Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #5

Display Manager

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 5.0). Information online at: `http://dragonfly.wpi.edu/book/`

4.7 Sending Events

While the only event Dragonfly handles right now is the step event, sent to each Object every iteration of the game loop, the engine will soon have more, and will need to send those events to Objects, as well. For efficiency and convenience, the code that currently resides in the GameManager to send events should be moved into the base Manager class. That way, derived Managers that handle events, say keyboard events, can send the events to the game objects.

To do this, the Manager is extended with an `onEvent()` method, shown in Listing 4.62. The code is the same as that in the GameManager that sends the step event to all Objects (Listing 4.51).

Listing 4.62: Manager onEvent()

```
// Send event to all Objects.
// Return count of number of events sent.
int Manager::onEvent(const Event *p_event) const
  count = 0

  all_objects = WorldManager getAllObjects()
  create ObjectListIterator li on all_objects list
  while not li.isDone() do
    li.currentObject() -> eventHandler() with p_event
    li.next()
    increment count
  end while

  return count
```

Once `onEvent()` is defined, the GameManager code in Listing 4.51) needs to be removed and replaced with:

Listing 4.63: GameManager providing step event

```
  ...
  // Provide step event to all Objects.
  EventStep s;
  onEvent(&s);
  ...
```

Note, the return value is not used here. However, the same `onEvent()` method can be used for user-defined events, such as the "nuke" event in the Saucer Shoot tutorial (see Section 3.3.8 on page 32). The count of events sent returned by `onEvent()` may be useful for game programmer code.

4.8 Display Management

While games are much more than just pretty visuals, graphical output is an important, if not the *most* important, element of a computer game. As previously noted, Dragonfly is a text-based game engine (see Section 3.2 for why), using the Simple and Fast Multimedia Library (SFML) to help with drawing characters on the screen, described next.

4.8.1 Simple and Fast Multimedia Library – Graphics

The Simple and Fast Multimedia Library (SFML) provides a relatively easy interface for displaying graphics and playing sounds, as well as gathering input from the keyboard and mouse. SFML has been ported to most major platforms, including Windows, Linux and Mac, and even to iOS and Android mobile platforms. SFML is free and open-source, under the zlib/png license.

For graphics output, SFML provides a graphics module for 2D drawing. The graphics module makes use of a specialized window class, `sf::RenderWindow`. Creating a window (which will pop it open on the screen) can be done with the code in Listing 4.64.

Listing 4.64: SFML window

```
#include <SFML/Graphics.hpp>
...
// Create SFML window.
int window_horizontal = 1024
int window_vertical = 768
sf::RenderWindow window(sf::VideoMode(horizontal, vertical), "Title -
    Dragonfly", sf::Style::Titlebar)

// Turn off mouse cursor for window.
window.setMouseCursorVisible(false)

// Synchronize refresh rate with monitor.
window.setVerticalSyncEnabled(true)

// When done...
window.close()
```

The first argument for an `sf::RenderWindow` is the video mode that defines the size of the window (the inner size, not including the title bar and borders). Listing 4.64 creates a window 1024x768 pixels. The second argument, the string "Title - Dragonfly", is the title of the window. The third argument provides the window style, here in the form of a title bar. The third argument is actually optional – not including it will provide the default style of a title bar and resize and close buttons.

For a text-only window, such as in Dragonfly, it is often useful to hide the mouse cursor when the mouse is over the window. This is done with the `setMouseCursorVisible()` call, passing in `false`. The cursor can be shown, of course, by passing in `true`, too.

If the game engine drawing rate is faster than the monitors refresh rate, there may be visual artifacts such as tearing. Synchronizing the SFML refresh rate with the monitor's refresh rate is done by `setVerticalSyncEnabled()`. This is only called once, after creating the window.

When use of the window is done, `close()` closes the window and destroys all the attached resources.

Before drawing any text, SFML needs to have the font loaded using the `sf::Font` class. Typically, the font is loaded from the disk using the `loadFromFile()` method as in Listing 4.65. The string "df-font.ttf"[13] is the name of the font file, supporting most standard

[13] The Dragonfly engine (`http://dragonfly.wpi.edu/engine/`) includes the default Dragonfly font file "df-font.ttf" that can be used for development.

formats. Note, the exact path to the font file must be provided since SFML cannot directly access any standard fonts installed on the system.

