



# Program a Game Engine from Scratch

Mark Claypool

## Chapter 4 - Engine

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/>

Copyright ©2012–2017 Mark Claypool and WPI. All rights reserved.

## Chapter 4

# Engine

Introduction understood (Chapter 1), development environment setup (Chapter 2) and tutorial complete (Chapter 3) means it is time start coding an engine! This chapter begins with an overview of **Dragonfly** then proceeds with sections on the design and development of each of the major game engine concepts and components.

### 4.1 Overview

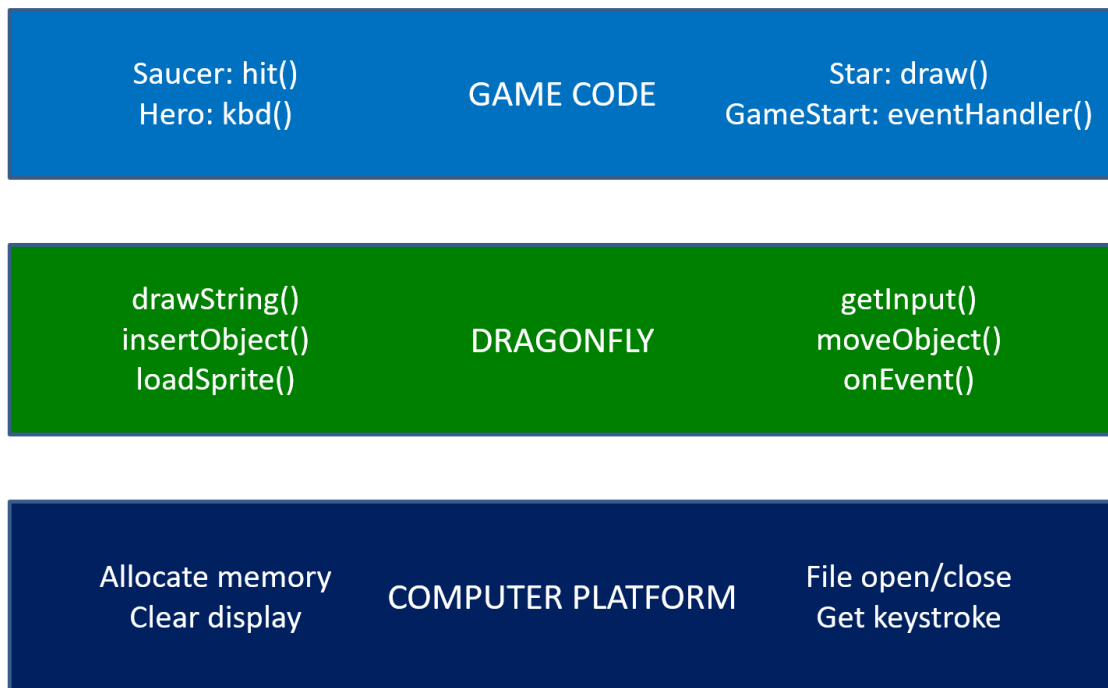


Figure 4.1: **Dragonfly** architectural overview

Figure 4.1 depicts an overview of the **Dragonfly** game engine relative to other parts of

Table 4.1: *Dragonfly* Classes

Type	Class	Description
Utility	Box	2d rectangle
	Vector	2d vector
	Clock	Timing support
	Music	Store and play music
	Sound	Store and play sound effects
	Frame	2d character image
	Sprite	Sequence of frames for animated image
Event	Event	Base class for engine events
	EventCollision	Event generated solid objects collide
	EventKeyboard	Event generated when keyboard has input
	EventMouse	Event generated when mouse has input
	EventOut	Event generated when object goes outside world
	EventStep	Event generated each game loop
Manager	EventManager	Base class for engine managers
	DisplayManager	Manager of the graphics display
	GameManager	Manager of the game loop
	InputManager	Manager of input (keyboard and mouse)
	LogManager	Manager of the logfile
	ResourceManager	Manager of resources (sprites)
	WorldManager	Manager of the game world
Object	Object	Base class for game engine objects
	ViewObject	View objects displayed on the Heads Up Display
	ObjectList	List container for Objects
	ObjectListIterator	Iterator for list of Objects

the system and game. The top layer, GAME CODE, depicts functionality that is handled by the game programmer, the middle layer, DRAGONFLY, by the game engine and the bottom layer, COMPUTER PLATFORM, by the operating system and computer hardware.

As discussed in Section 1.2, code in the GAME CODE layer provides specific game functionality that is not, and often cannot, be handled by the game engine. For example, for *Saucer Shoot* in Section 3.3, the Saucer interprets the action of a collision event in the `hit()` method. This action is specific to this game, as a collision for a Saucer, or any other object, in a different game could have very different results. Similarly, the Hero object must decide what to do on a keyboard event in the `kbd()` method, where other games would likely have different actions for different keyboard inputs.

Game engine code in the DRAGONFLY layer provides for functionality that is general to all games made with this engine. In this way, the engine provides convenient game functionality that makes it easier for the game programmer to develop the game and may



provide more efficient implementations in some cases. For **Dragonfly**, functionality for drawing characters on the screen, loading sprites into the engine, moving objects through the game world, and triggering events from one object to another, provide core functionality needed by the game programmer.

Low-level system functionality in the COMPUTER PLATFORM layer provides for hardware-specific features, such as clearing the terminal window, opening and closing files, allocating memory and gathering keystrokes. A game engine, such as **Dragonfly**, may be ported to different platforms (e.g., Mac) by re-writing platform-specific engine code. Once done, in many cases, this allows the same game code to then run on multiple, different computer platforms allowing a game to be easily ported.

Table 4.1 lists the **Dragonfly** classes for the core engine, grouped hierarchically and by function. Utilities are designed to provide convenient, low-level functionality used by many other classes. Events convey actions in the game engine to appropriate objects. Managers are responsible for the different functionalities of the game engine. Game objects, with some support classes, are managed by the game engine, with some specialized classes for custom displays (e.g., a “heads-up” display, or user interface buttons). The scene graph organizes Objects in the world for more efficient management and rendering.



## 4.2 Managers

Managers are the support systems for game engines, handling crucial tasks. This includes handling input, rendering graphics, logging data, managing the game world and game objects and more. Logically, the different functions can be broken up into different managers. The main class, Manager, is not instantiated. Instead, it serves as a base class for all derived game engine managers. Refer to Table 4.1 on page 49 for details.

The interface for the *Dragonfly* Manager class is shown in Listing 4.1. The Manager class provides `startUp()` and `shutDown()` methods, allowing the game programmer to control initialization and termination of all derived manager objects. For the base Manager, `startUp()` sets `is_started` to `true` and `shutDown()` sets `is_started` to `false`. The method `isStarted()` allows for a query to check if the manager has been successfully started (via `startUp()`). As this is the base class, in Manager, the methods for starting up and shutting do not do any “real” work – instead just manipulating the `is_started` boolean variable. The method `setType()` sets the private attribute `type` to the name “Manager”. The method is `protected` since only the base class and derived classes are allowed to change a manager’s type (typically, this is done in the constructor for each derived Manager).

Listing 4.1: Manager.h

```

0 namespace df {
1
2 class Manager {
3
4     private:
5         std::string m_type;           // Manager type identifier.
6         bool m_is_started;           // True when started successfully.
7
8     protected:
9         // Set type identifier of Manager.
10        void setType(std::string type);
11
12    public:
13        Manager();
14        virtual ~Manager();
15
16        // Get type identifier of Manager.
17        std::string getType() const;
18
19        // Startup Manager.
20        // Return 0 if ok, else negative number.
21        virtual int startUp();
22
23        // Shutdown Manager.
24        virtual void shutDown();
25
26        // Return true when startUp() was executed ok, else false.
27        bool isStarted() const;
28    };
29
30 } // end of namespace df

```



Line 0 of Listing 4.1 defines the **Dragonfly** namespace using the `df::` tag. This requires code outside of the namespace (e.g., game code) to use `df::` to access elements inside the namespace (e.g., `setSolidness(df::SPECTRAL)`). Typically large, 3rd-party libraries (such as a game engine) use namespaces to help developers avoid conflicts in names their own code may use with names the libraries use. The **Dragonfly** namespace is meant to prevent potential name conflicts with game code.

Note, for brevity in all future Listings in this book (with the exception of **Logfile.h**), the **Dragonfly** namespace `df::` is not shown, but it does appear in the actual engine `.h` files.

Many managers depend upon each other, so the startup order of individual managers matters. For example, a logfile manager is often needed first since all the other managers write log messages upon startup, either noting successful startup in the logfile or reporting errors in the logfile when there are problems. Or, a display manager may need memory allocated for sprites, so a memory manager would need to be invoked before the display managers.

In addition, unlike many game objects, it often makes sense to have only one instance of each manager. For example, having two display managers simultaneously writing to the graphics card/display device may not make sense nor even be supported by the hardware/operating system. Similarly, having two independent managers handle input from the keyboard and mouse may yield undesirable (or at least unpredictable) results.

Managers are generally global in scope because the service the manager provides may be sought in many places in both game and engine code. For example, the engine code and game code may both write messages to the logfile via the LogManager, and both the engine code and game code may draw characters on the screen via the DisplayManager. Given this, a natural inclination may be to make the manager instances global variables, as depicted in Listing 4.2.

Listing 4.2: Managers declared as global variables

```
0 // Outside main(), these are global variables.
1 df::DisplayManager display_manager;
2 df::LogManager log_manager;
3
4 main() {
5     ...
6 }
```

While declaring managers as global variables does provide for global scope, it does not give control over the order of invocation. The order of instantiation of global variables is determined by the compiler and not by the program order. For example, if the code in Listing 4.2 is compiled and run, the `log_manager` may be instantiated first and then the `display_manager` or vice versa. Moreover, using global variables from the engine – say, if DisplayManager wanted to write to the logfile – would *require* the game programmer to use the same names as expected by the engine, making the game code more brittle.



### 4.2.1 Singletons

The *singleton* design pattern can be used for the game engine manager to solve all the above problems: 1) the singleton restricts instantiation of a class to one, and only one, object; 2) the singleton allows control of the order of manager initialization for dependency cases where the order matters; and 3) the singleton allows for global access.

In order to restrict instantiation to one (and only one) instance, the singleton class needs to disallow typical operations that enable object creation of its class. In particular, access must be restricted to the constructor, copy and assignment operators – otherwise, a programmer can use them to make additional instances of the class. Restricting creation is done by making the specific operations `private` to the class.

In order to instantiate a singleton class in C++, the keyword `static` is used as a modifier to the variable representing the one instance of the class. Remember, `static` variables retain their value even after the function terminates – in effect, the lifetime extends across the entire run of the program. However, a `static` variable is not allocated until the function is first called. This last feature allows explicit control as to when the manager is started up. Also remember, the keyword `static` in front of a method or function is quite different. A `static` method does not require an instance of the class to use it, and a `static` function (or a `static` global variable) has a scope that restricted to the `.cpp` file it is declared in.

Listing 4.3 depicts the class template for a singleton class.

Listing 4.3: A Singleton class

```

0 class Singleton {
1     private:
2         Singleton();                // No constructing.
3         Singleton(Singleton const &copy);    // No copying.
4         Singleton&(Singleton const &assign); // No assigning.
5     public:
6         static Singleton &getInstance();    // Return instance.
7 };
8
9 // Return the one and only instance of the class.
10 Singleton &Singleton::getInstance() {
11     // Note, a static variable persists after method ends.
12     static Singleton single;
13     return single;
14 }

```

While the singleton class guarantees there will be one and only one instance of the class, when a manager is actually instantiated (the first time `getInstance()` is called for that class), there can sometimes be a lot of work to be done. Thus, most of the initialization work for any manager is done in the `startUp()` method, called after the first `getInstance()` call.

Each specific manager (e.g., `DisplayManager`) class inherits from the base `Manager` class using the singleton template. The virtual methods `startUp()` and `shutDown()` are defined in the derived class and are specific to that particular manager. For example, the logfile manager might open the logfile for writing, while the display manager might ready the display for graphical output.



## 4.3 Logfile Management

If a game is working well, meaning all game engine code and game programmer code is doing what it is supposed to, all meaningful output typically goes to the screen in the form of game actions – characters moving, bullets flying, menus popping up, etc. However, during development this is often not the case, as code (even game engine code) can have bugs, or confirmation of working code is needed before proceeding. While debuggers are essential for effective programming, many game programmers do not have the luxury of having the source code for the game engine so a debugger cannot trace through the engine code. Moreover, some bugs are timing dependent meaning they only happen at full speed or are caused by a long sequence of events, making them hard to trace by a debugger.

What is helpful in these cases is a game engine that provides meaningful output as to the workings (or not) of the engine, and also provides a flexible, easy-to-use mechanism for a game programmer to get output. However, standard methods of printing to the screen can often interfere with the game itself or are not even possible when a display device is in graphics mode. In order to get around this limitation, logfiles are often used, where descriptive messages from the engine are written to a file, and the engine provides a flexible, easy-to-use mechanism for their own messages. This is the essence of the LogManager, often the first manager developed since all other engine components make use of it.

For base functionality, upon startup the LogManager opens up the logfile, making sure writing is allowed to the appropriate directory. Advanced features could allow appending or overwriting of previous logfiles, name the logfile with a timestamp, and check if there is sufficient disk space for normal operations. Upon shutdown, the logfile is closed, effectively flushing any outstanding data to the disk.

For attributes, the LogManager only needs a file structure (e.g., `FILE *`) for access to the logfile.

### 4.3.1 Variable Number of Arguments

The most frequently used LogManager method is to support writing general-purpose messages to the logfile – whether the messages come from other parts of the engine or from the game programmer – via a `writeLog()` method. For example, the game programmer may want to write a string such as “Player is moving”, which is effectively one argument to `writeLog()`. Or, the game programmer may want to write “Player is moving to (x, y)” where x and y are integer variables that are passed in. In other words, the number of arguments that `writeLog()` supports is not known ahead of time, but can be one or more.

A function that supports a variable number of arguments is depicted in Listing 4.4. Note the “...” characters in the parameter list for the function. Handling a variable number of arguments in this way requires `#including` the system header file `stdarg.h`, and the system header file `stdio.h` is needed for `fprintf()`. In the body of the function, a `va_list` structure is created, then initialized with arguments in the `va_start` command, provided with the name of the last known argument (`fmt`, in this case). At this point, the function is ready to call a `printf()` to produce output, but instead of a fixed string, the `va_list` structure has the formatting parameters, so `vfprintf()` is used instead. When finished, the `va_end()` must be called to clean up the stack.





Listing 4.4: Function taking variable number of arguments

```

0 #include <stdio.h>
1 #include <stdarg.h>
2
3 void writeMessage(const char *fmt, ...) {
4     fprintf(stderr, "Message: ");
5     va_list args;
6     va_start(args, fmt);
7     vfprintf(stderr, fmt, args);
8     va_end(args);
9 }

```

Note, Listing 4.4 uses standard error (`stderr`), which typically defaults to the console Window, while a game engine (e.g., `Dragonfly`) will usually write to a file. The code can be adjusted, accordingly.

### 4.3.2 Human-friendly Time Strings (optional)

For a long running game, it is often helpful to have timestamps associated with messages in the logfile. These times can be in “game time”, such as game loop iterations, or in “wall-clock time” corresponding to the actual time of the day. `Dragonfly` does the latter, displaying a human-friendly time in front of each message.

An easy way to associate a time with a written message is to have a function, say `getTimeString()`, that `writeLog()` calls to get a string with a timestamp. The `getTimeString()` method uses the `time()` system call, which returns the number of seconds since January 1, 1970. In order to turn that big number into something that is easier for humans to read, the `localtime()` system call converts the seconds into calendar time, allowing extraction of hours, minutes and seconds. Listing 4.5 depicts the `getTimeString()` method. Note, no error checking is provided for the system calls.<sup>1</sup> The function `sprintf()` on line 12 is similar to `printf()`, but instead of printing to `stdout`, `sprintf()` prints to a string,<sup>2</sup> in this case `time_str`.

Listing 4.5: Function to provide human-readable time string

```

0 // Return a nicely-formatted time string: HH:MM:SS
1 char *getTimeString() {
2
3     // String to return, made 'static' so persists.
4     static char time_str[30];
5
6     // System calls to get time.
7     time_t now;
8     time(&now);
9     struct tm *p_time = localtime(&now);
10
11     // '02' gives two digits, '%d' for integers.
12     sprintf(time_str, "%02d:%02d:%02d",
13         p_time -> tm_hour,
14         p_time -> tm_min,

```

<sup>1</sup>All system calls should be error-checked, and errors handled appropriately, in case they fail.

<sup>2</sup>The ‘s’ in front of `printf()` is for ‘string.’



```

15  p_time -> tm_sec);
16
17  return time_str;
18 }

```

A complementary message in the logfile is the “game clock” – the number of iterations of the game loop – obtained from the GameManager. This can be displayed as an integer pre-pended to the log message. See the GameManager in Section 4.4.4 on page 4.4.4 for details.

The presence of both the time string and the game clock in the logfile can setup to be controlled by the game programmer by having the LogManager keep two boolean variables, `log_time_string` and `log_step_count`, which, if `true`, have pre-pend the time string or game clock, respectively, to the game programmer’s log message.

While the printing of time has been presented in the context of the logfile, functions like `getTimeString()` are useful beyond the LogManager class and do not access any attributes of the class. As such, they should be placed in a file called `utility.cpp` (with a corresponding `utility.h`). Other functions that provide utility services, but are not part of any class definitions, will reside in `utility.cpp` as they are created.

### 4.3.3 Flushing Output

Generally, writing data to a file does not immediately write the data out to the disk. The operating system typically buffers data, writing when the device is idle or when internal memory buffers are filled. Such buffering generally improves overall system performance, but can cause unexpected output (or, more precisely, lack of it) if a program, say a game engine, crashes before all data is written. For example, if the line `log_manager.-writeLog("Doing stuff")` is executed and then the program crashes (e.g., from a segfault), the string “Doing stuff” may not appear in the logfile even though that line has been executed. This can make it hard to trace where, exactly, the error (in this case, the error that caused the segfault) might have occurred.

To force the operating system to immediately write out buffered data to the disk, the `fflush()` system call can be used after each write. Used during development, this helps provide complete logfiles even during system crashes. Note, this does decrease efficiency (speed) somewhat, so might not be used when game development (or game engine development) is complete. Thus, the LogManager can provide the game programmer with an option to flush the logfile after each disk, or not.

The attributes and methods for the LogManager can now be described in Listing 4.6. The destructor closes the file if `p_f` is not `NULL`.

Listing 4.6: LogManager attributes and methods

```

0 private:
1   bool do_flush;    // True if fflush after each write.
2   FILE *p_f;        // Pointer to logfile structure.
3
4 public:
5   // If logfile is open, close it.
6   ~LogManager();
7

```



```

8 // Start up the LogManager (open logfile "dragonfly.log").
9 int startUp();
10
11 // Shut down the LogManager (close logfile).
12 void shutDown();
13
14 // Set flush of logfile after each write.
15 void setFlush(bool do_flush=true);
16
17 // Write to logfile. Supports printf() formatting.
18 // Return number of bytes written, -1 if error.
19 int writeLog(const char *fmt, ...);

```

#### 4.3.4 Conditional Compilation

Once implemented, using the newly-minted LogManager throughout the engine (or in game code) will quickly reveal a problem – the LogManager header file, `LogManager.h`, likely gets included by the compiler pre-processor multiple times, resulting in compiler warnings about “redeclaration of class LogManager”. In order to fix this, directives to the pre-processor can limit class (and other) definitions to be included only once by having code only compiled during certain conditions. For the LogManager (and other `Dragonfly` header files), this is done by using an `#ifndef` wrapper and a unique identifier. Consider the sample code in Listing 4.7. When `foo.h` is seen by the compiler the first time, `FILE_FOO_SEEN` is not defined, so the pre-processor defines it in the next line and proceeds to parse and processes the foo file (and defining `class Foo`), normally. The next time `foo.h` is seen by the pre-processor, `FILE_FOO_SEEN` is already defined so the contents of the foo file are not included, avoiding a duplicate definition of `class Foo`.

Listing 4.7: Once-only header files

```

0 // File foo.h
1 #ifndef FILE_FOO_SEEN
2 #define FILE_FOO_SEEN
3
4 // The entire foo file appears next.
5 class Foo {};
6
7 #endif // !FILE_FOO_SEEN

```

Such conditional compilation directives are often used for platform-specific parts of code. Listing 4.8 shows a code stub that would compile Linux-specific code if `LINUX` was defined (say, with a `-DLINUX` flag to a `g++` compiler), or Windows-specific code if either `_WIN32` or `_WIN64` was defined.

Listing 4.8: Conditional compilation for platform-specific code

```

0 #if defined(_WIN32) || defined(_WIN64)
1
2 // Windows specific code here.
3
4 #elif LINUX
5

```



```

6 // Linux specific code here.
7
8 #endif

```

Note, there is no real functional difference between `#ifdef NAME` and `#if defined(NAME)`, but `#ifdef` can only use a single condition while `#if defined` can use compound conditions (as in the example in Listing 4.8).

When using `#define` directives in *Dragonfly* for literal replacement (e.g., for the engine version number), the convention is to prefix names with a `DF_` (e.g., `DF_VERSION`). This naming convention is to reduce the risk of potential namespace conflicts between the engine programmer and the game programmer.

When using conditional compilation for header files, the convention is for system utilities to use underscores before and after the name (e.g., `_STRING_H_`), while user code (game code) should never use initial/post underscores. This naming convention is to avoid potential namespace conflicts between the engine developer and the game programmer. For *Dragonfly*, a double initial underscore and double post underscore is used.

### 4.3.5 The LogManager

The complete header file for the LogManager is shown in Listing 4.9. Notice the `#ifndef` and `#define` statements at the top for conditional compilation.<sup>3</sup>

The `#include <stdio.h>` on line 6 is for the `FILE` variable, `p_f`. `LOGFILE_NAME` on line 13 provides the name of the logfile, “dragonfly.log”.

The methods in the `private` section that allow implementation of the singleton pattern. The attributes provide the file descriptor and whether or not to flush output after each write. Whether flushing is done or not is specified in the `setFlush()` method, but defaults to not flushing (`do_flush` is `false`).

The LogManager constructor should set the type of the Manager to “LogManager” (i.e., `setType("LogManager")`). As in most classes, the constructor should also initialize all attributes, in this case `p_f` to `NULL` and `do_flush` to `false`.

While `startUp()` and `shutDown()` are defined in the Manager class, they are redefined in the LogManager to open the logfile and close the logfile, respectively. Manager `startUp()` and Manager `shutDown()` should be called from LogManager `startUp()` and LogManager `shutDown()`, respectively. Remember, in C++ even if a method is defined in a derived class (e.g., `startUp()` in the LogManager), the parent method can still be called explicitly (e.g., `Manager::startUp()`).

Listing 4.9: LogManager.h

```

0 // The logfile manager.
1
2 #ifndef __LOG_MANAGER_H__
3 #define __LOG_MANAGER_H__
4
5 // System includes.
6 #include <stdio.h>
7

```

<sup>3</sup>For brevity, subsequent *Dragonfly* header files are not shown with `#ifndef` directives.



```

8 // Engine includes.
9 #include "Manager.h"
10
11 namespace df {
12
13 const std::string LOGFILE_NAME = "dragonfly.log";
14
15 class LogManager : public Manager {
16
17 private:
18     LogManager(); // Private since a singleton.
19     LogManager(LogManager const&); // Don't allow copy.
20     void operator=(LogManager const&); // Don't allow assignment.
21     bool m_do_flush; // True if flush to disk after each write.
22     FILE *m_p_f; // Pointer to logfile struct.
23
24 public:
25     // If logfile is open, close it.
26     ~LogManager();
27
28     // Get the one and only instance of the LogManager.
29     static LogManager &getInstance();
30
31     // Start up the LogManager (open logfile "dragonfly.log").
32     int startUp();
33
34     // Shut down the LogManager (close logfile).
35     void shutDown();
36
37     // Set flush of logfile after each write.
38     void setFlush(bool do_flush=true);
39
40     // Write to logfile. Supports printf() formatting of strings.
41     // Return number of bytes written, -1 if error.
42     int writeLog(const char *fmt, ...) const;
43 };
44
45 } // end of namespace df
46 #endif // _LOG_MANAGER_H_

```

Listing 4.10: Example of using the LogManager writeLog()

```

0 // Get singleton instance of LogManager.
1 df::LogManager &log_manager = df::LogManager::getInstance();
2
3 // Example call with 1 argument.
4 log_manager.writeLog(
5     "DisplayManager::startUp(): Current window set");
6
7 // Example call with 2 arguments.
8 log_manager.writeLog(
9     "WorldManager::isValid(): WorldManager does not handle '%s'",
10     event_name.c_str());
11
12 // Example call with 3 arguments.
13 log_manager.writeLog(

```



```

14  "DisplayManager::startUp(): max X is %d, max Y is %d",
15      max_x, max_y);
16
17  ...
18
19  // Sample logfile output:
20  LogManager started
21  DisplayManager::startUp(): Current window set
22  WorldManager::isValid(): WorldManager does not handle 'mud event'
23  DisplayManager::startUp(): max X is 80, max Y is 24

```

### 4.3.6 Controlling Verbosity (optional)

Logfile messages can be invaluable for debugging, performance tuning or verifying that engine code or game code is working properly. However, logfiles can also be “noisy,” with many, many innocuous lines of information hiding the ones that may offer true value. This is especially true of messages that are printed each step of the game loop (typically, 30 times per second), or for messages printed each step of a loop that iterates through all game objects!

While messages can be removed to decrease some of this noise, sometimes messages are useful during later debugging and take time to put back into place. So, instead of removing messages what is often better is to control the verbosity level of messages written to the log – with low verbosity, only essential messages are printed, while with high verbosity, all messages are printed. One way to do this is to have an explicit verbosity setting, depicted in Listing 4.11, via an attribute named `log_level`, with get/set methods, in the `LogManager`. Verbosity is controlled by changing the log level, even dynamically during run time, with logfile messages only written out when the `LogManager` log level is greater than or equal to that of the message.

Listing 4.11: Controlling verbosity with log level

```

0  // (attribute of LogManager)
1  private:
2      int log_level;           // Logging level.
3      ...
4
5  // (Modify writeLog to take log level as a parameter.)
6  // Write to logfile.
7  // Only write if indicated log level >= LogManager log level.
8  // Supports printf() formatting of strings.
9  // Return number of bytes written (excluding pre-pends), -1 if error.
10 void LogManager::writeLog(int log_level, char *fmt, ...)
11
12 // Only print message when verbosity level high enough.
13 if log_level > this.log_level then
14     va_list args
15     ...
16 end if

```

In order to avoid repeating code, the version of `writeLog()` without the `log_level` should just invoke the `writeLog()` in Listing 4.11, providing `INT_MAX`, the maximum integer



(found in `limits.h`), as the log level.

The method of controlling verbosity in Listing 4.11 is effective but does have a bit of additional overhead, notably a comparison check against the global variable holding the verbosity level for each call. It is not likely this overhead is onerous, but it can be significant, particularly for messages written each step and for each iteration of an object list.

An alternative method is to use conditional compilation, as described in Section 4.3.4 (page 57), with `#ifdef` directives used to decide whether or not to compile in logfile messages. An example is shown in Listing 4.12.

Listing 4.12: Controlling verbosity with conditional compilation

```
0 #ifdef DEBUG_1
1 LogManager &log_manager = LogManager::getInstance();
2 log_manager.writeLog("WorldManager::markForDelete(): deleting object %d",
3   p_o -> getId());
4 #endif
```

The first line indicates the following lines (that actually write the logfile message) are to be compiled in *only* if `DEBUG_1` is defined. The developer can define `DEBUG_1` when testing code, looking for bugs, verifying functionality and so on. When the game is ready to ship, `DEBUG_1` can be left undefined and the code is not compiled into the game. This approach has none of the overhead in Listing 4.11 when messages are not to be written since the messages to write to the logfile are not included at all. As a downside, code can be slightly less readable if there are many `#ifdef` messages.

### 4.3.7 Development Checkpoint #1!

At this point, development of `Dragonfly` should commence! The setup from Chapter 2, used for the tutorial, can be used to setup the development environment for your engine. Steps:

1. Setup your development environment, as specified in Chapter 2. Successfully compiling the first tutorial game in Section 3.3.1 on page 14 will ensure that the necessary tools are in place and configured.
2. Discard the pre-compiled `Dragonfly` libraries and header files from step 1 and prepare a directory structure for your own engine development.
3. Create a Manager base class, both `Manager.h` and `Manager.cpp`. See Listing 4.1 for details on the class definition.
4. Create a LogManager derived class, inheriting from the Manager class. Use Listing 4.9 as a reference.
5. Implement a `writeLog()` function, initially, not part of the LogManager. Have `writeLog()` just produce output to the screen (standard output). Test thoroughly, with no arguments, single arguments and multiple arguments. Be sure to test with different data types (`ints`, `floats`, etc.), too.



6. Move `writeLog()` into the `LogManager` class as a method. In `game.cpp`, have `#include "LogManager.h"` at the top of the file to include the class definition. Then, instantiate (via `getInstance()`) and start up (via `startUp()`) the `LogManager`. Check the return values to ensure the calls are working.
7. Test with various calls to `writeLog()` in the `LogManager` are working, testing single and multiple arguments with different types. Verify that the expected output appears in the logfile, "dragonfly.log".
8. Implement and test any optional elements (e.g., `getTimeString()`), as desired.

Make sure all the code is working thoroughly and is clearly written (indented and commented). As suggested earlier, the `LogManager` is used heavily during development, both for engine code and for game code, and needs to be robust and reliable before moving forward. This is true of each Development Checkpoint – make sure code is debugged and tested thoroughly before proceeding!





## 4.4 Game Management

At a high level, “managing the game” is the job description of the entire game engine. Game programmers (and players) often think of this as “running the game”.

### 4.4.1 The Game Loop

The game manager “runs” the game, doing so by repeating the same basic set of actions in a loop (the *game loop*), over and over. A 10,000 foot view of the game loop is presented in Listing 4.13. Each iteration of the game loop is called a “step” (or a “tick”, as in the tick of a loop clock). During one step, the game loop: 1) gets input, say from the keyboard or the mouse (these are player actions in the game); 2) updates the game world state to move objects around, generate needed actions, respond to the input; 3) draws a new image (the current scene) on the graphics buffer; and 4) swaps out the old image for the new image. This process is repeated in the loop until the game is over.

Listing 4.13: The game loop

```
0 while (game not over) do
1   Get input // e.g., from keyboard/mouse
2   Update game world state
3   Draw current scene to back buffer
4   Swap back buffer to current buffer
5 end while
```

Note that the loop in Listing 4.13 runs as fast as it can, updating and drawing the game world as fast as the computer can get through the code. Early game development efforts were often targeted for a machine with a specific speed, where the time to execute a loop was known and objects could be moved an appropriate amount of time each loop. Of course, running the same game code on a faster machine (as would happen when computer speeds improved) meant the game would run faster! Moreover, if a step took more or less time than expected, the update rate of game objects would vary, causing them to move faster or slower.

In order to rectify this problem, the game loop is enhanced with loop timing information, shown in Listing 4.14. In this version of the game loop, one step of the loop is expected to take `TARGET_TIME` (e.g., 33 milliseconds). So, the time to execute the first 4 instructions is carefully measured and, at the end of the loop on line 6, the game is put to sleep (effectively, doing nothing) for whatever is remaining of the `TARGET_TIME`.

Listing 4.14: The game loop with timing

```
0 while (game not over) do
1   Get input // e.g., keyboard/mouse
2   Update game world state
3   Draw current scene to back buffer
4   Swap back buffer to current buffer
5   Measure loop_time // i.e., how long above steps took
6   Sleep for (TARGET_TIME - loop_time)
7 end while
```



An important decision is deciding how long `TARGET_TIME` should be. Setting it too high will result in the game loop progressing slowly, limiting animation rates and game update rates – the game will look less “smooth” and will feel sluggish to the player. Setting it too low will result in the game loop progressing rapidly, giving a smooth, responsive game, but may unnecessarily burden the computer and cause problems, such as visual glitches or unintended game slowdowns, if the game world is too complicated to be fully updated in one step.

Guidelines for setting `TARGET_TIME` can be drawn from video. Video performance is often reported in units of frame rate, the rate at which video images are updated on the screen. The units are typically frames per second (f/s). “Full motion” video, the rate seen in movies or television, is approximately 30 f/s. Frame rates higher than this provides little benefit to visual quality, while frame rates below this look “jerky” for some kinds of video content. Considering the rendered game images as video images, the game loop rate is analogous to video frame rates, provided guidance on the game loop rates. Notably, a reasonable expectation is to update the game screen 30 times per second – equivalently, setting `TARGET_TIME` to 33 milliseconds.

#### 4.4.2 Measuring Computer Time

In order to step through the game loop 30 times per second, the time for one loop iteration must be measured precisely. Modern operating systems provide several different ways (system calls) to measure time. For example, on Unix systems, the `time()` call returns the number of seconds since January 1st, 1970. Subsequent system calls can use that number to extract the hours, minutes and seconds or even the month, day, year. However, the resolution of the `time()` system call is only 1 second, meaning it is too coarse to provide timing on the order of the milliseconds needed for the game loop.

Fortunately, modern computer processors have high-resolution timers provided by hardware registers that count processor cycles. For instance, a 3 GHz processor increments the timer register 3 billion times per second, providing a resolution of 0.3 nanoseconds – plenty of precision for the game loop! The actual system calls to access these high-resolution timers varies with platform. Windows uses `QueryPerformanceCounter()` to get the timer value, and `QueryPerformanceFrequency()` to get the processor cycle rate. Xbox 360 and PS3 game consoles use the `mftb` (which stands for “move from time base”) register to obtain the timer value, with the hardware having a known processor cycle time. Linux uses `clock_gettime()` to get a high-resolution time value (needing to be linked in with the real-time library, `-lrt`).

In order to measure the time the game loop takes (everything between line 1 “Get input” and line 4 “Swap” in Listing 4.14), the method in Listing 4.15 is used. The method starts by recording the time (storing it in a variable). Next, the tasks to be timed are run (for a game loop, this is input, update and so forth). When the tasks are done, the time is again recorded. The elapsed time is obtained by subtracting the before time from the after time.

Listing 4.15: Method to measure elapsed time

```
0 // Record before time
1 // Do processing stuff (get input, update ...)
2 // Record after time
```



```
3 // Compute elapsed time: after - before
```

For a Linux platform, Listing 4.16 provides a code fragment to compute the elapsed time of a block of computation. Note, in this example, the units for elapsed time are in microseconds, which is often used for timing in game engines (as are milliseconds), but it could easily be adjusted to seconds or milliseconds. The system header file `time.h` is needed for the timing routines and compilation needs `-lrt`, to link in the real-time library. The timing function, `clock_gettime()`, fills in the components of a `timespec` structure, which includes fields for seconds (`tv_sec`) and nanoseconds (`tv_nsec`). Computing elapsed time is done by converting the seconds and nanoseconds to microseconds, and subtracting the initial value from the final value.

Listing 4.16: Measuring elapsed time in Linux

```
0 // Compile with -lrt
1 #include <time.h>
2
3 struct timespec before_ts, after_ts;
4 clock_gettime(CLOCK_REALTIME, &before_ts); // Start timing.
5 // Do stuff...
6 clock_gettime(CLOCK_REALTIME, &after_ts); // Stop timing.
7
8 // Compute elapsed time in microseconds.
9 long int before_msec = before_ts.tv_sec*1000000 + before_ts.tv_nsec/1000;
10 long int after_msec = after_ts.tv_sec*1000000 + after_ts.tv_nsec/1000;
11 long int elapsed_time = after_msec - before_msec;
```

For Mac, the system call `clock_gettime()` does not exist (nor does the `rt` library). Instead, the system call `gettimeofday()` (located in `<sys/time.h>`) should be used, as shown in Listing 4.17. A call to `gettimeofday()` fills a `struct timeval` with the number of seconds and microseconds.

