border`, `draw_value`, and `color` are provided.

The ViewObject has a custom `eventHandler()` (line 42) since ViewObjects respond to special view events provided by the game programmer to, say, update the player's points or other game-specific value.

Listing 4.174: ViewObject.h

```
0 // System includes.
1 #include <string>
2
3 // Engine includes.
4 #include "Object.h"
5 #include "Event.h"
6
7 // General location of ViewObject on screen.
8 enum ViewObjectLocation {
9   UNDEFINED=-1,
```

```
  TOP_LEFT,
  TOP_CENTER,
  TOP_RIGHT,
  CENTER_LEFT,
  CENTER_CENTER,
  CENTER_RIGHT,
  BOTTOM_LEFT,
  BOTTOM_CENTER,
  BOTTOM_RIGHT,
};

class ViewObject : public Object {

 private:
  std::string view_string;          // Label for value (e.g., "Points").
  int m_value;                      // Value displayed (e.g., points).
  bool m_draw_value;                // True if should draw value.
  bool m_border;                    // True if border around display.
  Color m_color;                    // Color for text (and border).
  ViewObjectLocation m_location;    // Location of ViewObject.

 public:
  // Construct ViewObject.
  // Object settings: SPECTRAL, max alt.
  // ViewObject defaults: border, top_center, default color, draw_value.
  ViewObject();

  // Draw view string and value.
  virtual int draw();

  // Handle 'view' event if tag matches view_string (others ignored).
  // Return 0 if ignored, else 1 if handled.
  virtual int eventHandler(const Event *p_e);

  // General location of ViewObject on screen.
  void setLocation(ViewObjectLocation new_location);

  // Get general location of ViewObject on screen.
  ViewObjectLocation getLocation() const;

  // Set view value.
  void setValue(int new_value);

  // Get view value.
  int getValue() const;

  // Set view border (true = display border).
  void setBorder(bool new_border);

  // Get view border (true = display border).
  bool getBorder() const;

  // Set view color.
  void setColor(Color new_color);

```

```
  // Get view color.
  Color getColor() const;

  // Set view display string.
  void setViewString(std::string new_view_string);

  // Get view display string.
  std::string getViewString() const;

  // Set true to draw value with display string.
  void setDrawValue(bool new_draw_value = true);

  // Get draw value (true if draw value with display string).
  bool getDrawValue() const;

};
```

Listing 4.175 shows the ViewObject constructor. First, it makes Object settings appropriate for a ViewObject. Specifically, it puts the Object at the highest altitude so it is visible above any other game objects, makes the ViewObject spectral so it does not collide with any other objects and sets its type to "ViewObject". Second, the ViewObject-specific settings are made, with a value of 0, a border being drawn, the location in the top center of the screen and the default color. Lastly, the ViewObject registers for interest in a view event, described in Section 4.16.1 on page 202.

Listing 4.175: ViewObject ViewObject()

```
// Construct ViewObject.
// Object settings: SPECTRAL, max altitude.
// ViewObject defaults: border, top_center, default color, draw_value.
ViewObject::ViewObject()

  // Initialize Object attributes.
  setSolidness(SPECTRAL)
  setAltitude(MAX_ALTITUDE)
  setType("ViewObject")

  // Initialize ViewObject attributes.
  setValue(0)
  setDrawValue()
  setBorder(true)
  setLocation(TOP_CENTER)
  setColor(COLOR_DEFAULT)

  // Register interest in view events.
  registerInterest(VIEW_EVENT) // if Section 4.15 implemented.
```

Pseudo code for ViewObject `setLocation()` method is shown in Listing 4.176. Basically, the switch statement starting on line 4 determines the (x,y) location. Only the first 2 (of 9 total) entries of the statement are shown, with the missing pieces following the same pattern. The y coordinate is at 1 if on the top of the window, or `world_manager.getView().getVertical()`-1 if on the bottom.

