Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #4

Dragonfly Egg

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

4.5.4 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. 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. 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. However, while updating the world (so, in the middle of a game loop iteration), the game world may be in an inconsistent state. This latter fact is important in handling how game objects are deleted (more on this, later).

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 4.45. 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 4.45: Game loop with update

```
0 ObjectList world_objects
1 while (game not over) {
2   ...
3   // Update world state.
4   for i = 0 to world_objects count
5     Object p_o = world_objects[i]
6     p_o -> Update()
7   end for
8   ...
9 }
```

As an abstraction, use of the `Update()` method for all game 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 a game object often consists of: moving the object (including checking and responding to collisions), then drawing the object on the screen. For a Saucer from Saucer Shoot in Section 3.3, this might look like the code in Listing 4.46. 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 same game loop iteration. The serial nature of updates for each game does not allow for tuning. Worse, in some cases, game objects cannot be drawn until the position of other game 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 4.46: Possible Update() method for Saucer

```
// Update saucer (should be called once per game loop).
void Saucer::Update()
  WorldManager move(this)
  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 game object itself. These are instead handled in phases by the game engine, depicted in Listing 4.47. Note, the *Update()* method for each game object can still be invoked, calling game code, to do any game-specific functionality that is needed.

Listing 4.47: Game loop with phases

```
ObjectList world_objects
while (game not over) do
  ...
  // Update Objects
  for i = 0 to world_objects count
    // Update
  end for

  // Move Objects
  for i = 0 to world_objects count
    // Move
  end for

  // Draw Objects
  for i = 0 to world_objects count
    // Draw
  end for
  ...

end while
```

4.5.5 Events

Games are inherently event-driven. In the previous section, each iteration of the game loop is often treated as an event. In other words, in Listing 4.47, 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, a network packet arrives, etc.

Generally, when an event occurs, an engine: 1) notifies all interested objects, and 2) those objects respond as appropriate, also called 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 the player is controlling 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 and remain unharmed.

The simple approach to dealing with game events is for the game engine to call the appropriate method for each game object when the event occurs. In Listing 4.47, this means that each step of the game loop (a step event) invokes the `Update()` method of each game object. Consider another example, where there is an explosion in a game, handled in the `Update()` method of an Explosion object, shown in Listing 4.48. In this case, all game objects within the radius of the explosion have their `onExplosion()` method invoked.

Listing 4.48: 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      for i = 0 to damaged_objects count
9        damaged_objects[i] -> onExplosion()
10     end for
11
12     ...
13   end if
```

Listing 4.48 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 9) are known at compile time. All this sounds ok, and Listing 4.48 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 game objects must have methods for all events. Specifically, for this example, it means every game object needs an `onExplosion()` method, even if not all objects use it. The fact that an game 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 game 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 for the game programmer – 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 game objects that are interested in that event. In the event handler, the event 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, ...), and can be extended by the game programmer. Representing events this way has several advantages over the approach in Listing 4.48.

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 game 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, 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., `std::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 54), in front of game engine event names (e.g., `"df::step"`). If needed (or for really large development efforts), more elaborate software tools 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 vector. 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 4.49 shows how this might be declared.[11] 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 4.49: 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 solider 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 4.50 illustrates how a chain of responsibility might look for a particular game. On 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 someGameObject, but are not consumed in that other objects can respond to the damage, too. On line 10, health pack events are consumed, so other objects in the chain do not handle them. Unrecognized events, line 15, are not handled.

[11] Note, methods to set the event type are not shown.