Listing 4.17: Measuring elapsed time in Mac OS

```
0 #include <sys/time.h>
1
2 struct timeval before_tv, after_tv;
3
4 gettimeofday(&before_tv, NULL); // Start timing.
5 // Do stuff...
6 gettimeofday(&after_tv, NULL); // Stop timing.
7
8 // Compute elapsed time in microseconds.
9 long int before_msec = before_tv.tv_sec*1000000 + before_tv.tv_usec;
10 long int after_msec = after_tv.tv_sec*1000000 + after_tv.tv_usec;
11 long int elapsed_time = after_msec - before_msec;
```

For Windows, the system call needed is `GetSystemTime()` (located in `<Windows.h>`) is used, as shown in Listing 4.18. A call to `GetSystemTime()` fills a `SYSTEMTIME` structure with the number of minutes, seconds, and milliseconds.

Listing 4.18: Measuring elapsed time in Windows

```
0 #include <Windows.h>
```



```

1
2 SYSTEMTIME before_st, after_st;
3 GetSystemTime(&before_st);
4 // Do stuff...
5 GetSystemTime(&after_st);
6
7 // Compute elapsed time in microseconds.
8 long int before_msec = (before_st.wMinute * 60 * 1000000)
9                       + (before_st.wSecond * 1000000)
10                      + (before_st.wMilliseconds * 1000);
11 long int after_msec = (after_st.wMinute * 60 * 1000000)
12                     + (after_st.wSecond * 1000000)
13                     + (after_st.wMilliseconds * 1000);
14 long int elapsed_time = after_msec - before_msec;

```

#### 4.4.3 The Clock Class

It is helpful for both the game engine and the game programmer to have a class that provides convenient access to high-resolution timing – the Clock class. Listing 4.19 provides the header file for the Clock class.<sup>4</sup> The clock functions as a sort of “stopwatch”, so the time is stored in the variable `previous_time`, set when a Clock object is instantiated to the current time. A call to the method `delta()` returns the elapsed time (in microseconds), and resets `previous_time` to the current time. A call to the method `split()` returns the time (in microseconds) since the last `delta()` call, but does not change the value of `previous_time`. The constructor should set `previous_time` to the current time, and both `delta()` and `split()` can be implemented using Listing 4.16, 4.17, or 4.18 (as appropriate), as a reference.

Listing 4.19: Clock.h

```

0 // The clock, for timing (such as in the game loop)
1
2 class Clock {
3
4     private:
5         long int m_previous_time; // Previous time delta() called (in microsec).
6
7     public:
8         // Sets previous_time to current time.
9         Clock();
10
11         // Return time elapsed since delta() was last called, -1 if error.
12         // Resets clock time.
13         // Units are microseconds.
14         long int delta();
15
16         // Return time elapsed since delta() was last called, -1 if error.
17         // Does not reset clock time.
18         // Units are microseconds.
19         long int split() const;
20 };

```

<sup>4</sup>Note, the conditional `#ifdef` directives described in Section 4.3.4 are not shown.



With a Clock class for timing, the last missing piece for providing timing control in the game loop is the ability to sleep (line 6 of Listing 4.14). Linux and Mac provide the `sleep()` system call, but it has only seconds of resolution, meaning it will not allow the game engine to sleep for, say, 20 milliseconds. Since game loop timing needs milliseconds of resolution, so does an appropriate sleep call.

On Linux and Mac, this can be done with `nanosleep()` which sleeps for a given number of nanoseconds.<sup>5</sup> A `#include <time.h>` is needed for `nanosleep()`. The system call `nanosleep()` takes in a pointer to a `struct timespec`, that has the amount of seconds plus nanoseconds to sleep. The example in Listing 4.20 shows a call to `nanosleep()` for 20 milliseconds.

Listing 4.20: `nanosleep()` example for Linux and Mac

```
0 // Sleep for 20 milliseconds.
1 struct timespec sleep_time;
2 sleep_time.tv_sec = 0;
3 sleep_time.tv_nsec = 20000000;
4 nanosleep(&sleep_time, NULL);
```

On Windows, sleeping can be done with `Sleep()` which sleeps for a given number of milliseconds. A `#include <Windows.h>` is needed for `Sleep()`. In order to obtain a millisecond resolution using `Sleep()`, the system call `timeBeginPeriod(1)` needs to be called once, when the game engine starts, to set the timer resolution to the minimum possible. The system call `timeEndPeriod(1)` is called once the game engine exits to clear the initial request for a minimal timer resolution. Both functions return `TIMERR_NOERROR` if successful or `TIMERR_NOCANDO` if the resolution specified is out of range. Note, the best places for these calls are when the GameManager starts up and when the GameManager shuts down, respectively (see Section 4.4.4 on page 69).

Listing 4.21: `Sleep()` example for Windows

```
0 // Sleep for 20 milliseconds.
1 int sleep_time = 20;
2 Sleep(sleep_time);
```

Listing 4.22 provides pseudo-code for how the Clock class and sleep functions can be used together in the game loop. The call to `clock.delta()` at the beginning of the loop starts the timing, while the call to `clock.split()` after most of the loop body provides the elapsed time, measuring how long the game loop took. The game engine then sleeps (via `nanosleep()` for Linux or Mac or `Sleep()` for Windows) for `TARGET.TIME - loop.time`.

Listing 4.22: The game loop with Clock

```
0 Clock clock
1 while (game not over) do
2     clock.delta()
3     // Get input (e.g., from keyboard/mouse)
4     // Update world state
5     // Draw current scene to back buffer
6     // Swap back buffer to current buffer
```

<sup>5</sup>There are 1 billion nanoseconds in 1 second.



```

7  loop_time = clock.split()
8  sleep(TARGET_TIME - loop_time)
9  end while

```

The expectation is that `(TARGET_TIME - loop_time)` is positive, since the `sleep()` call on line 8 of Listing 4.22 expects positive number. But what happens when it is not? First off, consider what it means for `(TARGET_TIME - loop_time)` to be negative. This happens when the time to do the work in the game loop (the input, update, draw and swap) takes longer than the expected time for one iteration of the game loop (longer than `TARGET_TIME`). When this happens, the game engine cannot keep up with the work required to run the game, resulting, at a minimum, in the displayed frame rate that the player sees to decrease. For example, if the `TARGET_TIME` is 33 milliseconds, providing a frame rate of 30 f/s, but the loop time (`loop_time`) takes 50 milliseconds, the frame rate is only be 20 f/s. With longer loop times, the frame rate drops further, decreasing the smoothness of the visual display for the player. The time between getting input from the player also decreases, probably making the game feel less responsive.

If the `loop_time` is greater than the `TARGET_TIME`, do the game objects themselves need to slow down also? Not necessarily. When updating the game world, the engine can be aware of the previous update time, thus knowing how much time has elapsed, and use this to decide how far, say, an object should move. This requires the game engine to pass along timing information to update functions and for those functions to use the information accordingly. However, in *Dragonfly*, the engine does not do this, so if the computer cannot keep up at the expected `TARGET_TIME` pace, the game will look, feel and run slower. Thus, as for programming all games using a game engine, *Dragonfly* game programmers must work within the constraints of the engine to ensure the load their game places on the engine does not cause performance issues.

#### 4.4.3.1 Fine Tuning the Game Loop (optional)

A subtle timing aspect that is important for some games is that when calling operating system sleep functions (e.g., `nanosleep()` or `Sleep()`), the actual amount of sleep time may be longer than requested depending upon other activity in the system and the operating system scheduler. In most cases, this does not matter much, since sleep differences are typically being only a matter of a few milliseconds. However, in some cases, such as trying to synchronize game state on two different machines in a multi-player game, more precision in the total time a game loop takes is required. So, a final adjustment to the loop timing can be made by determining how long the sleep function call actually took. Measurement can be done before and after the sleep call, with any extra time subtracted from the next game loop. Listing 4.23 shows how to put in this adjustment.

Listing 4.23: The game loop with Clock and sleep adjustment

```

0  Clock clock
1  while (game not over) do
2    clock.delta()
3    // Get input (e.g., from keyboard/mouse)
4    // Update world state
5    // Draw current scene to back buffer

```



```

6 // Swap back buffer to current buffer
7 loop_time = clock.split()
8 intended_sleep_time = TARGET_TIME - loop_time - adjust_time
9 clock.delta()
10 sleep(intended_sleep_time)
11 actual_sleep_time = clock.split()
12 adjust_time = actual_sleep_time - intended_sleep_time
13 end while

```

#### 4.4.4 The GameManager

With timing technologies developed for the game loop, implementation of the GameManager can now be started with the class definition provided in Listing 4.24.

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

The method `setGameOver()` lets the game programmer set the game over condition when ready (e.g., the player has indicated they want to quit) and `getGameOver()` returns the game over status.

The `run()` method is used to start the game, effectively running the game loop until the game is over, controlled by the boolean attribute `game_over`.

The GameManager needs start up methods, like all engine managers. The GameManager `startUp()` method instantiates (via `getInstance()`) and starts up (via `startUp()`) all the other game managers and in the right order. For now, the GameManager only starts up the LogManager. The `game_over` variable should be set to `false`. Most games typically use the default of 33 milliseconds (line 3), but games that want to run faster or slower may want to use an alternate frame time.\* If developing for Windows, GameManager `startUp()` should invoke `timeBeginPeriod(1)` (see page 67).

The `shutDown()` method does the reverse, shutting down the LogManager. It calls `setGameOver()` to indicate to any game objects that the game is over, which sets the `game_over` variable to `true`. If developing for Windows, GameManager `shutDown()` should invoke `timeEndPeriod(1)` (see page 67).

Upon success, Manager `startUp()` and Manager `shutDown()` should be called from GameManager `startUp()` and GameManager `shutDown()`, respectively.

Listing 4.24: GameManager.h

```

0 #include "Manager.h"
1
2 // Default frame time (game loop time) in milliseconds (33 ms == 30 f/s).
3 const int FRAME_TIME_DEFAULT = 33;
4
5 class GameManager : public Manager {
6
7 private:
8     GameManager(); // Private since a singleton.

```

---

\* Did you know (#2)? Large dragonflies have an average cruising speed of about 10 mph, with a maximum speed of about 30 mph. – “Frequently Asked Questions about Dragonflies”, *British Dragonfly Society*, 2013.



```

9  GameManager (GameManager const&); // Don't allow copy.
10 void operator=(GameManager const&); // Don't allow assignment.
11 bool game_over; // True, then game loop should stop.
12 int frame_time; // Target time per game loop, in milliseconds.
13
14 public:
15 // Get the singleton instance of the GameManager.
16 static GameManager &getInstance();
17
18 // Startup all GameManager services.
19 int startUp();
20
21 // Shut down GameManager services.
22 void shutDown();
23
24 // Run game loop.
25 void run();
26
27 // Set game over status to indicated value.
28 // If true (default), will stop game loop.
29 void setGameOver(bool new_game_over=true);
30
31 // Get game over status.
32 bool getGameOver() const;
33
34 // Return frame time.
35 // Frame time is target time for game loop, in milliseconds.
36 int getFrameTime() const;
37 };

```

#### 4.4.5 Development Checkpoint #2!

If you have not kept up already, **Dragonfly** development should continue! Steps:

1. Create the Clock class. Create a **Clock.h** header file based on Listing 4.19. Add **Clock.cpp** to the project and stub out each method so it compiles.
2. Implement and test, using a simple program that creates a Clock object, waits for awhile (use an appropriate sleep call), and calls **split()** and/or **delta()**. Verify the times meet expectations. A robust LogManager (developed during Development Checkpoint 4.3.7) can be used for output.
3. Create the GameManager class. Create a **GameManager.h** header file based on Listing 4.24. Add **GameManager.cpp** and stub out each method so it compiles.
4. In the **GameManager.cpp** file, have the **startUp()** method start the LogManager, and the **shutDown()** method stop the LogManager and call **setGameOver()**. Test that **startUp()** and **shutDown()** work as expected before proceeding.
5. Implement the game loop inside the GameManager **run()** method. The body of the loop does not do anything yet (although you can add some “dummy” statements), but the loop should time (via **delta()** and **split()**) and sleep properly. Be sure to





double-check any conversions of units (e.g., milliseconds to microseconds) used. The game loop uses a Clock object. Test thoroughly by timing (with a clock on the wall) that you get the expected number of loop iterations.

6. Add additional functionality to the GameManager, as desired. The frame time option to `startUp()` can be useful.

Since code developed during this Development Checkpoint drives the entire game, it should be tested thoroughly, making sure it is robust and clearly written before proceeding.



## 4.5 The Game World

The game world itself is full of objects: bad guys running around; walls that enclose buildings and spaces; trees, rocks and other obstacles; and the hero, rushing to save the day. The exact type of objects depends upon the type of game, of course, but in nearly all games, the game engine has many objects to manage (game objects are introduced in Section 4.5.1 on page 72).

The game world needs to store and access groups of objects. In addition, the game programmer needs to access game objects in order to make them react to input or perform game-specific functions. So, the game world needs to manage them efficiently and present them in a convenient fashion. The game programmer might want a list of all the solid objects, a list of all the objects within the radius of an explosion, or a list of all the Saucer objects. It is the job of the world manager to store the game objects and provide these lists in response to game programmer queries. Section 4.5.2 on page 76 introduces lists of game objects.

The game world not only stores and provides access to the game objects, it also needs to update them, moving them around, see if they collide, and more. Section 4.5.4 (page 85) introduces methods to update game objects, with Section 4.5.5 (page 86) providing details on events, of vital importance for understanding how a game engine connects to game programmer code.

### 4.5.1 Game Objects

Game objects are a fundamental game programmer abstraction for items in the game. For example, consider Saucer Shoot from Section 3.3. As in many games, there are opponents to defeat (i.e., Saucers), the player character (i.e., the Hero), projectiles that can be launched (i.e., Bullets), and other game objects (e.g., Explosions, Points, etc.). Other games may have obstacles or boundaries to bypass (e.g., walls, doors), items that can be picked up (e.g., gold coins, health packs), and even other characters to interact with (e.g., non-player characters). The game engine needs to access all these objects, for example, to get an object's position. The game engine also needs to update these objects, for example, to change the location as the object moves. Thus, a core attribute for a game object, and the first one used in *Dragonfly*, is the object's position, stored as a 2d vector.

#### 4.5.1.1 The Vector Class

The Vector class represents a 2d vector. When used for a position, the Vector is sufficient to hold a location in the game world. Some future version of *Dragonfly* could provide a third dimension, z, and/or provide coordinates as floating point numbers. The header file for the Vector class is described in Listing 4.25. Vector mostly holds the attributes *x* and *y*, with methods to get and set them. In addition to the default constructor (which set *x* and *y* to 0), on line 9, Vector has a constructor that sets *x* and *y* to initial values.

Listing 4.25: Vector.h

```
0 class Vector {
1
```



```

2 private:
3     float m_x; // Horizontal component.
4     float m_y; // Vertical component.
5
6 public:
7
8     // Create Vector with (x,y).
9     Vector(float init_x, float init_y);
10
11    // Default 2d (x,y) is (0,0).
12    Vector();
13
14    // Get/set horizontal component.
15    void setX(float new_x);
16    float getX() const;
17
18    // Get/set vertical component.
19    void setY(float new_y);
20    float getY() const;
21
22    // Set horizontal & vertical components.
23    void setXY(float new_x, float new_y);
24
25    // Return magnitude of vector.
26    float getMagnitude() const;
27
28    // Normalize vector.
29    void normalize();
30
31    // Scale vector.
32    void scale(float s);
33
34    // Add two Vectors, return new Vector.
35    Vector operator+(const Vector &other) const;
36 };

```

To make a Vector more generally useful, methods and operators on lines 26 to 35 are provided.

Vector `getMagnitude()`, shown in Listing 4.26, returns the magnitude (size) of the vector.

Listing 4.26: Vector `getMagnitude()`

```

0 // Return magnitude of vector.
1 void Vector::getMagnitude()
2     float mag = sqrt(x*x + y*y)
3     return mag

```

Vector `scale()`, shown in Listing 4.27, resizes (changes the magnitude) of the vector by the scale factor, leaving the direction the vector is pointing in the same.

Listing 4.27: Vector `scale()`

```

0 // Scale vector.
1 void Vector::scale(float s)
2     x = x * s

```



```
3 y = y * s
```

Vector `normalize()`, shown in Listing 4.28, takes a vector of any length and, keeping it pointing in the same direction, changes its length to 1 (also called a unit vector). The `if` check is to avoid a possible division by zero.

Listing 4.28: Vector `normalize()`

```
0 // Normalize vector.
1 void Vector::normalize()
2     length = getMagnitude()
3     if length > 0 then
4         x = x / length
5         y = y / length
6     end if
```

Overloading the addition operator (+ in `v1 + v2`) for an `Vector` is shown in Listing 4.29. The method is called on the first vector (the `Vector` on the left-hand side of the '+'), with the second vector provided as an argument (the `Vector` on the right-hand side of the '+'). The variable `v` holds the new `Vector`, with `x` and `y` values added from the components of the other two vectors and then returned.

Listing 4.29: Vector operator+

```
0 // Add two Vectors, return new Vector.
1 Vector Vector::operator+(const Vector &other) const {
2     Vector v // Create new vector.
3     v.x = x + other.x // Add x components.
4     v.y = y + other.y // Add y components.
5     return v // Return new vector.
```

**Other Vector operators (optional)** The addition operator (+) is core since adding two vectors is used for many operations. However, there are other operators that may be useful for a general `Vector` class, including: subtraction (-), multiplication (\*), division (/), comparison (== and !=) and not (!). The aspiring programmer may want to implement them, using Listing 4.29 as a reference.

#### 4.5.1.2 The Object Class

With the `Vector` class in place, the `Object` class can now be specified. The `Object` class definition is provided in Listing 4.30.

Listing 4.30: `Object.h`

```
0 // System includes.
1 #include <string>
2
3 // Engine includes.
4 #include "Vector.h"
5
6 class Object {
7
8     private:
```



```

9   int m_id;           // Unique game engine defined identifier.
10  std::string m_type; // Game-programmer defined identification.
11  Vector m_position;  // Position in game world.
12
13 public:
14  // Construct Object. Set default parameters and
15  // add to game world (WorldManager).
16  Object();
17
18  // Destroy Object.
19  // Remove from game world (WorldManager).
20  virtual ~Object();
21
22  // Set Object id.
23  void setId(int new_id);
24
25  // Get Object id.
26  int getId() const;
27
28  // Set type identifier of Object.
29  void setType(std::string new_type);
30
31  // Get type identifier of Object.
32  std::string getType() const;
33
34  // Set position of Object.
35  void setPosition(Vector new_pos);
36
37  // Get position of Object.
38  Vector getPosition() const;
39 };

```

Each Object has a unique id, initially set in the constructor ( `Object()` ), that may be of use in some games to uniquely identify an game object.<sup>6</sup> The id is obtained from a `static` integer that starts at 0 and is incremented each time `Object()` is called. The method `getId()` can be used to obtain an Object's id. The method `setId()` can be used to set an id manually, but in most cases this is never used.<sup>7</sup>

The type is a string primarily used to identify the Object in game code. For example, a Bullet object can `setType("Bullet")`, allowing a Saucer object to query a collision event to see whether or not the type was a “bullet”, destroying oneself as an action. For the base Object constructor, the type can just be set to `"Object"`.

The Object `setPosition()` and `getPosition()` methods should allow changing the attribute `position` (via set) and retrieving it (via get).

Objects will have many attributes eventually (such as altitude, solidness, bounding boxes for collisions, sprites for animating, ...) but for now merely keeping the position of the game world is sufficient.

Note that the destructor for Object is `virtual` on line 20. This is necessary since

<sup>6</sup>In most cases, the game programmer can uniquely identify an object by its memory address, but an integer may still be more convenient. Moreover, a network game cannot count on a game object that is replicated on another computer to have the same memory address.

<sup>7</sup>A common exception is for synchronizing game worlds in a networked computer game.



Objects are deleted by the engine, not the game programmer, and the `virtual` keyword makes sure the right destructor is called. If the destructor was not `virtual`, when an Object was deleted by the engine, only `~Object()` would be invoked and not the destructor for a game programmer object that inherited from it.

### 4.5.2 Lists of Objects

In order to handle the management and presentation of objects to the game programmer, the world manager needs a data structure that supports lists of objects, lists that can be created and passed around (inside the engine and to the game programmer) in a convenient to use and efficient to handle manner. In passing the lists to the game programmer, if the Objects themselves are changed (e.g., say, by decreasing the hit points of Objects damaged in an explosion), updates need to happen to the “real” objects as seen by the world manager. While there are numerous libraries that could be used for building efficient lists (e.g., the *Standard Template Library* or the *Boost C++ Library*), list performance is fundamental to game engine performance, both for *Dragonfly* as well as for other game engines, so implementing game object lists provides an in-depth understanding of game engine performance. Plus, there are additional programming skills to be gained by implementing lists from scratch.

There are different implementation choices possible for lists of game objects, including linked lists, arrays, hash tables, trees and more. For ease of implementation *and* performance efficiency, an array is used for lists of game objects in *Dragonfly*. In addition, the *iterator* design pattern is used to provide a way to access the list without exposing the underlying data representation. In general, not having iteration as part of the container class separates the functionality for the collection from the functionality for iteration. This simplifies the collection (not having it cluttered with iteration methods), allows several iterations to be active at the same time, and decouples collection algorithms from collection data structures, while still leaving the details of the container implementation encapsulated. This last point means the internals of the list data structure can be changed (e.g., replace the array with a linked list) without changing the rest of the game engine or any dependent game programmer code.

For reference, consider a basic list of integers (`ints`), shown in Listing 4.31. The list starts out empty, by setting the count of items to 0 in the constructor. Basic operations allow the programmer to `insert()` items, `remove()` items, and `clear()` the entire list. Clearing the list and adding items to the list are quite efficient. Removing items from the list is rather inefficient, requiring the entire list to be traversed each time. However, perhaps most importantly, copying the list, which in a game engine happens often as many lists are created and destroyed by both the engine and the game programmer, is efficient as compilers (and programmers) handle fixed sized chunks of memory efficiently.

Listing 4.31: List of integers implemented as an array

```
0 const int MAX = 100;
1
2 class IntList {
3
4 private:
5     int list[MAX];
```



```

6   int count;
7
8   public:
9   IntList() {
10      count = 0;
11   }
12
13   // Clear list.
14   void clear() {
15      count = 0;
16   }
17
18   // Add item to list.
19   bool insert(int x) {
20      if (count == MAX) // Check if room.
21         return false;
22      list[count] = x;
23      count++;
24      return true;
25   }
26
27   // Remove item from list.
28   bool remove(int x) {
29      for (int i=0; i<count; i++) {
30         if (list[i] == x) { // Found...
31            for (int j=i; j<count-1; j++) // ...so scoot over
32               list[j] = list[j+1];
33            count--;
34            return true; // Found.
35         }
36      }
37      return false; // Not found.
38   }
39 };

```

While arrays are efficient, it is still not a good design for the game engine to have entire objects inside the list. In other words, the `int` in Listing 4.31 should *not* be replaced by an `Object` representing a game object. Instead, lists of game objects are handled by having *pointers* to the game objects. So, `int` is replaced with `Object *`. Using pointers allows the game engine to reference the game object's attributes and methods, but is much more efficient for creating the many needed lists of objects needed. Basically, copying a list of pointers is much faster (and uses less memory) than copying a list of game objects. In addition, lists passed from the engine to, say, the game code refer to the original Objects (via the pointers) and not copies. Last but not least, having Object pointers allows for polymorphism, as in Listing 1.1 on page 7, when Object methods are resolved.

Listing 4.32: ObjectList.h

```

0   const int MAX_OBJECTS = 5000;
1
2   #include "Object.h"
3   #include "ObjectListIterator.h"
4
5   class ObjectListIterator;

```



```

6
7 class ObjectList {
8
9 private:
10     int m_count;           // Count of objects in list.
11     Object *m_p_obj[MAX_OBJECTS]; // Array of pointers to objects.
12
13 public:
14     friend class ObjectListIterator;
15
16     // Default constructor.
17     ObjectList();
18
19     // Insert object pointer in list.
20     // Return 0 if ok, else -1.
21     int insert(Object *p_o);
22
23     // Remove object pointer from list.
24     // Return 0 if found, else -1.
25     int remove(Object *p_o);
26
27     // Clear list (setting count to 0).
28     void clear();
29
30     // Return count of number of objects in list.
31     int getCount() const;
32
33     // Return true if list is empty, else false.
34     bool isEmpty() const;
35
36     // Return true if list is full, else false.
37     bool isFull() const;
38 };

```

Notice that line 14 of Listing 4.32 refers to an “ObjectListIterator” class that has not been defined. This class is defined in Section 4.5.2.1 and is used for efficient traversal of the list without exposing the attributes and internal structure of the list publicly. Line 5 is needed to act as a forward reference for the compiler, allowing compilation to proceed, as long as the ObjectListIterator class is defined before linking. Make sure this is also put inside the `df::` namespace (see page 51).

#### 4.5.2.1 ObjectList Iterators

The ObjectList class, as described in Section 4.5.2, is a fine container class, but does not allow traversal of the items ( Objects) in the list. Fortunately, this can be rectified by defining an *iterator* for the ObjectList class. In general, iterators “know” how to traverse through a container class without exposing the internal data methods and structure publicly. Iterators do this by being a **friend** of the container class, giving them access to the private and protected attributes of the class. Defining an iterator decouples traversing the container from the container iteration. This allows, for instance, the structure of the container to be changed (e.g., from a static array to a linked list) without redefining all the code that uses the container.





There can be more than one iterator for a given list instance, each keeping its own position. Note, however, that adding or deleting items to a list during iteration may cause the iterator to skip or repeat iteration of an item (not necessarily the one added) – the program should not crash, but the iteration may not touch each item once and only once.

There are 3 primary steps in coding an iterator for a container:

1. Understand container class (e.g., List)
2. Design iterator class for container class
3. Add iterator materials:
  - Add iterator as friend class of container class

To illustrate these steps, consider creating a `IntListIterator` for the `List` defined in Listing 4.31 on page 76. Step 1 is to understand the implementation of `List`, in terms of the attributes `p_obj[]` array and the `count` used to store and keep track of list members. Step 2 is to define an iterator for `IntList`, provided by `IntListIterator` in Listing 4.33. The constructor for `IntListIterator` (line 7) needs a pointer to the `IntList` object it will iterate over, which it stores in attribute `p_list`. Since the iterator does not change the contents of the list, this pointer is declared as `const`. The `index` attribute is used to keep track of where the iterator resides in the list during iteration. The `first()` method resets the iterator to the beginning of the list. Subsequently, `next()` and `isDone()` allow iteration until the end of the list. The method `currentItem()` returns the current item that the iterator is on. Note, although not shown for brevity, `index` should be error checked for bounds in `currentItem()` and `next()`.

Listing 4.33: Iterator for `IntList` class

```

0 class IntListIterator {
1
2 private:
3     const IntList *p_list; // Pointer to IntList iterating over.
4     int index;             // Index of current item.
5
6 public:
7     IntListIterator(const IntList *p_l) {
8         p_list = p_l;
9         first();
10    }
11
12    // Set iterator to first item.
13    void first() {
14        index = 0;
15    }
16
17    // Iterate to next item.
18    void next() {
19        if (index < p_list->count)
20            index++;
21    }
22

```



```

23 // Return true if done iterating, else false.
24 bool isDone() {
25     return (index == p_list -> count);
26 }
27
28 // Return current item.
29 int currentItem() {
30     return p_list -> item[index];
31 }
32 };

```

For step 3, in order for the `IntListIterator` to access the private member of the `IntList` class, namely the `item[]` array and the list `count`, `IntListIterator` must be declared as a `friend` class inside `IntList.h`.

```

0 friend class IntListIterator;
1 ...

```

Both `IntList.h` and `IntListIterator.h` need forward references to class `IntListIterator` and class `IntList`, respectively.

Once the `IntListIterator` is defined, a programmer that wants to iterate over an instance of `List`, say `my_list`, first creates an iterator:

```

0 IntListIterator li(&my_list);

```

Then, the programmer calls `first()`, `currentItem()`, and `next()` until `isDone()` returns true.

Listing 4.34: Iterator with `while()` loop

```

0 li.first();
1 while (!li.isDone()) {
2     int item = li.currentItem();
3     li.next();
4 }

```

A `for` loop provides for a bit shorter syntax:

Listing 4.35: Iterator with `for()` loop

```

0 for (li.first(); !li.isDone(); li.next())
1     int item = li.currentItem();

```

For `Dragonfly`, the complete header file for an `ObjectListIterator` is defined in Listing 4.36. Having the default constructor `private` on line 8 makes it explicit that a `List` must be provided to the iterator when created.

Listing 4.36: `ObjectListIterator.h`

```

0 #include "Object.h"
1 #include "ObjectList.h"
2
3 class ObjectList;
4
5 class ObjectListIterator {
6

```



```

7 private:
8   ObjectListIterator();           // Must be given list when created.
9   int m_index;                   // Index into list.
10  const ObjectList *m_p_list;     // List iterating over.
11
12 public:
13   // Create iterator, over indicated list.
14   ObjectListIterator(const ObjectList *p_l);
15
16   void first();                   // Set iterator to first item in list.
17   void next();                   // Set iterator to next item in list.
18   bool isDone() const;           // Return true if at end of list.
19
20   // Return pointer to current Object, NULL if done/empty.
21   Object *currentObject() const;
22 };

```

#### 4.5.2.2 Overloading + for ObjectList (optional)

A useful abstraction for game programmers is to combine two ObjectLists, the result being a third, combined list holding all the elements of the first list and all the elements of the second list. A method named `add()` could combine two lists, written as part of the ObjectList class (e.g., `ObjectList::add()`) or as a stand alone function. However, a natural abstraction is to use the addition ('+') operator, overloading it to combine ObjectLists in the expected way.

Operators are just functions, albeit special functions that perform operations on objects without directly calling the objects' methods each time. Unary operators act on a single piece of data (e.g., `myInt++`), while binary operators operate on two pieces of data (e.g., `newInt = myInt1 + myInt2`). For ObjectLists, overloading the binary addition '+' operator is helpful. The syntax for overloading an operator is the same as for declaring a method, except the keyword `operator` is used before the operator itself.

Overloading the addition operator for an ObjectList is shown in Listing 4.37. The method is called on the first list (the ObjectList on the left-hand side of the '+'), with the second list provided as an argument (the ObjectList on the right-hand side of the '+'). The variable `big_list` holds the combined list, starting out by copying the contents of the first list (`*this`) on line 4). The method then proceeds to iterate through the second list, inserting each element from the second list into the first list on line 10. Once finished iterating over all elements in the second list, the method returns the combined list `big_list` on line 14.

Listing 4.37: ObjectList operator+

```

0 // Add two lists, second appended to first.
1 ObjectList ObjectList::operator+(ObjectList list)
2
3 // Start with first list.
4 ObjectList big_list = *this
5
6 // Iterate through second list, adding each element.
7 ObjectListIterator li(&list)

```



```

8   for (li.first(); not li.isDone(); li.next())
9       Object *p_o = li.currentObject()
10      big_list.insert(p_o) // Add element from second, to first list
11  end for
12
13  // Return combined list.
14  return big_list

```

Since ObjectLists are implemented as arrays, a more efficient ‘+’ operation could allocate one array of memory large enough for both lists, then, using `memcpy()` or something similar, copy the first list then the second list into the allocated memory. Care must be taken to get the pointers and memory block length correct. That is left as option for the reader to explore outside this text.

However, as an advantage, the implementation in Listing 4.37 is agnostic of the actual implementation of ObjectList. The lists could be implemented as either arrays or linked lists with pointers or some other internal structure and the code would still work.

Once defined, the ObjectList ‘+’ operator can be called explicitly, such as:

```

0  ObjectList list_1, list_2;
1  ObjectList list_both = ObjectList+(list_1, list_2);

```

but a more natural representation is to call it as intended:

```

0  ObjectList list_1, list_2;
1  ObjectList list_both = list_1 + list_2;

```

### 4.5.2.3 Dynamically-sized Lists of Objects (optional)

A significant potential downside of the code shown in Listing 4.31 and Listing 4.32 is that the maximum size of the list needs to be specified at compile time. If the list grows larger than this maximum, items cannot be added to the list – the container class data structure cannot do anything besides return an error code. This is true even when there is memory available on the computer to store more list items. A full list is potentially problematic – for example, the world manager can no longer manage any more Ogres or the player cannot put more Oranges into a backpack. Specifying a larger maximum size and then recompiling the game engine and the game is hardly an option for most players!

What can be done instead is to make arrays that dynamically resize themselves to be larger as more items are required to be stored in the list. This has two tremendous advantages: 1) the maximum size of the list does not need to be known by the engine ahead of time, and 2) game object lists do not have to all be as large as the potential maximum size, but can instead be small when a small list is required and only become large when a large list is required, thus saving runtime memory and runtime processing time when lists are copied and returned. The downside of this approach is that more runtime overhead can be incurred when a list grows. If done right, however, this runtime overhead can be infrequent and fairly small.

The basic idea of dynamically sized lists is to allocate a relatively small array to start. Then, if the array gets full (via `insert()`), the memory is re-allocated to make the array larger. In order to avoid having the re-allocation happen every time a new item is inserted,



the re-allocation is for a large chunk of memory. A good guideline for the size of the larger chunk is twice the size of the list that is currently allocated.

In order to make this change, first, the `ObjectList` attribute for the list needs to be changed from an array to a list of pointers, such as `Object **p_list`.<sup>8</sup>

Next, the `ObjectList` constructor needs to allocate memory for the list dynamically. This can certainly be done via `new`, but memory can be efficiently resized using C's `realloc()`.<sup>9</sup> The initial allocation uses `malloc()` to create a list of size `MAX_COUNT_INIT`. `MAX_COUNT_INIT` is defined to be 1, but other sizes can certainly be chosen. The `ObjectList` destructor should `free()` up memory, if it is allocated.

Listing 4.38: Re-declaring list to be dynamic array label

```
0 max_count = MAX_COUNT_INIT; // initial list size (e.g., 1)
1 p_item = (Object **) malloc(sizeof(Object *));
```

In the `insert()` method, if the list is full (`isFull()` returns `true`) then the item array is re-allocated to be twice as large, shown in Listing 4.39.

Listing 4.39: Re-allocating list size to twice as large

```
0 Object **p_temp_item;
1 p_temp_item = (Object **)
2     realloc(p_item, 2*sizeof(Object *) * max_count);
3 p_item = p_temp_item;
4 max_count *= 2;
```

The default copy constructor and assignment operator provided by C++ only do a “shallow” copy, meaning any dynamically allocated data items are not copied. Since the revised `List` class has dynamically allocated memory for the items, a copy constructor and assignment operator need to be created, doing a “deep” copy. The copy constructor and assignment operator prototypes look like:

Listing 4.40: Copy and assignment operator prototypes

```
0 ObjectList::ObjectList(const ObjectList &other);
1 ObjectList &operator=(const ObjectList &rhs);
```

In the assignment operator, memory for the copy needs to be dynamically allocated and copied over, along with the static attributes:

Listing 4.41: Deep copy of list memory

```
0 p_item = (Object **) malloc(sizeof(Object *) * other.max_count);
1 memcpy(p_item, other.p_item, sizeof(Object *) * other.max_count);
2 max_count = other.max_count;
3 count = other.count;
```

<sup>8</sup>The variable name is changed from `list` to explicitly depict that this is a pointer with dynamically allocated memory.