Listing 4.65: SFML font

```
sf::Font font
if (font.loadFromFile("df-font.ttf") == false) then
  // Error
end if
```

To draw text, the `sf::Text` class is used, as in Listing 4.66. The method `setFont()` is used to select a previously loaded font, as in Listing 4.65. The method `setString()` provides the string to be displayed. The method `setCharacterSize()` sets the character size, in pixels not point size. The method `setColor()` sets the text color to a type `sf::Color`, with built in color choices of Black, White, Red, Green, Blue, Yellow, Magenta and Cyan. The method `setStyle()` sets the text style, in this case bold and underlined. `setPosition()` sets the location on the window (in pixels) to draw the text.

Listing 4.66: SFML text

```
sf::Text text

// Select pre-loaded font (from Listing 4.65).
text.setFont(font)

// Set display string.
text.setString("Hello world")

// Set character size (in pixels).
text.setCharacterSize(24)

// Set color.
text.setColor(sf::Color::Red)

// Set style.
text.setStyle(sf::Text::Bold | sf::Text::Underlined)

// Set position on window (in pixels).
text.setPosition(100, 50)
```

Once setup, the text can be drawn on the window. Drawing text requires a few steps, illustrated in Listing 4.67. The `clear()` method clears the window and is usually called each game loop right before drawing commences. Note, as an option, `clear()` can also be given a background color to paint the window (e.g., `clear(sf::Color::Blue)`). The `draw()` method draws the text on the window, but does not actually display it yet. The `display()` method displays on the window what has been drawn.

Listing 4.67: SFML drawing text

```
// Clear window and draw text.
window.clear();
window.draw(text);
window.display();
```

Putting it together, a "Hello, world!" sample can be created, shown in Listing 4.68, demonstrating the basic SFML graphics needed for Dragonfly. In Listing 4.68, the top part loads the font, as in Listing 4.65. The next part sets up the text field to display, as in Listing 4.66. The main loop at the `while()` repeats drawing the text as in Listing 4.67, then checking if the window has been closed. Once the window is closed, `main()` will return and the process stopped.

Listing 4.68: SFML Hello, world!

```
#include <iostream> // for std::cout
#include <SFML/Graphics.hpp>

int main() {

  // Load font.
  sf::Font font;
  if (font.loadFromFile("df-font.ttf") == false) {
    std::cout << "Error! Unable to load font 'df-font.ttf'." << std::endl;
    return -1;
  }

  // Setup text to display.
  sf::Text text;
  text.setFont(font); // Select font.
  text.setString("Hello, world!"); // Set string to display.
  text.setCharacterSize(32); // Set character size (in pixels).
  text.setColor(sf::Color::Green); // Set text color.
  text.setStyle(sf::Text::Bold); // Set text style.
  text.setPosition(96,134); // Set text position (in pixels).

  // Create window to draw on.
  sf::RenderWindow *p_window =
    new sf::RenderWindow(sf::VideoMode(400, 300), "SFML - Hello, world!");
  if (!p_window) {
    std::cout << "Error! Unable to allocate RenderWindow." << std::endl;
    return -1;
  }

  // Turn off mouse cursor for window.
  p_window -> setMouseCursorVisible(false);

  // Synchronize refresh rate with monitor.
  p_window -> setVerticalSyncEnabled(true);

  // Repeat as long as window is open.
  while (1) {

    // Clear window and draw text.
    p_window -> clear();
    p_window -> draw(text);
    p_window -> display();

    // See if window has been closed.
    sf::Event event;
    while (p_window -> pollEvent(event)) {
```

```
46       if (event.type == sf::Event::Closed) {
47         p_window -> close();
48         delete p_window;
49         return 0;
50       }
51     } // End of while (event).
52
53   } // End of while (1).
54
55 } // End of main().
```

Note, treating the SFML window as a pointer (`sf::RenderWindow *`) starting on Line 22 is not strictly necessary (after all, it is not a pointer in Listing 4.64), but it more closely mimics use by the DisplayManager (described in the next section) so is used in this example.