The x coordinate is at 1/6th, 3/6th, and 5/6th the horizontal distance (`world_manager.getView().getHorizontal()`), depending on if it is left, right or center, respectively. The y_delta variable is used to adjust the vertical distance by -1 if the ViewObject is at the top and does not have a border, and by +1 if the ViewObject is at the bottom and does not have a border. On line 21, the position is actually shifted and on line 24 the position of the ViewObject is moved to the new position. Note, as given, Listing 4.176 assumes new_location is one of the nine valid locations whereas in actual code this should be checked and no action should be taken if new_location is invalid.

Listing 4.176: ViewObject setLocation()

```
0  // General location of ViewObject on screen.
1  ViewObject::setLocation(ViewObjectLocation new_location)
2
3    // Set new position based on location.
4    switch (new_location)
5    case TOP_LEFT:
6      p.setXY(WorldManager getView().getHorizontal() * 1/6, 1)
7      if getBorder() is false then
8        y_delta = -1
9      end if
10     break;
11   case TOP_CENTER:
12     p.setXY(WorldManager getView().getHorizontal() * 3/6, 1)
13     if getBorder() is false then
14       y_delta = -1
15     end if
16     break;
17   ...
18   end switch
19
20   // Shift, as needed, based on border.
21   p.setY(p.getY() + y_delta)
22
23   // Set position of object to new position.
24   setPosition(p)
25
26   // Set new location.
27   location = new_location
```

The corresponding ViewObject getLocation() is not shown, but should merely return location.

ViewObject setBorder() does a bit more than just set border to the new value. As shown in Listing 4.177, it also calls setLocation() since, if the border has changed, the (x,y) location on the screen needs to be adjusted based on the new border value.

Listing 4.177: ViewObject setBorder()

```
// Set view border (true = display border).
void ViewObject::setBorder(bool new_border)

  if border != new_border then

    border = new_border

    // Reset location to account for border setting.
    setLocation(getLocation())
```

```
  end if
```

The ViewObject `draw()` method is shown in Listing 4.178. The first code block constructs the string to draw, created from the display string and the integer holding the value. The second block of code actually draws the string, invoking the `drawString()` (see Listing 4.79 on page 116) method from the DisplayManager, along with a border (if appropriate). Note, since the ViewObject's (x,y) location is in *screen* coordinates, as opposed to game world coordinates like most Objects, the ViewObject position needs to be translated to world coordinates via the utility function `viewToWorld()`. The function `viewToWorld()` does the reverse translation as `worldToView()`, in Listing 4.143 on page 173.

Listing 4.178: ViewObject draw()

```
0  // Draw view string and value.
1  int ViewObject::draw()
2
3    // Display view_string + value.
4    if border is true then
5      temp_str = " " + getViewString() + " " + toString(value) + " "
6    else
7      temp_str = getViewString() + " " + toString(value)
8    end if
9
10   // Draw centered at position.
11   Vector pos = viewToWorld(getPosition())
12   DisplayManager drawString(pos, temp_str, CENTER_JUSTIFIED,
13                             getColor())
14   if border is true then
15     // Draw box around display.
16     ...
17   end if
```

The `toString()` function used in Listing 4.178 on line 5 and line 7 is a useful utility function to put in `utility.cpp`. Basically, it creates a `stringstream`, adds a number to it, and return a string with the new contents. The full function is shown in Listing 4.179.

Listing 4.179: Utility toString()

```
#include <sstream>
using std::stringstream;

// Convert int to a string, returning string.
std::string toString(int i) {
   std::stringstream ss;  // Create stringstream.
   ss << i;               // Add number to stream.
   return ss.str();       // Return string with contents of stream.
}
```

While thus far, view objects could be done entirely outside the engine in "game programmer" code space, there is one part of the engine that is aware of ViewObjects – the WorldManager's `draw()` method. The extension required of the WorldManager to support views is shown in Listing 4.180. Without views, the `draw()` method checked each Object to see if they intersect the visible screen (see Listing 4.145 on page 174). ViewObjects may fail this check since their positions are relative to the screen, not the game world. So, instead, after checking for intersection, a `dynamic_cast` is made to see if the Object is a ViewObject. If so, it is drawn.