<sup>9</sup>Preliminary investigation running the Bounce benchmark and Saucer Shoot on both Linux Mint and Windows 7 suggests about 20% of the time when a list needs to expand, the memory block can be extended via `realloc()`, while 80% of the time the new block must be allocated elsewhere.



The assignment operator is similar, but with two additions before doing the “deep” copy: 1) the item being copied, `rhs`, must be checked to see if it is the same object (`*this`) to avoid copying the list over itself. Doing this check makes sense for efficiency and can also prevent some crashes in copying the memory over itself; 2) if the current object (`*this`) has memory allocated (`p_item` is not `NULL`), then that memory should be `free()`’d. Not doing this results in a memory leak.

Note, the above code needs error checking since calls to `malloc()` and `realloc()` can fail (returning `NULL`).

Lastly, the `ObjectList` destructor needs to check if (`p_item` is not `NULL`), and, if so, then that memory needs to be `free()`’d.

### 4.5.3 Development Checkpoint #3!

If you have not continued to do so, resume development now.

1. Create the `Vector` and `Object` classes, using the headers from Listing 4.25 and Listing 4.30, respectively. Add `Vector.cpp` and `Object.cpp` to the project and stub out each method so it compiles.
2. Implement both `Vector` and `Object`. Then, test even though the logic is fairly simple in both of these classes since they are primarily holders of attributes.
3. Create the `ObjectList` class, using the header from Listing 4.32. Refer to Listing 4.31 for method implementation details, remembering that `ObjectList` uses a static array of `Object` pointers. Add `.cpp` code to the project and stub out each method so it compiles.
4. Implement `ObjectList` and test. At this point, write a test program that inserts and removes elements from an `ObjectList`. Be sure to test boundary conditions – check if the `isFull()` method works, too, when the list reaches maximum size.
5. Create the `ObjectListIterator` class, using the header from Listing 4.36. Refer to Listing 4.33 for method implementation details. Add `.cpp` code to the project and stub out each method so it compiles.
6. Compile the iterator and write test code, referencing Listing 4.34 or Listing 4.35 for example code that uses the iterator. While iterating, adjust `Object` positions during run-time and print out values to verify code is working. Test cases where the list is empty, too.

Make sure to test all the above code thoroughly to be sure code is trustworthy (robust). Make sure the code is easy to read and commented sufficiently so it can be re-factored later as needed - the `Object` class is definitely re-visited as game objects grow in attributes and functionality.



### 4.5.4 Updating Game Objects

The world in real-life is dynamic, with objects changing continuously over time. Game worlds are often viewed the same way since they are also dynamic, but the game engine advances the game world in discrete steps, one step each game loop. At the end of a game loop, the static representation of the world is displayed to the player on the screen. Objects are consistent with each other at that time. While updating the world (so, in the middle of a game loop iteration), the game world may be in an inconsistent state. Viewed another way, each iteration of the game loop updates game objects to produce a sample of the dynamic game world, with *Dragonfly* taking 30 samples per second.

Updating the objects in the game world is one of the core functions of a game engine. Such updates: 1) Make the game dynamic since many objects change state during the course of the game. For example, an enemy can change position, moving towards the player's avatar. 2) Make the game interactive, since objects can respond to player input. For example, a player avatar object can be moved north in response to the player pressing the up arrow.

The simple approach to updating game objects is to have each object have an `Update()` method. In the game loop, the engine then iterates over all objects in the world, calling `Update()` for each object, as shown in Listing 4.42. In this case, the `Update()` method is responsible for updating the state of the object as appropriate. This could mean moving the object in a certain direction at a certain speed, or gathering input from the keyboard or mouse or doing whatever other unique action needs to happen each step of the game loop. Some actions, such as movement and keyboard input, could be generalized and handled by the game engine (as will be shown later in this chapter). Other actions, such as AI behavior specific to a game, would need to happen in the game code.

Listing 4.42: Game loop with update

```

0 ObjectList world_objects
1 while (game not over) {
2     ...
3     // Update world state.
4     ObjectListIterator li(&world_objects)
5     li.first()
6     while not li.isDone() do
7         li.currentObject() -> Update()
8         li.next()
9     end while
10    ...
11 }
```

As an abstraction, use of the `Update()` method for all objects is useful, since it gets at the heart of what a game engine does. However, the specific implementation of this straightforward idea has complications. These complications arise from subsystems that operate on behalf of all objects. For example, an update for an object often consists of: moving the object (including checking and responding to collisions), then drawing the object on the screen. For a Saucer object from Saucer Shoot in Section 3.3, this might look like the code in Listing 4.43. The proposed implementation looks harmless enough, but consider what is happening for all objects. Each object is moved, collided and drawn completely



before the next object is handled. This serial behavior does not allow for drawing efficiency. For example, it may be that an object is not drawn at all because it is occluded by another object or even destroyed by another object that moves later in the game loop. The serial nature does not allow for tuning. Worse, some objects cannot be drawn until the position of other objects are known. For example, drawing a passenger must be done once the position of the vehicle is known, or, the limbs of a 3d model may not be drawable until the position of the skeleton is known. Thus, efficiency and functionality require another solution.

Listing 4.43: Possible Update() method for Saucer

```

0 // Update saucer (should be called once per game loop).
1 void Saucer::Update()
2     WorldManager move(this)
3     WorldManager drawSaucer(this)

```

Instead, the subsystems that handle each task (e.g., move, draw) are done as separate functions by the game engine. The `Update()` method for each object does not need to ask the game engine to move, collide or draw the object itself. These are instead handled in phases by the game engine, depicted in Listing 4.44. Note, the `Update()` method for each object can still be invoked, calling game code, to do any game-specific functionality that is needed.

Listing 4.44: Game loop with phases

```

0 ObjectList world_objects
1 while (game not over) do
2     ...
3     // Have objects update themselves.
4     ObjectListIterator li(&world_objects)
5     for (li.first(); not li.isDone(); li.next())
6         li.currentObject() -> Update()
7     end for
8
9     // Move all objects.
10    for (li.first(); not li.isDone(); li.next())
11        WorldManager move(li.currentObject())
12    end for
13
14    // Draw all objects.
15    for (li.first(); not li.isDone(); li.next())
16        WorldManager draw(li.currentObject())
17    end for
18    ...
19 end while

```

### 4.5.5 Events

Games are inherently event-driven. Events can be almost anything. In the previous section, each iteration of the game loop is often treated as an event. In other words, in Listing 4.44, each iteration of the game loop triggers an event that is the `Update()` method for each object. A typical game has many events, such as a key is pressed, a mouse is clicked, an object collides with another object, a bomb explodes, an avatar picks up a health pack, etc.





Generally, when an event occurs, an engine: 1) notifies all interested objects, and 2) those objects respond, also called *event handling*. When a specific event occurs, different objects respond in different ways, and in some cases, may not even respond at all. For example, when a keypress event occurs, the hero object may move or fire, but most other objects do not respond. When a car object collides with a rock object, the car object may stop and take damage while the rock object may move slightly backward.

The simple approach to dealing with game events is for the game engine to call the appropriate method for each object when the event occurs. In Listing 4.44, this means that each step of the game loop (a step event) invokes the `Update()` method of each object. Consider another example, where there is an explosion in a game, handled in the `Update()` method of an Explosion object, shown in Listing 4.45. In this case, all objects within the radius of the explosion have their `onExplosion()` method invoked.

Listing 4.45: Explosion Update()

```

0 void Explosion::Update()
1   ...
2   if (explosion_went_off) then
3
4       // Get list of all objects in range.
5       ObjectList damaged_objects = getObjectsInRange(radius)
6
7       // Have them each react to explosion.
8       ObjectListIterator li(&damaged_objects)
9       for (li.first(); not li.isDone(); li.next())
10          li.currentObject() -> onExplosion()
11       end for
12
13       ...
14   end if

```

Listing 4.45 illustrates *statically typed, late binding*. The code is “late binding” since the compiler does not know what code is to be invoked at compile time – invocation is bound to the right method, depending upon the object (e.g., Saucer or Hero), at run time. The code is “statically typed” since the type of the object (an Object) and name of the method (`onExplosion()`, on line 10) are known at compile time. All this sounds ok, and Listing 4.45 looks ok, so what is the problem?

In a nutshell, for a general purpose game engine the problem with this approach is *inflexibility*. The statically typed requirement means that all objects must have methods for all events. Specifically, for this example, it means every object needs an `onExplosion()` method, even if not all objects use it. The fact that an object may not use it is perhaps not so bad, since it can just be ignored (the `onExplosion()` method essentially being a “no-op” for that object). However, some games will not even *have* explosions, but this approach still requires all objects to have that method. In fact, it requires that all events that any game made with the engine be known, and defined, at compile time. If a game is to be made using an event that is not defined, then too bad – the engine will not support it. That makes it quite difficult for the game engine to be general purpose, able to support a variety of games, much less a variety of game genres.

What is needed is *dynamically* typed, late binding. While some languages support



dynamic typing automatically (e.g., C#), others, such as C++, must implement dynamic typing manually. Fortunately, this can be done fairly easily by treating events as objects. When an event occurs, it is passed to all objects that are interested in that event. In the event handler, the object is inspected for the event type and attributes, and an appropriate action is taken. This paradigm is often called *message passing*.

In order to provide the flexibility needed without forcing the engine to recognize all event types at compile time, the event is encapsulated in an Event object. The Event object has the information required to represent the event type (e.g., explosion, health pack, collision, ...) with the ability to have additional attributes unique to each event (e.g., radius and damage, healing amount, location, ...). Representing events this way has several advantages over the approach in Listing 4.45.

1. *Single event handler*: Each game object does not need a separate method for each event (for example, objects do not all need an `onExplosion()` method). Instead, game objects have a generic event handler method (e.g., `virtual int eventHandler(Event *p_e)`), declared as `virtual` so it can be overridden, as needed, by derived game code objects.
2. *Persistence*: Event data pertaining to the event can be easily stored (say, in a list inside an object) and handled later.
3. *Blind forwarding*: An object can pass along an event without even “understanding” what the event does. Note, this is *exactly* what the engine does when it passes events to game objects! For example, a jeep object may get a “dismount” event. The jeep itself does not know how to dismount nor have any code to recognize such an event, but it can pass the event, unmodified, to each of the passengers that it does know about. The passengers, say people objects, do know how to handle a dismount event, and so take the appropriate action.

There are several options for representing the “type” for each event. One approach is to make each type an integer. Integers are small and are efficiently handled by the computer during runtime. Using an `enum` can make the integer type more programmer-friendly. For example, an `enum EventType` can be declared with values of `COLLISION`, `MOVE_UP`, `MOUSE_CLICK`, ... declared. Each “name” is assigned a unique integer value by the compiler. Game programmers can extend the types to include game specific definitions (e.g., `EXPLOSION`). While easy to read and efficient, `enum` types are relatively brittle in that the actual values are order dependent, meaning if the order of the names is re-arranged, the integer values corresponding to each change. This is not a problem if the code using them is re-compiled accordingly, but can cause problems for things like save game files or databases, or types stored in source code across systems. Worse, C++ does not readily allow for `enums` to be extended, meaning the game programmer cannot easily add custom event types to types already declared by the engine.

Another option, one used in *Dragonfly*, is to store event types as strings (e.g., `string event.type`). Strings as a type are dynamic in that they are parsed at runtime, allowing free form use by game programmers. Thus, events can be added easily, such as “explosion” or “the dog ate my homework”. The downside is that strings are relatively expensive to parse



compared with integers. However, string comparisons (the most common operation when checking events at runtime) are usually fast. Another downside is that game programmers, especially for large teams, may have potential event name conflicts with each other or even with the engine. A bit of care where a team of game programmers agrees upon a naming convention can usually solve this problem. For event names, *Dragonfly* uses a “df:” prefix, as it does for a *namespace* (see page 51), in front of game engine event names (e.g., “df:step”). If needed (or for really large development efforts), more elaborate software systems can even be used to avoid conflicts, checking code for conflicts ahead of time and detecting human errors.

The arguments needed for each event depend upon the type. For example, an explosion event may need a radius and damage, while a collision event needs the two objects involved and perhaps a force. The easiest mechanism to support this is to have a new derived class for each event, where the class inherits from the base event class. Listing 4.46 shows how this might be declared.<sup>10</sup> In the game code, an event handler looks at the event type. If it is, for example, an “explosion” and the object should recognize and handle explosion events, then the object can be inspected for a location, damage and radius.

Listing 4.46: Simple event class

```

0 class Event {
1     std::string event_type;
2 };
3
4 class EventExplosion : public Event {
5     Vector location;
6     int damage;
7     float radius;
8 };

```

As discussed earlier, game objects are often connected to each other, so a vehicle may get a “dismount” event, but it is really intended for the passengers, or a soldier may get a “heal” event that does not need to be passed to her backpack or to the pistol inside. A dependency chain, often called a *chain of responsibility* design pattern, can be drawn between events, illustrating their relationship. In this case, vehicle–soldier–backpack–pistol. Events that start at the head of the chain are passed down the chain, stopping when “consumed” or when the end of the chain is reached. For example, a “heal” event starts at the vehicle, is forwarded blindly to the soldier where it is consumed, and not passed further. An “explosion” event starts at the vehicle, where it takes damage, but then is passed along to each object in the chain since all take damage, too.

Listing 4.47 illustrates how a chain of responsibility might look for a particular game. One line 2, some events are “consumed” (completely handled) by the base class and no further action is required. On line 6, damage events invoke a response from someGameOb-ject, but are not consumed in that other objects can respond to the damage, too. In line 10, health pack events are consumed, so other objects in the chain do not handle them. Unrecognized events, line 15, are not handled.

Listing 4.47: Chain of responsibility

<sup>10</sup>Note, methods to set the event type are not shown!



```

0 bool someGameObject::eventHandler(Event *p_event)
1     // Call base class' handler first.
2     if (BaseClass::eventHandler(p_event))
3         return true // If base consumed, then done.
4
5     // Otherwise, try to handle event myself.
6     if p_event -> getType() is EVENT_DAMAGE then
7         takeDamage(p_event -> getDamageInfo())
8         return false // Responded to event, but ok to forward.
9     end if
10    if p_event -> getType() is EVENT_HEALTH_PACK then
11        doHeal(p_event -> getHealthInfo())
12        return true // Consumed event, so don't forward.
13    end if
14    ...
15    return false // Didn't recognize this event.

```

The code in Listing 4.47 is almost right – but the compiler will throw up an error at lines 7 and 11. See Section 4.5.5.3 on page 92 for details on the casting that needs to be done before the compiler will successfully compile the code.

#### 4.5.5.1 Events in Dragonfly

Listing 4.48 provides the header file for the Event class. The event type `string_type` is a string and is set to `UNDEFINED_EVENT` on line 2 in the constructor. The `setType()` and `getType()` methods change and return the event type, respectively. The `virtual` keyword in front of the destructor in line 14 ensures that if a pointer to a base Event is deleted (say, in the engine), the destructor to the child gets called, as appropriate.

Listing 4.48: Event.h

```

0 #include <string>
1
2 const std::string UNDEFINED_EVENT = "df::undefined";
3
4 class Event {
5
6     private:
7         std::string m_event_type; // Holds event type.
8
9     public:
10        // Create base event.
11        Event();
12
13        // Destructor.
14        virtual ~Event();
15
16        // Set event type.
17        void setType(std::string new_type);
18
19        // Get event type.
20        std::string getType() const;
21

```



22 };

The base Event class is passed around to game objects. It is expected that game code inherits from Event in defining game specific events, such as EventNuke in Saucer Shoot (Section 3.3.8 on page 32). Dragonfly does recognize (and pass several) specific events that are derived from Event. These “built in” events are depicted in Figure 4.2. Most of them are defined later in this chapter as they are introduced, except for the “step” event, which is defined next in Section 4.5.5.2.

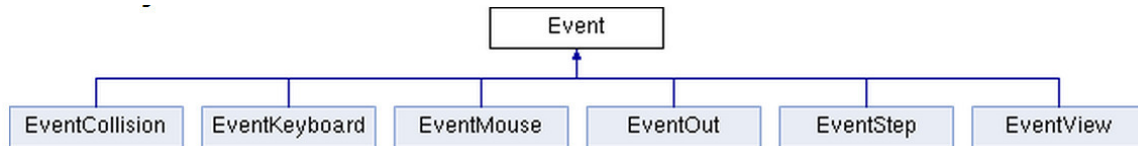


Figure 4.2: Dragonfly events

#### 4.5.5.2 Step Event

Often, a game object does something every step of the game loop. For example, a sentry object may look around to see if there is a bad guy within sight, or a bomb object may see if enough time has passed and it is time to explode. Dragonfly supports this by providing a “step” event each game loop for game objects that want to handle it.

Listing 4.49 provides the header file for the EventStep class. EventStep is derived from the Event base class. The `private` attribute `step_count` is to record the current iteration number of the game loop. Methods are provided to get and set `step_count`, as well as a constructor to set the initial `step_count`, if desired. Other “work” done in the constructor is to set `event_type` to `STEP_EVENT`.

Listing 4.49: EventStep.h

```

0 #include "Event.h"
1
2 const std::string STEP_EVENT = "df::step";
3
4 class EventStep : public Event {
5
6     private:
7         int m_step_count; // Iteration number of game loop.
8
9     public:
10        // Default constructor.
11        EventStep();
12
13        // Constructor with initial step count.
14        EventStep(int init_step_count);
15
16        // Set step count.
17        void setStepCount(int new_step_count);
18

```



```

19 // Get step count.
20 int getStepCount() const;
21 };

```

Inside the engine, the step event is handled like any other event, in terms of being stored and sent to Objects' event handlers. Inside the event handler code for an Object is where, if required, the step event is recognized and acted upon. For example, as shown in Listing 4.50, the Points object in Saucer Shoot (Chapter 3) recognizes the step event in its `eventHandler()`, counting the number of times called so it can increment the score every 30 steps (1 second).

Listing 4.50: Points eventHandler()

```

0 int Points::eventHandler(const Event *p_e) {
1     ...
2     // If step, increment score every second (30 steps).
3     if (p_e -> getType() == df::STEP_EVENT) {
4         if (p_e -> getStepCount() % 30 == 0)
5             setValue(getValue() + 1)
6         ...
7     }

```

The GameManager sends step events to each interested game object, once per game loop. Basically, inside the game loop, the GameManager iterates over each of the Objects in the game world and sends each of them an `EventStep`, with the step count set (via `setStepCount()`) to the current game loop iteration count. Listing 4.51 shows pseudo code for sending step events to all objects in the game world. Line 2 gets all the objects to iterate over from the WorldManager (see Section 4.5.6 on page 94). Line 3 creates an instance of the step event (`EventStep`) that will be passed to each Object, with `loop_count` referring to the current iteration number of the game loop. Lines 4 to 8 iterate through all Objects in the world, passing the step event to the Object event handlers in line 6.

Listing 4.51: Sending step events

```

0     ...
1     // Send step event to all objects.
2     all_objects = WorldManager getAllObjects()
3     create EventStep s(game_loop_count)
4     create ObjectListIterator li on all_objects list
5     while not li.isDone() do
6         li.currentObject() -> eventHandler() with s
7         li.next()
8     end while
9     ...

```

### 4.5.5.3 Casting

C++ is a strongly typed language. Among other things, this means that values of one type (e.g., `float`) can only be assigned to variables that are of the same type (e.g., `float f`) or of a different type that has a known conversion (e.g., `int i`, where values after the decimal point are truncated). When a type is assigned to a variable of a different type where there



is no known conversion, one of two things can happen. If the types are of different sizes and structures, such as a `struct` type being assigned to an `int`, then the compiler produces an error message and halts. If the types are the same size, such as a `enum` type being assigned to an `int`, then the compiler produces a warning message but continues to compile the code. At runtime, then, the conversion does happen (`enum` to `int`, in this example) even if that is not what the programmer intended.

In game code, when the engine provides a generic event, the event handler often needs to convert the generic event to a specific event when it determines what type it is. With `Dragonfly`, the `eventHandler()` for an Object is invoked with a pointer to a generic event (e.g., an `Event *`). Once the `eventHandler()` determines the event type (e.g., a step event, `EventStep`) by invoking the `getType()` method, it can treat the event as the specific type.

In C++, this can be done with a *type-cast* (or just *cast* for short) which converts one type to another. C++ has different varieties of type-casts, but the one needed in this case is the *dynamic cast*.<sup>11</sup> The syntax for a dynamic cast is `dynamic_cast <new_type> (expression)`. For example, a dynamic cast from a base class to a derived class is written as in Listing 4.52. The value of `p_b` of type `Base *` is converted to a different type, type `p_d`.

Listing 4.52: Cast from base class to derived class

```
0 class Base {};
1 class Derived : public Base {};
2 Base *p_b = new Base;
3 Derived *p_d = dynamic_cast <Derived *> (p_b);
```

Generally, a `dynamic_cast` is used for converting pointers within an inheritance hierarchy, almost exclusively for handling polymorphism (see Chapter 1).

For a game developed using `Dragonfly`, a cast is often needed in a game object's `eventHandler()`. The `eventHandler()` takes as input a pointer to a generic event, or an `Event *`. Once the type of the event is determined by invoking the method `getType()` and examining the string returned, the game code often acts on the event, as appropriate. For example, in Listing 4.53 the Bullet's `eventHandler()` from Saucer Shoot (Section 3.3 on page 14) checks if the event type is a collision event (`COLLISION_EVENT` – see Section 4.10.1.2 on page 136 for details on the collision event). If so, it acts upon it in the `hit()` method. Since the Bullet needs to access methods specific to the collision event to obtain the Object collided with for destruction, the `hit()` method takes a pointer to a collision event, not a generic event. This means the `Event *` passed to the `eventHandler()` must be cast as an `EventCollision *`.

Listing 4.53: Cast from Event to EventCollision

```
0 int Bullet::eventHandler(const df::Event *p_e)
1 ...
2 if p_e->getType() is df::COLLISION_EVENT then
3     EventCollision *p_collision_e = dynamic_cast <const df::EventCollision
4         *> (p_e)
5     hit(p_collision_e)
6     return 1
```

<sup>11</sup>The C-style cast (e.g., `int x = (int) 4.2`) is generally replaced with a `static_cast` in C++.



```

6   end if
7   ...

```

### 4.5.6 The WorldManager

At this point, development of the game world has provided game objects, lists and iterators for those objects, and events along with a means of passing them to game objects. For *Dragonfly*, this means the WorldManager can be designed and implemented. The WorldManager manages game objects, inserting them into the game world, removing them when done, moving them around, and passing along events generated by the game code. For now, this is all the WorldManager does. Soon, however, the WorldManager's functionality will expand to manage world attributes, such as size and camera location, drawing and animating object sprites and providing collisions and other game engine events.

The WorldManager is a singleton (see Section 4.2.1), so the methods on lines 6 to 8 are private and line 15 provides the instance of the WorldManager. For now, the WorldManager only has two attributes: 1) Line 10 *updates* is a list holding all the game objects in the world; and 2) Line 11 *deletions* is a list of the game objects to delete at the end of the current update phase.

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

Listing 4.54: WorldManager.h

```

0  #include "Manager.h"
1  #include "ObjectList.h"
2
3  class WorldManager : public Manager {
4
5  private:
6      WorldManager(); // Private (a singleton).
7      WorldManager(WorldManager const&); // Don't allow copy.
8      void operator=(WorldManager const&); // Don't allow assignment.
9
10     ObjectList m_updates; // All Objects in world to update.
11     ObjectList m_deletions; // All Objects in world to delete.
12
13  public:
14     // Get the one and only instance of the WorldManager.
15     static WorldManager &getInstance();
16
17     // Startup game world (initialize everything to empty).
18     // Return 0.
19     int startUp();
20
21     // Shutdown game world (delete all game world Objects).
22     void shutDown();
23
24     // Insert Object into world. Return 0 if ok, else -1.
25     int insertObject(Object *p_o);
26
27     // Remove Object from world. Return 0 if ok, else -1.

```





```

28  int removeObject(Object *p_o);
29
30  // Return list of all Objects in world.
31  ObjectList getAllObjects() const;
32
33  // Return list of all Objects in world matching type.
34  ObjectList objectsOfType(std::string type) const;
35
36  // Update world.
37  // Delete Objects marked for deletion.
38  void update();
39
40  // Indicate Object is to be deleted at end of current game loop.
41  // Return 0 if ok, else -1.
42  int markForDelete(Object *p_o);
43 };

```

The methods `insertObject()` and `removeObject()` provide a means to insert and remove objects in the world, respectively.

The `markForDelete()` method is called whenever an Object needs to be destroyed in the course of running the game. For example, when a projectile object collides with a target object (e.g., Bullet with Saucer), the projectile may mark both itself and the target for deletion.

The method `getAllObjects()` returns the `updates` ObjectList. A similar method, `objectsOfType()` returns a list of Objects matching a certain type. This method, shown in Listing 4.55, iterates through all Objects in the `updates` list and each Object that matches in type is added to the ObjectList, returned at the end.

Listing 4.55: WorldManager objectsOfType()

```

0  // Return list of Objects matching type.
1  // List is empty if none found.
2  ObjectList objectsOfType(std::string type) const
3
4  ObjectList list
5  ObjectListIterator li(&updates)
6  for (li.first(); not li.isDone(); li.next())
7      if li.currentObject() equals type then
8          list.insert(li.currentObject())
9      end if
10 end for
11
12 return list

```

The `update()` method is called from the GameManager (Section 4.4.4) once per game loop. In general, the update phase moves objects, generates collision events, etc. For now, it will only remove objects that have been marked for deletion.

The GameManager invokes WorldManager `startUp()` right after the LogManager is started. At this point, the WorldManager does not do much in `startUp()`, except for calling `Manager::startUp()`. Later versions of the WorldManager will set some of the game world attributes.

When invoked (typically by the GameManager), WorldManager `shutDown()` deletes



all the Objects in the game world. Typically, the game code does not preserve the address of game objects created to populate the world so only the WorldManager can do so. Pseudo code for WorldManager `shutDown()` is shown in Listing 4.56.

Listing 4.56: WorldManager shutDown()

```

0 // Shutdown game world (delete all game world Objects).
1 void WorldManager::shutDown()
2
3 // Delete all game objects.
4 ObjectList ol = updates // copy list so can delete during iteration
5 ObjectListIterator li(&ol)
6 for (li.first(); not li.isDone(); li.next())
7     delete li.currentObject()
8 end for
9
10 Manager::shutDown()

```

At this point, the Object class needs to be extended to support events. A public event handling method, `eventHandler()` is declared as in Listing 4.57.

Listing 4.57: Event handler prototype

```

0 // Handle event (default is to ignore everything).
1 // Return 0 if ignored, else 1 if handled.
2 virtual int eventHandler(const Event *p_e);

```

The implementation body of the `eventHandler()` method should do nothing, merely returning 0 indicating that the event was not handled. However, the keyword `virtual` ensures that derived classes (such as Saucer and Hero) can define their own specific event handlers. The keyword `const` indicates the event handler cannot modify the attributes of the event pointed to (`p_e`) – this is because the same event may be passed to multiple Objects.

The Object constructor needs to be modified also. Specifically, it needs to add the Object itself to the game world. A code fragment for this is shown in Listing 4.58. Since parent constructors are automatically called from derived classes, a derived object created by the game programmer (e.g., a Hero) calls the Object constructor, causing the object to automatically have itself added to the game world.

Listing 4.58: Object Object()

```

0 // Construct Object. Set default parameters and
1 // add to game world (WorldManager).
2 Object::Object()
3
4 // Add self to game world.
5 WorldManager insertObject(this)

```

Similarly, the destructor needs to remove the Object from the game world. A code fragment for this is shown in Listing 4.59. In a fashion similar to the constructor, when a derived object is destroyed, the parent destructor is called, removing the object from the game world.



Listing 4.59: Object ~Object()

```

0 // Destroy Object.
1 // Remove from game world (WorldManager).
2 Object::~~Object()
3
4 // Remove self from game world
5 WorldManager removeObject(this)

```

#### 4.5.6.1 Deferred Deletion

During the update phase of a game loop, a game object (with a base Object class) may be tempted to delete itself (calling `delete`) or another game object, perhaps as a result of a collision or after a fixed amount of time. But such an operation would likely be carried out somewhere in the middle of the update loop, so the iterator would be in the middle of going through the list of game world Objects. This may mean other Objects that are later in the iteration act on the recently deleted Object!

To illustrate these issues, consider an example game where darts are thrown at colored balloons for points. When a dart and a balloon collide, the WorldManager sends a collision event to both the dart and the balloon. The balloon, upon getting the collision event, destroys itself. The dart upon getting a collision, may also destroy itself. So far so good. However, what if the dart also queries the balloon to see check the balloon's color so as to add the right number of points. If the balloon has been deleted there is no way to do this! In fact, the code will compile and run, but will most likely result in a memory violation error and crash. Moreover, objects, in general, should very rarely use `delete this` to be removed. It is legal, but should only be done carefully under delicate circumstances.

A cleaner, safer method of removing game objects from the game world is via the `markForDelete()` method in the WorldManager. Basically, an Object that is ready to be destroyed or an Object that is ready to destroy another Object indicates this by telling the WorldManager to delete the Object at the end of the current update phase. Pseudo-code for the WorldManager's `markForDelete()` is shown in Listing 4.60. The top code block makes sure not to add the Object more than once. Failure to do so would mean that if an Object was marked more than once, `delete` would be called on an already de-allocated block of memory. If the last line of the method is reached, the Object had not been added so the list so it is added.

Listing 4.60: WorldManager markForDelete()

```

0 // Indicate Object is to be deleted at end of current game loop.
1 // Return 0 if ok, else -1.
2 int WorldManager::markForDelete(Object *p_o)
3
4 // Object might already have been marked, so only add once.
5 create ObjectListIterator li on deletions list
6 while not li.isDone() do
7     if li.currentObj() is p_o then // Object already in list.
8         return 0 // This is still "ok".
9     end if
10    li.next()
11 end while

```



```

12
13 // Object not in list, so add.
14 deletions.insert(p_o)

```

With the addition of the above code, some “unusual” code in the tutorial can be explained. Specifically, when a Saucer is created via `new` the pointer is not saved (i.e., `new Saucer;`). Normally, this would look like a potential source of a memory leak in that memory is allocated, but it is not clear it can be de-allocated with a corresponding `delete` since the pointer value is lost. However, having now written the constructor for Object, the pointer `this` is passed to the WorldManager where it is stored in the `updates` list. When the time comes to destroy the Object, the request is made to the WorldManager to mark this Object for deletion, which then does call `delete`.

#### 4.5.6.2 The Update Phase

With the new Object code in place, and the `markForDelete()` method available, the WorldManager’s `update()` can be defined. Pseudo-code for WorldManager `update()` is shown in Listing 4.61.

Lines 5 to 9 iterate through all objects that have been marked for deletion, actually deleting them by calling `delete` in line 9. Line 12 clears the deletion list (so there are no Objects in it) to get ready for the next phase.

Listing 4.61: WorldManager update()

```

0 // Update world.
1 // Delete Objects marked for deletion.
2 void WorldManager::update()
3
4 // Delete all marked objects.
5 create ObjectListIterator li on deletions list
6 while not li.isDone() do
7     delete li.currentObject()
8     li.next()
9 end while
10
11 // Clear list for next update phase.
12 deletions.clear()

```

#### 4.5.7 Program Flow for Game Objects

This section provides a summary of the program flow in *Dragonfly* for Objects when they are created and destroyed.

When a game object, derived from Object (e.g., Saucer), is created:

1. The game program (e.g., `game.cpp`) invokes `new`, say `new Saucer`.
2. The base Object constructor, `Object()`, is invoked first before the game object constructor, (e.g., `Saucer()`).
3. The Object constructor, `Object()`, calls WorldManager `insertObject()` to request being added to the game world.



4. WorldManager `insertObject()` calls `insert()` on the `updates` ObjectList, thus adding the game object to the game world.

When a game object is finished, ready to be destroyed:

1. The game program (e.g., game code in Saucer) calls WorldManager `markForDelete()`, indicating the Object is ready to be deleted.
2. WorldManager `markForDelete()` calls `deletions.insert()` to add the object to the `deletions` ObjectList.
3. GameManager `run()` calls WorldManager `update()` at the end of the current game loop iteration.
4. At the end of the `update()` method, the WorldManager iterates through the `deletions` ObjectList, calling `delete` on each Object in the list. The `delete` triggers the derived Object's destructor (e.g., `~Saucer()`).<sup>12</sup>
5. The derived Object's destructor (e.g., `~Saucer()`), calls the base class destructor, Object `~Object()`.
6. The Object destructor, `~Object()`, calls WorldManager `removeObject()`, requesting the WorldManager to remove the Saucer from the game world.
7. WorldManager `removeObject()` calls `remove()` on the `updates` ObjectList, removing the Saucer from the game world.

## 4.6 Development Checkpoint #4 – Dragonfly Egg!

Your Dragonfly development should continue.

1. Create the base Event class referring to the header file in Listing 4.48. Add `Event.cpp` to the project and stub out each method so it compiles. Testing should primarily ensure that it compiles, but make a stand alone program that sets (`setType()`) and gets (`getType()`) the event type for thoroughness.
2. Create the derived EventStep class based on the header file in Listing 4.49. Add `EventStep.cpp` to the project and stub out each method so it compiles. As for the Event class, testing should primarily ensure that it compiles, but create test code to be sure event types can be get and set for this derived class.
3. Add an event handler method to the Object class, based on Listing 4.57. Test by creating a simple game object derived from the Object class (e.g., a Saucer) and spawning (via `new`) several in a program. Define the class' `eventHandler()` methods to recognize a step event. Pass in both EventStep events and Events and see that they are recognized properly. Verify this with output messages to the screen and/or logfile.

---

<sup>12</sup>Remember, in C++, `delete` invokes an objects destructor *and* frees memory allocated by `new`.



4. Create the WorldManager class based on the header file in Listing 4.54. Add `WorldManager.cpp` to the project and stub out all methods, making sure the code compiles.
5. Write the bodies for WorldManager methods `insertObject()`, `removeObject()`, `getAllObjects()` and `objectsOfType()`. Create a stand alone program that tests that these methods work. Test by inserting multiple objects and removing some and then all, verifying each method works as expected. Use messages written to either the screen or logfile, both inside the methods and outside the WorldManager to get feedback.
6. Write code to extend the Object constructor and destructor to add and remove itself from the WorldManager automatically. Refer to Listing 4.58 and Listing 4.59 as needed. Test by using the derived game objects (e.g., Saucers) and spawning (via `new`) them in a program. Verify they are removed when deleted via `delete` for now.
7. Write the WorldManager `markForDelete()` method, referring to Listing 4.60 as needed. Write the WorldManager `update()`, too, at this time since `update()` and `markForDelete()` are easiest to test together. Test by spawning several derived game objects (e.g., Saucers), then marking some of them for deletion. Calling `update()` should see those Objects removed. Verify this with extensive messages to the screen and/or logfile.
8. Add functionality to the GameManager run loop. This includes doing the following once per game loop: 1) getting a list of all Objects from the WorldManager and sending each Object a step event (see Listing 4.51), and 2) calling WorldManager `update()`.

At this point, it is suggested to review the Dragonfly code base thus developed. First, to refresh the design and implementation done thus far. Second, to be sure code has been integrated into a single engine and the full set of functionalities implemented have been tested. If implementation has keep pace with the book, development should have come a long way! A game programmer can write game code to:

1. Start the GameManager. The GameManager should start the LogManager and the WorldManager, in that order.
2. Populate the game world. This means creating a class derived from Object (e.g., a Saucer) and spawning one or more objects (via `new`). The class constructor for Object has each instance add itself to the WorldManager. The Objects can set their initial positions.
3. Run the GameManager (via `run()`). The GameManager executes the game loop with controlled timing (using the Clock class). Each iteration, the GameManager gets the list of game objects from the WorldManager, then iterates through the list, sending each Object a step event.
4. The GameManager also calls `update()` in the WorldManager, which iterates through the list of all Objects marked for deletion, removing each of them via `delete`.