4.8.2 The DisplayManager

This section introduces the DisplayManager. Before doing so, however, providing a means for Dragonfly to support color is provided.

4.8.2.1 Color

Life is better with color, and so are most games!* Since the DisplayManager will support such game-enhancing color, it is helpful for the engine and the game programmer to define Dragonfly colors in a separate header file. Listing 4.69 shows `Color.h` which has an `enum Color` that provides for the built-in colors Dragonfly recognizes. For drawing functions where no color is specified, `COLOR_DEFAULT` is used.

Listing 4.69: Color.h

```
// Colors Dragonfly recognizes.
enum Color {
  UNDEFINED_COLOR = -1,
  BLACK = 0,
  RED,
  GREEN,
  YELLOW,
  BLUE,
  MAGENTA,
  CYAN,
  WHITE,
};

// If color not specified, will use this.
const Color COLOR_DEFAULT = WHITE;
```

The DisplayManager is a singleton class derived from Manager. Thus, as described in Section 4.2.1 on page 53, the DisplayManager has private constructors and a `getInstance()`

* **Did you know (#4)?** Newly-emerged Dragonflies usually have muted colors and can take days to gain their bright, adult colors. Some adults change color as they age. – "Frequently Asked Questions about Dragonflies", *British Dragonfly Society*, 2013.

method to return the one and only instance. The header file, including class definition, is provided in Listing 4.70.

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

Line 6 has a `#include` for `Vector.h` since the DisplayManager draws characters on the screen at a given (x,y) location provided by a Vector object. A `#include` for `Color.h` is also included since the Dragonfly colors are used for drawing.

The next section, starting at line 8, provides the default settings for the Dragonfly window rendered on the screen. These include horizontal and vertical pixels, horizontal and vertical characters, window style, color and title and the font file to used for drawing characters. Note, the background color (`WINDOW_BACKGROUND_COLOR_DEFAULT` on on Line 14) is actually of type `sf::Color` and not type `df::Color` in order to make drawing more efficient by not having to map the background color, which does not change much, for every character drawn.

The private attributes starting on line 24 store the important window attributes. Note that the SFML window is stored as a pointer on line 25 since this allows the window to be allocated during startup, instead of when the DisplayManager is instantiated.

The `startUp()` method gets the SFML display ready, calling many of the SFML functions shown in Listings 4.64 and 4.65.

The `shutDown()` method closes the SFML window calling `close()` and de-allocates memory.

The `drawCh()` routine uses SFML fonts and the SFML `draw()` method (see Listing 4.67) to draw the indicated character at the (x,y) location specified by the position and color.

The methods `getHorizontal()` and `getVertical()` return the horizontal and vertical character limits of the window, respectively. Similarly, the methods `getHorizontalPixels()` and `getVerticalPixels()` return the horizontal and vertical pixel limits of the window, respectively.

For drawing, the DisplayManager uses `p_window`, a pointer to an SFML `sf::RenderWindow`. Most 2d and 3d graphics setups have two buffers – one for the current window being displayed and the second for the one being drawn. When the new window is ready to be displayed, it is swapped with the current window. The DisplayManager does not need this mechanism since the `draw()` method effectively does this swapping. The `swapBuffers()` provides this feature. The method `getWindow()` returns the SFML window, which can be useful for game code that wants to make use of additional SFML features beyond drawing characters.