Listing 4.180: WorldManager extensions to draw() to support ViewObjects

```
...
// Only draw if Object would be visible (intersects view).
if boxIntersectsBox(box, view) or           // Object in view,
  dynamic_cast <ViewObject *> (p_temp_o))    // or is ViewObject.
  p_temp_o -> draw()
end if
...
```

4.16.1 View Event

View events are used by game programmers to signal the change in a view value. For example, if the player scored 10 points, say by destroying a Saucer, a view event would be created, given a value of 10, and passed to all ViewObjects (using `onEvent()`). Listing 4.181 provides the header file for the EventView class, derived from the Event class (Listing 4.48 on page 92). Remember, in the constructor of a ViewObject, the Object already registered for interest in a `VIEW_EVENT` (see Listing 4.175 on page 199). `VIEW_EVENT` is defined in Listing 4.181 on line 2.

Like many other Events, the EventView is mostly a container, holding a string (`tag`) which is a label associated with a specific ViewObject, an integer (`value`) that is used to modify the value in the ViewObject, and a boolean (`delta`) that determines whether the value either adjusts the ViewObject value (if `delta` is `true`) or replaces it (if `delta` is `false`). Methods are provided to get and set these values. The default constructor assigns `VIEW_EVENT`, 0 and `false` to `tag`, `value` and `delta`, respectively, and an alternate constructor is provided to create an EventView with attribute values specified.

Listing 4.181: EventView.h

```
0  #include "Event.h"
1
2  const std::string VIEW_EVENT = "df::view";
3
4  class EventView : public Event {
5
6   private:
7    std::string m_tag;  // Tag to associate.
8    int m_value;        // Value for view.
9    bool m_delta;       // True if change in value, else replace value.
10
11  public:
12   // Create view event with tag VIEW_EVENT, value 0 and delta false.
13   EventView();
14
15   // Create view event with tag, value and delta as indicated.
16   EventView(std::string new_tag, int new_value, bool new_delta);
17
18   // Set tag to new tag.
```

```
  void setTag(std::string new_tag);

  // Get tag.
  std::string getTag() const;

  // Set value to new value.
  void setValue(int new_value);

  // Get value.
  int getValue() const;

  // Set delta to new delta.
  void setDelta(bool new_delta);

  // Get delta.
  bool getDelta() const;
};
```

With EventView specified, the ViewObject `eventHandler()` can now be defined as shown in Listing 4.182. The first `if-then` confirms that the event is a `VIEW_EVENT`. If so, line 7 casts the generic Event pointer as an EventView pointer.

An EventView is then be checked to see if its tag matches the view string associated with this ViewObject – if so, this event was intended for this object. At that point, the two options are for `delta` to indicate that the EventView value is to change the ViewObject's value by that amount (if `true`), or that the EventView value is to replace the ViewObject's value (if `false`). Either way, the event his handled and 1 is returned at line 20. If line 27 is reached, the event was not handled so 0 is returned.[21].

Listing 4.182: ViewObject eventHandler()

```
0 // Handle ‘view’ events if tag matches view_string (others ignored).
1 // Return 0 if ignored, else 1 if handled.
2 int ViewObject::eventHandler(const Event *p_e)
3
4   // See if this is ‘view’ event.
5   if p_e->getType() is VIEW_EVENT then
6
7     EventView *p_ve = static_cast <const EventView *> p_e
8
9     // See if this event is meant for this object.
10    if p_ve -> getTag() is getViewString() then
11
12      if p_ve -> getDelta() then
13        setValue(getValue() + p_ve->getValue())  // Change in value.
14      else
15        setValue(p_ve->getValue())               // New value.
16
17      end if
18
19      // Event was handled.
20      return 1
```

[21]If the parent Object `eventHandler()` did any work, it should be called but in the case of the engine at this point, it does not

```

    end if

  end if

  // If here, event was not handled. Call parent eventHandler().
  return 0
```

An example helps illustrate the use of ViewObjects and EventViews. Say a game programmer wants to have points associated with player achievements in a game and have the points displayed in the top right of the screen. The game programmer might use the code in Listing 4.183 at the top to create the view object, before the game actually starts. This code creates a ViewObject, associates "points" with the object, initializes the value to 0, positions it at the top right of the screen and makes it yellow. The ViewObject code automatically registers the object for interest in view events.