5. Objects handle the step event in their `eventHandler()` methods. The derived game object (e.g., Saucer) should actually define the behavior. At this point, a game object can “move” by changing its position to demonstrate functionality. Objects can write messages (e.g., (x,y) position) to the screen or logfile to show behavior.
6. After some condition (e.g., a game object has moved 100 steps), the game can be stopped by invoking GameManager `setGameOver()` method.
7. The engine can gracefully shut everything down by invoking GameManager `shutDown()`. This should shutdown the WorldManager and the LogManager, in that order.

For the game programmer, this means creating one or more derived game objects classes, and one or more “games” (each with a separate `main()`) that can be used to test, debug and demonstrate robust behavior from the engine.

The full set of the above functionality is a good start – the foundation of a game engine. Put another way, the base code thus far is a *Dragonfly* egg\* that, with the help of the rest of this book, will hatch and grow into a fully functioning *Dragonfly* game engine.

---

\* **Did you know (#3)?** Dragonflies start out their lives as eggs laid in water. A Dragonfly can lay as many as 100,000 eggs. – “Frequently Asked Questions about Dragonflies”, *British Dragonfly Society*, 2013.



## 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()`

```

0 // Send event to all Objects.
1 // Return count of number of events sent.
2 int Manager::onEvent(const Event *p_event) const
3     count = 0
4
5     all_objects = WorldManager getAllObjects()
6     create ObjectListIterator li on all_objects list
7     while not li.isDone() do
8         li.currentObject() -> eventHandler() with p_event
9         li.next()
10        increment count
11    end while
12
13    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

```

0 ...
1 // Provide step event to all Objects.
2 EventStep s;
3 onEvent(&s);
4 ...

```

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

```

0 #include <SFML/Graphics.hpp>
1 ...
2 // Create SFML window.
3 int window_horizontal = 1024
4 int window_vertical = 768
5 sf::RenderWindow window(sf::VideoMode(horizontal, vertical), "Title -
   Dragonfly", sf::Style::Titlebar)
6
7 // Turn off mouse cursor for window.
8 window.setMouseCursorVisible(false)
9
10 // Synchronize refresh rate with monitor.
11 window.setVerticalSyncEnabled(true)
12
13 // When done...
14 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”<sup>13</sup> is the name of the font file, supporting most standard

<sup>13</sup>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

```

0 sf::Font font
1 if (font.loadFromFile("df-font.ttf") == false) then
2     // Error
3 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

```

0 sf::Text text
1
2 // Select pre-loaded font (from Listing 4.65).
3 text.setFont(font)
4
5 // Set display string.
6 text.setString("Hello world")
7
8 // Set character size (in pixels).
9 text.setCharacterSize(24)
10
11 // Set color.
12 text.setColor(sf::Color::Red)
13
14 // Set style.
15 text.setStyle(sf::Text::Bold | sf::Text::Underlined)
16
17 // Set position on window (in pixels).
18 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

```

0 // Clear window and draw text.
1 window.clear();
2 window.draw(text);
3 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!

```

0 #include <iostream> // for std::cout
1 #include <SFML/Graphics.hpp>
2
3 int main() {
4
5     // Load font.
6     sf::Font font;
7     if (font.loadFromFile("df-font.ttf") == false) {
8         std::cout << "Error! Unable to load font 'df-font.ttf'." << std::endl;
9         return -1;
10    }
11
12    // Setup text to display.
13    sf::Text text;
14    text.setFont(font); // Select font.
15    text.setString("Hello , world!"); // Set string to display.
16    text.setCharacterSize(32); // Set character size (in pixels).
17    text.setColor(sf::Color::Green); // Set text color.
18    text.setStyle(sf::Text::Bold); // Set text style.
19    text.setPosition(96,134); // Set text position (in pixels).
20
21    // Create window to draw on.
22    sf::RenderWindow *p_window =
23        new sf::RenderWindow(sf::VideoMode(400, 300), "SFML - Hello , world!");
24    if (!p_window) {
25        std::cout << "Error! Unable to allocate RenderWindow." << std::endl;
26        return -1;
27    }
28
29    // Turn off mouse cursor for window.
30    p_window -> setMouseCursorVisible(false);
31
32    // Synchronize refresh rate with monitor.
33    p_window -> setVerticalSyncEnabled(true);
34
35    // Repeat as long as window is open.
36    while (1) {
37
38        // Clear window and draw text.
39        p_window -> clear();
40        p_window -> draw(text);
41        p_window -> display();
42
43        // See if window has been closed.
44        sf::Event event;
45        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`

```

0 // Colors Dragonfly recognizes.
1 enum Color {
2     UNDEFINED_COLOR = -1,
3     BLACK = 0,
4     RED,
5     GREEN,
6     YELLOW,
7     BLUE,
8     MAGENTA,
9     CYAN,
10    WHITE,
11 };
12
13 // If color not specified, will use this.
14 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;

```



```

10 const int WINDOW_VERTICAL_PIXELS_DEFAULT = 768;
11 const int WINDOW_HORIZONTAL_CHARS_DEFAULT = 80;
12 const int WINDOW_VERTICAL_CHARS_DEFAULT = 24;
13 const int WINDOW_STYLE_DEFAULT = sf::Style::Titlebar;
14 const sf::Color WINDOW_BACKGROUND_COLOR_DEFAULT = sf::Color::Black;
15 const std::string WINDOW_TITLE_DEFAULT = "Dragonfly";
16 const std::string FONT_FILE_DEFAULT = "df-font.ttf";
17
18 class DisplayManager : public Manager {
19
20 private:
21     DisplayManager(); // Private (a singleton).
22     DisplayManager(DisplayManager const&); // Don't allow copy.
23     void operator=(DisplayManager const&); // Don't allow assignment
24     sf::Font m_font; // Font used for ASCII graphics.
25     sf::RenderWindow *m_p_window; // Pointer to SFML window.
26     int m_window_horizontal_pixels; // Horizontal pixels in window.
27     int m_window_vertical_pixels; // Vertical pixels in window.
28     int m_window_horizontal_chars; // Horizontal ASCII spaces in window.
29     int m_window_vertical_chars; // Vertical ASCII spaces in window.
30
31 public:
32     // Get the one and only instance of the DisplayManager.
33     static DisplayManager &getInstance();
34
35     // Open graphics window ready for text-based display.
36     // Return 0 if ok, else -1.
37     int startUp();
38
39     // Close graphics window.
40     void shutDown();
41
42     // Draw character at window location (x,y) with color.
43     // Return 0 if ok, else -1.
44     int drawCh(Vector world_pos, char ch, Color color) const;
45
46     // Return window's horizontal maximum (in characters).
47     int getHorizontal() const;
48
49     // Return window's vertical maximum (in characters).
50     int getVertical() const;
51
52     // Return window's horizontal maximum (in pixels).
53     int getHorizontalPixels() const;
54
55     // Return window's vertical maximum (in pixels).
56     int getVerticalPixels() const;
57
58     // Render current window buffer.
59     // Return 0 if ok, else -1.
60     int swapBuffers();
61
62     // Return pointer to SFML drawing window.
63     sf::RenderWindow *getWindow() const;
64 };

```



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 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()

```

0 // Open graphics window ready for text-based display.
1 // Return 0 if ok, else return -1.
2 int DisplayManager::startUp()
3
4 // If window already created, do nothing.
5 if p_window is not NULL then
6     return 0
7 end if
8
9 create window // an sf::RenderWindow for drawing
10
11 turn off mouse cursor
12
13 synchronize refresh rate with monitor
14
15 load font
16
17 if everything successful then
18     invoke Manager::startUp()
19     return ok
20 else
21     return error
22 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

```

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

```



```

2
3 // Compute character width, based on window size and font.
4 float charWidth();
5
6 // Convert ASCII spaces (x,y) to window pixels (x,y).
7 Vector spacesToPixels(Vector spaces);
8
9 // Convert window pixels (x,y) to ASCII spaces (x,y).
10 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()

```

0 // Render current window buffer.
1 // Return 0 if ok, else -1.
2 int DisplayManager::swapBuffers()
3
4 // Make sure window is allocated.
5 if p_window is NULL then
6     return -1
7 end if
8
9 // Display current window.
10 p_window -> display()
11
12 // Clear window to get ready for next draw.
13 p_window -> clear()
14
15 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()

```

0 public:
1     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()

```

0 // Draw all objects.

```



```

1 void WorldManager::draw()
2
3     create ObjectListIterator li on updates // All game objects.
4
5     while not li.isDone() do
6         Object *p_temp_o = li.currentObj()
7         p_temp_o -> draw()
8         li.next()
9     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

```

0 enum Justification {
1     LEFT_JUSTIFIED,
2     CENTER_JUSTIFIED,
3     RIGHT_JUSTIFIED,
4 };
5
6 ...
7

```



```

8 class DisplayManager : public Manager {
9
10 ...
11
12 // Draw string at window location (x,y) with default color.
13 // Justified left, center or right.
14 // Return 0 if ok, else -1.
15 int drawString(Vector pos, std::string str, Justification just,
16               Color color) const;
17 ...
18 };

```

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
7 // Get starting position.
8 Vector starting_pos = pos
9 switch (just)
10 case CENTER_JUSTIFIED:
11     starting_pos.setX(pos.getX() - str.size()/2)
12     break
13 case RIGHT_JUSTIFIED:
14     starting_pos.setX(pos.getX() - str.size())
15     break
16 case LEFT_JUSTIFIED:
17     default:
18     break
19 end switch
20
21 // Draw string character by character.
22 for i = 0 to str.size()
23     Vector temp_pos(starting_pos.getX() + i, starting_pos.getY())
24     drawCh(temp_pos, str[i], color)
25 end for
26
27 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

```

0 private:
1   int m_altitude;           // 0 to MAX supported (lower drawn first).
2
3 public:
4   // Set altitude of Object, with checks for range [0, MAX_ALTITUDE].
5   // Return 0 if ok, else -1.
6   int setAltitude(int new_altitude);
7
8   // Return altitude of Object.
9   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

```

0 // In draw() ...
1 for alt = 0 to MAX_ALTITUDE
2
3   // Normal iteration through all objects.
4   ...
5
6   if li.currentObject() -> getAltitude() is alt then
7
8     // Normal draw.
9     ...
10
11   end if

```



```

12
13 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

```

0 private:
1   sf::Color window_background_color; // Background color of window.
2   ...
3
4 public:
5   // Set default background color. Return true if ok, else false.
6   bool setBackgroundColor(int new_color);
7   ...

```

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:

```

0 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`

```

0 #include <unistd.h> // Needed in Linux/Mac.
1
2 #include "DisplayManager.h"
3
4 int main() {
5     DM.startUp();
6     DM.drawCh(df::Vector(10,5), '*', df::WHITE);
7     DM.swapBuffers();
8     sleep(2); // Use Sleep(2000) if in Windows.

```



```
9 | DM.shutdown();  
10| }
```

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.





## 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, 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 103), 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 4.64 on page 103)). 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 4.84.

Listing 4.84: 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()`. `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 4.85. 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 2. 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 21. For the keyboard, the key that is pressed/released is in the `key.code` for the event, and for the mouse, the button that pressed/released is `mouseButton.button` for the event.

Listing 4.85: SFML input for games

```
0 // Go through all Window events, extracting recognized ones.
1 sf::Event event
2 while (p_window -> pollEvent(event)) do
3
4     // Key was pressed.
5     if event.type is sf::Event::KeyPressed then
6         // Do keypressed stuff. e.g., examine event.key.code
7     end if
8
9     // Key was released.
10    if event.type is sf::Event::KeyReleased then
11        // Do key released stuff. e.g., examine event.key.code
12    end if
13
14    // Mouse was moved.
15    if event.type is sf::Event::MouseMoved then
16        // Do mouse moved stuff. e.g., examine event.mouseButton.button
17    end if
18
19    // Mouse was clicked.
20    if event.type is sf::Event::MouseButtonPressed then
```



```

21     if event.mouseButton.button is sf::Mouse::Right
22         // Do right mouse clicked stuff.
23     end if
24 end if
25
26 end while
27
28 // Key is pressed.
29 if sf::Keyboard::isKeyPressed(keycode) then
30     // Do key is pressed stuff.
31 end if
32
33 // Mouse button is pressed.
34 if sf::Mouse::isButtonPressed(button) then
35     // Do mouse is pressed stuff.
36 end if

```

While the code in Listing 4.85 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. This is illustrated by the last two blocks of pseudo-code in Listing 29. Note, this code is *outside* of the `while()` loop. Here, a specific key can be polled (e.g., `sf::Keyboard::Left`) to see if it is currently being held down. Similarly, a specific mouse button can be polled (e.g., `sf::Mouse::Left`) to see if it is currently being held down

### 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 53). The header file, including class definition, is provided in Listing 4.86.

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 puts the terminal back into normal mode by turning the cursor back on using `cursor_set()`. Finally, the method `getInput()` uses SFML to obtain keyboard and mouse input and is called by the GameManager once per game loop.

Listing 4.86: InputManager.h

```

0 #include "Manager.h"
1
2 class InputManager : public Manager {
3
4     private:
5         InputManager();           // Private (a singleton).
6         InputManager (InputManager const&); // Don't allow copy.
7         void operator=(InputManager const&); // Don't allow assignment
8
9     public:
10        // Get the one and only instance of the InputManager.

```



```

11  static InputManager &getInstance();
12
13  // Get window ready to capture input.
14  // Return 0 if ok, else return -1.
15  int startUp();
16
17  // Revert back to normal window mode.
18  void shutDown();
19
20  // Get input from the keyboard and mouse.
21  // Pass event along to all Objects.
22  void getInput();
23 };

```

For starting up, an SFML window needs to be created first. 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 `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 4.87. 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 4.87: InputManager startUp()

```

0  // Get window ready to capture input.
1  // Return 0 if ok, else return -1.
2  int InputManager::startUp()
3
4      if DisplayManager is not started then
5          return error
6      end if
7
8      sf::RenderWindow window = DisplayManager getWindow()
9
10     disable key repeat in window
11
12     invoke 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 4.88 and are similar to those in Listing 4.85 (on page 120).

First, SFML window events are checked first. 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 4.62 on page 102 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.

Next, “live” actions are polled, first for each key on the keyboard and then for each mouse button. Corresponding Dragonfly events are generated and sent, as appropriate.

Listing 4.88: InputManager `getInput()`

```

0 // Get input from the keyboard and mouse.
1 // Pass event along to all Objects.
2 void InputManager::getInput()
3
4 // Check past window events.
5 while event do
6
7     if key press then
8
9         create EventKeyboard (key and action)
10        send EventKeyboard to all Objects
11
12    else if key release then
13
14        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.
30
31 // Check current key press events for each key.
32 if isKeyPressed(sf::Keyboard::Up) then
33
34    create EventKeyboard (key and action)
35    send EventKeyboard to all Objects
36
37 end if
38 ...
39

```



```

40 // Check current mouse press events for each button.
41 if mouse button pressed (sf::Mouse::Left) then
42
43     create EventMouse (x, y and action)
44     send EventMouse to all Objects
45
46 end if
47 ...

```

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 4.89 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`, `KEY_RELEASED`, and `KEY_DOWN`. 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.

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.

Listing 4.89: EventKeyboard.h

```

0 #include "Event.h"
1
2 const std::string KEYBOARD_EVENT = "df::keyboard";
3
4 // Types of keyboard actions Dragonfly recognizes.
5 enum EventKeyboardAction {
6     UNDEFINED_KEYBOARD_ACTION = -1, // Undefined
7     KEY_PRESSED,                    // Was down
8     KEY_RELEASED,                   // Was released

```



```

9   KEY_DOWN,                // Is down
10 };
11
12 // Keys Dragonfly recognizes.
13 namespace Keyboard {
14     enum Key {
15         UNDEFINED_KEY = -1,
16         SPACE, RETURN, ESCAPE, TAB, LEFTARROW, RIGHTARROW, UPARROW, DOWNARROW,
17         PAUSE, MINUS, PLUS, TILDE, PERIOD, COMMA, SLASH, LEFTCONTROL,
18         RIGHTCONTROL, LEFTSHIFT, RIGHTSHIFT, F1, F2, F3, F4, F5, F6, F7, F8,
19         F9, F10, F11, F12, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q,
20         R, S, T, U, V, W, X, Y, Z, NUM1, NUM2, NUM3, NUM4, NUM5, NUM6, NUM7,
21         NUM8, NUM9, NUM0,
22     };
23 } // end of namespace Keyboard
24
25 class EventKeyboard : public Event {
26
27     private:
28         Keyboard::Key m_key_val;                // Key value.
29         EventKeyboardAction m_keyboard_action;  // Key action.
30
31     public:
32         EventKeyboard();
33
34         // Set key in event.
35         void setKey(Keyboard::Key new_key);
36
37         // Get key from event.
38         Keyboard::Key getKey() const;
39
40         // Set keyboard event action.
41         void setKeyboardAction(EventKeyboardAction new_action);
42
43         // Get keyboard event action.
44         EventKeyboardAction getKeyboardAction() const;
45 };

```

#### 4.9.2.2 Mouse Event

Listing 4.90 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 `MouseButton` defined by `MouseButton`, respectively. Mouse actions recognized are `CLICKED`, `PRESSED`, 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 in side 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_position`.

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 `MOUSE_EVENT`.

Listing 4.90: EventMouse.h

```

0  #include "Event.h"
1
2  const std::string MOUSE_EVENT = "df::mouse";
3
4  // Set of mouse actions recognized by Dragonfly.
5  enum EventMouseAction {
6      UNDEFINED_MOUSE_ACTION = -1,
7      CLICKED,
8      PRESSED,
9      MOVED,
10 };
11
12 // Set of mouse buttons recognized by Dragonfly.
13 namespace Mouse {
14     enum Button {
15         UNDEFINED_MOUSE_BUTTON = -1,
16         LEFT,
17         RIGHT,
18         MIDDLE,
19     };
20 } // end of namespace Mouse
21
22 class EventMouse : public Event {
23
24     private:
25         EventMouseAction m_mouse_action; // Mouse action.
26         Mouse::Button m_mouse_button;    // Mouse button.
27         Vector m_mouse_xy;               // Mouse (x,y) coordinates.
28
29     public:
30         EventMouse();
31
32         // Load mouse event's action.
33         void setMouseAction(EventMouseAction new_mouse_action);
34
35         // Get mouse event's action.
36         EventMouseAction getMouseAction() const;
37
38         // Set mouse event's button.
39         void setMouseButton(Mouse::Button new_mouse_button);
40
41         // Get mouse event's button.
42         Mouse::Button getMouseButton() const;
43
44         // Set mouse event's position.
45         void setMousePosition(Vector new_mouse_xy);
46
47         // Get mouse event's y position.
48         Vector getMousePosition() const;
49 };

```





At this point, a view of complete version of the game loop is warranted, shown in Listing 4.91. 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 4.91: The complete game loop

```

0 Clock clock // Section 4.4.3 on page 66.
1 while (game not over) do // Line 11 of Listing 4.24 on page 69.
2     clock.delta() // Line 14 of Listing 4.19 on page 66.
3     InputManager getInput() // Listing 4.88 on page 123.
4     WorldManager update() // Listing 4.61 on page 98.
5     WorldManager draw() // Listing 4.76 on page 112.
6     DisplayManager swapBuffers() // Listing 4.74 on page 112.
7     loop_time = clock.split() // Line 19 of Listing 4.19 on page 66.
8     sleep(TARGET_TIME - loop_time) // Listings 4.20 and 4.21 on page 67+.
9 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 53. Add **InputManager.cpp** to the project and include stubs for all methods in Listing 4.86. Make sure the class, with stubs, compiles.
2. Write **startUp()** and **shutDown()** methods for the InputManager, referring to Listing 4.87 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 4.88 for the structure and Listing 4.85 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.



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.



## 4.10 Kinematics

Kinematics is the physics that describes the motion of objects without taking into account forces or masses. In relation to our engine, up until now, moving an object has to be done in game code with the game programmer writing code to move an game object each step event. For example, if a Saucer needs to move, say 1 space to the left every 4 steps, when the Saucer receives the step event in the `eventHandler()`, it moves, as in Listing 4.92. Basically, the Saucer looks for the step event, then decrementing a `countdown` variable until it reaches 0. At that time, it moves left one space and resets the `countdown` to 4.

Listing 4.92: Saucer movement without velocity support

```

0 // Move 1 space every step.
1 bool Saucer::eventHandler(const df::Event *p_e) {
2     if (p_e->getType() == df::STEP_EVENT) {
3         countdown--;
4         if (countdown == 0) { // Only move when time (i.e., not every step)
5             df::Vector position;
6             position.setX(getPosition().getX() - 1); // Move 1 space left.
7             setPosition(position);
8             countdown = 4;
9         }
10        return 1; // Event handled.
11    }
12    return 0; // Event not handled.
13 }

```

While this can work, all of it can be tedious for a game programmer and prone to errors. Fortunately, a significant service provided by most game engines, including *Dragonfly*, is moving game objects automatically, in the right direction at the right speed. This allows a game programmer to provide a game object with a velocity and have the object move an appropriate amount in an appropriate direction each game step.

Next, *Object* is extended with additional attributes for velocity, shown in Listing 4.93 – namely velocity and speed, both in *Vectors*. These should both be initialized to 0 in the *Object* constructor. They can be set by normal getters and setters. In addition, getting and setting the *Object* velocity is done via `getVelocity()` and `setVelocity()`, respectively, which use *Vector* operations `scale()` and `normalize()` (see Listing 4.27 on page 73 and Listing 4.28 on page 74), as appropriate.

Listing 4.93: Object extensions to support kinematics

```

0 private:
1     Vector direction; // Direction vector.
2     float speed;      // Object speed in direction.
3
4 public:
5     // Set speed of Object.
6     void setSpeed(float speed);
7
8     // Get speed of Object.
9     float getSpeed() const;
10
11    // Set direction of Object.

```



```

12 void setDirection(Vector new_direction);
13
14 // Get direction of Object.
15 Vector getDirection() const;
16
17 // Set direction and speed of Object.
18 void setVelocity(Vector new_velocity);
19
20 // Get velocity of Object based on direction and speed.
21 Vector getXVelocity() const;
22
23 // Predict Object position based on speed and direction.
24 // Return predicted position.
25 Vector predictPosition()

```

The real magic happens in an Object support method – one that computes where an Object will be after a game loop’s worth of velocity is added, but without actually moving it. This method, called `predictPosition()`, is shown in Listing 4.94. Basically, it indicates where an Object will be after it moves with its speed and direction (velocity).

Listing 4.94: Object predictPosition()

```

0 // Predict Object position based on speed and direction.
1 // Return predicted position.
2 Vector Object::predictPosition()
3
4 // Add velocity to position.
5 Vector new_pos = position + getVelocity()
6
7 // Return new position.
8 return new_pos

```

Once support for velocities is in the Object class, the WorldManager needs to be updated to use it. This is done by adding functionality to the `update()` method (see Section 4.5.6.2 on page 98). Listing 4.95 shows the necessary pseudo code.

Listing 4.95: WorldManager extensions to update() to support kinematics

```

0 // Update object positions based on their velocities.
1
2 // Iterate through all objects.
3 create ObjectListIterator i on updates list
4
5 while not li.isDone() do
6
7 // Add velocity to position.
8 Vector new_pos = p_o -> predictPosition()
9
10 // If Object should change position, then move.
11 if new_pos != getPosition() then
12 moveObject() to new_pos
13 end if
14
15 li.next()
16

```



```

17   end while // End iterate.
18   ...

```

Basically, `update()` iterates through all Objects. For each, it calls `predictPosition()` to check if the Object should move or not. If so, it moves the object by calling `moveObject()`. The functionality inside `moveObject()` is discussed in the next section, Section 4.10.1.

Using velocity support in `Dragonfly` for the game programmer is easy. Instead of the game object having to move itself by handling every step event in the `eventHandler()` and determining when and where to move, the game programmer can just set the speed and direction (or velocity) for game objects (derived from the Object class, of course). For example, for a Saucer traveling right to left across the screen, the programmer sets the x component of the velocity as in Listing 4.96. There is no longer a need for a game object to handle the step event for moving.

Listing 4.96: Saucer movement with velocity support

```

0 // Set movement left 1 space left every 4 frames.
1 setSpeed(0.25);
2 setDirection(df::Vector(-1.0, 0));
3
4 // The above two lines are equivalent to:
5 setVelocity(df::Vector(-0.25, 0));

```

**Newtonian Mechanics Physics (optional)** Velocity support is complete, but could be enhanced. Possible options could extend the velocity support to include an acceleration component (another `Vector`) for each game object. With acceleration, a game object's velocity would change slightly (it would accelerate or decelerate according to the direction) each step. This could be especially useful for providing, say, gravity pulling an avatar down (a fixed acceleration in the vertical direction) for a platformer game.

While the kinematics in `Dragonfly` provides support for a lot of games, additional classical Newtonian physics mechanics include forces and masses. In fact, while acceleration can simply be added to objects as mentioned in the previous paragraph, in the real world it is achieved through application of a force on an object through the equation:

$$F = m \cdot a$$

where  $F$  is the applied force,  $m$  is the object mass and  $a$  is the acceleration. Adding a mass attribute to game objects would be a first step to using forces as a precursor so acceleration and velocity.

And that is really just the tip of the iceberg. More advanced physics techniques that could be incorporated into `Dragonfly` include elastic and non-elastic collisions, rigid body, soft body and ragdoll simulation, joints as constraints, and more. The aspiring programmer is encouraged to check out one of the many books on physics for game engines, such as the one by Ian Millington [6].



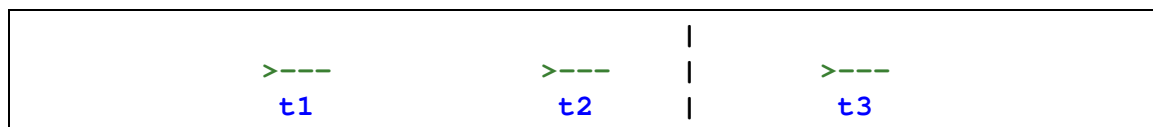
### 4.10.1 Collisions

A closely related service to moving a game object is detecting when game objects collide and, when they do, changing object behavior as appropriate. However, determining when objects collide is not as easy as it may seem.

- Geometries of objects can be complex – square or circular objects are efficient in terms of computing area (or volume, if 3d) and locations in the game world, but more complex objects, for example humanoids and trees, take many more computations to compute exact locations in the game world.
- Objects can be moving fast – objects that move great distances mean there is a larger area (or volume, if 3d) from the old position to the new position that may result in potential collisions.
- There can be many objects – the more objects there are, the more opportunities there are for collisions and the more computations the game engine needs to do each step to determine if objects collide. A naïve solution could take  $O(n^2)$ , meaning every object (all  $n$  of them) is checked for a collision with every other object, so  $n \times n$  comparisons are made each step.

There are many possibilities when it comes how to implement collision detection. *Dragonfly* uses what is perhaps the most commonly used technique for games – *overlap testing*. Overlap testing is relatively easy, both conceptually (for the game programmer as well as the game engine programmer) and in terms of implementation. However, it may exhibit more errors than some other forms of collision. Basically, with overlap testing, when an object moves, the engine checks whether or not its area (or volume, if 3d) overlaps that of any other object. If so, there has been a collision. If not, then the object can move unimpeded. When a collision has occurred, both objects get notified of the event.

Despite the advantages (simplicity and speed of execution), overlap testing can fail when objects move too fast relative to their size. Consider the game example below where an arrow is fired at a window. The arrow has a velocity of 15, meaning it moves 15 spaces horizontally each step.



At time *t1*, the arrow (*>---*), moving left to right, approaches the window (*|*). One step later, at time *t2*, the arrow's velocity moves it 15 spaces to the right and a collision seems imminent. However, there is not yet an overlap between the arrow and the glass, so no collision is detected. At time *t3*, the arrow's velocity carries it another 5 spaces to the right, past the window and as there is still no overlap, no collision is detected. Effectively, the arrow went right through the glass window with breaking it (or even hitting it)!

There are several possible solutions to this problem. A game programmer, knowing that the engine is using overlap testing, can put a constraint on game object sizes and speeds



such that the fastest object moves slow enough (per step) to overlap with the thinnest object. For the example above, the window would need to be 13 spaces thick or the arrow would need to only move 3 spaces per step, or some combination of the two. While this would ensure the arrow would always hit the window, it might not be practical for all games (e.g., the arrow speed might be too slow for other uses and/or the window too thick for the environment).

Another solution is to reduce the step size. The step size dictates how often the game world is updated. Game programmers want aspects such as the speed of an object to be relative to real time in terms of how fast the player sees the arrow move across the window, and not relative to the game step. This means, for example, that a velocity of 1 with a step frequency of 30 f/s moves 30 spaces per second. If the frequency was doubled, to 60 f/s, then the game programmer would want to adjust the velocity to be 0.5 so the player still sees the object move 30 spaces per second. What this will do, however, is make the object move fewer spaces each step, making it less likely for overlap testing to fail. However, this comes at a cost – namely, increased computation as the game loop runs more often (in this example, twice as often), possibly leading to the engine not being able to keep up with the update rate. \*

A more complex solution (incidentally, *not* supported by [Dragonfly](#)) is to have a different step size for different objects. Thus, fast objects and/or small objects would be updated more frequently than slow objects. This adds complexity, and computation overhead, in the update step, as well, but could be applied selectively.

The overlap test itself is fairly easy to compute with simple volumes, like circles (or spheres, if 3d). More complex volumes can require more computation. For example, a humanoid-shaped object and a tree-shaped object may be made up of hundreds or even thousands of smaller polygons. If every polygon in an object needs to be tested for overlap with the polygons of every other object to determine if the objects collide, this scales  $O(n^2)$  with  $n$  polygons per object.

To overcome this, complex geometries are often simplified in order to reduce the number of comparisons needed. Rather than doing this simplification in the model itself (which can compromise how it looks), each object can instead be given a bounding volume that approximates the object. For example, the humanoid-shaped object can be approximated by an ellipsoid and the tree-shaped object by a sphere. For a depiction of this last case, consider the brown, ASCII tree in Figure 4.3. Computing whether a single character collides with this tree would involve checking all the nooks and crannies of the branches – effectively examining every character in the tree for a collision as if each character was a separate object. Instead, the entire tree can be bound by a simpler shape for computing collisions – in this case, an ellipse, shown by the dashed oval surrounding the tree. Determining if another object collides with this tree is now much simpler, merely computing if it intersects the ellipse. It might be noted that the oval itself does not fully encompass the tree – some of the branches are poking out. This would mean that an object could hit those external branches and not collide with the tree. For many games, this might be just fine and could be preferred over drawing a bigger ellipse.

---

\* **Did you know (#6)?** Adult dragonflies have 6 legs but cannot walk. – “30,000 Facets Give Dragonflies a Different Perspective: The Big Compound Eye in the Sky”, *ScienceBlogs*, July 8, 2009.



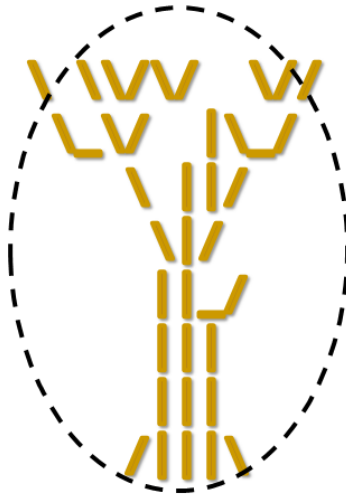


Figure 4.3: Object with bounding ellipse

Commonly used bounding volumes are circles (or spheres, if 3d) where an overlap test compares the distances between the centers with the sum of the radii. Another common bounding volume is a rectangular box, or a *bounding box*.<sup>14</sup> In *Dragonfly*, each Object has a bounding box encompassing the game object, representing the Object with a simplified geometry.

In *Dragonfly*, if the edges of any two boxes overlap, there is a collision. A optional refinement could be to provide an game programmer option for precise collision testing. In this case, the bounding boxes would be tested for overlap normally. If there was an overlap, then a more refined test could be done to see if the individual characters in the object sprite overlapped, indicating a collision. Depicting this, consider the two objects in Figure 4.4 on the left. The Saucer and the Hero ship are clearly not touching – boxes around each do not intersect. However, a short time later, the game world state may be as on the right. Here, the boxes do overlap. Under normal *Dragonfly* operation, this overlap itself would indicate a collision. However, looking more closely, the characters themselves do not overlap. So, an alternative option could be that if the boxes overlap, then more precise consideration of each character is done to see if there is, in fact, a collision.

Once a collision is detected, action is taken to resolve the collision. What the actions are, exactly, often depends upon what the game objects are. For example, if two billiard balls collide, then computation as to where, exactly they hit is done first, followed by computation of new velocities for both balls, accompanied by playing a “clinking” sound. As another example, if a rocket slams into a rock wall, the rocket is destroyed, the wall takes on a charred texture and an explosion animation object is created. As a third example, a character walks through an invisible wall, a magic, propagating ripple effect is triggered, but no velocities or other impacts are recorded.

In *Dragonfly*, the engine detects collisions through overlap testing of bounding boxes,

<sup>14</sup>Bounding boxes are also known as *hitboxes* since they are used to determine if an object is “hit”.





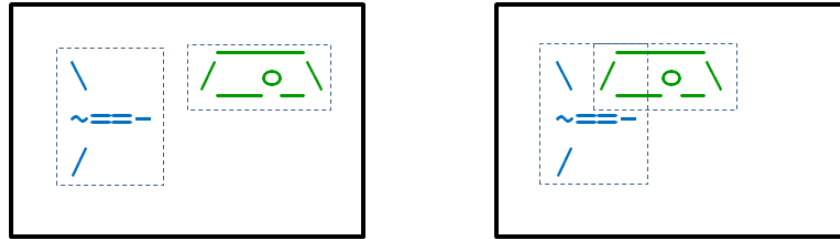


Figure 4.4: Refined collision detection with boxes

then provides a collision event to both Objects involved in the collision. Resolution involves sending the collision event to both Objects. The Objects themselves receive the event in their `eventHandler()` methods and can react appropriately.

#### 4.10.1.1 Collidable Entities

However, not all Objects are collidable entities. Consider display elements on the screen, such as player score or indication of time left. In most cases, these objects do not collide with, say, the player avatar. Similarly, background objects that provide scenery, such as the Stars in Saucer Shoot (Section 3.3), do not collide with other game elements. To support this, *Dragonfly* has a notion of “solidness”, with three states defined via an `enum`, as in Listing 4.97. This `enum` is defined in `Object.h`.

Listing 4.97: Solidness states

```
0 enum Solidness {
1     HARD,      // Object causes collisions and impedes.
2     SOFT,      // Object causes collisions, but doesn't impede.
3     SPECTRAL,  // Object doesn't cause collisions.
4 };
```

Collisions only happen between solid Objects – Objects that are `SPECTRAL` cannot collide. So, for example, Saucer Shoot’s Stars are `SPECTRAL` and not solid. Solid Objects are either `HARD` or `SOFT`. A `HARD` object cannot occupy the same space as another other `HARD` Object, but any number of `SOFT` Objects can occupy a space, with our without at most one `HARD` Object.

In addition to notifying colliding solid (`HARD` or `SOFT`) Objects of the event, resolution then disallows two `HARD` objects to occupy the same space. Basically, if a first solid Object moves onto (collides with) a second `HARD` Object, the first object is moved back to its original location – the collision still happens, but the movement does not take place.