Listing 4.50: Chain of responsibility

```
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 4.50 is almost right – but the compiler will throw up an error at lines 7 and 11.

4.5.5.1 Events in Dragonfly

Listing 4.51 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 4.51: 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 34). Dragonfly recognizes (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.

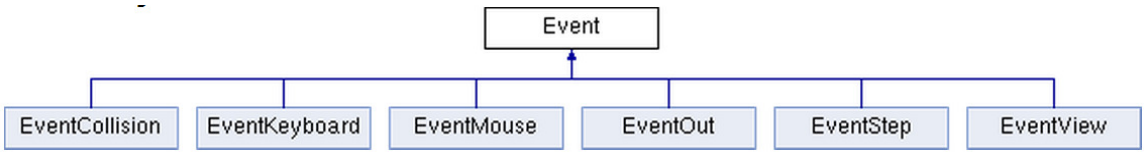

Figure 4.2: Dragonfly events

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 is 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 4.52 provides the header file for the EventStep class. EventStep is derived from the Event base class. The `private` attribute `m_step_count` is to record the current iteration number of the game loop. Methods are provided to get and set `m_step_count`, as well as a constructor to set the initial `m_step_count`, if desired. Other "work" done in the constructor is to set the base event type (using `setType()`) to `STEP_EVENT`.

Listing 4.52: 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 4.53, 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).

Tip 12! Dereferencing pointers and invoking methods. To invoke an Object method from inside the game engine, the object pointer is dereferenced. Normal dereferencing uses '`*`', and normal method invocation uses '`.`'. However, combined, the preferred syntax is to use '`->`'. For example, to check the type of a game object named `p_e`, the code could be written as `(*p_e).getType()`. However, the preferred syntax, and identical functionality, is written as `p_e->getType()`.

Listing 4.53: 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 4.54 shows pseudo code for sending step events to all objects in the game world. Line 3 gets all the Objects to iterate over from the WorldManager (see Section 4.5.6 on page 100). Line 2 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 6 iterate through all Objects in the world, passing the step event to the Object event handlers in line 5.

Listing 4.54: Sending step events

```
0   ...
1   // Send step event to all Objects.
2   create EventStep s(game_loop_count)
3   wo = WorldManager getAllObjects()
4   for i = 0 to wo count
5     wo[i] -> eventHandler() with s
6   end for
7   ...
```

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.

> **Tip 13! Heeding warnings.** In general, warning messages from a compiler should *not* be ignored. The compiler is indicating something is potentially amiss when it throws up a warning. A programmer should pay attention to any warning, resolving it whenever possible (and usually it *is* possible), such as, for example, casting to indicate to the compiler that an implicit conversion is intended. Even if the current warnings are harmless in that the code still executes fine, by ignoring them, it makes it more likely that future warnings, that may *not* be harmless go unnoticed.

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*.[12] 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 4.55. The value of `p_b` of type `Base *` is converted to a different type, type `p_d`.

Listing 4.55: 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

[12] The C-style cast (e.g., `int x = (int) 4.2`) is generally replaced with a `static_cast` in C++.

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 4.56 the Bullet's `eventHandler()` from Saucer Shoot (Section 3.3 on page 15) checks if the event type is a collision event (`COLLISION_EVENT` – see Section 4.10.1.2 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 4.56: 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_ce = dynamic_cast<const df::EventCollision *>(p_e)
4     hit(p_ce)
5     return 1
6   end if
7   ...
```

4.5.6 The WorldManager

At this point, development of the game world has provided game objects, lists for those game 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 objects 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 `m_updates` is a list holding all the game objects in the world; and 2) Line 11 `m_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 4.57: 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
```

```
  ObjectList m_updates;     // All Objects in world to update.
  ObjectList m_deletions;   // All Objects in world to delete.

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

  // Startup game world (initialize everything to empty).
  // Return 0.
  int startUp();

  // Shutdown game world (delete all game world Objects).
  void shutDown();

  // Insert Object into world. Return 0 if ok, else -1.
  int insertObject(Object *p_o);

  // Remove Object from world. Return 0 if ok, else -1.
  int removeObject(Object *p_o);

  // Return list of all Objects in world.
  ObjectList getAllObjects() const;

  // Return list of all Objects in world matching type.
  ObjectList objectsOfType(std::string type);

  // Update world.
  // Delete Objects marked for deletion.
  void update();

