Program a Game Engine from Scratch

Mark Claypool

Chapter 6 - Taking Flight

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/`

Chapter 6

Taking Flight

6.1 Testing

6.1.1 Overview

Testing, often overlooked by the novice programmer, is known to be critical for the experienced programmer, particularly as code complexity (e.g., size!) increases. Figure 6.1 coarsely depicts this relationship for the development of a single project (i.e., a single-software code base).

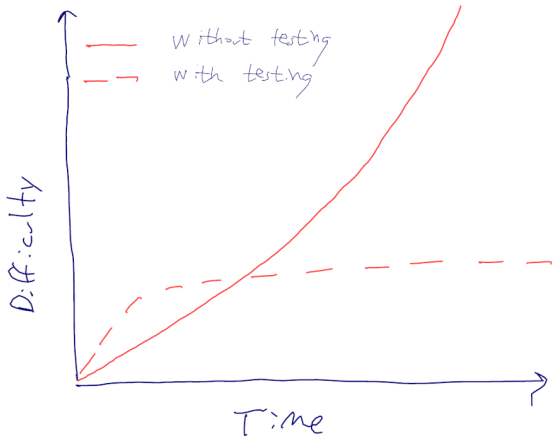

Figure 6.1: The difficulty of code development over time.

The horizontal axis represents time spent on developing code on the project and the vertical represents the difficulty in writing bug-free, functional code. There are two trend lines shown – the solid curved line represents the difficulty over time relationship for developing code without formalized testing. Note that the slope is getting steeper and steeper as time progresses and the code gets more complex. With formalized testing, the difficulty flattens out. Why? Because structured, thorough testing ensures that code that is written and expanded upon works as expected, without bugs or unexpected behaviors, allowing

additional code to be added to this base. Effectively, adding new code is no more complex than the code developed before it.

A keen observer will note that the difficulty for developing with formalized testing is a bit more difficult initially (see the bottom left corner of Figure 6.1) than development without testing. This is because it takes bit more effort to define tests that are needed and setup a testing framework. However, once done, this extra time will pay immense dividends as the project progresses. In short, testing takes a bit more time up front, but saves time later on. Put it another way, for a large project, there is not enough time *not* to write tests. This section also provides a framework that can be leveraged to help with some of this testing, reducing the time-costs of adding new tests.

Also, another misconception may be that testing is for someone else, a quality-assurance (QA) team or something similar. That's not the case. Testing is for developers not just testers – aspiring programmers should embrace testing as just one of the aspects of development.

Moreover, testing early and often has another benefit – finding bugs early in the development process costs a lot less (in time, and time is money) than finding a bug later in the process. Roughly, finding a bug in an initial class implementation costs about 10-times less than finding a bug when verifying a functioning system, and about 40-times less than when finding a bug in final integration testing!

6.1.2 Types of Tests

Definitions of test names and types vary, but broadly there are at least these types of tests relevant to systems such as Dragonfly:

- Unit test – a unit test tests the smallest unit of program code, such as a function or a class method. For example, a unit test may check that the LogManager `writeLog()` method with two arguments produces the expected output, checking the message and any pre-pended frame or time string.
- Component test – A component test tests a set of units integrated together, such as a module or a subsystem that is composed of an isolated set of classes. For example, the WorldManager and all its supporting classes and even the Dragonfly engine could be tested via a component test.
- Integration test – An integration test tests several component layers together. For example, a integration test could verify that a Saucer Shoot gameplay (Hero and Saucers) plus the accompanying Dragonfly engine work as expected.
- System test – A system test tests that several stand alone applications are integrated, often the full system. For example, test Saucer Shoot game (Splash screen, Game Start, and core gameplay) with the Dragonfly engine could be tested in a system test.

Another important test type is:

- Regression test – regression testing verifies that all previously developed code still works, despite additional functionality that has been added or bugs that have fixed.

When a bug (a regression) is found and corrected, a test case is written to verify that the bug is, indeed, fixed. That test is preserved and run during subsequent development, ensuring that later code does not re-introduce the fixed bug.

6.1.3 Unit Tests

Since unit tests are the lowest-level building block they are the right place to start in writing tests.