The Object class extensions needed to support solidness are shown in Listing 4.98. The attribute `solidness` determines if the Object is treated as solid (`HARD` or `SOFT`) or non-solid (`SPECTRAL`). By default, Objects should be `HARD` (set in the Object constructor). The method `isSolid()` provides a convenient boolean test for whether or not the Object is solid. Methods to get and set the solidness are provided in `getSolidness()` and `setSolidness()`, respectively.



Listing 4.98: Object class extensions to support solidness

```

0 private:
1     Solidness m_solidness; // Solidness of object.
2
3 public:
4     bool isSolid() const; // True if HARD or SOFT, else false.
5
6     // Set object solidness, with checks for consistency.
7     // Return 0 if ok, else -1.
8     int setSolidness(Solidness new_solid);
9
10    // Return object solidness.
11    Solidness getSolidness() const;

```

#### 4.10.1.2 Collision Event

When *Dragonfly* detects a collision, it sends a collision event to each Object involved. Listing 4.99 provides the header file for the *EventCollision* class, derived from the *Event* class (Listing 4.48 on page 90). Like other event classes, the *EventCollision* is a “container” and does not have any significant functionality itself. The attributes store information about the collision: *pos* stores the position where the collision occurred; *p\_obj1* is a pointer to the Object moving, the one causing collision; and *p\_obj2* is a pointer to the Object being collided with. Methods are provided to get and set each of the attributes.

Listing 4.99: EventCollision.h

```

0 #include "Event.h"
1 #include "Object.h"
2
3 const std::string COLLISION_EVENT = "df::collision";
4
5 class EventCollision : public Event {
6
7 private:
8     Vector m_pos; // Where collision occurred.
9     Object *m_p_obj1; // Object moving, causing collision.
10    Object *m_p_obj2; // Object being collided with.
11
12 public:
13    // Create collision event at (0,0) with o1 and o2 NULL.
14    EventCollision();
15
16    // Create collision event between o1 and o2 at position p.
17    // Object o1 'caused' collision by moving into object o2.
18    EventCollision(Object *p_o1, Object *p_o2, Vector p);
19
20    // Set object that caused collision.
21    void setObject1(Object *p_new_o1);
22
23    // Return object that caused collision.
24    Object *getObject1() const;
25
26    // Set object that was collided with.

```



```

27 void setObject2(Object *p_new_o2);
28
29 // Return object that was collided with.
30 Object *getObject2() const;
31
32 // Set position of collision.
33 void setPosition(Vector new_pos);
34
35 // Return position of collision.
36 Vector getPosition() const;
37 };

```

#### 4.10.1.3 Collisions in the WorldManager

**Dragonfly** handles collisions inside the **WorldManager**. In order to do this, the **WorldManager** needs to be extended with several methods. The first method is **isCollision()** which tests whether or not an **Object** has a collision at a given location. If so, all **Objects** at that location are returned in an **ObjectList** (there can be more than one **Object** at a location since multiple **SOFT** **Objects** can occupy the same location, with up to one **HARD** **Object**). The method **moveObject()** moves an **Object** to another location, as long as there is not a collision between two **HARD** **Objects** at this location.

Listing 4.100: **WorldManager** extensions for collision support

```

0 public:
1 // Return list of Objects collided with at position 'where'.
2 // Collisions only with solid Objects.
3 // Does not consider if p_o is solid or not.
4 ObjectList isCollision(Object *p_o, Vector where) const;
5
6 // Move Object.
7 // If collision with solid, send collision events.
8 // If no collision with solid, move ok else don't move Object.
9 // If Object is Spectral, move ok.
10 // Return 0 if move ok, else -1 if collision with solid.
11 int moveObject(Object *p_o, Vector where);

```

An additional utility function (in **utility.h** and **utility.cpp**) called **positionsIntersect()** is helpful for the **WorldManager** **isCollision()** method, as shown in Listing 4.101. Basically, if two **Positions** are the same (their x coordinates are within 1 space of each other *and* their y coordinates are within 1 space of each other) then the method returns **true**, otherwise it returns **false**. This method may seem somewhat trivial, but it serves as a placeholder for method introduced in Section 4.132 on page 162, **boxIntersectsBox()**, that will test whether or not two bounding boxes overlap.

Listing 4.101: Utility **positionsIntersect()**

```

0 // Return true if two positions intersect, else false.
1 bool positionsIntersect(Vector p1, Vector p2)
2 if abs(p1.getX() - p2.getX()) <= 1 and
3     abs(p1.getY() - p2.getY()) <= 1
4     return true

```



```

5   else
6       return false
7   end if

```

The WorldManager `isCollision()` method is provided in Listing 4.102. Since the method returns a list of all Objects that are collided with, line 6 creates an empty list (`collision_list`) to start. The next block of code iterates through all the Objects in the game. Each Object is first checked if it is the same Object passed into `isCollision()` – if it is, it is ignored since an Object cannot collide with itself. If it is not, then the Object is checked to see if it is at the same location, and that it is solid. If both of these are true, the Object is added to the `collision_list`. When the loop is finished, the list of all Objects collided with are returned in `collision_list`. Note, if no Objects have been collided with, `collision_list` is an empty list.

Listing 4.102: WorldManager `isCollision()`

```

0 // Return list of Objects collided with at position 'where'.
1 // Collisions only with solid Objects.
2 // Does not consider if p_o is solid or not.
3 ObjectList isCollision(const Object *p_o, Vector where) const
4
5 // Make empty list.
6 ObjectList collision_list
7
8 // Iterate through all objects.
9 create ObjectListIterator i on updates list
10
11 while not li.isDone() do
12
13     Object *p_temp_o = li.currentObj()
14
15     if p_temp_o is not p_o then // Do not consider self.
16
17         // Same location and both solid?
18         if positionsIntersect(p_temp_o->getPosition(), where)
19             and p_temp_o -> isSolid() then
20
21             add p_temp_o to collision_list
22
23         end if // No solid collision.
24
25     end if // Not self.
26
27     li.next()
28
29 end while // End iterate.
30
31 return collision_list

```

The `isCollision()` method is used in the `moveObject()` method, shown in Listing 4.103. The `moveObject()` method first checks if the calling Object (the one that wants to be moved) is solid (calling `isSolid()`) – if not, there are no further checks needed and the Object can move. If the Object is solid, `isCollision()` is called to produce a list of



solid Objects collided with. That collision list is iterated through,<sup>15</sup> and each Object is sent a collision event starting at line 22. If both Objects are **HARD** (line 30), then the move will not be allowed by setting **do\_move** to false in line 31. If a move is allowed (no **HARD** collisions), then the actual move happens at the end of the method, on line 47.

Listing 4.103: WorldManager moveObject()

```

0 // Move Object.
1 // If collision with solid, send collision events.
2 // If no collision with solid, move ok else don't move Object.
3 // If Object is Spectral, move ok.
4 // Return 0 if move ok, else -1 if collision with solid.
5 int WorldManager::moveObject(Object *p_o, Vector where)
6
7     if p_o -> isSolid() then // Need to be solid for collisions.
8
9         // Get collisions.
10        ObjectList list = isCollision(p_o, where)
11
12        if not list.isEmpty() then
13
14            boolean do_move = true // Assume can move.
15
16            // Iterate over list.
17            create ObjectListIterator li on list
18            while not li.isDone() do
19
20                Object *p_temp_o = li.currentObj()
21
22                // Create collision event.
23                EventCollision c(p_o, p_temp_o, where)
24
25                // Send to both objects.
26                p_o -> eventHandler(&c)
27                p_temp_o -> eventHandler(&c)
28
29                // If both HARD, then cannot move.
30                if p_o is HARD and p_temp_o is HARD then
31                    do_move = false // Can't move.
32                end if
33
34                li.next()
35
36            end while // End iterate.
37
38            if do_move is false then
39                return -1 // Move not allowed.
40            end if
41
42        end if // No collision.
43
44    end if // Object not solid.
45

```

<sup>15</sup>Note, if the collision list is empty, the iteration loop immediately ends since there is nothing to collide with.



```

46 // If here, no collision between two HARD objects so allow move.
47 p_o -> setPosition(where)
48
49 return ok // Move was ok.

```

**Disallow Movement onto Soft Objects (optional)** The construct of soft Objects allows many solid objects to reside in one location (if they are **SOFT**). This allows a solid Object to move onto a group of soft objects, generating collisions for each one in the process.

However, a game programmer may not want to allow some Objects to move onto the soft Objects. **Dragonfly** can be extended to support this functionality, first by adding an additional attribute to the Object class, shown in Listing 4.104, along with basic functions to get and set the attribute. If the attribute **no\_soft** is **true** (it should default to **false**), the Object is not allowed to move onto soft game objects (but will still generate collisions – effectively, soft game objects are treated as hard game objects in this case).

Listing 4.104: Object class extensions to support no soft

```

0 private:
1   bool no_soft;    // True if won't move onto soft objects.
2
3 public:
4   // Set 'no soft' setting (true – cannot move onto SOFT Objects).
5   void setNoSoft(bool new_no_soft=true);
6
7   // Get 'no soft' setting (true – cannot move onto SOFT Objects).
8   bool getNoSoft() const;

```

Then, the WorldManager **moveObject()** method is refactored. In Listing 31 on page 139, similar to the block of code around line 31 that indicates the Object should not move if both Objects are solid, an additional check is made if the Object being moved is soft and the Objects are soft. This is shown in Listing 4.105.

Listing 4.105: WorldManager extensions to moveObject() to support no soft

```

0 ...
1 // If object does not want to move onto soft objects, don't move.
2 if p_o->getNoSoft() and p_temp_o->getSolidness() is SOFT then
3   do_move = false
4 end if
5 ...

```

#### 4.10.1.4 World Boundaries – Out of Bounds Event

Generally, game objects expect to stay inside the game world. This is not to say that all object are on the *screen* at the same time – for example, think of a side scroller where a lot of the level is off the screen – but game objects stay in the known game world. As such, when a game object moves outside of the game world it is helpful to tell it this via an event. Of course, the object can ignore the event and stay outside the game world. But in many cases, an out of bounds object will want to take action, such as moving back inside the game world or, in the case of a bullet that would otherwise fly off forever, destroy itself.



Specifically for *Dragonfly*, when an Object inside the game world moves outside the game world, the WorldManager generates an out of bounds event, an EventOut. The move is still allowed, giving the Object freedom to stay outside of the game world should it so choose, but providing the information in the form of the event in case the Object wants to act.

Listing 4.106 provides the header file for the EventOut class. As for all *Dragonfly* events, EventOut is derived from the base Event class. Unlike some other events, EventOut only has information that the event occurred – i.e., that the object moved from inside to outside the game world. The Object itself already knows where (it has its (x,y) position) so that does not need to be indicated. The constructor sets `event_type` to `OUT_EVENT`. There are no other attributes and not additional methods since all the game programmer would need to do is consult the type (defined in the parent class, `Event::getType()`) to determine what happened.

Listing 4.106: EventOut.h

```

0 #include "Event.h"
1
2 const std::string OUT_EVENT = "df::out";
3
4 class EventOut : public Event {
5
6     public:
7         EventOut();
8 };

```

As suggested above, game world boundaries can be different than window boundaries – this functionality is supported by *Dragonfly* in subsequent development (see Section 4.13.4 on page 165). However, at this point in *Dragonfly*, the game world boundaries are determined by the window boundaries. In order to determine the boundaries of the world, the DisplayManager routines `getHorizontal()` and `getVertical()` can be used.

The WorldManager `moveObject()` is extended slightly. After a move is allowed (line 47 in Listing 4.103), the Object's position is checked against the horizontal and vertical limits obtained from the DisplayManager. If the Object is out of bounds, an EventOut object is generated and sent to the Object's event handler as in Listing 4.107.

Listing 4.107: Generate and send EventOut

```

0 ...
1 // Generate out of bounds event and send to Object.
2 EventOut ov
3 p_o -> eventHandler(&ov)
4 ...

```

Note, the WorldManager only sends an EventOut once, when the Object first moves from inside to outside the game world. If the Object moves outside then stays outside, no additional events are generated. Presumably, the Object already knows it is outside the game world upon receiving the first out event, so if it stays outside it does not need to be reminded.



## 4.11 Development Checkpoint #7 – Dragonfly Naiad!

Continue with your *Dragonfly* development by adding to your code base to create a *Dragonfly Naiad*.<sup>\*</sup> Steps:

1. Extend the Object class to support kinematics, as in Listing 4.93. Add code to Object to predict the position if it applies a step of velocity, as in Listing 4.94. Add extensions to `update()` to do the velocity step for all Objects, as in Listing 4.95.
2. Test the velocity code thoroughly by making Objects with different starting locations and different velocities. Verify visually and via logfile messages that Objects move an appropriate amount each step. Test with different velocity values (x and y), positive and negative and greater than 1 and less than 1.
3. Extend the WorldManager to support collisions, as in Listing 4.100. Implement support function `positionsIntersect()` in `utility.cpp`, as per Listing 4.101. Implement `isCollision()` in the WorldManager based on Listing 4.102. Support for solidness needs to be added to the Object class, based on Listing 4.98 and Listing 4.97. Lastly, write `moveObject()` based on Listing 4.103. EventCollision needs to be created for this, following Listing 4.99, and can be tested outside of the game engine before using it in `moveObject()`. Since the code for all the above is fairly extensive, add liberal `writeLog()` statements to provide meaningful output to verify it is working.
4. Write numerous test cases to verify that collisions work properly. Start first with a solid Object that attempts to move onto another solid Object. Verify the move is not allowed (visually and in the logfile) and make sure both Objects get an event with appropriate pointers. Next, test that soft Objects can move on each other, but that all soft Objects colliding get collision events. Lastly, check with test examples that spectral Objects do not generate collisions.
5. Create an EventOut class based on Listing 4.106. Add `EventOut.cpp` to the project and stub out each method so it compiles. Add code to the WorldManager `moveObject()` method that sends the out of bounds event to objects that move out of the game world. Refer to Listing 4.107, as needed.
6. Test the out of bounds additions by making a game object that starts inside the game world, but soon moves out. Output messages should be seen in the logfile, but events should be checked and handled by the Object `eventHandler()` method. Make different Objects that move in and out, multiple times and verify only one EventOut message is generated each time an Object moves from inside to outside.

After completing the above steps (and all the previous Development Checkpoints), you will have a fully functional game engine!

Features include:

- Objects can draw themselves in 2d, as colored text characters.

---

<sup>\*</sup> **Did you know (#7)?** A *naiad* is a Dragonfly in larval stage.





- Objects can appear above (foreground) or behind (background) when drawing.
- Objects can get input from the keyboard and mouse.
- Objects are moved automatically based on their velocities.
- Objects that move out of the game world get an “out of bounds” event.
- Objects have solidness – soft, hard, or spectral – affecting movement and collisions.
- Solid Objects that collide get a collision event, providing information on both Objects enabling them to react appropriately.

The above functionality to support objects, graphics, input, and interaction with collisions allows creation of a wide variety of games. Consider, for example, making the game Saucer Shoot from Section 3.3. The core gameplay for Saucer Shoot can be made (aside from the Points and Nuke ViewObjects), with the main exception that *Dragonfly Naiad!* only supports single character game objects without animation, rather than animated, multi-character sprites.





The individual cells in a frame are characters. These could be stored in a two dimensional array. However, in order to use the speed and efficiency of the C++ string library class, the entire frame is stored as a single string.

The definition for the Frame class is provided in Listing 4.109. While the attribute `frame_str` holds the frame data in a one dimensional list of characters, the attributes `width` and `height` determine the shape of the frame rectangle. There are two constructors: the default constructor creates an empty frame (`height` and `width` both zero with an empty `frame_str`), while the method on line 14 allows construction of a frame with an initial string of characters and a given width and height. The Frame is essentially a container, and most of the methods allow getting and setting the attributes.

Listing 4.109: Frame.h

```

0  #include <string>
1
2  class Frame {
3
4  private:
5      int m_width;           // Width of frame.
6      int m_height;          // Height of frame.
7      std::string m_frame_str; // All frame characters stored as string.
8
9  public:
10     // Create empty frame.
11     Frame();
12
13     // Create frame of indicated width and height with string.
14     Frame(int new_width, int new_height, std::string frame_str);
15
16     // Set width of frame.
17     void setWidth(int new_width);
18
19     // Get width of frame.
20     int getWidth() const;
21
22     // Set height of frame.
23     void setHeight(int new_height);
24
25     // Get height of frame.
26     int getHeight() const;
27
28     // Set frame characters (stored as string).
29     void setString(std::string new_frame_str);
30
31     // Get frame characters (stored as string).
32     std::string getString() const;
33 };

```

### 4.12.2 The Sprite Class

Sprites are sequences of frames, typically rendered such that if a sequence is drawn fast enough, it looks animated to the eye. The Sprite class in `Dragonfly` is just a repository for



the Frame data. Sprites do not know how to draw themselves, nor even what the display rate should be for the frames – that functionality is provided by the Object class. Sprites record the dimension of the Sprite (typically, the same dimension of the Frames), along with providing the ability to add and retrieve individual Frames. An Sprite sequence may look like the example in Listing 4.110.

Listing 4.110: Sprite sequence example

```

0      /_/_/_\      /_/_/_\      /_/_/_\      /_/_/_\      /_/_/_\
1      /_/_/_\      /_/_/_\      /_/_/_\      /_/_/_\      /_/_/_\

```

The Sprite class header file is shown in Listing 4.111. The class needs `Frame.h` as well as `<string>`. The attributes `width` and `height` typically mirror the sizes of the frames composing the sprite. The frames themselves are stored in an array, `frame[]`, which is dynamically allocated when the Sprite object is created. In fact, the default constructor, `Sprite()`, is `private` since it *cannot* be called – instead, Sprites must be instantiated with the maximum number of frames they can hold as an argument (e.g., `Sprite(5)`). This maximum is stored in `max_frame_count`, while the actual number of frames is stored in `frame_count`. The color, which is the color of all frames in the sprite, is stored in `color`. Each sprite can be identified by a text label, `label` – for example, “ship” for the Hero’s sprite in Saucer Shoot (Section 3.3). Most of the methods are to get and set the attributes, with `addFrame()` putting a new frame at the end of the Sprite’s `frame` array.

Listing 4.111: Sprite.h

```

0 // System includes.
1 #include <string>
2
3 // Engine includes.
4 #include "Frame.h"
5
6 class Sprite {
7
8     private:
9         int m_width;           // Sprite width.
10        int m_height;          // Sprite height.
11        int m_max_frame_count; // Max number frames sprite can have.
12        int m_frame_count;     // Actual number frames sprite has.
13        Color m_color;         // Optional color for entire sprite.
14        Frame *m_frame;        // Array of frames.
15        std::string m_label;    // Text label to identify sprite.
16        Sprite();              // Sprite always has one arg.
17
18     public:
19        // Destroy sprite, deleting any allocated frames.
20        ~Sprite();
21
22        // Create sprite with indicated maximum number of frames.
23        Sprite(int max_frames);
24
25        // Set width of sprite.
26        void setWidth(int new_width);
27

```



```

28 // Get width of sprite.
29 int getWidth() const;
30
31 // Set height of sprite.
32 void setHeight(int new_height);
33
34 // Get height of sprite.
35 int getHeight() const;
36
37 // Set sprite color.
38 void setColor(Color new_color);
39
40 // Get sprite color.
41 Color getColor() const;
42
43 // Get total count of frames in sprite.
44 int getFrameCount() const;
45
46 // Add frame to sprite.
47 // Return -1 if frame array full, else 0.
48 int addFrame(Frame new_frame);
49
50 // Get next sprite frame indicated by number.
51 // Return empty frame if out of range [0, frame_count].
52 Frame getFrame(int frame_number) const;
53
54 // Set label associated with sprite.
55 void setLabel(std::string new_label);
56
57 // Get label associated with sprite.
58 std::string getLabel() const;
59 };

```

The Sprite constructor is shown in Listing 4.112. The width, height and frame count are initialized to zero. The `frame` array is allocated by `new` to the indicated size. Like all memory allocation, this should be checked for success – if the needed memory cannot be allocated (not shown), an error message is written to the logfile and the maximum frame count is set to 0. The Sprite should initially have the default color, `COLOR_DEFAULT`, as defined in `Color.h` (Listing 4.69 on page 106).

Listing 4.112: Sprite Sprite()

```

0 // Create sprite with indicated maximum number of frames.
1 Sprite::Sprite(int max_frames)
2     set frame_count to 0
3     set width to 0
4     set height to 0
5     frame = new Frame [max_frames]
6     set max_frame_count to max_frames
7     set color to COLOR_DEFAULT

```

The Sprite destructor is shown in Listing 4.113. The only logic the destructor has is to check that frames are actually allocated (`frame` is not `NULL`) and, if so, `delete` the `frame` array.



Listing 4.113: Sprite ~Sprite()

```

0 // Destroy sprite, deleting any allocated frames.
1 Sprite::~Sprite()
2     if frame is not NULL then
3         delete [] frame
4     end if

```

Once a Sprite is created, frames are typically added to it one at a time until the entire animation sequence has all been added. Pseudo code for Sprite `addFrame()`, which adds one Frame, is shown in Listing 4.114. The method first checks if the array is filled – if so, an error is returned. Otherwise, the new frame is added and the frame count is incremented. Remember, as in all C++ arrays, the index of the first item is 0, not 1.

Listing 4.114: Sprite addFrame()

```

0 // Add a frame to the sprite.
1 // Return -1 if frame array full, else 0.
2 int Sprite::addFrame(Frame new_frame)
3     if frame_count is max_frame_count then // Is Sprite full?
4         return error
5     else
6         frame[frame_count] = new_frame
7         increment frame_count
8     end if

```

Sprite `getFrame()` is shown in Listing 4.115. The first block of code checks if the frame number is outside of the range `[0, frame_count-1]` – if so, an empty frame is returned. Otherwise, the frame at the index indicated by `frame_count` is returned.

Listing 4.115: Sprite getFrame()

```

0 // Get next sprite frame indicated by number.
1 // Return empty frame if out of range [0, frame_count].
2 Frame Sprite::getFrame(int frame_number) const
3
4     if ((frame_number < 0) or (frame_number >= frame_count)) then
5         Frame empty // Make empty frame.
6         return empty // Return it.
7     end if
8
9     return frame[frame_number]

```

### 4.12.3 The ResourceManager

With data structures for frames and sprites in place, a manager to handle resources is needed – the ResourceManager. The ResourceManager is a singleton derived from Manager, with private constructors and a `getInstance()` method to return the one and only instance (see Section 4.2.1 on page 53). The header file, including class definition, is provided in Listing 4.116.

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



Listing 4.116: ResourceManager.h

```

0 // System includes.
1 #include <string>
2
3 // Engine includes.
4 #include "Manager.h"
5 #include "Sprite.h"
6
7 // Maximum number of unique assets in game.
8 const int MAX_SPRITES = 1000;
9
10 // Delimiters used to parse Sprite files -
11 // the ResourceManager 'knows' file format.
12 const std::string FRAMES_TOKEN = "frames";
13 const std::string HEIGHT_TOKEN = "height";
14 const std::string WIDTH_TOKEN = "width";
15 const std::string COLOR_TOKEN = "color";
16 const std::string END_FRAME_TOKEN = "end";
17 const std::string END_SPRITE_TOKEN = "eof";
18
19 class ResourceManager : public Manager {
20
21 private:
22     ResourceManager(); // Private (a singleton).
23     ResourceManager(ResourceManager const&); // Don't allow copy.
24     void operator=(ResourceManager const&); // Don't allow assignment.
25     Sprite *m_p_sprite[MAX_SPRITES]; // Array of sprites.
26     int m_sprite_count; // Count of number of loaded sprites.
27
28 public:
29     // Get the one and only instance of the ResourceManager.
30     static ResourceManager &getInstance();
31
32     // Get ResourceManager ready to manager for resources.
33     int startUp();
34
35     // Shut down ResourceManager, freeing up any allocated Sprites.
36     void shutDown();
37
38     // Load Sprite from file.
39     // Assign indicated label to sprite.
40     // Return 0 if ok, else -1.
41     int loadSprite(std::string filename, string label);
42
43     // Unload Sprite with indicated label.
44     // Return 0 if ok, else -1.
45     int unloadSprite(std::string label);
46
47     // Find Sprite with indicated label.
48     // Return pointer to it if found, else NULL.
49     Sprite *getSprite(std::string label) const;
50 };

```

The ResourceManager uses strings for labels so needs `#include <string>`. In addition, it inherits from `Manager.h` and has methods and attributes to handle Sprites, so also



`#includes Sprite.h.`

The ResourceManager stores Sprites in an array of pointers (the attribute `p_sprite[]`), bounded by `MAX_SPRITES`. The other `consts` are delimiters used to parse the sprite files – the ResourceManager “understands” the file format and uses the delimiters as tokens during parsing. See Listing 3.3 on page 17 for an example of a sprite file. For parsing, `frames`, `height`, `width` and `color` define parameters for the sprite, while `end` and `eof` are used to mark the end of each frame and the end of the file, respectively.

The method `startUp()` gets the ResourceManager ready for use – basically, just setting `sprite_count` to 0 and calling `Manager::startUp()`. The complement, `shutDown()`, frees up any allocated Sprites (e.g., `delete p_sprite[1]`) and calls `Manager::shutDown()`.

The method `loadSprite()` reads in a sprite from a file indicated by `filename`, stores it in a Sprite and associates a label string with it. The method `unloadSprite()` unloads a Sprite with a given `label` from memory, freeing it up. The method `getSprite()` returns a pointer to a Sprite with the given label.

The ResourceManager `loadSprite()` is shown in Listing 4.117. The name of the sprite file is passed in as a string (`filename`), along with a string with the label name to associate with the sprite once it is successfully loaded (`label`).

Listing 4.117: ResourceManager `loadSprite()`

```

0 // Load Sprite from file.
1 // Assign indicated label to sprite.
2 // Return 0 if ok, else -1.
3 int ResourceManager::loadSprite(std::string filename, std::string label)
4     open file filename
5     read header
6     new Sprite (with frame count)
7     while not eof do
8         read frame
9         add frame to Sprite
10    end while
11    close file
12    add label to Sprite

```

The method proceeds by opening the file indicated by `filename`. After that, the header is read and parsed. Once the number of frames is known from the header, on line 6 a new Sprite is created (e.g., if the sprite has 5 frames, then `new Sprite(5)`). Then, the method loops through the rest of the file, reading a frame at a time and adding it to the Sprite. When the `eof` token is found (actually, `END_SPRITE_TOKEN`), the parsing stops and the loop exits. The final step on line 12 is to associate `label` with the Sprite.

Note, error checking should be done throughout, looking at file format, length of line, number of lines, frame count and general file read errors. If any errors are encountered, the line number in the file should be reported along with a descriptive error in the logfile. Listing 4.118 shows an example of a possible error message. In this example, line 12 of the file has `“/o___\”` which is 7 characters (there is an extra ‘ ’ at the end, a common error), while the header file had indicated the width was only 6. Making the error message as descriptive as possible is helpful to game programmers that may have “bugs” in their sprite files. Upon encountering an error, all resources should be cleaned up (i.e., `delete` the Sprite and close the file), as appropriate.





Listing 4.118: Example error reported when parsing Sprite file

```

0 Error line 12. Line width 7, expected 6.
1 Line is: "/o---\ "

```

Coding the `loadSprite()` method is much easier with a few “helper” functions, shown in Listing 4.119. Function `readLineInt()` reads a single line from a file and expects it to be in the form of “tag number” and returns the number (e.g., the line “frames 5” called with a tag of “frames” would return 5). Function `readLineStr()` does the same thing, but expects `number` to be a string, returning the string found. Function `readFrame()` reads a frame of a given width and height from a file, returning the `Frame`. All three methods take in the line number so that if they encounter an error, they can report (in the logfile) on which line number the error occurred. They use a pointer to the line number so they can increment it for each line read. None of these methods are part of the `ResourceManager`, but rather are stand alone utility functions. They are not general engine utility functions either (i.e., it is unlikely a game programmer would ever use them), so do not really belong in `utility.cpp`. Instead, they can be placed directly into `ResourceManager.cpp`.

Listing 4.119: ResourceManager helper functions for loading sprites

```

0 // Read single line 'tag number', return number (e.g., "frames 5").
1 int readLineInt(ifstream *p_file, int *p_line_num,
2               const char *tag);
3
4 // Read single line 'tag string', return string (e.g., "color green").
5 std::string readLineStr(ifstream *p_file, int *p_line_num,
6                       const char *tag);
7
8 // Read frame until 'eof', return Frame.
9 Frame readFrame(ifstream *p_file, int *p_line_number,
10               int width, int height);

```

Function `readLineInt()` is shown in Listing 4.120. The function reads one line from the file into the string `line`. Note, error checking (`p_file -> good()` needs to be done on this operation. The `if` conditional uses `line.compare()` to see if the `tag` is the same as the first part of the `line` (e.g., `line.compare(0, strlen(tag), tag)`). If the tag is not the one expected, an error is returned. If the tag is the one expected, the remainder of the string after the tag is converted into an integer using `atoi()` – e.g.,

```

0 atoi(line.substr(strlen(tag)+1).c_str())

```

The number extracted with `atoi()` is returned. The line number in the file should be appropriately incremented.

Listing 4.120: `readLineInt()`

```

0 // Read single line of form 'tag number', return number.
1 int readLineInt(ifstream *p_file, int *p_line_number,
2               const char *tag)
3
4     string line
5
6     getline() into line

```



```

7  if not line.compare(line, tag) then
8      return error // Not right tag.
9  end if
10
11  number = atoi() on line.substr()
12
13  increment line number
14
15  return number

```

The same logic is used for `readLineStr()` with the exception that the final string after the tag is not converted to an integer, but is instead just returned (e.g., `line.substr(strlen(tag) + 1)`).

The method `readFrame()` is shown in Listing 4.121. The function is provided the height of the frame via `height` as a parameter. So, via a `for` loop, it loops for a count of the height of the frame, reading in a line at a time. Each line represents one row of the frame. If any line is not the right width (also passed in as a parameter, via `width`), an error is returned in the form of an “empty” Frame. If the frame is read in successfully, an additional line is read from the file, shown on line 16. Since the frame is over, this line should be `END_FRAME_TOKEN` (“end”), otherwise there is an error in the file format. Note, error checking (`p_file -> good()`) needs to be done on the file read operations. Errors of any kind should result in an error code (empty Frame) returned by the function. If line 21 is reached, the frame has been read and parsed successfully, so a Frame containing the frame is created and returned. The line number should be used to report (in the logfile) where any errors occurred in the input file, and should be appropriately incremented as the parsing progresses.

Listing 4.121: `readFrame()`

```

0  // Read frame until 'eof', return Frame.
1  Frame readFrame(ifstream *p_file, int *p_line_number, int width, int height
2  )
3
4  string line, frame_str
5
6  for i = 1 to height
7
8      getline() into line // Error check.
9      if line width != width then
10         return error (empty Frame)
11     end if
12
13     frame_str += line
14
15 end for
16
17 getline() into line // Error check.
18 if line is not END_FRAME_TOKEN then
19     return error (empty Frame)
20 end if
21
22 create frame (width, height, frame_str)
23
24 return frame

```



**Dragonfly** should run on both Windows, Linux or Mac. Unfortunately, text files are treated slightly differently on Windows versus Linux and Mac. In Windows text files, the end of each line has a newline (`'\n'`) and a carriage return (`'\r'`), while in Unix and Mac, the end of each line only has a newline. In order to allow **Dragonfly** to work with text files created on either operating system, pseudo code for an optional utility, `discardCR()`, is shown in Listing 4.122. A `string`, typically just read in from a file, is passed in via reference (`&str`). The function examines the last character of this string and, if it is a carriage return, it removes it via `str.erase()`.

Listing 4.122: `discardCR()` (optional)

```

0 // Remove '\r' from line (if there - typical of Windows).
1 void discardCR(std::string &str)
2   if str.size() > 0 and str[str.size()-1] is '\r' then
3     str.erase(str.size() - 1)
4   end if

```

Once in place, `discardCR()` can be called each time after reading a line from a file.

The complement of `loadSprite()` is `unloadSprite()`, which is much simpler. `unloadSprite()` is shown in Listing 4.123. The method loops through the Sprites in the ResourceManager. If the label being looked for (`label`) matches the label of one of the Sprites (`getLabel()`) then that is the Sprite to be unloaded. The Sprite's memory is deleted via `delete`. Then, in a loop starting on line 11, the rest of the Sprites in the array are moved down one. Since one Sprite was unloaded, the sprite count is decremented on line 11. If the loop terminates without a label match, the sprite to be unloaded is not in the ResourceManager and an error is returned.

Listing 4.123: ResourceManager `unloadSprite()`

```

0 // Unload Sprite with indicated label.
1 // Return 0 if ok, else -1.
2 int ResourceManager::unloadSprite(std::string label)
3
4   for i = 0 to sprite_count-1
5
6     if label is p_sprite[i] -> getLabel() then
7
8       delete p_sprite[i]
9
10      // Scoot over remaining sprites.
11      for j = i to sprite_count-2
12        p_sprite[j] = p_sprite[j+1]
13      end for
14
15      decrement sprite_count
16
17      return success
18
19    end if
20
21  end for
22
23  return error // Sprite not found.

```



The final method needed by the ResourceManager is `getSprite()`, with pseudo code show in Listing 4.124. The method loops through all the Sprites in the ResourceManager. The first Sprite that matches the label `label` is returned. If line 10 is reached, the label was not found and an error (`NULL`) is returned.

Listing 4.124: ResourceManager `getSprite()`

```

0 // Find Sprite with indicated label.
1 // Return pointer to it if found, else NULL.
2 Sprite *ResourceManager::getSprite(std::string label) const
3
4 for i = 0 to sprite_count-1
5     if label is p_sprite[i] -> getLabel() then
6         return p_sprite[i]
7     end if
8 end for
9
10 return NULL // Sprite not found.

```

#### 4.12.4 Development Checkpoint #8!

Continue *Dragonfly* development. Steps:

1. Make the Frame class, referring to Listing 4.109. Add `Frame.cpp` to the project and stub out each method so it compiles. Implement and test the Frame class outside of any game engine components, making sure it can be loaded with different strings and frame dimensions.
2. Make the Sprite class, referring to Listing 4.111. Add `Sprite.cpp` to the project and stub out each method so it compiles. Code and test the simple attributes first (`ints` and `label`).
3. Implement the constructor `Sprite(int max_frames)` next, allocating the array. Implement `addFrame()` based on Listing 4.114 and `getFrame()` based on Listing 4.115. Test that you can create Sprites of different numbers of frames and add individual Frames to them. Be sure the upper limit on frame count is protected. Testing should be done outside of the other engine components, and Frames can be arbitrary strings at this point.
4. Make the ResourceManager, as a singleton (described in Section 4.2.1 on page 53), referring to Listing 4.116. Add `ResourceManager.cpp` to the project and stub out each method so it compiles.
5. Build helper functions from Listing 4.119, using logic from Listing 4.120 and Listing 4.121. Test each helper function thoroughly, outside of the ResourceManager or any other engine code. Use Sprite files from Saucer Shoot (e.g., the Saucers (Section 3.3.2 on page 16)) rather than custom sprites. Purposefully introduce errors – to the headers, frames, and frame count – and verify that the helper functions catch all errors and report helpful error messages in the logfile on the right lines.



6. Return to the ResourceManager and implement `loadSprite()` based on Listing 4.117, using the helper functions. Test thoroughly with a variety of Sprite files, verifying working code through logfile messages.
7. Implement `getSprite()` based on Listing 4.124 and `unloadSprite()` based on Listing 4.123. Test each method thoroughly. Write test code that uses all the ResourceManager methods, loading a Sprite, getting frames, and unloading a Sprite. Repeat for multiple sprites.