  // Indicate Object is to be deleted at end of current game loop.
  // Return 0 if ok, else -1.
  int markForDelete(Object *p_o);
};
```

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 `m_updates` ObjectList. A similar method, `objectsOfType()` returns a list of Objects matching a certain type. This method, shown in Listing 4.58, iterates through all Objects in the `m_updates` list and each Object that matches in type is added to the ObjectList, returned at the end.

Listing 4.58: WorldManager objectsOfType()

```
// Return list of Objects matching type.
// List is empty if none found.
ObjectList objectsOfType(std::string type)

  ObjectList list
  for i = 0 to m_updates count
```

```
    if m_updates[i] -> getType() is type then
      list.insert(m_updates[i])
    end if
  end for

  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()` `delete`s all the Objects in the game world. Typically, the game code does not preserve the addresses of game objects created to populate the world so only the WorldManager can do so. Pseudo code for WorldManager `shutDown()` is shown in Listing 4.59.

Listing 4.59: WorldManager shutDown()

```
// Shutdown game world (delete all game world Objects).
void WorldManager::shutDown()

  // Delete all game objects.
  ObjectList ol = m_updates // Copy list so can delete during iteration.
  for i = 0 to ol count
    delete ol[i]
  end for

  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 4.60.

Listing 4.60: Event handler prototype

```
// Handle event (default is to ignore everything).
// Return 0 if ignored, else 1 if handled.
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 4.61. 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 4.61: Object Object()

```
// Construct Object. Set default parameters and
// add to game world (WorldManager).
Object::Object()

  // Add self to game world.
  WorldManager insertObject(this)
```

Similarly, the destructor needs to remove the Object from the game world. A code fragment for this is shown in Listing 4.62. 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 4.62: Object ~Object()

```
// Destroy Object.
// Remove from game world (WorldManager).
Object::~Object()

  // Remove self from game world.
  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 iteration 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 (say, popping red balloons earns more points than popping green balloons). 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 during gameplay. 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 4.63. 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 4.63: 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   for i = 0 to m_deletions count
6     if m_delections[i] is p_o then // Object already in list.
7       return 0                     // This is still "ok".
8     end if
9   end for
10
11  // Object not in list, so add.
12  m_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 `m_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 4.64.

Lines 5 to 8 iterate through all Objects that have been marked for deletion, actually deleting them by calling `delete` in line 8. Line 11 clears the deletion list (so there are no Objects in it) to get ready for the next phase.

Listing 4.64: WorldManager update()

```
0 // Update world.
1 // Delete Objects marked for deletion.
2 void WorldManager::update()
3
4   // Delete all marked Objects.
5
6   for i = 0 to m_deletions count
7     delete m_deletions[i]
8   end for
9
10  // Clear list for next update phase.
11  m_deletions.clear()
```

4.5.7 Program Flow for Game Object Lifetime

This section provides a summary of the lifetime 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., before `Saucer()`).
3. The Object constructor, `Object()`, calls WorldManager `insertObject()` to request being added to the game world.
4. WorldManager `insertObject()` calls `insert()` on the `m_updates` ObjectList, thus adding the game object to the game world.
5. Any remaining code is the derived constructor, `Saucer()`, is run.

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 `m_deletions.insert()` to add the object to the `m_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 `m_deletions` ObjectList, calling `delete` on each Object in the list. The `delete` triggers the derived Object's destructor (e.g., `~Saucer()`).[13]
5. After the derived Object's destructor (e.g., `~Saucer()`) is run, it 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 `m_updates` ObjectList, removing the saucer from the game world.

[13] Remember, in C++, `delete` invokes an object's destructor *and* frees memory allocated by `new`.

4.5.8 Dragonfly Testing

This section provides some basic advice for getting started with game engine testing suitable for completing Dragonfly Egg. The idea is to test functionality, both large and small, in a modular fashion and, where possible, isolate the test code from the game engine code. Small here, means individual methods, but also building up combined use of methods to test integrated functionality. Large here means integrating functionality from several classes (e.g., testing the GameManager). A detailed treatment of testing is provided in the "Taking Flight" chapter.

For Dragonfly Egg development, initial testing is most easily done by isolating test code in a function and calling it from `main()`. Each test, large and small, should be in a *separate* function, allowing each function to be called individually. Commenting out individually test functions can be done to "turn off" tests that are not needed at that time, but still keeping the code around for later use. This latter idea – running a full set of tests, including tests that have previously "passed" – is commonly called regression testing and can be valuable for catching bugs that might arise in previously written code when adding new, seemingly unrelated, code.

An example of the suggested testing is shown in Listing 4.65. There are two tests written, `testClock_timing()` and `testStepEvent()`, that correspondingly test if the timing aspects of the Clock class and step events are working properly. Each function returns `true` if the test passes and `false` if it fails. In `main()`, the individual functions are called with the success/failure of the individual tests indicated. A tally of passes and failures could easily be added (e.g., `tests_taken++` and `tests_passed++`) and a summary provided (e.g., `test_passed` out of `test_taken`). The test functions themselves should write liberally to the logfile in order to confirm that tests pass and, when they fail, help to figure out *why*.

Listing 4.65: Isolating test functions