To change the value of the points ViewObject, say when an enemy object is destroyed, the game programmer places the second block of code (starting on line 8) into the enemy object destructor. When the enemy object is destroyed and the destructor is called, an EventView is created, intended for the points ViewObject, providing a value of 10 that will be added to the ViewObject value, since `delta`, the last parameter, is `true`. The event is given to the ViewObject (actually all ViewObjects, but only the "points" ViewObject will react) via the `onEvent()` call in the WorldManager.

Listing 4.183: Using ViewObjects

```
0  // Before starting game...
1  df::ViewObject *p_vo = new df::ViewObject; // Used for points.
2  p_vo -> setViewString("Points");
3  p_vo -> setValue(0);
4  p_vo -> setLocation(df::TOP_RIGHT);
5  p_vo -> setColor(df::COLOR_YELLOW);
6  ...
7
8  // In destructor of enemy object...
9  df::EventView ev("Points", 10, true);
10 df::WorldManager onEvent(&ev);
```

4.16.2 Buttons (optional)

A common user interface option is the button, represented graphically on the screen and selected with a mouse. Computer users and game players are familiar with buttons, using them for all sorts of game-related input. Buttons can provide in-game input, for example for casting a spell, or before the game starts, for example for choosing what character to be.

For Dragonfly, the button is similar to a ViewObject in that it is drawn on top of the rest of the game objects and does not interact with the game world. The button needs to respond to the mouse, too, so that it can recognize when the mouse is over it and when it has been clicked.

Listing 4.184 shows the Button class, derived from the ViewObject class. The Button adds two attributes for colors – one for the Button color when the button is highlighted (the mouse is over it) (highlight_color, and one to keep track of the default color when the button is not highlighted (default_color). Methods to get and set these attributes are provided. The constructor needs to set default attribute values and register for interest in mouse events.

Listing 4.184: Button.h

```
class Button : public ViewObject {

 private:
  Color m_highlight_color;    // Color when highlighted.
  Color m_default_color;      // Color when not highlighted.

 public:
  Button();

  // Handle "mouse" events.
  // Return 0 if ignored, else 1.
  int eventHandler(const Event *p_e);

  // Set highlight (when mouse over) color for Button.
  void setHighlightColor(Color new_highlight_color);

  // Get highlight (when mouse over) color for Button.
  Color getHighlightColor() const;

  // Set color of Button.
  void setDefaultColor(Color new_default_color);

  // Get color of Button
  Color getDefaultColor() const;

  // Return true if mouse over Button, else false.
  bool mouseOverButton(const EventMouse *p_e) const;

  // Called when Button clicked.
  // Must be defined by derived class.
  virtual void callback() = 0;
};
```

The mouseOverButton() method is a helper to facilitate the Button in changing between the highlight (when the mouse moves over it) and default colors (when the mouse is not over it). Its functionality is depicted in Listing 4.185. A pointer to EventMouse event is a parameter, with the return type boolean as true if the mouse is inside the button, otherwise false.

The first block of code creates a bounding box for the Button which is wide enough for the string and adjusted for with width and height if the Button has borders (an attribute of the parent ViewObject). The next block of code simply calls boxContainsPosition() (see Listing 4.140 on page 170) using the newly constructed Box and the mouse's position, and returns the appropriate boolean.

Listing 4.185: Button mouseOverButton()

```
// Return true if mouse over Button, else false.
```

```
bool MouseOverButton::mouseOverButton(const EventMouse *p_e) const

  // Create Box for Button.
  width = getViewString().size()
  height = 1
  if getBorder() then     // if Button has border
    width = width + 4     // box wider by 2 spaces and |
    height = height + 2   // box taller by 2 rows of ---
  end if
  Vector corner(getPosition().getX() - width/2,
                getPosition().getY() - height/2)
  Box b(corner, width, height)

  // If mouse inside button box, return true, else false.
  if boxContainsPosition(b, p_e -> getMousePosition())
    return true
  else
    return false
```

With that method in place, the `eventHandler()` method, shown in Listing 4.186, is ready to handle mouse actions. Since the Button only handles mouse events, this is checked at the start, and any non-mouse event is not handled (`return 0`).