### 4.12.5 Using Sprites

At this point, the game programmer can load Sprites into the ResourceManager in a few simple steps. The first step is to create a sprite file, such as the one in Listing 3.3 on page 17. The second is to load the sprite into the ResourceManager so the game can make use of it. Example code to load the saucer sprite for Saucer Shoot (Section 3.3) is shown in Listing 3.2 on page 16.

To actually use Sprites, say to draw them in an animated fashion on the window, *Dragonfly* needs to be extended in a couple of ways. Up to now, the DisplayManager can only draw a single character on the window. While characters are the basic drawing unit for *Dragonfly*, many Objects have a frame of animation as a basic drawing unit. So, for convenience, the DisplayManager is extended to be able to draw a frame. The pseudo code for `drawFrame()` is shown in Listing 4.125 and a public method prototype is added to *DisplayManager.h*.

Listing 4.125: DisplayManager drawFrame()

```

0 // Draw single sprite frame at window location (x,y) with color.
1 // If centered true then center frame at (x,y).
2 // Return 0 if ok, else -1.
3 int DisplayManager::drawFrame(Vector world_pos, Frame frame,
4                               bool centered, Color color) const;
5
6 // Error check empty string.
7 if frame is empty then
8     return error
9 end if
10
11 // Centered? Then offset (x,y) by 1/2 (width,height).
12 if centered then
13     x_offset = frame.getWidth() / 2
14     y_offset = frame.getHeight() / 2
15 else
16     x_offset = 0
17     y_offset = 0
18 end if
19
20 // Frame data stored in string.
21 std::string str = frame.getString()
22
23 // Draw row by row, character by character.
24 for y = 0 to frame.getHeight()-1
25     for x = 0 to frame.getWidth()-1
26         Vector temp_pos(world_pos.getX() - x_offset + x,
27                         world_pos.getY() - y_offset + y)
28         drawCh(temp_pos, str[y * frame.getWidth() + x], color)
29     end for // x
30 end for // y

```

The first block of code starting on line 6 checks if the Frame is empty to avoid subsequent parsing errors. This can be checked with the `empty()` method call on the Frame string and, if true, an error (-1) is returned.

Next, a Sprite needs to be associated with a game Object. This means extending the



Object class to have additional private attributes to handle Sprites, shown in Listing 4.126. The attribute `p_sprite` indicates what Sprite is associated with the Object. Most Sprites are drawn centered on the Object position, but this is just an option specified by `sprite_center`. In the normal course of animation, drawing proceeds sequentially through all the frames in a sprite until the end, then loops. The attribute `sprite_index` keeps track of which frame is currently being drawn. By default, a sprite frame is advanced sequentially each game loop. However, for many animations, this will be too fast. In order to slow down the animation, the attribute `sprite_slowdown` provides a slowdown rate. For example, a slowdown of 5 would mean the animation is only advanced by 1 frame for every 5 steps of the game loop. A slowdown of 1 means no slowdown, and a slowdown of 0 has a special meaning, to stop the animation altogether. The attribute `sprite_slowdown_count` is a counter used in conjunction with the slowdown rate to provide animation through cycling the frames. Methods to get and set each attribute are also provided. The `setSprite()` methods also sets the bounding box for the Object (described in the upcoming Section 4.13.2).

Listing 4.126: Object class extensions to support Sprites

```

0 private:
1   Sprite *p_sprite;    // Sprite associated with object.
2   bool sprite_center;  // True if sprite centered on object.
3   int  sprite_index;   // Current index frame for sprite.
4   int  sprite_slowdown; // Slowdown rate (1 = no slowdown, 0 = stop).
5   int  sprite_slowdown_count; // Slowdown counter.
6
7 public:
8   // Set Object Sprite to new one.
9   // If set_box is true, set bounding box to size of Sprite.
10  // Set sprite index to 0 (first frame).
11  void setSprite(Sprite *p_new_sprite, bool set_box=true);
12
13  // Return pointer to Sprite associated with this Object.
14  Sprite *getSprite() const;
15
16  // Set Sprite to be centered at Object position.
17  void setCentered(bool centered=true);
18
19  // Indicates if sprite is centered at Object position.
20  bool isCentered() const;
21
22  // Set index of current Sprite frame to be displayed.
23  void setSpriteIndex(int new_sprite_index);
24
25  // Return index of current Sprite frame to be displayed.
26  int  getSpriteIndex() const;
27
28  // Slows down sprite animations.
29  // Sprite slowdown is in multiples of GameManager frame time.
30  void setSpriteSlowdown(int new_sprite_slowdown);
31  int  getSpriteSlowdown() const;
32  void setSpriteSlowdownCount(int new_sprite_slowdown_count);
33  int  getSpriteSlowdownCount() const;
34
35  // Draw single sprite frame.

```



```

36 // Drawing accounts for center & slowdown, and advances Sprite frame.
37 virtual void draw();

```

In addition, while the `draw()` method has previously been defined in Listing 4.75 on page 112, it is currently empty. Up until now, game objects needed to define their own `draw()` methods to display something on the window. But with a Sprite now associated with an Object, the `draw()` method can now be defined to draw the animated sprite. Basically, this involves a call to DisplayManager `drawFrame()`, then advancing the sprite index to the next frame. If done strictly this way, the rate of change on the animation will be one frame every game loop (so, 30 updates per second). Many 2d animations are not designed to progress that fast – basically, they want a lower frame rate. So, the `draw()` method allows the game programmer to control the rate the sprite animation progresses through its frames.

Object `draw()` is shown in Listing 4.127. Line 12 asks the DisplayManager to draw the current frame at the indicate position and in the indicated sprite color.

The block of code at line 16 checks if the sprite slowdown value is set to 0 – if so, this indicates the animation is frozen, not to be advanced, so the method is done.

Otherwise, the slowdown counter is advanced and on line 25 checked against the slowdown value to see if it is time to advance the sprite frame. Advancing increments the index, with the code starting at line 32 taking care of looping from the end of the animation sequence to the beginning.

The last two actions at the end of the method set the slowdown counter and the sprite indices to their values for the next call to `draw()`.

Listing 4.127: Object `draw()`

```

0 // Draw single sprite frame.
1 // Drawing accounts for: centering, slowdown, advancing Sprite Frame.
2 virtual void Object::draw()
3
4 // If sprite not defined, don't continue further.
5 if p_sprite is NULL then
6     return
7 end if
8
9 index = getSpriteIndex()
10
11 // Ask graphics manager to draw current frame.
12 DisplayManager drawFrame(pos, p_sprite->getframe(index),
13                           sprite_center, p_sprite->getColor())
14
15 // If slowdown is 0, then animation is frozen.
16 if getSpriteSlowdown() is 0 then
17     return
18 end if
19
20 // Increment counter.
21 count = getSpriteSlowdownCount()
22 increment count
23
24 // Advance sprite index, if appropriate.
25 if count >= getSpriteSlowdown() then

```





```

26     count = 0 // Reset counter.
27
28     increment index // Advance frame.
29
30     // If at last frame, loop to beginning.
31     if index >= p_sprite -> getFrameCount() then
32         index = 0
33     end if
34
35
36 end if
37
38 // Set counter for next draw().
39 setSpriteSlowdownCount(count)
40
41 // Set index for next draw().
42 setSpriteIndex(index)

```

Note, `draw()` is still defined as `virtual`. This allows a derived class (a game object) to still define its own `draw()` method. In such a case, the game object's `draw()` would get called. The game programmer could write code for object-specific functionality (say, displaying a health bar above an avatar), and still call the built-in Object `draw()` explicitly, via `Object::draw()`.

#### 4.12.5.1 Transparency (optional)

In some cases, the characters making up a sprite do not occupy the full extent of their box. For example, a stick figure will have a bounding box around the whole figure, but there will be empty regions around the head, under the arms, etc. By default, `Dragonfly` will draw such blank spaces, occluding whatever characters may have been drawn below it (e.g., the background), when it may look better to not draw the blanks. For images, providing this functionality is typically done by declaring one color to be “transparent” where that color, wherever found in the image, is not rendered on the window, allowing any underlying image to be seen instead. For `Dragonfly`, transparency is done in a similar fashion, with the option of a character being specified as the transparency character – whenever this character is part of the sprite frames it is not rendered, thus not occluding any underlying characters.

In order to support drawing with transparency, the `DisplayManager drawFrame()` method must be refactored as shown in Listing 4.128 (refer to Listing 4.125 on page 156 for the original method). The refactored `drawFrame()` method takes an additional parameter indicating the transparent character, with a default value of 0 (*not* the character ‘0’) meaning no transparency. Inside the loops iterating over the frame, before a character is drawn (via `drawCh()`), it is verified that either the transparency is not 0 or the character is not the transparent character.

Listing 4.128: `DisplayManager` extension to support transparency

```

0 // Draw single sprite frame at window location (x,y) with color.
1 // If centered true, then center frame at (x,y).
2 // Don't draw transparent characters (0 means none).
3 // Return 0 if ok, else -1.
4 int DisplayManager::drawFrame(Vector world_pos, Frame frame,

```



```

5         bool centered, Color color,
6         char transparent) const;
7
8     // Draw character by character.
9     for y = 0 to frame.getHeight()-1
10        for x = 0 to frame.getWidth()-1
11            if (transparent not defined) or
12                (str[y*frame.getWidth() + x] != transparent) then
13                // drawCh normally
14                ...
15            end if
16        ...

```

The transparency character itself is an attribute of an Object. Extensions needed to the Object class are shown in Listing 4.129. Transparency is stored in a `char` attribute, `sprite_transparency` (set to 0 in the Object constructor), with methods to get and set it.

Listing 4.129: Object class extensions to support transparency

```

0 private:
1     char sprite_transparency;    // Sprite transparent character (0 if none).
2
3 public:
4     // Set Sprite transparency character (0 means none).
5     void setTransparency(char transparent=' ');
6
7     // Get Sprite transparency character (0 means none).
8     char getTransparency() const;

```

Lastly, the Object `draw()` method needs to be changed to correspond to the refactored DisplayManager `drawFrame()`, providing `getTransparency()` as a parameter.

#### 4.12.6 Development Checkpoint #9!

Continue *Dragonfly* development, getting the engine to support Sprites. Steps:

1. Extend the DisplayManager by coding `drawFrame()`, referring to Listing 4.125. Test outside of an actual Sprite by writing a game object (inherited from Object) `draw()` method that requests drawing of a variety of sizes and contents for frames.
2. Extend the Object class to support Sprites, adding attributes from Listing 4.126 and stubbing out the methods. Make sure that it compiles, first, then implement the methods to get and set the simple attributes.
3. Write the code for the revised Object `draw()` in Listing 4.127 that uses Sprites to draw. Write code for a game object (inherited from Object) that associates with a Sprite. Integrate this game object into a game and test the functionality of the Object `draw()`. Debugging can be visual (what is seen on the screen), but use logfile messages to help determine when/where there are problems.
4. Test a variety of game objects with a variety of Sprites (from the Saucer Shoot tutorial or created by hand). Verify the Sprites can be advanced, slowed down and stopped and are drawn without visual glitches. Test and debug thoroughly before proceeding.



## 4.13 Boxes

Boxes (also known as rectangles) are useful for providing a variety of 2d or 3d game features. Boxes can be used to determine the bounds of an object for collisions, as discussed in Section 4.10.1. Boxes can also be used to determine the boundary of the game world, helping detect when a game object goes out of bounds and/or off the boundary of the visual window. This latter feature is useful for when the game world is larger than what can be seen on the window, such as is typically the case for adventure-type games that feature exploring a large work, or for side-scrolling platformers. In this case, the player's view of the world is through a view window that moves with, say, the player's avatar. In order to support these features, *Dragonfly* has a Box class.

### 4.13.1 The Box Class

The definition for the Box class is provided in Listing 4.130. The Box uses a Vector attribute (*corner*) for the upper left corner, with horizontal and vertical attributes stored as integers. The default constructor creates an empty (zero width, zero height) box at (0,0). It is often useful to create a Box with given attributes, so the other constructor allows specification of position, horizontal and vertical attributes upon instantiation. Since the Box is just a container, the rest of the methods just get and set the attribute values.

Listing 4.130: Box.h

```

0  #include "Vector.h"
1
2  class Box {
3
4  private:
5      Vector m_corner;    // Upper left corner of box.
6      float m_horizontal; // Horizontal dimension.
7      float m_vertical;   // Vertical dimension.
8
9  public:
10     // Create box with (0,0) for the corner, and 0 for horiz and vert.
11     Box();
12
13     // Create box with an upper-left corner, horiz and vert sizes.
14     Box(Vector init_corner, float init_horizontal, float init_vertical);
15
16     // Set upper left corner of box.
17     void setCorner(Vector new_corner);
18
19     // Get upper left corner of box.
20     Vector getCorner() const;
21
22     // Set horizontal size of box.
23     void setHorizontal(float new_horizontal);
24
25     // Get horizontal size of box.
26     float getHorizontal() const;
27
28     // Set vertical size of box.

```



```

29 void setVertical(float new_vertical);
30
31 // Get vertical size of box.
32 float getVertical() const;
33 };

```

### 4.13.2 Bounding Boxes

Boxes are used for the “size” of an Object, also known as a *bounding box* since it bounds the borders of a game object. The bounding box determines the region an Object occupies and is used in the computation to figure out if an Object collides with another Object.

Extensions to the Object class to support bounding boxes are shown in Listing 4.131. The bounding box is stored in the `private` attribute `box`, with methods provided to get and set it.

Listing 4.131: Object class extensions to support bounding boxes

```

0 private:
1   Box box;    // Box for sprite boundary & collisions.
2
3 public:
4   // Set Object's bounding box.
5   void setBox(Box new_box);
6
7   // Get Object's bounding box.
8   Box getBox() const;

```

The Object bounding box is initialized to a unit square (a Box with horizontal and vertical of 1), but typically the game programmer wants the bounding box to be the size of the Object as drawn. Thus, by default, the `setSprite()` method from Listing 4.126 (page 157) sets `box` to the width and height of the indicated Sprite.

A major change needed to support bounding boxes regards collisions. Instead of Objects only colliding if their positions overlap, with boxes, Objects collide if their bounding boxes overlap. The idea is to replace the call to `positionsIntersect()` in the WorldManager `moveObject()` method (Listing 4.103 on page 139) with `boxIntersectsBox()`.

While there are numerous possible algorithms to test if two boxes overlap, an intuitive and fairly fast method is as follows: consider Figure 4.5, where the upper left corner of a box is (`x1`, `y1`) and the bottom right corner is (`x2`, `y2`). An overlap of box A and box B only occurs if the left edge of B is contained within the width of A *and* the top edge of box B is contained within the height of box A. If both of those are true, then the two boxes overlap, otherwise they do not. And vice versa for A within B.

Using this idea, a new function `boxIntersectsBox()` is created in `utility.cpp`, with pseudo code shown in Listing 4.132.

Listing 4.132: `boxIntersectsBox()`

```

0 // Return true if boxes intersect, else false.
1 bool boxIntersectsBox(Box A, Box B)
2
3   // Test horizontal overlap (x_overlap).

```



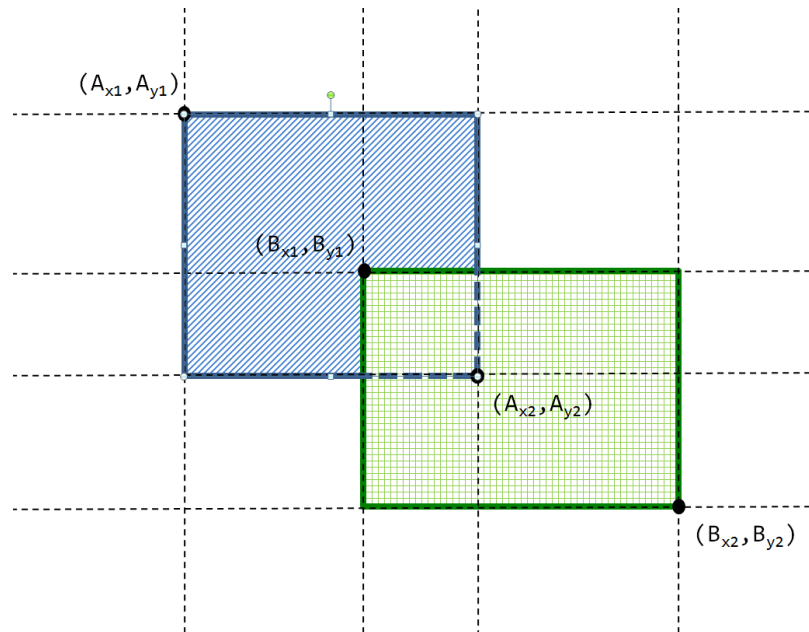


Figure 4.5: Determine if boxes overlap

```

4  Bx1 <= Ax1 && Ax1 <= Bx2      // Either left side of A in B?
5  Ax1 <= Bx1 && Bx1 <= Ax2      // Or left side of B in A?
6
7  // Test vertical overlap (y_overlap).
8  By1 <= Ay1 && Ay1 <= By2      // Either top side of A in B?
9  Ay1 <= By1 && By1 <= Ay2      // Or top side of B in A?
10
11 if (x_overlap) and (y_overlap) then
12     return true                // Boxes do intersect.
13 else
14     return false               // Boxes do not intersect.
15 end if

```

In replacing the call to `positionsIntersect()` in the WorldManager `isCollision()` method (Listing 4.102 on page 138) with `boxIntersectsBox()`, it is important to remember that the bounding boxes for Objects are relative to the Objects themselves. For example, the top left corner of a bounding box for an Object with a 1-character Sprite is (0,0) and the top left corner of a bounding box for a 3x3 character Sprite (centered on the Object) is (-1,-1). Neither of these boxes are in terms of the game world coordinates.

In order to compute collisions, the bounding box position needs to be converted to a game world position. A useful utility (in `utility.cpp`) for this conversion is `getWorldBox()` that converts the bounding box positioned relative to an Object to box positioned relative to the game world.

Listing 4.133: `getWorldBox()`

```

0 // Convert relative bounding Box for Object to absolute world Box.
1 Box getWorldBox(const Object *p_o)

```



```

2
3   Box temp_box = p_o -> getBox()
4   Vector corner = box.getCorner()
5   corner.setX(corner.getX() + p_o -> getPosition().getX())
6   corner.setY(corner.getY() + p_o -> getPosition().getY())
7   temp_box.setCorner(corner)
8
9   return temp_box

```

In addition, a close relative can be made that converts a relative bounding Box for an Object to absolute world Box at position **where**.

```

0 // Convert relative bounding Box for Object to absolute world Box.
1 Box getWorldBox(const Object *p_o, Vector where)

```

For ease of implementation, the first `getWorldBox()` can all the second, providing `p_o->getPosition()` as the argument to **where**.

Once created, collision detection in the WorldManager `isCollision()` method is modified as in Listing 4.134.

Listing 4.134: WorldManager `isCollision()` with bounding boxes

```

0 // Return list of Objects collided with at position 'where'.
1 // Collisions only with solid Objects.
2 // Does not consider if p_o is solid or not.
3 ObjectList isCollision(const Object *p_o, Vector where) const
4 ...
5
6 // World position bounding box for object at where
7 Box b = getWorldBox(p_o, where)
8
9 // World position bounding box for other object
10 Box b_temp = getWorldBox(p_temp_o)
11
12 if boxIntersectsBox(b, b_temp) and p_temp_o->isSolid() then
13 ...

```

### 4.13.3 Utility Functions (optional)

The function `boxIntersectsBox()` is not only helpful for **Dragonfly** in detecting collisions, it can be a generally useful utility for a game programmer. The name suggests other utilities shown in Listing 4.135 that are not necessarily used by the game engine but can be used by game programmers. On line 1 is a simple function that tests whether a value lies between the other two, useful in computing whether or not two boxes intersect (see Listing 4.132). On Line 4 is a function that converts the relative bounding box of an Object along with its position into a box placed in the world (see Listing 4.133).

Listing 4.135: Utility functions

```

0 // Return true if value is between min and max (inclusive).
1 bool valueInRange(float value, float min, float max);
2
3 // Convert relative bounding Box for Object to absolute world Box.

```



```

4 Box getWorldBox(const Object *p_o);
5 Box getWorldBox(const Object *p_o, Vector where);
6
7 // Return true if Box contains Position.
8 bool boxContainsPosition(Box b, Vector p);
9
10 // Return true if Line segments intersect.
11 // (Parallel line segments don't intersect).
12 bool lineIntersectsLine(Line line1, Line line2);
13
14 // Return true if Line intersects Box.
15 bool lineIntersectsBox(Line line, Box b);
16
17 // Return true if Circle intersects or contains Box.
18 bool circleIntersectsBox(Circle circle, Box b);
19
20 // Return distance between any two positions.
21 float distance(Vector p1, Vector p2);

```

The rest of the utility function prototypes shown in Listing 4.135 are not needed for *Dragonfly*, but may be useful to game programmers. Lines 7 through 18 are variations of the `boxIntersectsBox()` function, but with different shapes.<sup>16</sup> On Line 21 is a function that returns the distance between any two positions.

#### 4.13.4 Views

In a game like *Pac-Man*, the entire game board is visible on the window. However, there are many games where this is not the case, games in which the game world is larger than what can be seen on the window. Think of a game where the player explores a gameworld, too vast to be contained to one window. In such a case, the window acts as a “viewport” over the world, with the camera moving to show different world views in response to player actions. Sometimes, the camera will move with an avatar, say, keeping the avatar in the window as the world behind it moves. This is what happens in a platformer such as *Super Mario*, where the player controls the main avatar (Mario), jumping and falling vertically and running left and right in a large game world while the camera follows the avatar. Similarly, in the case of an adventure game such as *The Legend of Zelda*, the player controls the main avatar (Link), exploring a very large world in the course of rescuing the princess. Omnipresent games, where the player has a top-down view of part of the game world, such as is the case of many real time strategy games, have the camera show part of the game world on the window while the entire game world is much larger. For *Dragonfly*, since it uses text-based cells, the view afforded by the camera is limited to the size of the initial window. When the game world is larger than this window, the game engine needs to map the world coordinates to the window coordinates.

Extensions to the `WorldManager` to support views are shown in Listing 4.136. The limit provided by the terminal window is called the *view boundary* and the limit provided by the game world is called the *world boundary*. Both boundaries are stored as `Box` attributes, privately kept in the `WorldManager`. Methods to get and set the boundaries are provided.

<sup>16</sup>Classes for `Line` and `Circle` are not provided in this book, but are left as exercises for the aspiring programmer.



By default, in the `WorldManager` constructor, the size of both boundaries, length and width, should be set to 0.

Listing 4.136: `WorldManager` extensions to support views

```

0 private:
1   Box boundary;    // World boundary.
2   Box view;        // Player view of game world.
3
4 public:
5   // Set game world boundary.
6   void setBoundary(Box new_boundary);
7
8   // Get game world boundary.
9   Box getBoundary() const;
10
11  // Set player view of game world.
12  void setView(Box new_view);
13
14  // Get player view of game world.
15  Box getView() const;

```

The `GameManager` sets the default world boundary and the view boundary to be the size of the initial window, obtained from the `DisplayManager` via `getHorizontal()` and `getVertical()` (see Section 4.8.2 on page 106).<sup>\*</sup> The `GameManager` does this in its own `startUp()` method, after both the `DisplayManager` and `WorldManager` have been successfully started.

The world boundary as a box provides an immediate opportunity to refactor the “out of bounds” event from Section 4.10.1.4 (page 140). Listing 4.137 depicts the new pseudo-code.

Listing 4.137: `WorldManager` `moveObject()` refactored for `EventOut`

```

0 // Move Object.
1 // ...
2 // If moved from inside world boundary to outside, generate EventOut.
3 int WorldManager::moveObject(Object *p_o, Vector where)
4
5   ...
6
7   // Do move.
8   Box orig_box = getWorldBox(p_o) // original bounding box
9   p_o -> setPosition(where)        // move object
10  Box new_box = getWorldBox(p_o)    // new bounding box
11
12  // If object moved from inside to outside world, generate
13  // "out of bounds" event.
14  if boxIntersectsBox(orig_box, boundary) and // Was in bounds?
15     not boxIntersectsBox(new_box, boundary) // Now out of bounds?
16     EventOut ov                               // Create "out of bounds"
17         event

```

<sup>\*</sup> **Did you know (#8)?** The *Globe Skimmer* dragonfly has the longest migration of any insect, back and forth across the Indian Ocean, about 11,000 miles. – “14 Fun Facts About Dragonflies”, *Smithsonian.com*, October 5, 2011.





```

17     p_o -> eventHandler(&ov)           // Send to Object
18     end if
19
20     ...

```

Game objects' positions are specified in relation to the world coordinates. In other words, the position attribute in an Object that provides an (x,y) location means that object should be at (x,y) in the world, but not necessarily at (x,y) on the window. With views, the world (x,y) position need to be mapped to the view/window (x,y) position.

Consider an example in Figure 4.6. The game world is 35x25 and the origin (0,0) is in the upper left corner. There are three Objects in the world: A is at (15,10), B is at (8,5) and C is at (25,2). The coordinates depicted for all the objects are relative to the game world. The view, 10x10, what the player sees on the window, is smaller than the game world, 35x25. The view origin, position (0,0) on the window, is at position (10,3) in game world coordinates. To correctly display objects on the window, the view origin position is subtracted from each Object's position before drawing. For example, for Object A, subtracting (10,3) from (15,10) puts A at location (5,7) on the window. For Object B, subtracting (10,3) from (8,5) puts B at (-2,2). Since the x coordinate is negative, B is not drawn. For Object C, subtracting (10,3) from (25,2) puts C at (15,-1). Since the y coordinate is negative and 15 is greater than the window width, C is not drawn.

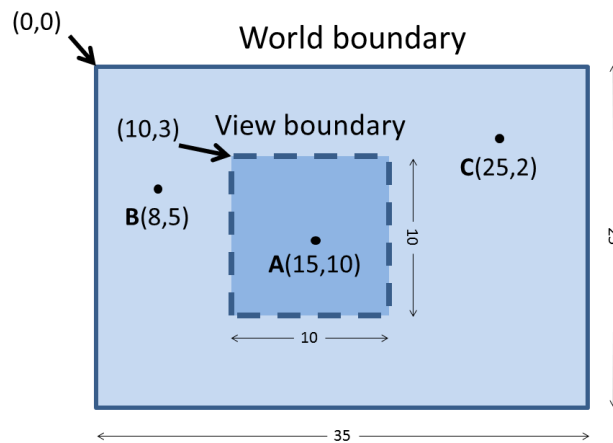


Figure 4.6: View boundary in relation to world boundary

This world-to-view translation is most easily done in the DisplayManager right before drawing a character on the window. The `utility.cpp` method `worldToView()`, shown in Listing 4.138, converts a world (x,y) position to a view (x,y) position on the window based on the current view.

Listing 4.138: `worldToView()`

```

0 // Convert world position to view position.
1 Vector worldToView(Vector world_pos)

```



```

2  view_origin = WorldManager getView().getCorner()
3  view_x = view_origin.getX()
4  view_y = view_origin.getY()
5  Vector view_pos(world_pos.getX()-view_x, world_pos.getY()-view_y)
6  return view_pos

```

The DisplayManager `drawCh()` is re-factored to call `worldToView()` right before each character that is drawn, shown in Listing 4.139.

Listing 4.139: DisplayManager extensions to `drawCh()` to support views

```

0  int DisplayManager::drawCh(Vector world_pos, char ch, Color color) const
1  {
2  Vector view_pos = worldToView(world_pos)
3  ...

```

With views, not all Objects need to be drawn every loop. For example, in Figure 4.6, objects B and C are off the visible window so while calling `drawCh()` would not cause errors, it is a waste of time. In the WorldManager `draw()` method, instead of automatically drawing all objects, first the bounding box for each object is checked for intersection with the current view. If there is intersection, the Object is drawn. If there is not intersection, the Object is not drawn. This logic, done inside the “altitude” loop, is shown in Listing 4.140.

Listing 4.140: WorldManager extensions to `draw()` to support views

```

0  ...
1  // Bounding box coordinates are relative to Object,
2  // so convert to world coordinates.
3  temp_box = getWorldBox(p_temp_o)
4
5  // Only draw if Object would be visible on window (intersects view).
6  if boxIntersectsBox(temp_box, view) then
7      p_temp_o -> draw()
8  end if
9  ...

```

In order to give the game programmer control over the view, the WorldManager is extended as indicated in Listing 4.141. The method `setViewPosition()` positions the view at a specific (x,y) world coordinate, and `setViewFollowing()` automatically moves the view to keep the indicated game object, stored in the private attribute `p_view_following`, in the center of the window. The latter is useful when the game programmer wants the camera to follow an avatar as it moves around the world, such as is typical in a platformer.

Listing 4.141: WorldManager extensions to support view following Object

```

0  private:
1  Object *p_view_following; // Object view is following.
2
3  public:
4  // Set view to center window on position view_pos.
5  // View edge will not go beyond world boundary.
6  void setViewPosition(Vector view_pos);
7
8  // Set view to center window on Object.
9  // Set to NULL to stop following.

```



```

10 // If p_new_view_following not legit, return -1 else return 0.
11 int setViewFollowing(Object *p_new_view_following);

```

Pseudo code for the method `setViewPosition()` is shown in Listing 4.142. The method takes in a position in the game world and sets the view to be centered on that position. In the first two blocks of code, the method makes sure that the edges of the view do not go outside of the edges of the game world, horizontally and then vertically. If it does, then the boundary is moved to flush with the edge.

Listing 4.142: WorldManager `setViewPosition()`

```

0 // Set view to center window on position view_pos.
1 // View edge will not go beyond world boundary.
2 void WorldManager::setViewPosition(Vector view_pos)
3
4 // Make sure horizontal not out of world boundary.
5 x = view_pos.getX() - view.getHorizontal()/2
6 if x + view.getHorizontal() > boundary.getHorizontal() then
7     x = boundary.getHorizontal() - view.getHorizontal()
8 end if
9 if x < 0 then
10     x = 0
11 end if
12
13 // Make sure vertical not out of world boundary.
14 y = view_pos.getY() - view.getVertical()/2
15 if y + view.getVertical() > boundary.getVertical() then
16     y = boundary.getVertical() - view.getVertical()
17 end if
18 if y < 0 then
19     y = 0
20 end if
21
22 // Set view.
23 Vector new_corner(x, y)
24 view.setCorner(new_corner)

```

Pseudo code for the method `setViewFollowing()` is shown in Listing 4.143. The first block of code starting on line 6 checks if `p_new_view_following` is `NULL` – if so, the game programmer intends to stop having the view follow any particular Object.

The second block of code starting on line 12 iterates over all the Objects in the world. Each Object is compared to the `p_new_view_following` to make sure the game programmer has requested to follow a legitimate object. The boolean variable `found` is set to true if the Object is matched with one of the known game objects.

The third block of code starting on line 17 sets the `p_view_following` variable if the Object has been found and, if so, sets the view position to be centered on that Object.

If the Object is not found, the method returns -1 (an error).

Listing 4.143: WorldManager `setViewFollowing()`

```

0 // Set view to follow Object.
1 // Set to NULL to stop following.
2 // If p_new_view_following not legit, return -1 else return 0.
3 int WorldManager::setViewFollowing(Object *p_new_view_following)

```



```

4
5 // Set to NULL to turn 'off' following.
6 if p_new_view_following is NULL then
7     p_view_following = NULL
8     return ok
9 end if
10
11 // ...
12 // Iterate over all Objects. Make sure p_new_view_following
13 // is one of the Objects, then set found to true.
14 // ...
15
16 // If found, adjust attribute accordingly and set view position.
17 if found then
18     p_view_following = p_new_view_following
19     setViewPosition(p_view_following -> getPosition())
20     return ok
21 end if
22
23 // If we get here, was not legit. Don't change current view.
24 return -1

```

The last adjustment is to the WorldManager `moveObject()` method. Here, at the very end of the method, if the Object has been successfully moved and the Object being moved is the same as the Object being followed, then the view is adjusted to the new position of the Object. This logic is shown in Listing 4.144.

Listing 4.144: WorldManager extensions to `moveObject()` to support views

```

0 ...
1 // If view is following this object, adjust view.
2 if p_view_following is p_o then
3     setViewPosition(p_o -> getPosition())
4 end if
5 ...

```

#### 4.13.4.1 Advanced View Control (optional)

As implemented, when following an Object with the view, the camera keeps the Object dead-center on the screen at all times (subject to the world boundaries, of course). For the player, a camera locked in this mode can be tedious for a game where the player is moving the view a lot. There are a variety of advanced camera control techniques that *Dragonfly* may be extended to provided including *zoning*, *blending* and *rails*. The interested developers is encouraged to see Phil Wilkins excellent talk on dynamic camera systems [9] with many more details in Mark Haigh-Hutchinson’s book on real-time cameras [5].

An additional technique shown here is *dynamics*. With dynamics, the camera still follows an Object, but does not require the Object to be strictly in the middle of the screen. Instead, the Object can stay within a smaller rectangle inside the screen without the camera moving. This provides some “slack” that allows the Object to be within a center area before the camera needs to move.



To support these dynamics, the WorldManager is extended with a `view_slack` attribute which is a Vector representing the (x, y) dimensions of the inner rectangle. The default value for `view_slack`, set in the WorldManager constructor, should be (0, 0). Attributes to get and set `view_slack`, (`getViewSlack()` and `setViewSlack()`, respectively) should also be created.

Then, WorldManager `moveObject()` is re-factored to support dynamics, as shown in Listing 4.145.

Listing 4.145: WorldManager extensions to `moveObject()` to support view dynamics

```

0  ...
1  // If view is following this object, adjust view as appropriate.
2  if p_view_following is p_o then
3
4      // Get center of view.
5      view_center_x = view.getCorner().getX() + view.getHorizontal()/2
6      view_center_y = view.getCorner().getY() + view.getVertical()/2
7
8      // Compute inner "slack" box edges.
9      left = view_center_x - view.getHorizontal() * view_slack.getX()/2
10     right = view_center_x + view.getHorizontal() * view_slack.getX()/2
11     top = view_center_y - view.getVertical() * view_slack.getY()/2
12     bottom = view_center_y + view.getVertical() * view_slack.getY()/2
13
14     new_pos = p_o -> getPosition()
15
16     // Move view right/left?
17     if (new_pos.getX() < left)
18         view_center_x -= left - new_pos.getX()
19     else if (new_pos.getX() > right)
20         view_center_x += new_pos.getX() - right
21
22     // Move up/down?
23     if (new_pos.getY() < top)
24         view_center_y -= top - new_pos.getY()
25     else if (new_pos.getY() > bottom)
26         view_center_y += new_pos.getY() - bottom
27
28     // Set new view position.
29     setViewPosition(Vector(view_center_x, view_center_y))
30 end if // following p_o
31 ...

```

The first block of code, lines 4 to 12 compute the edges of the inner box (the “slack” in the camera dynamics). The next block of code, lines 16 to 26 compare the position of the Object the camera is following to these edges, moving the edge as appropriate to keep the Object within the inner box. The last bit of code, line 29, actually adjusts the view to the new location.

#### 4.13.4.2 Using Views

The view support added to `Dragonfly` can be used by the game programmer to provide the player with a game world larger than the window. Assume, for example, that in Saucer



Shoot (see Section 3.3 on page 14), the game programmer wants the game world to be about twice as large vertically as a window. This can be done by explicitly setting the world boundary in `game.cpp`, as shown in Listing 4.146, setting the view boundary to be the typical size of 80x24.