Listing 4.70: DisplayManager.h

```
0 // System includes.
1 #include <SFML/Graphics.hpp>
2
3 // Engine includes.
4 #include "Color.h"
5 #include "Manager.h"
6 #include "Vector.h"
7
8 // Defaults for SFML window.
9 const int WINDOW_HORIZONTAL_PIXELS_DEFAULT = 1024;
```

```
const int WINDOW_VERTICAL_PIXELS_DEFAULT =  768;
const int WINDOW_HORIZONTAL_CHARS_DEFAULT = 80;
const int WINDOW_VERTICAL_CHARS_DEFAULT = 24;
const int WINDOW_STYLE_DEFAULT = sf::Style::Titlebar;
const sf::Color WINDOW_BACKGROUND_COLOR_DEFAULT = sf::Color::Black;
const std::string WINDOW_TITLE_DEFAULT =  "Dragonfly";
const std::string FONT_FILE_DEFAULT = "df-font.ttf";

class DisplayManager : public Manager {

 private:
  DisplayManager();                          // Private (a singleton).
  DisplayManager(DisplayManager const&);     // Don't allow copy.
  void operator=(DisplayManager const&);     // Don't allow assignment
  sf::Font m_font;                  // Font used for ASCII graphics.
  sf::RenderWindow *m_p_window;     // Pointer to SFML window.
  int m_window_horizontal_pixels; // Horizontal pixels in window.
  int m_window_vertical_pixels;   // Vertical pixels in window.
  int m_window_horizontal_chars;  // Horizontal ASCII spaces in window.
  int m_window_vertical_chars;    // Vertical ASCII spaces in window.

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

  // Open graphics window ready for text-based display.
  // Return 0 if ok, else -1.
  int startUp();

  // Close graphics window.
  void shutDown();

  // Draw character at window location (x,y) with color.
  // Return 0 if ok, else -1.
   int drawCh(Vector world_pos, char ch, Color color) const;

  // Return window's horizontal maximum (in characters).
  int getHorizontal() const;

  // Return window's vertical maximum (in characters).
  int getVertical() const;

  // Return window's horizontal maximum (in pixels).
  int getHorizontalPixels() const;

  // Return window's vertical maximum (in pixels).
  int getVerticalPixels() const;

  // Render current window buffer.
  // Return 0 if ok, else -1.
  int swapBuffers();

  // Return pointer to SFML drawing window.
  sf::RenderWindow *getWindow() const;
};
```

In more detail, the `startUp()` method does the steps shown in Listing 4.71. The first block of code is a redundancy check to see if the SFML window (`p_window`) is already allocated. If so, that indicates an SFML window was already created (probably, due to DisplayManager `startup()` already having been called) and the method returns, but indicates no error. Note, `p_window` should be initialized to `NULL` in the DisplayManager constructor.

After that, the mouse cursor is turned off and the drawing refresh rate is synchronized with the monitor (see (see Listing 4.64 on page 103), and the engine font is loaded (see Listing 4.65 on page 104) from the file (`FONT_FILE_DEFAULT`).

If the window can be created and the font loaded, the Manager `startUp()` method is called, which sets `is_started` is to `true`. Later, upon a successful `shutDown()` it is set to `false` by calling Manager `shutDown()`.

Listing 4.71: DisplayManager startUp()

```
// Open graphics window ready for text-based display.
// Return 0 if ok, else return -1.
int DisplayManager::startUp()

  // If window already created, do nothing.
  if p_window is not NULL then
    return 0
  end if

  create window // an sf::RenderWindow for drawing

  turn off mouse cursor

  synchronize refresh rate with monitor

  load font

  if everything successful then
    invoke Manager::startUp()
    return ok
  else
    return error
  end if
```

As noted, Dragonfly is text-based in that game programmers render graphics through displaying 2-d ASCII art on the game window. Since SFML is fundamentally pixel based, not text-based, it is useful to have functions that convert (x,y) pixel coordinates to (x,y) text coordinates and vice versa. In turn, these functions make use of helper functions to compute the character height and width (in pixels) based on the dimensions of the game window. Listing 4.72 shows the full list of helper functions. These are declared in `DisplayManager.h` and defined in `DisplayManager.cpp`, but are utility-type functions, not part of the DisplayManager class. Since they are not general game-programmer utilities, they are also not part of `utility.h`.

Listing 4.72: DisplayManager drawing helper functions

```
// Compute character height, based on window size and font.
float charHeight();
```

```

// Compute character width, based on window size and font.
float charWidth();

// Convert ASCII spaces (x,y) to window pixels (x,y).
Vector spacesToPixels(Vector spaces);

// Convert window pixels (x,y) to ASCII spaces (x,y).
Position pixelsToSpaces(Vector pixels);
```

The function `charHeight()` computes and returns the height (in pixels) of each character, which is number of vertical pixels (DisplayManager `getVerticalPixels()`) divided by the number of vertical characters (DisplayManager `getVertical()`). Similarly the function `charWidth()` computes and returns the width (in pixels) of each character, which is number of horizontal pixels (DisplayManager `getHorizontalPixels()`) divided by the number of horizontal characters (DisplayManager `getHorizontal()`).