Next, the mouse event is checked to see if the mouse is inside the Button using `mouseOverButton()`. If it is not, then the Button color is changed to the default and the method returns (having still handled the event).

If the mouse is inside the Button, the Button color is changed to the highlight color and if the mouse action is `CLICKED`, then the Button `callback()` is invoked.

Remember, although not shown, the Event pointer `p_e` needs to be casted when used as an EventMouse (see Section 4.5.5.3 on page 94).

Listing 4.186: Button eventHandler()

```
// Handle "mouse" events.
// Return 0 if ignored, else 1.
int Button::eventHandler(const Event *p_e)

  // Check if mouse event.
  if p_e -> getType() is not MSE_EVENT then
    return 0  // not handled
  end if

  // Check if mouse over button.
  if mouseOverButton(p_e) then

    // Highlight on.
    setColor(highlight_color)

    // Check if clicked.
    if p_e -> getMouseAction() is CLICKED then

      // Invoke callback.
      callback()

    end if
```

```

  // Highlight off.
  setColor(default_color)

  // Event handled.
  return 1
```

Lastly, note that the `callback()` method on line 30 of Listing 4.184 is declared as pure virtual (`=0`) meaning `callback()` *must* be defined before Button can be used. This is because there is really no generic behavior common for all buttons when clicked, but instead the game programmer must implement the button-specific behavior wanted.

An example can help illustrate how the Button class can be used. Consider a typical start screen in a game, such as the start screen for Saucer Shoot in Section 3.3.11 on page 39, where the player can choose to either "play" or "quit". A quit button can be made as in Listings 4.188 (header file) and 4.187 (code). In the header file, QuitButton is derived from Button. The only method that must be defined is `callback()`, but in this case there is a default constructor since some Button defaults are changed (such as the button text).

Listing 4.187: QuitButton.h – Example Quit button for game start screen

```
#include "Button.h"

class QuitButton : public df::Button {

 public:
  QuitButton();
  void callback();
};
```

In the source code (Listing 4.187), the constructor sets the text displayed in the button to "quit" and places the button in the bottom center of the screen. Other options could include changing the button's color(s) and the presence of a border. The `callback()` method is invoked when the button is clicked. In this case, it sets game over to true, which causes the game loop to exit and the game engine to shutdown (see Section 4.4.4 on page 70).

Listing 4.188: QuitButton.cpp – Example Quit button for game start screen

```
#include "GameManager.h"
#include "QuitButton.h"

QuitButton::QuitButton() {
  setViewString("Quit");
  setLocation(df::BOTTOM_CENTER);
}

// On callback, set game over to true.
void QuitButton::callback() {
  GM.setGameOver();
```

4.16.3 Text Entry (optional)

Another common user interface option is the text entry widget, typically represented as a blank box that allows players to type in a string. Text entry is sometimes used for in-game options, such as typing in an action for a classic text adventure, but more often for extra-game options, such as entering the network address of a server in a multi-player game or typing in player initials in a high score table.

Like buttons, text entry widgets are presented to the player above the rest of the game objects and do not interact with the game world, like the Dragonfly ViewObject. Unlike the Button, the text entry widget does not need a mouse, but does need to respond to keyboard input as keys are pressed.

Listing 4.189 shows the TextEntry class, derived from the ViewObject class. TextEntry adds three attributes related to the text – `text` for the text characters, `limit` to limit how many characters can be entered and `numbers_only`, a boolean that if true, indicates that only numbers are accepted. Methods to get and set these attributes are provided. The constructor needs to set default attribute values and register for interest in keyboard events and step events (the latter to handle blinking the cursor). The `text` attribute needs to be initialized with all spaces (up to length `limit`) so that the text entry box is drawn properly – this is done in `setLimit()`, in case the game programmer changes the limit.