Listing 4.146: Explicitly setting game world boundaries

```
0 // Set world boundaries to 80 horizontal, 50 vertical.
1 Vector corner(0,0)
2 Box world_boundary(corner, 80, 50)
3 WorldManager setBoundary(world_boundary)
4
5 // Set view to 80 horizontal, 24 vertical.
6 Box view(corner, 80, 24)
7 WorldManager setView(view)
```

With the WorldManager controlling the world boundaries, code that generates the out event (see Section 4.10.1.4 on page 140) should be refactored to use the WorldManager `getBoundary()` instead of the DisplayManager window limits.

With the world larger than the window, the intent is probably to always keep the Hero centered vertically in the window. This could be done by extending the original `move()` method (Listing 3.11 on page 44) with the code defined in Listing 4.147. Basically, when the Hero moves vertically, the new code adjusts the view by the same vertical amount.

Listing 4.147: Example Hero extension to `move()` to support views

```
0 // Always keep Hero centered in window.
1 void Hero::move(float dy)
2 // Move as before...
3
4 // Adjust view.
5 Box new_view = WorldManager getView();
6 Vector corner = new_view.getCorner();
7 corner.setY(corner.getY() + dy);
8 new_view.setCorner(corner);
9 WorldManager setView(new_view);
```

Alternatively, the WorldManager can just be told to follow the Hero by calling `@@setViewFollowing()` in the Hero's constructor, as in Listing 4.148.

Listing 4.148: Example Hero extension to support views

```
0 // Always keep Hero centered in window.
1 void Hero::Hero()
2 // ...
3 WorldManager setViewFollowing(this);
```

#### 4.13.5 Development Checkpoint #10!

Continue *Dragonfly* development, using Boxes to first provide bounding boxes for Objects and next to provide view and world boundaries. Steps:



1. Add the Box class, referring to `Box.h` in Listing 4.130. Add `Box.cpp` to the project and stub out each method so it compiles. The Box is really just a container, but test the Box class thoroughly, anyway. Do this outside of the engine, making sure that the attributes can all be get and set properly and that the constructor with corner, horizontal and vertical dimensions specified works.
2. Extend the Object class support bounding boxes, referring to Listing 4.131. Test that Object bounding boxes can be set and retrieved properly. Be sure to extend `setSprite()` to set the Object box to the dimensions of the associated Sprite.
3. Write the utility function `boxIntersectsBox()` (Listing 4.132) that determines if two boxes overlap. Test this with a program that uses Boxes of a variety of dimensions and locations with all sorts of intersection combinations (including containment). Verify all test cases work before proceeding – this function gets called a *lot* in a typical game.
4. Replace `positionsIntersect()` with `boxIntersectsBox()` in the WorldManager. First, verify using former test code that the engine still works with single character Objects. Then, create test code with multi-character Sprites, testing a variety of Objects and collisions. Use one non-moving Object with one moving Object at first to make debugging simpler.
5. Add views, starting by extending the WorldManager to support views as in Listing 4.136. Test the get and set methods for the `view` and `boundary` attributes.
6. Write and test the DisplayManager `worldToView()`, referring to Listing 4.138 as needed. Put calls to `worldToView` in the DisplayManager `drawCh()`, as per Listing 4.139. Test that previous code without views still works, then test that having a view that is not positioned at the world's origin works as expected. At this point, use just a hard-coded view in the WorldManager.
7. Extend the WorldManager `draw()` method to only draw objects that are in the view, as shown in Listing 4.140. Test with a variety of Objects that are in the view, completely out of the view, and partially in/out of the view.
8. Add settings to the WorldManager that enable setting the view, including attributes and methods from Listing 4.141. Refer to details from Listing 4.142 and Listing 4.143, as needed. Extend WorldManager `moveObject()` as in Listing 4.144.
9. To test, modify the Saucer Shoot tutorial to use views. First, set the game world boundaries as in Listing 4.146, then modify the Hero `move()` method as in Listing 4.147. Test thoroughly, making sure the window is smaller than the game world settings. Once convinced all works, revert back to the former `move()` method and have the view follow the `Hero()` as in Listing 4.148. Test this thoroughly, too.



## 4.14 Audio

While sight and feel (interaction) are core elements of most games, sound is nearly as important. *Dragonfly* supports audio\* using the built-in capabilities of the Simple and Fast Multimedia Library (SFML).

### 4.14.1 Simple and Fast Multimedia Library – Audio

SFML provides audio support and recognizes two distinct types: 1) *sounds*, which are typically small (fitting in memory) and for games, are played in response to a game action. Think of these as sound effects. Examples from Saucer Shoot (Section 3.3) include the “fire” sound when the Hero shoots a Bullet and the “explode” sound when a Saucer is destroyed. The class `sf::Sound` supports this type of audio. 2) *music*, which is typically longer (e.g., an entire song) and, for games, is played continuously, often in the background, as game action takes place. An example from Saucer Shoot is the background music that plays during the initial game start screen. The class `sf::Music` supports this type of audio. These differences between sound and music influence how they are handled technically by the SFML classes. For example, sound effects are usually small enough to load into memory, while music, being larger, is streamed directly from disk. SFML supports most common audio file formats – the full list can be found in online documentation. Note, `<SFML/Audio.hpp>` is needed as an `#include` for all SFML audio.

For `sf::Sound`, the sound data is not stored directly in the object but via a separate class called `sf::SoundBuffer`. The sound buffer holds the audio samples in an array of 16-bit integers. Each audio sample is the amplitude of the sound wave at a given point in time. Sound data from a file (e.g., a `.wav` file) can be loaded into a `sf::SoundBuffer` with the method `loadFromFile()`. Use of this method is shown in the top part of Listing 4.149.

Once the audio data is loaded, the buffer can be assigned to an `sf::Sound` object via `setBuffer()` and then played via `play()`. The latter half of Listing 4.149 shows a code fragment to do this. Note, sounds can also be played together without any issues.

Listing 4.149: SFML playing a sound

```
0 #include <SFML/Audio.hpp>
1
2 sf::SoundBuffer buffer;
3 if (buffer.loadFromFile("sound.wav") == false)
4     // Error!
5
6 sf::Sound sound;
7 sound.setBuffer(buffer);
8 sound.play();
```

Unlike `sf::Sound`, `sf::Music` does not pre-load audio data but instead streams directly from a file. So, this means opening a file and then just playing it, as in Listing 4.150.

---

\* **Did you know (#9)?** Dragonflies cannot hear, at least not the same way humans can. However, Dragonflies do have receptors in their antennae and legs that are sensitive to pressure changes, such as air pressure changes from sounds. These receptors supplement their vision. – Ann Cooper. *Dragonflies – Q&A Guide: Fascinating Facts About Their Life in the Wild*, Stackpole Books, September 2014.





Listing 4.150: SFML playing music

```

0 #include <SFML/Audio.hpp>
1
2 sf::Music music;
3 if (music.openFromFile("music.wav") == false)
4     // Error!
5
6 music.play();

```

Looping for both sound and music can be done with `setLoop()`, indicating `true` to loop (repeat) the audio from the beginning when at the end and `false` to stop the audio when at the end. Both sounds and music can be stopped with `stop()` and paused with `pause()`.

One key difference between `sf::Music` and `sf::Sound` is that SFML does not allow copying of `sf::Music` objects (presumably, this is to help SFML manage resources more efficiently). This does mean that the code samples in Listing 4.151 result in compile-time errors if tried.

Listing 4.151: SFML `sf::Music` not copyable

```

0 sf::Music music;
1 sf::Music music_copy = music; // Error!
2
3 void makeItSo(sf::Music music_parameter) {
4     ...
5 }
6 makeItSo(music); // Error!

```

### 4.14.2 Dragonfly Audio

To add **Dragonfly** support for audio, SFML audio support is wrapped by two classes (Sound and Music), with sound and music assets managed by the ResourceManager. Wrapping the SFML audio classes in this way provides for a simpler interface for game programming and, equally important, means that if **Dragonfly** were to use an alternate library for audio support, game code written for **Dragonfly** would not need to be changed. For game code that wishes to exploit alternate features of SFML audio, the base SFML types (`sf::Sound` and `sf::Music`) are exposed.

#### 4.14.2.1 The Sound Class

**Dragonfly** provides a Sound class for supporting basic sound effects, with the header file shown in Listing 4.152. The primary attributes provide for a `sf::Sound` (`sound`) and a `sf::SoundBuffer` (`sound_buffer`). The method `loadSound()` calls `loadFromFile()`, using the indicated filename and then sets the sound buffer with `setBuffer()`. See Listing 4.149 for examples. The `string label` is text to identify the sound for the game programmer, similar to the label used by the game programmer to identify a Sprite (see Listing 4.111 on page 146). The methods `setLabel()` and `getLabel()` are used to set and get the label, respectively. The methods `play()`, `stop()`, and `pause()`, call the corresponding methods on the `sound` object. The method `play()` has an option to loop the sound, too, which is done



via `setLoop()`. Looping is off by default. To allow the game programmer to manipulate the `sf::Sound` object directly, `getSound()` returns `sound`.

Important! If developing on Windows, a call to `resetBuffer()` needs to be made to avoid a debug assertion when removing the sound. This call should be placed in the Sound destructor (`~Sound()`).

Listing 4.152: Sound.h

```

0 // System includes.
1 #include <string>
2 #include <SFML/Audio.hpp>
3
4 class Sound {
5
6     private:
7         sf::Sound m_sound;           // The SFML sound.
8         sf::SoundBuffer m_sound_buffer; // SFML sound buffer associated with
9                                         sound.
10        std::string m_label;          // Text label to identify sound.
11
12     public:
13         Sound();
14         ~Sound();
15
16         // Load sound buffer from file.
17         // Return 0 if ok, else -1.
18         int loadSound(std::string filename);
19
20         // Set label associated with sound.
21         void setLabel(std::string new_label);
22
23         // Get label associated with sound.
24         std::string getLabel() const;
25
26         // Play sound.
27         // If loop is true, repeat play when done.
28         void play(bool loop=false);
29
30         // Stop sound.
31         void stop();
32
33         // Pause sound.
34         void pause();
35
36         // Return SFML sound.
37         sf::Sound getSound() const;
38 };

```

#### 4.14.2.2 The Music Class

**Dragonfly** provides a Music class for supporting music, with the header file shown in Listing 4.153. The primary attribute is `sf::Music` (`music`). The method `loadMusic()` calls `openFromFile()`, using the indicated filename. See Listing 4.150 for examples.



Note, as mentioned above, SFML does not allow copying of `sf::Music` objects (See Listing 4.151). That is why the Music copy and assignment operators are private. As a note, making the non-private can work, too, but then exposes potentially confusing SFML errors to the game program when linking.

The `string label` is text to identify the music for the game programmer, as for Sounds (see Listing 4.152) and Sprites (see Listing 4.111 on page 146). The methods `setLabel()` and `getLabel()` are used to set and get the label, respectively. The methods `play()`, `stop()`, and `pause()`, call the corresponding methods on the `music` object. The method `play()` has an option to loop the sound, too, which is done via `setLoop()`. Looping is on by default. To allow the game programmer to manipulate the `sf::Music` object directly, `getMusic()` returns a pointer to `music`. A pointer is used because SFML does not allow `music` to be copied.

Listing 4.153: Music.h

```

0 // System includes.
1 #include <string>
2 #include <SFML/Audio.hpp>
3
4 class Music {
5
6     private:
7         Music(Music const&);           // SFML doesn't allow music copy.
8         void operator=(Music const&);  // SFML doesn't allow music assignment.
9         sf::Music m_music;             // The SFML music.
10        std::string m_label;            // Text label to identify music.
11
12     public:
13         Music();
14
15         // Associate music buffer with file.
16         // Return 0 if ok, else -1.
17         int loadMusic(std::string filename);
18
19         // Set label associated with music.
20         void setLabel(std::string new_label);
21
22         // Get label associated with music.
23         std::string getLabel() const;
24
25         // Play music.
26         // If loop is true, repeat play when done.
27         void play(bool loop=true);
28
29         // Stop music.
30         void stop();
31
32         // Pause music.
33         void pause();
34
35         // Return pointer to SFML music.
36         sf::Music *getMusic();
37 };

```



### 4.14.2.3 Extending the ResourceManager for Audio

With Sound and Music in place, the ResourceManager is extended to manager sound and music resources. The needed extensions are shown in Listing 4.154. Audio is handled similarly to sprites, with fixed sized arrays for Sound and Music objects and count variables for each. The counts should be initialized to 0 upon `startUp()`. The “load” methods load the Sound and Music resources from files and the “unload” methods do the reverse. Two “get” methods provide pointers to both Sound and Music objects identified by a label.

Listing 4.154: ResourceManager extensions to support audio

```

0  const int MAX_SOUNDS = 128;
1  const int MAX_MUSICS = 128;
2
3  private:
4      Sound sound[MAX_SOUNDS];    // Array of sound buffers.
5      int sound_count;            // Count of number of loaded sounds.
6      Music music[MAX_MUSICS];    // Array of music buffers.
7      int music_count;            // Count of number of loaded musics.
8
9  public:
10     // Load Sound from file.
11     // Return 0 if ok, else -1.
12     int loadSound(std::string filename, std::string label);
13
14     // Remove Sound with indicated label.
15     // Return 0 if ok, else -1.
16     int unloadSound(std::string label);
17
18     // Find Sound with indicated label.
19     // Return pointer to it if found, else NULL.
20     Sound *getSound(std::string label);
21
22     // Associate file with Music.
23     // Return 0 if ok, else -1.
24     int loadMusic(std::string filename, std::string label);
25
26     // Remove label for Music with indicated label.
27     // Return 0 if ok, else -1.
28     int unloadMusic(std::string label);
29
30     // Find Music with indicated label.
31     // Return pointer to it if found, else NULL.
32     Music *getMusic(std::string label);

```

The `loadSound()` method to load a sound from a file is shown in Listing 4.155. Error checking is done to ensure the `sound` array is not filled. On line 9, the call to `Sound loadSound()` is made. If successful, the Sound is added to the array. Any error condition returns -1, while success returns 0.

Listing 4.155: ResourceManager loadSound()

```

0  // Load Sound from file.
1  // Return 0 if ok, else -1.
2  int ResourceManager::loadSound(std::string filename, std::string label)

```



```

3
4  if sound_count is MAX_SOUNDS then
5      writeLog("Sound array full.")
6      return -1 // Error.
7  end if
8
9  if sound[sound_count].loadSound(filename) is -1 then
10     writeLog("Unable to load from file")
11     return -1 // Error
12 end if
13
14 // All set.
15 sound[sound_count].setLabel(label)
16 increment sound_count
17 return 0

```

The complement of `loadSound()` is `unloadSound()`, shown in Listing 4.156. The method loops through the Sounds in the Resource Manager. If the label being looked for (`label`) matches the label of one of the Sounds (`getLabel()`) then that is the Sound to be unloaded. SFML does not have a method to actually free up memory for sounds, so the rest of the Sounds in the array are moved down one. Lastly, the sound count is decremented by one. If the loop terminates without a label match, the sound to be unloaded is not in the Resource Manager and an error is returned.

Listing 4.156: Resource Manager `unloadSound()`

```

0 // Remove Sound with indicated label.
1 // Return 0 if ok, else -1.
2 int ResourceManager::unloadSound(std::string label)
3
4     for i = 0 to sound_count-1
5
6         if label is sound[i].getLabel() then
7
8             // Scoot over remaining sounds
9             for j = i to sound_count-2
10                 sound[j] = sound[j+1]
11             end for
12
13             decrement sound_count
14
15             return success
16
17         end if
18
19     end for
20
21     return error // Sound not found.

```

The final method needed by the Resource Manager for sound is `getSound()`, with pseudo code show in Listing 4.157. The method loops through all the Sounds in the Resource Manager. The first Sound that matches `label` is returned. If line 10 is reached, the label was not found and an error (`NULL`) is returned.



Listing 4.157: ResourceManager getSound()

```

0 // Find Sound with indicated label.
1 // Return pointer to it if found, else NULL.
2 Sound *getSound(std::string label);
3
4 for i = 0 to sound_count-1
5     if label is sound[i].getLabel() then
6         return (&sound[i])
7     end if
8 end for
9
10 return NULL // Sprite not found.

```

Methods to `loadMusic()`, `unloadMusic()` and `getMusic()` are similar to `loadSound()`, `unloadSound()` and `getSound()`, respectively. The exception is that since Music is not copyable, the elements cannot be “scooted over” in the array. Instead, the found label is just set to empty (“”). This means that the empty label is not allowed in `loadMusic()` to distinguish from an unloaded Music.

### 4.14.3 Using Audio

At this point, the game programmer can load sounds and music into the ResourceManager in a few simple steps. The first step is to obtain/create an audio file, such as those provided by the *Dragonfly* tutorial (see Section 3). The second step is to load the audio file, as a Sound or Music, into the ResourceManager so the game can make use of it. Example code to load sound effects for Saucer Shoot is shown in Listing 3.7 on page 42 and example code to load music is shown in Listing 3.9 on page 43.

Once loaded, the game programmer can play audio at an appropriate point. For example, Saucer Shoot plays music during the game start screen (see Listing 3.8 on page 43) and plays a sound effect when the player fires a bullet (see Listing 3.10 on page 44).

### 4.14.4 Development Checkpoint #11!

Continue *Dragonfly* development to support audio. Steps:

1. Make the Sound and Music classes, referring to Listings 4.152 and 4.153, respectively. Separately implement and test both classes outside of the game engine. This means playing various sound effects and music. The audio files from the Saucer Shoot tutorial (see Section 3.3.12 on page 42) can be used for this. Make sure to test error conditions (e.g., the file cannot be found), too.
2. Extend the ResourceManager to support sound effects, referring to Listing 4.154 as needed. Write and test methods to `loadSound()`, `unloadSound()`, and `getSound()`. Refer to Listings 4.155, 4.156, and 4.157, as needed.
3. Extend the ResourceManager to support music, referring to Listing 4.154 as needed. Write and test methods to `loadMusic()`, `unloadMusic()`, and `getMusic()`. Base the music support implementation off the corresponding sound support previously implemented.



## 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 24). 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 (also called a *publish-subscribe* design pattern).

Figure 4.7 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 4.7.

Figure 4.8 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 4.7, 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 4.7, 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`



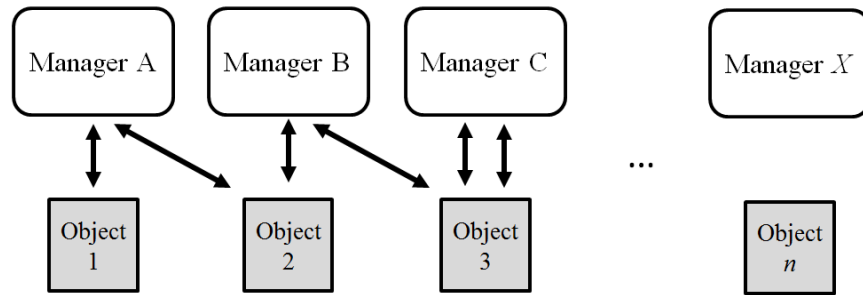


Figure 4.7: Event interest

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

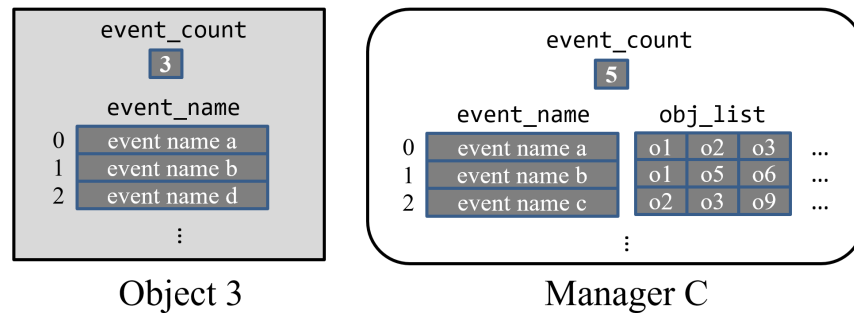


Figure 4.8: Event interest (zoom)

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 4.158 shows code fragments that provide attributes and methods for the Manager class to support interest management.

Listing 4.158: 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 *i*th element of `event[]` corresponds to the *i*th 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 4.159. 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.<sup>17</sup> 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 4.159: Manager `registerInterest()`

```

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

```

<sup>17</sup>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 4.160. 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 4.160: 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     scout over all items in object_list[]
13     scout 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 4.161. 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 4.161: 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 4.162. 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 4.162: Manager `isValid()`

```

0 // If this event is allowed by the manager, return 1 else 0
1 // (this is virtual and can be overwritten by child classes).
2 virtual bool isValid(std::string event_name);

```

The pseudo code for each derived manager class `isValid()` is shown in Listing 4.163. 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 4.163: 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)

```



```

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 4.164.

Listing 4.164: 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)
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 4.165. 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 4.165: Object class extensions for `registerInterest()`

```

0 private:
1   int event_count;
2   std::string event_name[MAX_OBJ_EVENTS];
3
4 public:
5   // Register for interest in event.
6   // Keeps track of manager and event.
7   // Return 0 if ok, else -1
8   int registerInterest(std::string event_type);
9
10  // Unregister for interest in event.
11  // Return 0 if ok, else -1
12  int unregisterInterest(std::string event_type);

```

The Object's `registerInterest()` method is provided in Listing 4.166. 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 32) 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 4.166: Object registerInterest()

```

0 // Register for interest in event.
1 // Keeps track of manager and event.
2 // Return 0 if ok, else -1
3 int Object::registerInterest(std::string event_type)
4
5   // Check if room.
6   if event_count is MAX_OBJ_EVENTS then
7     return -1
8   end if
9
10  // Register with appropriate manager.
11  if event_type is STEP_EVENT then
12    GameManager registerInterest(this, event_type)
13  else if
14    ...
15  else
16    WorldManager registerInterest(this, event_type)
17  end if
18
19  // Keep track of added event.
20  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 4.167. 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 76 for an example of doing “scooting” for an array of integers.

Listing 4.167: 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 4.168. 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 4.168: 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 `EventStep 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 4.158 for the methods needed. Add the necessary attributes, then write `registerInterest()` and `unregisterInterest()` based on Listing 4.159 and Listing 4.160, 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 4.161 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 4.163 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 4.165. Implement the `registerInterest()` and `unregisterInterest()` methods, referring to Listing 4.166 and Listing 4.167, 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 4.168 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.





## 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 4.169) 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 4.169. 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 4.169: *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,

```



```

10 TOP_LEFT,
11 TOP_CENTER,
12 TOP_RIGHT,
13 CENTER_LEFT,
14 CENTER_CENTER,
15 CENTER_RIGHT,
16 BOTTOM_LEFT,
17 BOTTOM_CENTER,
18 BOTTOM_RIGHT,
19 };
20
21 class ViewObject : public Object {
22
23 private:
24     std::string view_string;           // Label for value (e.g., "Points").
25     int m_value;                       // Value displayed (e.g., points).
26     bool m_draw_value;                 // True if should draw value.
27     bool m_border;                     // True if border around display.
28     Color m_color;                     // Color for text (and border).
29     ViewObjectLocation m_location;     // Location of ViewObject.
30
31 public:
32     // Construct ViewObject.
33     // Object settings: SPECTRAL, max alt.
34     // ViewObject defaults: border, top_center, default color, draw_value.
35     ViewObject();
36
37     // Draw view string and value.
38     virtual void draw();
39
40     // Handle 'view' event if tag matches view_string (others ignored).
41     // Return 0 if ignored, else 1 if handled.
42     virtual int eventHandler(const Event *p_e);
43
44     // General location of ViewObject on screen.
45     void setLocation(ViewObjectLocation new_location);
46
47     // Get general location of ViewObject on screen.
48     ViewObjectLocation getLocation() const;
49
50     // Set view value.
51     void setValue(int new_value);
52
53     // Get view value.
54     int getValue() const;
55
56     // Set view border (true = display border).
57     void setBorder(bool new_border);
58
59     // Get view border (true = display border).
60     bool getBorder() const;
61
62     // Set view color.
63     void setColor(Color new_color);
64

```



```

65 // Get view color.
66 Color getColor() const;
67
68 // Set view display string.
69 void setViewString(std::string new_view_string);
70
71 // Get view display string.
72 std::string getViewString() const;
73
74 // Set true to draw value with display string.
75 void setDrawValue(bool new_draw_value = true);
76
77 // Get draw value (true if draw value with display string).
78 bool getDrawValue() const;
79
80 };

```

Listing 4.170 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 196.

Listing 4.170: ViewObject ViewObject()

```

0 // Construct ViewObject.
1 // Object settings: SPECTRAL, max altitude.
2 // ViewObject defaults: border, top_center, default color, draw_value.
3 ViewObject::ViewObject()
4
5 // Initialize Object attributes.
6 setSolidness(SPECTRAL)
7 setAltitude(MAX_ALTITUDE)
8 setType("ViewObject")
9
10 // Initialize ViewObject attributes.
11 setValue(0)
12 setDrawValue()
13 setBorder(true)
14 setLocation(TOP_CENTER)
15 setColor(COLOR_DEFAULT)
16
17 // Register interest in view events.
18 registerInterest(VIEW_EVENT) // if Section 4.15 implemented.

```

Pseudo code for ViewObject `setLocation()` method is shown in Listing 4.171. 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 4.171 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 4.171: 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 4.172, 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 4.172: ViewObject setBorder()

```

0 // Set view border (true = display border).
1 void ViewObject::setBorder(bool new_border)
2
3 if border != new_border then
4
5     border = new_border
6
7     // Reset location to account for border setting.
8     setLocation(getLocation())

```



```

9
10 end if

```

The ViewObject `draw()` method is shown in Listing 4.173. 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 114) 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 4.138 on page 167.

Listing 4.173: ViewObject draw()

```

0 // Draw view string and value.
1 void 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
15 if border is true then
16     // Draw box around display.
17     ...
18 end if

```

The `toString()` function used in Listing 4.173 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 4.174.

Listing 4.174: Utility toString()

```

0 #include <sstream>
1 using std::stringstream;
2
3 // Convert int to a string, returning string.
4 std::string toString(int i) {
5     std::stringstream ss; // Create stringstream.
6     ss << i; // Add number to stream.
7     return ss.str(); // Return string with contents of stream.
8 }

```

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 4.175. Without views, the `draw()` method checked each Object to see if they intersect the visible screen (see Listing 4.140 on page 168). 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 4.175: WorldManager extensions to draw() to support ViewObjects

```

0  ...
1  // Only draw if Object would be visible (intersects view).
2  if boxIntersectsBox(box, view) or           // Object in view,
3     dynamic_cast <ViewObject *> (p_temp_o))    // or is ViewObject.
4     p_temp_o -> draw()
5  end if
6  ...

```

### 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 4.176 provides the header file for the EventView class, derived from the Event class (Listing 4.48 on page 90). Remember, in the constructor of a ViewObject, the Object already registered for interest in a `VIEW_EVENT` (see Listing 4.170 on page 193). `VIEW_EVENT` is defined in Listing 4.176 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 4.176: 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.

```



```

19 void setTag(std::string new_tag);
20
21 // Get tag.
22 std::string getTag() const;
23
24 // Set value to new value.
25 void setValue(int new_value);
26
27 // Get value.
28 int getValue() const;
29
30 // Set delta to new delta.
31 void setDelta(bool new_delta);
32
33 // Get delta.
34 bool getDelta() const;
35 };

```

With EventView specified, the ViewObject `eventHandler()` can now be defined as shown in Listing 4.177. 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.<sup>18</sup>

Listing 4.177: 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

```

<sup>18</sup>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



```

21     end if
22
23
24 end if
25
26 // If here, event was not handled. Call parent eventHandler().
27 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 4.178 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 4.178: 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 4.179 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 4.179: Button.h

```

0 class Button : public ViewObject {
1
2 private:
3     Color m_highlight_color;    // Color when highlighted.
4     Color m_default_color;     // Color when not highlighted.
5
6 public:
7     Button();
8
9     // Handle "mouse" events.
10    // Return 0 if ignored, else 1.
11    int eventHandler(const Event *p_e);
12
13    // Set highlight (when mouse over) color for Button.
14    void setHighlightColor(Color new_highlight_color);
15
16    // Get highlight (when mouse over) color for Button.
17    Color getHighlightColor() const;
18
19    // Set color of Button.
20    void setDefaultColor(Color new_default_color);
21
22    // Get color of Button
23    Color getDefaultColor() const;
24
25    // Return true if mouse over Button, else false.
26    bool mouseOverButton(const EventMouse *p_e) const;
27
28    // Called when Button clicked.
29    // Must be defined by derived class.
30    virtual void callback() = 0;
31 };

```

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 4.180. 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 4.135 on page 164) using the newly constructed Box and the mouse's position, and returns the appropriate boolean.

Listing 4.180: Button mouseOverButton()

```

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

```



```

1 bool MouseOverButton::mouseOverButton(const EventMouse *p_e) const
2
3 // Create Box for Button.
4 width = getViewString().size()
5 height = 1
6 if getBorder() then // if Button has border
7     width = width + 4 // box wider by 2 spaces and |
8     height = height + 2 // box taller by 2 rows of —
9 end if
10 Vector corner(getPosition().getX() - width/2,
11               getPosition().getY() - height/2)
12 Box b(corner, width, height)
13
14 // If mouse inside button box, return true, else false.
15 if boxContainsPosition(b, p_e -> getMousePosition())
16     return true
17 else
18     return false

```

With that method in place, the `eventHandler()` method, shown in Listing 4.181, 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 92).

Listing 4.181: Button `eventHandler()`

```

0 // Handle "mouse" events.
1 // Return 0 if ignored, else 1.
2 int Button::eventHandler(const Event *p_e)
3
4 // Check if mouse event.
5 if p_e -> getType() is not MOUSE_EVENT then
6     return 0 // not handled
7 end if
8
9 // Check if mouse over button.
10 if mouseOverButton(p_e) then
11
12     // Highlight on.
13     setColor(highlight_color)
14
15     // Check if clicked.
16     if p_e -> getMouseAction() is CLICKED then
17
18         // Invoke callback.
19         callback()
20
21     end if

```



```

22
23 // Highlight off.
24 setColor(default_color)
25
26 // Event handled.
27 return 1

```

Lastly, note that the `callback()` method on line 30 of Listing 4.179 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 38, where the player can choose to either “play” or “quit”. A quit button can be made as in Listings 4.183 (header file) and 4.182 (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 4.182: `QuitButton.h` – Example Quit button for game start screen

```

0 #include "Button.h"
1
2 class QuitButton : public df::Button {
3
4 public:
5     QuitButton();
6     void callback();
7 };

```

In the source code (Listing 4.182), 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 69).

Listing 4.183: `QuitButton.cpp` – Example Quit button for game start screen

```

0 #include "GameManager.h"
1 #include "QuitButton.h"
2
3 QuitButton::QuitButton() {
4     setViewString("Quit");
5     setLocation(df::BOTTOM_CENTER);
6 }
7
8 // On callback, set game over to true.
9 void QuitButton::callback() {
10     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 4.184 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 4.184: TextEntry.h

```

0 // Engine includes.
1 #include "EventMouse.h"
2 #include "ViewObject.h"
3
4 class TextEntry : public ViewObject {
5
6 private:
7     std::string m_text;           // Text entered.
8     int m_limit;                 // Character limit in text.
9     bool m_numbers_only;        // True if only numbers.
10    int m_cursor;                // Cursor location.
11    char m_cursor_char;          // Cursor character.
12    int m_blink_rate;            // Cursor blink rate.
13
14 public:
15     TextEntry();
16
17     // Set text entered.
18     void setText(std::string new_text);
19
20     // Get text entered.
21     std::string getText() const;
22
23     // Handle "keyboard" events.
24     // Return 0 if ignored, else 1.
25     int eventHandler(const Event *p_e);
26
27     // Called when TextEntry enter hit.
28     // Must be defined by derived class.
29     virtual void callback() = 0;
30
31     // Set limit of number of characters allowed.
32     void setLimit(int new_limit);
33
34     // 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 void 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 4.185.

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 4.185: `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())

```



```

11     else
12         text.replace(cursor, 1, 1, ' ')
13     end if
14     blink = blink + 1
15     if blink == getBlinkRate() then
16         blink = -1 * getBlinkRate()
17     end if
18
19     return 1
20
21 end if
22
23 // If keyboard event, handle.
24 if p_e -> getType() is KEYBOARD_EVENT and
25     p_e -> getKeyboardAction() is KEY_PRESSED then
26
27     // If return key pressed, then callback.
28     if p_e -> getKey() is Keyboard::RETURN then
29         callback()
30         return 1
31     end if
32
33     // If backspace, remove character.
34     if p_e -> getKey() is Keyboard::BACKSPACE then
35         if cursor > 0 then
36             if cursor < limit then
37                 text.replace(cursor, 1, 1, ' ')
38             end if
39             cursor = cursor - 1
40             text.replace(cursor, 1, 1, ' ')
41         end if
42         return 1
43     end if
44
45     // If no room, cannot add characters.
46     if cursor >= limit then
47         return 1
48     end if
49
50     // Get key as string.
51     std::string str = toString(p_k -> getKey())
52
53     // If entry should be number, confirm.
54     if numbers_only && not isdigit(str[0]) then
55         return 1
56     end if
57
58     // Replace spaces with characters.
59     text.replace(cursor, 1, str)
60     cursor++
61
62     // All is well.
63     return 1
64 end if
65

```



```

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 92).

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 4.186. 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 4.186: TextEntry draw()

```

0 // Draw viewstring + text entered.
1 void TextEntry::draw()
2
3 // Get original view string.
4 std::string view_str = getViewString()
5
6 // Add text.
7 setViewString(view_str + text)
8
9 // Draw.
10 ViewObject::draw()
11
12 // Restore original view string.
13 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 4.187 (header file) and 4.188 (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 4.187: NameEntry.h – Example TextEntry for player initials

```

0 #include "TextEntry.h"
1
2 class NameEntry : public df::TextEntry {
3
4 public:
5     NameEntryButton();

```



```

6   void callback();
7   };

```

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 4.188: NameEntry.cpp – Example TextEntry for player initials

```

0  #include "LogManager.h"
1  #include "NameEntry.h"
2
3  NameEntry::NameEntry() {
4      setViewString("Enter initials");
5      setLocation(df::CENTER_CENTER);
6      setLimit(3);
7  }
8
9  // On callback, write initials to logfile.
10 void QuitButton::callback() {
11     LM.writeLog("High score: %s", getText().c_str());
12 }

```

#### 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 4.169. 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 4.170 and then `setLocation()`, based on Listing 4.171. Get your code to compile and verify by visual inspection of code.
3. Based on Listing 4.174, 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 4.173. 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 4.176. Add `EventView.cpp` to the project. Define the `eventHandler()` based on Listing 4.177. Verify the code compiles and use visual inspection on the methods.





6. Referring to Listing 4.178, 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.



## 4.17 End Game (optional)

Up to this point, *Dragonfly* is a fairly full-featured, completely functional game engine. A few potential enhancements remain, however, that bring in elements common to many game engines and improve performance, appearance and functionality.

### 4.17.1 Scene Graphs (optional)

Scene graphs are data structures that arrange elements of a graphics scene in order to provide more efficient rendering. For example, when drawing objects in a 3d scene, a scene graph might arrange the objects based on distance from the camera. Rendering the frame then draws the objects that are farthest away from the camera first, proceeding to the objects that are closest to the camera since the closer objects may occlude those behind.

Consider *Dragonfly*, where Objects have altitude (see Section 4.8.5 on page 115). Objects that are at lower altitude are drawn first before Objects at higher altitude, allowing the higher altitude to be layered “on top” of the lower ones, as necessary. Without a scene graph, *Dragonfly* implements altitude by iterating through all the Objects for each altitude implementation, as in Listing 4.81 on page 115 – effectively, doing  $n \times \text{MAX\_ALTITUDE}$  comparisons, where  $n$  is the number of Objects in the game world. With a scene graph, the objects can be arranged by altitude, making the `WorldManager` `draw()` method only go through the list of Objects once, so only doing  $n$  comparisons.