Then, to convert spaces to pixels in `spacesToPixels()`, the x coordinate is multiplied by `charWidth()` and the y coordinate is multiplied by `charHeight()`. Conversely, in `pixelsToSpaces()`, the x coordinate is divided by `charWidth()` and the y coordinate is divided by `charHeight()`.

With the helper functions in place, the `drawCh()` method does the steps shown in Listing 4.73.

The first step starting on line 4 makes sure the SFML window has been allocated (it should have been if the DisplayManager has been successfully started).

Next, on line 10 spaces are converted to pixels. This provides the location on the SFML window where the character will be drawn.

In SFML, ASCII text is "see through" in that any characters behind show through, generally unexpected for the game programmer. To avoid this, a rectangle in the same color as the window background is drawn, effectively hiding any previously drawn characters. An `sf::RectangleShape` is used for this, setting the size, color and position with `setSize()`, `setFillColor()` and `setPosition()`, respectively. The method `draw()` on line 12 draws the rectangle on the window first, before the character is drawn on top.

The character to be drawn is embedded in an `sf::Text` object in the steps starting on line 20, using `setString()` to actually set the text string to the desired character. Making the character bold with on line 23 is optional, but it tends to make all the graphics "pop" a bit more.

Before actually drawing the text character, it needs to be scaled to the right size using `setCharacterSize()`. The scaling depends upon whichever is smaller, the character width or the character height, checked in line 26.

The drawing color specified in Dragonfly (e.g., `df::YELLOW`) needs to be mapped to the corresponding SFML color (e.g., `sf::Color::Yellow`). This is easily and efficiently done in a `switch()` statement, shown on line 32.

Lastly, the text is positioned at the right pixel location (line 43) and the character is drawn with the `sf::Text draw()` method.

Note, although not strictly necessary, both the `sf::rectangle` and the `sf::text` objects are declared as `static` so as not to re-allocate them each time.

Listing 4.73: DisplayManager drawCh()

```
0  // Draw a character at window location (x,y) with color.
1  // Return 0 if ok, else -1.
2  int DisplayManager::drawCh(Vector world_pos, char ch, Color color)
3
4    // Make sure window is allocated.
5    if p_window is NULL then
6      return -1
7    end if
8
9    // Convert spaces (x,y) to pixels (x,y).
10   Vector pixel_pos = spacesToPixels(world_pos)
11
12   // Draw background rectangle since text is "see through" in SFML.
13   static sf::RectangleShape rectangle
14   rectangle.setSize(sf::Vector2f(charWidth(), charHeight()))
15   rectangle.setFillColor(WINDOW_BACKGROUND_COLOR_DEFAULT)
16   rectangle.setPosition(pixel_pos.getX() - charWidth()/10,
17                         pixel_pos.getY() + charHeight()/5)
18   p_window -> draw(rectangle)
19
20   // Create character text to draw.
21   static sf::Text text("", font)
22   text.setString(ch)
23   text.setStyle(sf::Text::Bold) // Make bold, since looks better.
24
25   // Scale to right size.
26   if (charWidth() < charHeight()) then
27     text.setCharacterSize(charWidth() * 2)
28   else
29     text.setCharacterSize(charHeight() * 2)
30   end if
31
32   // Set SFML color based on Dragonfly color.
33   switch (color)
34   case YELLOW:
35     text.setColor(sf::Color::Yellow)
36     break;
37   case RED:
38     text.setColor(sf::Color::Red)
39     break;
40   ...
41   end switch
42
43   // Set position in window (in pixels).
44   text.setPosition(pixel_pos.getX(), pixel_pos.getY())
45
46   // Draw character.
47   p_window -> draw(text)
```

Note, the multiplier 2 on Lines 27 and 29 scale the text to typical terminal dimensions (such as you might see in a Linux shell). These characters, and characters in general, tend to be rectangle shaped, somewhat taller than they are wide. For a game that needs square cells, the multiplier can be set to 1. The characters will still appear normal, but there will be some horizontal (empty) padding to make the characters effective squares on the screen.