Listing 4.189: TextEntry.h

```
// Engine includes.
#include "EventMouse.h"
#include "ViewObject.h"

class TextEntry : public ViewObject {

 private:
  std::string m_text;              // Text entered.
  int m_limit;                     // Character limit in text.
  bool m_numbers_only;             // True if only numbers.
  int m_cursor;                    // Cursor location.
  char m_cursor_char;              // Cursor character.
  int m_blink_rate;                // Cursor blink rate.

 public:
  TextEntry();

  // Set text entered.
  void setText(std::string new_text);

  // Get text entered.
  std::string getText() const;

  // Handle "keyboard" events.
  // Return 0 if ignored, else 1.
  int eventHandler(const Event *p_e);

  // Called when TextEntry enter hit.
  // Must be defined by derived class.
  virtual void callback() = 0;

  // Set limit of number of characters allowed.
  void setLimit(int new_limit);

  // Get limit of number of characters allowed.
```

```
35   int getLimit() const;
36
37   // Set cursor to location.
38   void setCursor(int new_cursor);
39
40   // Get cursor location.
41   int getCursor() const;
42
43   // Set blink rate for cursor (in ticks).
44   void setBlinkRate(int new_blink_rate);
45
46   // Get blink rate for cursor (in ticks).
47   int getBlinkRate() const;
48
49   // Return true if only numbers can be entered.
50   bool numbersOnly() const;
51
52   // Set to allow only numbers to be entered.
53   void setNumbersOnly(bool new_numbers_only = true);
54
55   // Set cursor character.
56   void setCursorChar(char new_cursor_char);
57
58   // Get cursor character.
59   char getCursorChar() const;
60
61   // Draw viewstring + text entered.
62   virtual int draw();
63 };
```

The `callback()` method on line 29 is as for the Button class – declared as pure virtual (`=0`) meaning `callback()` *must* be defined before TextEntry can be used. As for a Button, the text entry specific behavior wanted must be implemented by the game programmer.

Most of the methods are implemented in a straightforward manner, with the exception of the `eventHandler()`, shown in Listing 4.190.

If the event is a step event, the code block from lines 7 to 17 handles the cursor blinking –the cursor in this case, is a character that toggles between an underscore (or whatever the cursor character is set to) and a space. The method uses a `static` variable to keep track of the blink count, counting up from a negative value. When the count passes zero, it toggles the cursor (blinks it).

Listing 4.190: TextEntry eventHandler()

```
0  // Handle "keyboard" events.
1  // Return 0 if ignored, else 1.
2  int TextEntry::eventHandler(const Event *p_e)
3
4    // If step event, blink cursor.
5    if p_e -> getType() is df::STEP_EVENT then
6
7      // Blink on or off based on rate.
8      static int blink = -1 * getBlinkRate()
9      if blink >= 0 then
10       text.replace(cursor, 1, 1, getCursorChar())
```

```
    else
      text.replace(cursor, 1, 1, ' ')
    end if
    blink = blink + 1
    if blink == getBlinkRate() then
      blink = -1 * getBlinkRate()
    end if

    return 1

  end if

  // If keyboard event, handle.
  if p_e -> getType() is KEYBOARD_EVENT and
     p_e -> getKeyboardAction() is KEY_PRESSED then

    // If return key pressed, then callback.
    if p_e -> getKey() is Keyboard::RETURN then
      callback()
      return 1
    end if

    // If backspace, remove character.
    if p_e -> getKey() is Keyboard::BACKSPACE then
      if cursor > 0  then
   if cursor < limit then
     text.replace(cursor, 1, 1, ' ')
        end if
   cursor = cursor - 1
   text.replace(cursor, 1, 1, ' ')
      end if
      return 1
    end if

    // If no room, cannot add characters.
    if cursor >= limit then
      return 1
    end if

    // Get key as string.
    std::string str = toString(p_k -> getKey())

    // If entry should be number, confirm.
    if numbers_only && not isdigit(str[0]) then
      return 1
    end if

    // Replace spaces with characters.
    text.replace(cursor, 1, str)
    cursor++

    // All is well.
    return 1
  end if
```

```
66   // If we get here, event is not handled.
67   return 0
```

If the event is a keyboard event, there are several possible actions. Remember, although not shown, the Event pointer p_e needs to be casted when used as an EventKeyboard (see Section 4.5.5.3 on page 94).