For a game engine, a scene graph often arranges objects for more efficient queries, also. Objects that are not solid do not cause collisions. Without any other organization, detecting whether a moving object collides with any other object must look through all objects, regardless of whether they are solid or not. Thus far, *Dragonfly* is implemented this way, too, as in Listing 4.102 (page 138), iterating through all Objects, checking for collision with every Object, even the non-solid ones. Other common organizations group Objects by location in the game world, allowing selection and iteration over only those Objects at or near a specific location.

Since a scene graph organizes Objects, whether for drawing or query efficiency, it is naturally part of the world manager. In fact, an easy way of viewing a scene graph is that it replaces a simple list of game Objects with a more complex data structure where the Objects are organized and indexed in different ways. In *Dragonfly*, this means replacing `ObjectList updates` on line 10 of Listing 4.54 (page 94) with `SceneGraph scene_graph`.

The header file for `SceneGraph` is shown in Listing 4.189. `SceneGraph` needs to `#include` both `Object.h` and `ObjectList.h`. The definition of `MAX_ALTITUDE` on line 3 has been moved from `WorldManager.h` to `SceneGraph.h`.

To support efficient queries by the `WorldManager` (e.g., to provide a list of all the solid objects), starting at line 8, the `SceneGraph` defines three lists of Objects. The first, `objects`, is a list of all the Objects in the game – formerly, this was `updates` in the `WorldManager`. The second, `solid_objects`, is a list of just the solid Objects in the game. The third, `visible_objects`, is an array of `ObjectList`s, with each element being a list of Objects at that altitude. Methods to add and remove objects to the scene graph are provided by `insertObject()` and `removeObject()`, respectively.

To support queries that may be made by the `WorldManager` (or even the game program-



mer), SceneGraph includes methods: `activeObjects()`, which returns all active Objects; `inactiveObjects()`, which returns all inactive Objects; `solidObjects()`, which returns all solid Objects; and `visibleObjects()`, which returns all visible Objects at a given altitude. The methods all return an empty ObjectList if there are no Objects matching the query. The method `updateAltitude()` is invoked when an Object re-positions itself to a new altitude and the method `updateSolidness()` is invoked when an Object updates its solidness.

Listing 4.189: SceneGraph.h

```

0 #include "Object.h"
1 #include "ObjectList.h"
2
3 const int MAX_ALTITUDE = 4;
4
5 class SceneGraph {
6
7     private:
8         ObjectList m_objects;           // All objects
9         ObjectList m_solid_objects;     // Solid objects.
10        ObjectList m_visible_objects[MAX_ALTITUDE+1]; // Visible objects.
11
12     public:
13         SceneGraph();
14         ~SceneGraph();
15
16         // Insert Object into SceneGraph.
17         int insertObject(Object *p_o);
18
19         // Remove Object from SceneGraph.
20         int removeObject(Object *p_o);
21
22         // Return all active Objects. Empty list if none.
23         ObjectList activeObjects() const;
24
25         // Return all active, solid Objects. Empty list if none.
26         ObjectList solidObjects() const;
27
28         // Return all active, visible Objects at altitude. Empty list if none.
29         ObjectList visibleObjects(int altitude) const;
30
31         // Return all inactive Objects. Empty list if none.
32         ObjectList inactiveObjects() const;
33
34         // Re-position Object in SceneGraph to new altitude.
35         // Return 0 if ok, else -1.
36         int updateAltitude(Object *p_o, int new_alt);
37
38         // Re-position Object in SceneGraph to new solidness.
39         // Return 0 if ok, else -1.
40         int updateSolidness(Object *p_o, Solidness new_solidness);
41
42         // Re-position Object in SceneGraph to new visibility.
43         // Return 0 if ok, else -1.
44         int updateVisible(Object *p_o, bool new_visible);

```



```

45 // Re-position Object in SceneGraph to new activeness.
46 // Return 0 if ok, else -1.
47 int updateActive(Object *p_o, bool new_active);
48 };
49

```

Implementation of SceneGraph `insertObject()` is shown in Listing 4.190. The method first inserts the Object into the `objects` list, since that is the “master” list that contains all Objects. Then, if the Object is solid, it is added to the `solid_objects` list. Next, the Object’s altitude is checked – if it is not in range (calling `valueInRange(altitude, 0, MAX_ALTITUDE)`, see line 1 in Listing 4.135 on page 164), it returns an error (-1). Otherwise, the object is inserted into the `visible_objects` list at the correct altitude. Note, the calls to `ObjectList::insert()` need to be error checked. If they encounter an error, an appropriate message should be written to the logfile and `insertObject()` should return -1.

While it may seem that keeping 3 object lists is inefficient, remember that game objects are stored as pointers to Objects, thus manipulating and copying such lists is not actually doing the much more expensive operation of copying the memory space for each Object. As a refresher, see Section 4.5.2 on page 4.5.2 for details on the ObjectList implementation.

Listing 4.190: SceneGraph `insertObject()`

```

0 // Insert Object into SceneGraph.
1 int SceneGraph::insertObject(Object *p_o)
2
3 // Add object to list.
4 insert p_o into objects list
5
6 // If solid, add to solid objects list.
7 if p_o -> isSolid() then
8     insert p_o into solid_objects list
9 end if
10
11 // Check altitude.
12 if not valueInRange(p_o->getAltitude(), 0, MAX_ALTITUDE) then
13     return error
14 end if
15
16 // Add to visible objects at right altitude.
17 insert p_o into visible_objects[p_o->getAltitude()] list

```

Implementation of the SceneGraph `removeObject()` is basically the “undo” of the `insertObject()` method, as shown in Listing 4.191. The indicated Object (`p_o`) is removed from the `objects`, `solid_objects` and `visible_objects` lists. As always, the calls to `ObjectList::remove()` need to be error checked, writing an appropriate message to the logfile and returning -1 on encountering an error.

Listing 4.191: SceneGraph `removeObject()`

```

0 // Remove object from SceneGraph.
1 int SceneGraph::removeObject(Object *p_o)
2
3 remove p_o from objects list
4

```



```

5  if p_o is solid then
6      remove p_o from solid_objects list
7  end if
8
9  remove p_o from visible_objects[p_o->getAltitude()] list

```

The methods `allObjects()` and `solidObjects()` just return `objects` and `solid_objects`, respectively. `visibleObjects()` first checks that the parameter `altitude` is in range (calling `valueInRange(altitude, 0, MAX_ALTITUDE)`, see line 1 in Listing 4.135 on page 164), then returns `visible_objects[altitude]`.

Objects may change their attributes, such as a `SPECTRAL` Object becoming `SOFT` or an Object changing altitude from 3 to 4. All such changes need to modify the contents of the SceneGraph lists, `solid_objects` and `visible_objects[]`, respectively. Listing 4.192 shows the implementation for updating the solidness of an Object. The first block of code checks if the Object is solid and, if so, removes it from the `solid_objects` list. The second block of code checks if the new solidness value for the Object is solid (`HARD` or `SOFT`) and, if so, inserts it into the `solid_objects` list. Error checking on the `ObjectList::insert()` calls is needed, as usual. Note, the solidness attribute for the Object is *not* changed – that is a private value for Object and is done in the Object `setSolidness()` method.

Listing 4.192: SceneGraph updateSolidness()

```

0  // Re-position Object in SceneGraph to new solidness.
1  // Return 0 if ok, else -1.
2  int SceneGraph::updateSolidness(Object *p_o, Solidness new_solidness)
3
4  // If was solid, remove from solid objects list.
5  if p_o->isSolid() then
6      remove p_o from solid_objects list
7  end if
8
9  // If is solid, add to list.
10 if new_solidness is HARD or new_solidness is SOFT then
11     insert p_o into solid_objects list
12 end if

```

Listing 4.193 shows the implementation for updating the altitude of an Object. First, the altitude values for both the new and old altitudes are checked for validity. It may seem odd to check the old value, since it seems it must be right, but it could have been corrupted someplace – if it was, trying to remove the object from the `visible_objects[]` list at the altitude may result in a crash. If both old and new are in the valid range, the object is first removed from `visible_objects[]` at the old altitude, then added to `visible_objects[]` at the new altitude. Error checking on the `ObjectList::insert()` calls are needed, as usual.

Listing 4.193: SceneGraph updateAltitude()

```

0  // Re-position object in scene graph to new altitude.
1  // Return 0 if ok, else -1.
2  int SceneGraph::updateAltitude(Object *p_o, int new_alt)
3
4  // Check if new altitude in valid range.

```



```

5  if not valueInRange(new_alt, 0, MAX_ALTITUDE) then
6      return error
7  end if
8
9  // Make sure old altitude in valid range.
10 if not valueInRange(p_o->getAltitude(), 0, MAX_ALTITUDE)) then
11     return error
12 end if
13
14 // Remove from old altitude.
15 remove p_o from visible_objects[p_o->getAltitude()]
16
17 // Add to new altitude.
18 insert p_o into visible_objects[new_alt]

```

Calls to `updateSolidness()` and `UpdateAltitude()` are made from Object, specifically Object `setSolidness()` and Object `setAltitude()`, respectively. The needed extension to Object `setSolidness()` to support SceneGraph is shown in Listing 4.194. The first block of code checks if the new solidness is valid (`HARD`, `SOFT` or `SPECTRAL`). If not, an error is returned. Otherwise, the `updateSolidness()` method of the SceneGraph is called and `solidness` is set in the Object.

Listing 4.194: Object class extension to `setSolidness()` to support SceneGraph

```

0 // Set object solidness, with checks for consistency.
1 // Return 0 if ok, else -1.
2 int Object::setSolidness(Solidness new_solidness)
3
4 // If solidness not valid, then ignore.
5 if new_solidness not (HARD or SOFT or SPECTRAL) then
6     return error
7 end if
8
9 // Update scene graph and solidness.
10 scene_graph.updateSolidness(this, new_solidness)
11 solidness = new_solidness

```

Extension to Object `setAltitude()` to support SceneGraphs is shown in Listing 4.195. The first block of code checks if the new altitude is in a valid range. If not, an error is returned. Otherwise, the SceneGraph `updateAltitude()` method is called and `altitude` is set in the Object.

Listing 4.195: Object class extension to `setAltitude()` support SceneGraphs

```

0 // Set object altitude.
1 // Checks for in range [0, MAX_ALTITUDE].
2 // Return 0 if ok, else -1.
3 int Object::setAltitude(int new_altitude)
4
5 // If altitude outside of range, then ignore.
6 if not valueInRange(new_altitude, 0, MAX_ALTITUDE) then
7     return error
8 end if
9
10 // Update scene graph and altitude.

```



```

11 scene_graph.updateAltitude(this, new_altitude)
12 altitude = new_altitude

```

With the SceneGraph in place, the Dragonfly WorldManager needs to be refactored to use the SceneGraph to manage game world Objects instead of storing the ObjectLists directly. Listing 4.196 shows the change in the WorldManager needed to use a SceneGraph. Basically, the attribute `ObjectList updates` is replaced with `SceneGraph scene_graph`. The method `getSceneGraph()` returns a reference to `scene_graph`.

Listing 4.196: WorldManager extensions to support SceneGraph

```

0 private:
1   SceneGraph scene_graph; // Storage for all Objects.
2
3 public:
4   // Return reference to the SceneGraph.
5   SceneGraph &getSceneGraph() const;

```

Then, internally, each of the WorldManager methods in Listing 4.197 needs to be refactored to support the SceneGraph. The methods `insertObject()` and `removeObject()` call and return `scene_graph.insertObject()` and `scene_graph.removeObject()`, respectively. The methods `update()` and `setViewFollowing()` call `scene_graph.allObjects()` to iterate through all the world Objects. The method `draw()` iterates through each altitude, calling `scene_graph.visibleObjects()` for each altitude. The method `isCollision()` checks for collisions only with Objects in the ObjectList returned from `scene_graph.solidObjects()`.

Listing 4.197: WorldManager methods to refactor to support SceneGraph

```

0 ObjectList getAllObjects()
1 int insertObject(Object *p_o)
2 int removeObject(Object *p_o)
3 int setViewFollowing(Object *p_new_view_following)
4 void update()
5 void draw()
6 ObjectList isCollision(const Object *p_o, Vector where) const

```

#### 4.17.1.1 Inactive Objects (optional)

For many games, it is useful for the game program to have game objects be ignored by the engine for some time, but without removing the objects altogether. For example, the Saucer Shoot tutorial game (Section 3.3.11 on page 38) has the main menu become inactive when the game is being played, becoming active again after the player's ship has been destroyed. Such inactive objects are not drawn by the engine, are neither moved nor considered in collisions, nor do they receive any events.

In order to support inactive Objects in Dragonfly, the Object class is extended with an attribute and methods to support whether an Object is active or inactive, shown in Listing 4.198. The boolean attribute `is_active` is `true` when the Object is active (note, all the Objects that have been dealt with to this point are active) and `false` when the Object is inactive and not acted upon by the engine. This value can be set via the `setActive()` method and queried via the `isActive()` method.



Listing 4.198: Object class extensions to support inactive objects

```

0 private:
1   bool is_active;    // If false , Object not acted upon.
2
3 public:
4   // Set activeness of Object. Objects not active are not acted upon
5   // by engine.
6   // Return 0 if ok, else -1.
7   int setActive(bool active=true);
8
9   // Return activeness of Object. Objects not active are not acted upon
10  // by engine.
11  bool isActive() const;

```

As shown in Listing 4.199, the method `setActive()` allows the game programmer to set the Object activeness, changing `is_active` as appropriate. Objects are active (`is_active` is `true`) by default, set in the constructor. In addition, the SceneGraph is obtained from the WorldManager and the SceneGraph `updateActive()` method is called.

Listing 4.199: Object setActive()

```

0 // Set activeness of Object. Objects not active are not acted upon
1 // by engine.
2 // Return 0 if ok, else -1.
3 int Object::setActive(bool active)
4
5   // Update scene graph.
6   scene_graph = WorldManager getSceneGraph()
7   scene_graph.updateActive(this, active)
8
9   // Set active value.
10  is_active = active

```

The SceneGraph is refactored to have an additional ObjectList, one that holds only inactive Objects while the main object list will hold active Objects. Listing 4.200 shows the changes to the SceneGraph attributes for this. The `objects` ObjectList has been renamed to `active_objects` to differentiate it from the ObjectList holding the inactive objects, `inactive_objects`.

Listing 4.200: SceneGraph extensions to support inactive Objects

```

0 private:
1   ObjectList active_objects;    // All active Objects.
2   ObjectList inactive_objects; // All inactive Objects.
3
4 public:
5   // Return all active Objects. Empty list if none.
6   ObjectList activeObjects() const;
7
8   // Return all inactive Objects. Empty list if none.
9   ObjectList inactiveObjects() const;
10
11  // Re-position Object in SceneGraph to new activeness.
12  // Return 0 if ok, else -1.
13  int updateActive(Object *p_o, bool new_active);

```





The methods `activeObjects()` and `inactiveObjects()` return `active_objects` and `inactive_objects`, respectively.

Listing 4.201 shows the SceneGraph `updateActive()` method. The first block of code checks if the activeness is being changed. If not, there is nothing more to do and an “ok” (0) is returned. The second block of code does the actual work. If the Object was active and became inactive, `remove()` is called on `active_objects`, `visible_objects[]` and, if solid, `solid_objects` and the Object is inserted into the `inactive_objects` list. Otherwise, if the Object was inactive and became active, `insert()` is called on `active_objects`, `visible_objects[]` and, if solid, `solid_objects` and the Object is removed from the `inactive_objects` list. All method calls should be error checked and an error (-1) returned, as appropriate. Otherwise, “ok” is returned at the end.

Listing 4.201: SceneGraph `updateActive()`

```

0 // Re-position object in scene graph based on activeness.
1 // Return 0 if ok, else -1.
2 int SceneGraph::updateActive(Object *p_o, bool new_active)
3
4 // If activeness unchanged, nothing to do (but ok).
5 if p_o->isActive() is new_active then
6     return ok
7 end if
8
9 // If was active then now inactive, so remove from lists.
10 if p_o->isActive() then
11
12     active_objects.remove(p_o)
13
14     visible_objects[p_o->getAltitude()].remove(p_o)
15
16     if p_o->isSolid() then
17         solid_objects.remove(p_o)
18     end if
19
20     // Add to inactive list
21     inactive_objects.insert(p_o)
22
23 else // Was active, so add to lists.
24
25     active_objects.insert(p_o)
26
27     visible_objects[p_o->getAltitude()].insert(p_o)
28
29     if p_o->isSolid() then
30         solid_objects.insert(p_o)
31     end if
32
33     // Remove from inactive list
34     inactive_objects.remove(p_o)
35
36 end if
37
38 // All is well.
39 return ok

```



The WorldManager `getAllObjects()` method is refactored, as in Listing 4.202. A boolean parameter `inactive` is provided to indicate whether the method should return only active Objects (`inactive` is `false`, the default) or both active and inactive Objects (`inactive` is `true`).

Listing 4.202: WorldManager extensions to support inactive Objects

```

0 // Return list of all Objects in world.
1 // If inactive is true, include inactive Objects.
2 // Return NULL if list is empty.
3 ObjectList getAllObjects(bool inactive=false);

```

The revised `getAllObjects()` is shown in Listing 4.203. The inactive case can use the overloaded '+' operator from Section 4.5.2.2 (page 81).

Listing 4.203: WorldManager extensions to `getAllObjects()` to support inactive Objects

```

0 // Return list of all Objects in world.
1 // If inactive is true, include inactive Objects.
2 // Return NULL if list is empty.
3 ObjectList WorldManager::getAllObjects(bool inactive) const
4
5 if inactive then
6     return scene_graph.activeObjects() + scene_graph.inactiveObjects()
7 else
8     return scene_graph.activeObjects()
9 end if

```

The Manager `onEvent()` method needs to be modified to check if an interested Object is actually active before sending it an event. This is shown on line 3 of Listing 4.204.

Listing 4.204: Manager extension to `onEvent()` to support inactive Objects

```

0 ...
1 create ObjectListIterator li on obj_list[i]
2 while not li.isDone() do
3     if li.currentObject -> isActive() then
4         invoke i.currentObject() -> eventHandler() with p_event
5     end if
6     li.next()
7 end while
8 ...

```

Lastly, WorldManager `shutDown()` should be revised to call `getAllObjects(true)` to delete both active and inactive Objects when the engine is shut down.

#### 4.17.1.2 Invisible Objects (optional)

Another useful property for many game objects is to become invisible. For a game object, invisibility could be a special power, say, for the hero or a bad guy to vanish from sight – but as such, it is rather rare. However, invisibility is commonly used to limit the player's ability to see objects that may be on the window, but should not yet be shown to the player



because of the player's avatar's orientation, or because of terrain or other "fog of war" type of effect.

To support invisibility, a new attribute is added to Object with methods for getting and setting it, shown in Listing 4.205. The method `isVisible()` returns the value of `is_visible`.

Listing 4.205: Object class extensions to support invisibility

```

0 private:
1   bool is_visible;      // If true, object gets drawn.
2
3 public:
4   // Set visibility of Object. Objects not visible are not drawn.
5   // Return 0 if ok, else -1.
6   int setVisible(bool visible=true);
7
8   // Return visibility of Object. Objects not visible are not drawn.
9   bool isVisible() const;

```

As shown in Listing 4.206, the method `setVisible()` allows the game programmer to set the Object visibility, changing `is_visible` as appropriate. Objects are visible (`is_visible` is `true`) by default. In addition, the SceneGraph is obtained from the WorldManager and the SceneGraph `updateVisible()` method is called.

Listing 4.206: Object setVisible()

```

0 // Set visibility of Object. Objects not visible are not drawn.
1 // Return 0 if ok, else -1.
2 int Object::setVisible(bool visible)
3
4   // Update scene graph.
5   scene_graph = WorldManager getSceneGraph()
6   scene_graph.updateVisible(this, visible)
7
8   // Set visibility value.
9   is_visible = visible

```

Listing 4.207 shows the SceneGraph `updateVisible()` method. The first block of code checks if the visibility is being changed. If not, there is nothing more to do and an "ok" (0) is returned. The second block of code does the actual work. If the Object was visible and went invisible, `remove()` is called on the ObjectList, otherwise `insert()` is called, at the right altitude (`p_o->getAltitude()`). All method calls should be error checked and an error (-1) returned, as appropriate. Otherwise, "ok" is returned at the end.

Listing 4.207: SceneGraph updateVisible()

```

0 // Re-position Object in scene graph based on visibility.
1 // Return 0 if ok, else -1.
2 int SceneGraph::updateVisible(Object *p_o, bool new_visible)
3
4   // If visibility unchanged, nothing to do (but ok).
5   if p_o->isVisible() is new_visible then
6     return ok
7   end if

```



```

8
9 // If was visible then now invisible , so remove from list.
10 if p_o->isVisible() then
11     visible_objects[p_o->getAltitude()].remove(p_o)
12 else // Was invisible , so add to list.
13     visible_objects[p_o->getAltitude()].insert(p_o)
14 end if
15
16 // All is well.
17 return ok

```

### 4.17.2 Development Checkpoint #14!

To develop the SceneGraph for *Dragonfly*, use the following steps:

1. Create the SceneGraph class, referring to Listing 4.189 as needed. Add `SceneGraph.cpp` to the project and stub out each method so the SceneGraph compiles.
2. Implement the SceneGraph `insertObject()` and `removeObject()` methods based on Listing 4.190 and Listing 4.191, respectively. Test outside of the game engine by adding and removing Objects.
3. Implement the SceneGraph `updateSolidness()` method, based on Listing 4.192 and `updateAltitude()`, based on Listing 4.193.
4. Extend the Object class `setSolidness()` to support a SceneGraph, referring to Listing 4.194. Do the same for `setAltitude()`, referring to Listing 4.195.
5. Extend the WorldManager to support a SceneGraph, as in Listing 4.196. Refactor the methods shown in Listing 4.197, as appropriate.
6. Test by verifying that previous code that worked without SceneGraphs *still* works, such as test code from the last development checkpoint (on page 206).

If support for inactive Objects is desired (optional), continue development:

1. Extend Object to support activeness based on Listing 4.198, implementing `setActive()` based on Listing 4.199.
2. Refactor the SceneGraph based on Listing 4.200, implementing `updateActive()` based on Listing 4.201.
3. Refactor the WorldManager based on Listing 4.202, extending `getAllObjects()` to return inactive Objects, too, based on Listing 4.203.
4. Test with game code that sets another game object to inactive and back, say, based upon key presses.

If support for invisibility is desired (optional), continue development:



1. Extend Object to support invisibility based on Listing 4.205.
2. Implement Object `setVisible()` based on Listing 4.206 and SceneGraph `updateVisible()`, based on Listing 4.207.
3. Test with game code that has game objects set themselves to invisible and back, say, depending upon key presses or positions on the screen.



### 4.17.3 Gracefully Shutting Down (optional)

As of now, the game programmer needs to provide the mechanism for the player to exit the game and terminate the engine.. For example, in Saucer Shoot (Chapter 3, the player can press ‘Q’ to exit. However, it can be useful for the player (and the developer) to allow standard method of closing windows to work with *Dragonfly*, also.

#### 4.17.3.1 Closing the Game Window (optional)

In many cases, a user will close an application window by clicking on the “close” button, typically a button with a ‘x’ in the upper right corner of the window border. However, up until this point, the *Dragonfly* window opened by the `DisplayManager` does not provide for a close button. However, SFML does support both providing and handling a windows close button, so *Dragonfly* can be extended to support the same.

In order to provide support for the close button, first, in `DisplayManager.h` `WINDOW_STYLE_DEFAULT` is modified to also include `sf::Style::Close` bitwise or-ed with the `Titlebar`. In other words, `WINDOW_STYLE_DEFAULT` should look like:

```
0 const int WINDOW_STYLE_DEFAULT = sf::Style::Titlebar|sf::Style::Close;
```

This provides a close button in the SFML game window that a user can click. When the user does so, this sends a `sf::Event::Closed` event to the window that can be detected along with other keyboard and mouse input in the `InputManager`. Needed extensions to the `InputManager` `getInput()` are shown in Listing 4.208. Basically, within the `while(event)` loop, the event type `sf::Event::Closed` is looked for (in addition to the keyboard and mouse events – see Listing 4.88 on page 123). If it is found, *Dragonfly* is shut down by calling `GameManager` `setGameOver()`. The header file `GameManager.h` needs to be `#included`.

Listing 4.208: `InputManager` extensions to `getInput()` to window close

```
0 // Get input from the keyboard and mouse.
1 // Pass event along to all Objects.
2 void InputManager::getInput()
3
4 // Check past window events.
5 while event do
6
7 // Special case – see if Window closed.
8 if event.type is sf::Event::Closed then
9     GameManager setGameOver()
10    return
11 end if
12 ...
```

#### 4.17.3.2 Catching Ctrl-C (optional)

In Linux and Mac, a common way to terminate a programming running in a terminal window (also called a “shell”) is by holding down the control key and pressing ‘c’ (also known as `ctrl-c`). Pressing `ctrl-c` actually sends a signal to the program that is currently running – called a *process* – in the window. The signal is a simple form of communication



between the operating system and the process, basically signifying to the process that this event (the `ctrl-c`) occurred.

Most programs interpret the `ctrl-c` as an indication to terminate – *Dragonfly* is no exception and it will end when `ctrl-c` is pressed. However, by recognizing a signal, a process has a chance to take some actions before being terminated. In particular, for *Dragonfly*, this means it can call `shutDown()` for each Manager, closing the logfile and reverting the terminal back to standard mode before the process exits. Not responding to `ctrl-c` in this way means that the process terminates abruptly, probably leaving the cursor off in the terminal window and/or having the colors changed, and possibly not writing any unwritten data to the logfile.

The mechanism to “catch” `ctrl-c` is to setup a signal handler, giving a function name to the operating system so that function can be called when a signal is passed to the process. The specific system call to do this depends upon what operating system the process is running on. On Linux, the system call is `system()`. For Windows, the system call is `SetConsoleCtrlHandler()`. The general semantics are that when the signal occurs, the current execution of the program is interrupted and the signal handler function is called. When the signal handler finishes, the program resumes where it left off. In the case of *Dragonfly*, and many other programs, the signal handler will terminate the process (via `exit()`) so it will not return.

Note, if a process does not handle `ctrl-c`, which is the default behavior, the operating system will terminate the process anyways – it is just that the process loses the opportunity to gracefully shutdown. In fact, when a computer is shutdown, the operating system first sends a `ctrl-c` signal to all processes, allowing them to shut themselves down. If they do not terminate, it next sends a “kill” signal that forcibly shuts them down, a signal they cannot handle.

First, the signal handler function is defined. Since the signal handler can be invoked from anywhere, it cannot be a method of any class, but must be a global function. This could suggest it be placed in `utility.cpp`, but since it only calls `GameManager shutDown()`, it should be placed in `GameManager.cpp`. The definition is shown in Listing 4.209 where, as stated earlier, the `GameManager` is shutdown and the process exits via the `exit()` system call.

Listing 4.209: Function `doShutDown()` in `GameManager.cpp`

```
0 // Called explicitly to catch ctrl-c, so exit when done.
1 void doShutDown(int sig)
2     GameManager shutDown()
3     exit(sig)
```

The second step is to tell the operating system to call the signal handler, `doShutDown()`, when it receives a `ctrl-c`. This is done by adding code to `startUp()`, shown in Listing 4.210. A `#include` of `signal.h` is needed for the system calls. In `startUp()`, a variable of type `struct sigaction` is created (here, named `action`) to hold the parameters for the system call. In this case, the main parameter to specify is the name of the handler, `doShutDown`. Note, `doShutDown()` must be defined above `startUp()` in `GameManager.cpp` or a function prototype must appear before `startUp()` – not doing this will result in a compiler error complaining about `doShutDown()` being undefined. The call `sigemptyset()`



initializes the set of signals to empty, and setting `sa_flags` to 0 indicates there are no special modifiers to the behavior when the signal comes. The final call, `sigaction()` enables the signal, with the first parameter, `SIGINT`, referring to `ctrl-c`, the second the `struct sigaction` data structure, and the third, `NULL`, indicating there is no interest in storing the previous action. The call to `sigaction()` should be error checked as it returns -1 on error and 0 on success.

Listing 4.210: GameManager extensions to support handling `ctrl-c` (Linux and Mac)

```

0  ...
1  #include <signal.h>
2  ...
3  // Startup all GameManager services.
4  int GameManager::startUp()
5  ...
6  // Catch ctrl-C (SIGINT) and shutdown.
7  struct sigaction action
8      action.sa_handler = doShutDown    // The signal handler.
9      sigemptyset(&action.sa_mask)    // Clear signal set.
10     action.sa_flags = 0              // No special modification to behavior.
11     sigaction(SIGINT, &action, NULL) // Enable the handler.
12     ...

```

Native Windows, handles signals a bit differently with the system call `SetConsoleCtrlHandler()` taking the name of a general-purpose signal handler, where it matches the signal type to `CTRL_C_EVENT` before calling `GameManager shutdown()`. Other signals that may be of interest for Windows processes are `CTRL_CLOSE_EVENT` (the program is being closed), `CTRL_LOGOFF_EVENT` (the user is logging off), and `CTRL_SHUTDOWN_EVENT` (the operating system is shutting down).

#### 4.17.4 Random Numbers (optional)

Many games make frequent use of random numbers. For example, spins of a virtual roulette wheel needs to bounce the ball randomly to different numbers each time; a maze generation algorithm needs randomness to decide on how to carve twists and turns; or a computer opponent needs some random behavior to make it difficult to figure out its next move. Without randomness, the roulette wheel will always fall on the same number, the maze will always be exactly the same, and the computer opponent will be boringly predictable.

However, true randomness is difficult for computers – after all, at their core, computers are binary – either a 0 or a 1 and nothing in between. Instead, computers provide *pseudo-randomness* by generating sequences of numbers that, to the eye (and to the game player) look random. The most sophisticated of such algorithms even withstand external tests that make it difficult to tell the sequences apart from true randomness. For example, consider the sequence generated by:

$$X_n = ((5 \times X_{n-1}) + 1) \bmod 16 \quad (4.1)$$

Assume  $X_0$  is 5. Then  $X_1 = ((5 \times 5) + 1) \bmod 16$ , or  $X_1 = 26 \bmod 16$ , or  $X_1 = 10$ . Doing the same computation for  $n = 1, 2, \dots$  gives the numbers 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14... It is difficult to look at this pattern and guess what number is next – to the eye





(and to the game player) it looks random. If the sequence was restarted, say, the next time a game was played, the sequence would be the same and the game player might then be able to predict what number would come next having seen the sequence during the previous game. Note, however, that the sequence generated totally depends upon the starting value ( $X_0$ ). The starting value, also called the “seed” since it provides the basis for the sequence that follows, can be changed each time the game is run to provide a different, unpredictable sequence. Of course, being able to make the seed random would appear to put us back at square one. However, a good “random” seed is to start the sequence based on the time of day in seconds. Since each time the game is run the time will be different, thus providing a different seed and, hence, a different random sequence.

In order to make the “random” sequence easy to use, two functions are provided – one to set the seed, and one to return the next number in the sequence. An example of a `rand()` function to generate random numbers is shown in Listing 4.211. The variable `g_next` is declared globally since it needs to be maintained from call to call. By default, `g_next` is initialized to a value in seconds since 1970 (see `time()` described in Section 4.3.2 on page 55), a starting value that will seem random to the player and vary from game run to game run. If the starting value (or actually at any time) needs to be explicitly controlled by the programmer, the function `srand()` sets `g_next` explicitly to the value `seed`.

Listing 4.211: Pseudo-random number functions

```

0 // Global variable to hold random number for sequence.
1 static unsigned long g_next = time()
2
3 // Set seed to get a different starting point in sequence.
4 void srand(int seed)
5     g_next = seed
6
7 // Generate ‘random’ number.
8 int rand()
9     g_next = ((5 * g_next) + 1) mod 16

```

While aspiring programmers could research good random number generating algorithms and code their own `rand()` and `srand()` functions, standard library functions already exist that do a good job. Namely, `srand()` can be used set the seed and `random()` (Linux) and `rand()` (Windows) to return the next random number in the sequence. The random function returns an arbitrarily large integer, but typically a game programmer wants a random number bounded to something smaller. For example, a dice roll would be in the range 1–6. In order to map, say, `random()` to the range 1–6, the game programmer calls `(rand() % 6) + 1`.

`Dragonfly` itself does not use `rand()`. Instead, all the engine needs to do is control the seed, setting it to the time (in seconds) by default, while providing a means for the game programmer to control the starting seed value when the engine starts up. This is done by extending the `GameManager` `startUp()` in a couple of ways.

First, `GameManager` `startUp()` calls `srand()` with the `time()` call (e.g., `srand(-time(NULL))`) in order to provide a random-looking starting point for the programmer to make subsequent calls to `rand()`. `GameManager.cpp` needs to `#include` both `<time.h>` and `<stdlib.h>` for the `time()` and `random()`, `srand()` function calls, respectively.



Second, as shown in Listing 4.212, a new `startUp()` method is provided, this one allowing the game programmer to set the seed explicitly (by calling `srand(seed)`).<sup>19</sup> The rest of the method is exactly the same as the normal GameManager `startUp()` method.

Listing 4.212: GameManager extension to support random seeds

```

0 public:
1 // Startup all GameManager services.
2 // seed = random seed (default is seed with system time).
3 int startUp(time_t seed=0);

```

## 4.18 Development Checkpoint #15 – Dragonfly!

(Optional)

Finish off your Dragonfly development!

1. Implement the `doShutDown()` function based on Listing 4.209. Test by having a game object call `doShutDown()` explicitly and verify in the logfiles that the engine shuts down properly.
2. Extend the GameManager to support handling closing the game window and `ctrl-c` (Linux), following guidelines from Listing 4.208 and Listing 4.210, respectively. Be sure to error as appropriate, writing informative errors to the logfile as needed. Test with a simple “game” that does not terminate, but can be ended by closing and/or `ctrl-c`. Verify these actions are actually caught via messages in the logfile.
3. Extend the GameManager to support random seeds, based on Listing 4.212. Test with a simple “game” that generates random numbers. Verify that the numbers produced are the same each run by using a fixed, explicit seed.

After completing the above steps (and all the previous Development Checkpoints), you will have a fully functional, full featured game engine! Features include everything from Dragonfly Naiad (Section 4.11 on page 142), plus:

- Velocity support for game objects.
- Objects with bounding boxes, enabling multi-character sizes.
- Collisions for boxes, not just single characters.
- Objects with sprites, giving them animated frames.
- Support for audio, including sound effects and music.
- View objects for display elements, each with a value and HUD-type location.
- Camera control for the game world, enabling “views” over a subset of the world with explicit camera control and support for the camera following a specific object.

<sup>19</sup>The type `time_t` is the type returned by `time()`, but can be thought of as an integer.



- (Optional) Objects register for interest in events (e.g., step events), getting notification for only those events.
- (Optional) Scene graph support for more efficient queries.
- (Optional) An inactive feature for game objects to temporarily not have the engine act upon them.
- (Optional) Invisible objects that are not drawn but otherwise interact with the world normally.
- (Optional) Ability to gracefully close the game through closing the window and/or catching `ctrl-c`.
- (Optional) Automatic random number seeding, with the ability of the game programmer to control initial seeds.

If implemented fully, including all the optional components, the Saucer Shoot game from Chapter 3 should compile and run with *your Dragonfly!*. Of course, that is just the start – bigger and better games are waiting to be built using your engine! Or, your new found engine knowledge is ready to be applied to learning a commercial engine or in a related game development project.

Have fun!