The `swapBuffers()` method does the steps shown in Listing 4.74. Basically, after checking if the window `p_window` is allocated, `display()` is invoked to make all changes since the previous refresh visible to the player). Then, `clear()` is called immediately to get ready for the next drawing. It may seem counter-intuitive to clear the window right after drawing, but remember that there are actually two buffers in play here – one that is currently being displayed, made so by the `display()` call, and one that is going to be drawn on and then displayed the next game loop. This second buffer is the one that is cleared with the `clear()` call.

Listing 4.74: DisplayManager swapBuffers()

```
// Render current window buffer.
// Return 0 if ok, else -1.
int DisplayManager::swapBuffers()

  // Make sure window is allocated.
  if p_window is NULL then
    return -1
  end if

  // Display current window.
  p_window -> display()

  // Clear window to get ready for next draw.
  p_window -> clear()

  return 0 // Success.
```

4.8.3 Using the DisplayManager

With the DisplayManager in place, the Object class can be extended to support using it. Specifically, the Object is given a `virtual` method for drawing, shown in Listing 4.75.

Listing 4.75: Object draw()

```
public:
  virtual void draw();
```

The Object `draw()` method does nothing itself, but can be overridden by derived classes. For example, the Star in Saucer Shoot (Section 3.3) defines the `draw()` method as in Listing 3.6 on page 37. When the `draw()` method for a Star is called, the Star invokes the `drawCh()` method of the DisplayManager, giving it the position of the Star and the character to be drawn (a '`.`').

With `draw()` defined for each game object, the engine can handle redrawing each Object every game loop. To do this, the WorldManager is extended with a `draw()` method of its own which iterates through all game objects, calling an Object's `draw()` method each iteration. Listing 4.76 illustrates the pseudo code.

Listing 4.76: WorldManager draw()

```
// Draw all objects.
```

```
void WorldManager::draw()

  create ObjectListIterator li on updates // All game objects.

  while not li.isDone() do
    Object *p_temp_o = li.currentObj()
    p_temp_o -> draw()
    li.next()
  end while
```

The game loop in the GameManager needs a couple of additional lines, first to invoke the WorldManager `draw()` method and then to call the DisplayManager `swapBuffers()` method. Listing 4.77 shows the game loop, with line 5 calling WorldManager `draw()` and Line 6 calling DisplayManager `swapBuffers()`. Note, line 4 calls the WorldManager `update()` method, as described in Section 4.5.6.2 on page 98. Line 3 gets the input from the player, which is described next in Section 4.9.

Listing 4.77: The game loop with drawing

```
0 Clock clock
1 while (game not over) do
2   clock.delta()
3   // Get input from keyboard/mouse.
4   WorldManager update()
5   WorldManager draw()
6   DisplayManager swapBuffers()
7   loop_time = clock.split()
8   sleep(TARGET_TIME - loop_time)
9 end while
```

4.8.4 Drawing Strings

Note, for now, the DisplayManager only supports drawing a single character (like a Star '`.`'). Later, the DisplayManager will be extended to support drawing sprites. However, at this time, a practical exercise is to extended the DisplayManager to draw a string at a given (x,y) location. Specifically, it will be used for ViewObjects in Section 4.16 (page 191). More generally, a string drawing routine is useful for a game that wants to draw strings on the screen, such as for instructions or the player's name. Listing 4.78 shows the `drawString()` method prototype. The enumerated type `enum Justification` allows drawing the string to the left of the (x,y) position, centered on the (x,y) position or to the right of the (x,y) position.

Listing 4.78: DisplayManager extensions to support drawing strings

```
enum Justification {
  LEFT_JUSTIFIED,
  CENTER_JUSTIFIED,
  RIGHT_JUSTIFIED,
};

...