The code starting on Line 27 checks if the return key is pressed. If so, the callback() method is invoked.

The code starting on Line 33 checks if the backspace key is pressed. If so, there is an additional check if the cursor is at the beginning of the string. If not, the character immediately to the left of the cursor is replaced.

The code on Line 45 makes sure that there is still room to add more text. If not (the limit is reached) the method ends.

Otherwise, the code at the bottom of the method adds the keyboard character pressed by replacing the space in the string at cursor with the character pressed.

The TextEntry draw() method also has a bit of work to do beyond the ViewObject draw() method. The required logic is shown in Listing 4.191. Basically, the original ViewObject text (set to "Enter text:" or something similar in the child class constructor) is loaded, the text entered so far is added, and then drawn.

Listing 4.191: TextEntry draw()

```
// Draw viewstring + text entered.
int TextEntry::draw()

  // Get original view string.
  std::string view_str = getViewString()

  // Add text.
  setViewString(view_str + text)

  // Draw.
  ViewObject::draw()

  // Restore original view string.
  setViewString(view_str)
```

An example can help illustrate how the TextEntry class can be used. Consider a high score table where the player, upon hitting a score worthy of the table, is asked to enter his/her initials (3 characters). A text entry widget can be made as in Listings 4.192 (header file) and 4.193 (code). In the header file, NameEntry is derived from TextEntry. The only method that must be defined is callback(), but in this case the limit (3 characters) needs to be set, too.

Listing 4.192: NameEntry.h – Example TextEntry for player initials

```
#include "TextEntry.h"

class NameEntry : public df::TextEntry {

 public:
  NameEntryButton();
```

```
  void callback();
};
```

In the source code, the constructor sets the text entry widget in the center of the screen and indicates the player should enter initials (setting the character limit to 3). The `callback()` method is invoked when the return key is pressed – in this case, a message is written to the logfile, but probably the game programmer would do something else with the initials, such as add them to a table.

Listing 4.193: NameEntry.cpp – Example TextEntry for player initials

```
#include "LogManager.h"
#include "NameEntry.h"

NameEntry::NameEntry() {
  setViewString("Enter initials");
  setLocation(df::CENTER_CENTER);
  setLimit(3);
}

// On callback, write initials to logfile.
void QuitButton::callback() {
  LM.writeLog("High score: %s", getText().c_str());
}
```

4.16.4 Development Checkpoint #13!

Continue development of Dragonfly, incorporating ViewObjects. Steps:

1. Create a ViewObject class (`ViewObject.h` and `ViewObject.cpp`), inheriting from Object, based on Listing 4.174. Add `ViewObject.cpp` to the project. Stub out all the methods first and get it to compile.

2. Write the ViewObject constructor, based on Listing 4.175 and then `setLocation()`, based on Listing 4.176. Get your code to compile and verify by visual inspection of code.

3. Based on Listing 4.179, write the utility function `toString()` and put it in `utility.cpp` and `utility.h`. Test with a stand alone program, outside of any other aspect of the game engine, to be sure it properly converts a range of integers to string values.

4. Write the ViewObject `draw()` method, referring to Listing 4.178. Remember, since `draw()` gets called automatically in WorldManager `draw()`, first test your code by creating a ViewObject (via `new`) before calling the GameManager `run()` method. Verify that the ViewObject appears, testing its location in all six fixed locations around the screen, for arbitrary strings and values.

5. Create a EventView class, based on Listing 4.181. Add `EventView.cpp` to the project. Define the `eventHandler()` based on Listing 4.182. Verify the code compiles and use visual inspection on the methods.

6. Referring to Listing 4.183, construct an example that uses a ViewObject with a test program that changes the value of the object. Test with a variety of view events, with different values and deltas. Verify that a ViewObject only handles events that are targeted toward it, ignoring others.