```
0  // Test fucntion prototypes.
1  bool testClock_timing()
2  bool testStepEvent()
3
4  main() {
5    if (testClock_timing())
6      puts("Pass")
7    else
8      puts("Fail")
9
10   if (testStepEvent())
11     puts("Pass")
12   else
13     puts("Fail")
14 }
```

An example of a `testClock_timing()` function is shown in Listing 4.66. The function creates a Clock object, with the timing aspects in the `sleep()` and `split()` calls starting on line 6. If the split time is not 1 on line 13, then an error is logged and `false` is returned. Otherwise, the function passes and `true` is returned. Note, `__func__` on line 19 is a built-in constant string that holds the name of the calling function (i.e., "testClock_timing()" in this example).

Listing 4.66: testClock_timing()

```
// Test the Clock class using second granularity.
// (Note, finer timer granularity should be tested, too.)
bool testClock_timing(void) {
  df::Clock clock;

  clock.delta(); // Start time.
  sleep(1); // Adjust to Mac/Linux/Windows.
  int t = (int) clock.split() / 1000000; // About 1 second.

  // Print time to logfile for debugging.
  LM.writeLog("split time t is %d", t);

  // See if reported 1 second as expected.
  if (t != 1) {
    LM.writeLog("split time t is %d", t);
    return false;
  }

  // If we get here, test has passed.
  LM.writeLog("%s passed.\n", __func__);
  return true;
}
```

As a final note, be aware that writing good tests – tests that inform whether or not code is working – takes time and skill, just like writing game engine code. The more you do, the better you get. Similarly, interpreting test failures takes time and skill. For tests that fail, additional work is involved in fixing the bugs they might be revealed. Make sure to re-run failed test after fixing the bug to be sure it is really fixed! And keep such tests around for future regression testing. All of this becomes easier with practice.

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 4.51. 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 4.52. 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 4.60. 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.

4. Create the WorldManager class based on the header file in Listing 4.57. 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 4.61 and Listing 4.62 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 4.63 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 4.54), 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" itself 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 (derived from Object), 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.

Tip 14! Source code control. In developing Dragonfly (and most other software projects of significant size), it is strongly urged to use a source code control (also called *version control*) system. Source code control systems help mange changes to computer programs, associating files by time and version names. While such features are critical for development teams, they are often helpful for independent developers, too, providing invaluable check pointing for working code. Checking in working versions of Dragonfly, say *Egg*, can help by preserving working code if future development breaks the code base. Similarly, source code control can provide a backup in case code is lost (e.g., a disk failure). Local source code control systems that do not back up over the network (e.g., `RCS`) should be periodically copied to another location (alternate local storage or, better, offline to another machine or cloud storage) in case of computer hardware failure.

* **Did you know (#4)?** 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.