```

```
class DisplayManager : public Manager {

  ...

  // Draw string at window location (x,y) with default color.
  // Justified left, center or right.
  // Return 0 if ok, else -1.
  int drawString(Vector pos, std::string str, Justification just,
                 Color color) const;
  ...
};
```

Listing 4.79 shows the code for `drawString()`. The opening switch statement determines the starting position for the string. If it is center justified, the starting position is moved to the left by one-half the length of the string. If it is right justified, the starting position is moved to the left by the length of the string. If it is left justified no modifications to the starting position are made. This is the default (and is also the behavior if any invalid `Justification` value is given).

Once the starting position is determined, the `for` loop starting on line 21 writes out the string a character at a time, moving the x position over by one each time.

Listing 4.79: DisplayManager drawString()

```
0  // Draw string at window location (x,y) with color.
1  // Justified left, center or right.
2  // Return 0 if ok, else -1.
3  int DisplayManager::drawString(Vector pos, std::string str,
4                                 Justification just,
5                                 Color color) const
6    // Get starting position.
7    Vector starting_pos = pos
8    switch (just)
9    case CENTER_JUSTIFIED:
10     starting_pos.setX(pos.getX() - str.size()/2)
11     break
12   case RIGHT_JUSTIFIED:
13     starting_pos.setX(pos.getX() - str.size())
14     break
15   case LEFT_JUSTIFIED:
16   default:
17     break
18   end switch
19
20   // Draw string character by character.
21   for i = 0 to str.size()
22     Vector temp_pos(starting_pos.getX() + i, starting_pos.getY())
23      drawCh(temp_pos, str[i], color)
24   end for
25
26   return ok
```

4.8.5 Drawing in Layers

Up until now, there is no easy way to make sure one object is drawn before another. For example, if the Saucer Shoot game from Section 3.3 was made, a Star could appear on top of the Hero. In order to provide layering control that allows the game programmer to explicitly determine which objects are drawn on top of which, Dragonfly has an "altitude" feature. Objects at low altitude are drawn before objects at higher altitude. Higher altitude objects drawn in the same location "overwrite" the lower ones before the screen is refreshed. For example, in Saucer Shoot, Stars are always drawn at low altitude so that they will always appear to be behind all other objects (e.g., Saucers and Bullets). Note that this feature is not a 3rd dimension – Dragonfly is still a 2d game engine – since layering is only used for drawing and not for moving and, more importantly, not for collisions. In other words, Objects can potentially collide with any Object, regardless of altitude.

In order to implement altitude, each game Object is given an altitude attribute and methods allow for getting and setting the altitude, all shown in Listing 4.80. The method `setAltitude()` checks that the new altitude is within the supported range, 0 to the maximum supported. The maximum supported should be defined as `const int MAX_ALTITUDE` in `WorldManager.h` and set to 4. In the Object constructor, the initial altitude should be set to 1/2 of `MAX_ALTITUDE`.

Listing 4.80: Object class extensions to support altitude

```
private:
 int m_altitude;        // 0 to MAX supported (lower drawn first).

public:
 // Set altitude of Object, with checks for range [0, MAX_ALTITUDE].
 // Return 0 if ok, else -1.
 int setAltitude(int new_altitude);

 // Return altitude of Object.
 int getAltitude() const;
```

With the altitude attributes and methods in place, in the WorldManager `draw()` method, an outer loop is added to go through each of the altitudes, low to high, as shown in Listing 4.81. If the Object's altitude matches the loop iterator, it is drawn. Drawing from low to high means Objects at higher altitudes are drawn "on top" of Objects at lower altitudes.

Listing 4.81: WorldManager extensions to support altitude

```
 // In draw()...
 for alt = 0 to MAX_ALTITUDE

   // Normal iteration through all objects.
   ...

     if li.currentObject() -> getAltitude() is alt then

       // Normal draw.
       ...

     end if
```

```
end for // Altitude outer loop.
```

While the looping method in Listing 4.81 is effective and simple (good attributes for most programs), it is not particularly efficient. Each object is drawn only once, just as it was before altitude was added. However, as specified by the outer loop, the WorldManager `draw()` method iterates through all objects 5 times (0 to `MAX_ALTITUDE`). This can be fixed by storing the objects according to their altitudes and fetching them only once. Such efficiency is a common feature of a scene graph and is addressed in the Dragonfly SceneGraph in Section 4.17.1 (page 208).

4.8.6 Colored Backgrounds (optional)

The default color scheme for Dragonfly has a black background. For some games – for example, a game of naval warfare on the high seas – a different color background, perhaps blue, may be more appropriate. To add support for alternate background colors, the DisplayManager can be extended as shown in Listing 4.82. The extension includes a private attribute for the background is added as well as a method to set it.

Listing 4.82: DisplayManager extension to support background colors

```
private:
 sf::Color window_background_color; // Background color of window.
 ...

public:
 // Set default background color.  Return true if ok, else false.
 bool setBackgroundColor(int new_color);
 ...
```

The `setBackgroundColor()` method maps the Dragonfly color, of type `Color` to the SFML color, of type `sf::Color`. The call to `clear()` in `swapBuffers()` is modified to pass in `window_background_color`.

Once in place, the game programmer can make a different colored background, say blue, by adding the call:

```
DM.setBackgroundColor(df::BLUE);
```

after starting up the game engine.

4.8.7 Development Checkpoint #5!

Continue with your Dragonfly development of your *Dragonfly Egg** from Section 4.6, by adding functionality for managing graphics from Section 4.8. Steps:

1. Modify GameManager `run()` to provide step events as in Listing 4.63. Create a test game object that receives these events. Verify with logfile or game object output,

* **Did you know (#5)?** There are about 5000 known species of dragonflies and damselflies, with an estimate of about 5500 and 6500 species in total. – "The Dragonfly Website", http://dragonflywebsite.com/faq.htm

counting the number of step events that should be received multiplied by the wall clock time (e.g., running for 30 seconds should yield 900 step events).

2. Create a DisplayManager derived class, inheriting from the Manager class. Implement DisplayManager as a singleton, described in Section 4.2.1 on page 53. Add `DisplayManager.cpp` to the project, and include stubs for all methods in Listing 4.70. Make sure the class (with stubs) compiles.

3. Write `startUp()` and `shutDown()` methods for the DisplayManager, referring to Listing 4.71 as needed. Implement `getWindow()` and then `swapBuffers()` based on Listing 4.74. Outside the GameManager, test that the DisplayManager can be started up, writing a character on the window using `getWindow()`, an `sf::Text` object and `draw()`, and then shut down.

4. Implement `getHorizontal()` and `getVertical()`. Test by starting the DisplayManager, making calls to `getHorizontal()` and `getVertical()` and writing them to the logfile. Verify the values reported correspond to the window size.

5. Implement `drawCh()` as in Listing 4.73. Verify that it works by replacing the drawing test code in the previous steps with calls to `drawCh()`. Once tested, implement `drawString()` which utilizes `drawCh()`. Test with a variety of strings and justifications.

6. Add the empty `draw()` method and create a game object derived from Object (e.g., a Star) with a `draw()` method that calls the DisplayManager `drawCh()`. Implement the `draw()` method in the WorldManager based on Listing 4.76. Modify the game loop in the GameManager to call the WorldManager `draw()` method and the DisplayManager `swapBuffers()`, as in Listing 4.77. Test that the custom game object is drawn properly as the game runs.

7. If implementing drawing in layers, add support for Object altitude, as in Listing 4.80. Extend the WorldManager to support altitude also, referring to Listing 4.81 as needed. Make an derived object (e.g., a Star) at a lower altitude than another derived object. Place the objects on top of each other and verify that the background object is obscured by the foreground object. Test several different layers at several different locations, along with an object that changes its altitude as the game runs. Verify that objects cannot set their altitude outside of the `[0, MAX_ALTITUDE]` range limits.

Listing 4.83: Testing the DisplayManager

```
#include <unistd.h> // Needed in Linux/Mac.

#include "DisplayManager.h"

int main() {
  DM.startUp();
  DM.drawCh(df::Vector(10,5), '*', df::WHITE);
  DM.swapBuffers();
  sleep(2);  // Use Sleep(2000) if in Windows.
```

```
  DM.shutDown();
}
```

At this point, you should now be able to actually see game objects in the window! This is an important milestone in development of an engine, and one that lets you develop and test by verifying object interactions visually. However, output to the logfile becomes even more important as writing debugging messages to window is more difficult.