For Dragonfly, unit tests are written from the game engine programmer's perspective. Contrast that to a functional test, one that confirms the system works as expected, is written more from the perspective of the game programmer.

Unit tests isolates part of program, testing a single behavior. For C++, this typically means a single function or class method. In addition, a unit test will clearly indicate pass or fail, with the latter accompanied by a reason for the failure (e.g., unexpected output). For a decent sized program, there may be many unit tests so unit tests should run quickly.

Although this section appears in the book after the engine development, in practice unit tests should be developed in *parallel* with code development. Moreover, the suite of unit tests (and regression tests) previously developed should be run (successfully!) before any code is checked into a source code repository.

Some guidelines for unit tests are:

- test edge conditions (e.g., boundaries on a list are when it is full or empty).
- test invalid inputs and other error conditions (e.g., drawing a character in an invalid color or at a location that is outside the game world).
- test multiple invocations of the code under test (e.g., repeatedly adding the same object to a list).
- test previously detected bugs found in your code (i.e., regression tests).

If a unit test fails, it is time to take action. If the immediate cause of the failure is not clear, additional output (e.g., logging) might be needed. Or, smaller tests may need to be written to more narrowly isolate the failure. Either way, the code should be fixed since it is likely that subsequent code that builds upon the unit will also then fail. Keeping unit tests around (and re-running periodically) ensures bugs are not introduced later and that fixed bugs are not re-introduced.

Broadly, a unit test has 4 phases:

1. *Setup test conditions.* For example, starting up the game engine or clearing log files.
2. *Call the method or function being tested.* For example, call the method `LM.writeLog( "This is unit test number %d", 12)`.
3. *Verify that the output results are correct.* For example, check the string written to the file "dragonfly.log" and make sure it has exactly "This is unit test number 12".
4. *Clean up.* For example, deleting any temporary files or shutting down game engine.

Note, while step 2 and 3 are critical (i.e., invoking the test and then making sure it works), steps 1 and 4 may not be needed for all unit tests.

The unit tests themselves should be maintained as if they are code (they are!). For example, if a previously developed class is re-factored, say, by modifying a method, then the unit tests that exercised that method should be updated, as well. Unit tests are not just written, and run once, but rather are written, run, re-run, modified, re-run, re-run, modified again, re-run and so on, all during engine development.

A good developer should resist pressure to "add tests later". It may seem that writing tests slows down the development process but in reality, for a large code base, it speeds it up. Yes, it takes a bit of extra time to write and run tests for a recently completed class method, but it helps make sure the class really is complete, and allows development moving forward to concentrate on newly added code having more confidence the existing classes work as they are supposed to. And any failed tests should be fixed as soon as possible. Ignoring the failed test, even if other code appears to work, makes development moving forward harder since it becomes difficult to know whether new errors are from new code or from the previous code that has fails its tests.

Along the same lines, frameworks to run tests automatically can be quite useful, even for moderate scale development (such as for Dragonfly). A good test framework will, for each unit test, invoke the 4 unit tests phases listed above and report all test results. Once automated, a development process may run unit tests daily, say at the end of a workday or overnight, providing test results in a daily log file. Similarly, before committing any code to a source code control system, unit tests should be written and run – this includes prior tests that may not seem related to the newly added code since errors caused by code can often manifest elsewhere, outside the code itself (see Section 5.1 on page 250).

6.1.3.1 Unit Test Manager

A quick Internet search will show there are dozens of unit testing frameworks that could be used. Creating a custom unit test framework is not too difficult, either. In the spirit of this book (programming a game engine from scratch), this section describes the design and use of the *Unit Test Manager* (UTM), a basic framework for doing unit testing. UTM is available for download at:

`http://dragonfly.wpi.edu/games/`

Figure 6.2 depicts an example of output from a UTM session. This session has been configured to run five tests, number 0 to 4. Tests 0-2 and 4 all passed, but test 3 failed. A summary of the number of tests passed and the test total is at the bottom. The name of the test is on the far right - e.g., `test_GameManager_getSetGameOver` failed. Although not shown, details on the test output is written into a logfile, named "utm.log" by default.

The first step in using UTM is code to set it up. Listing 6.1 depicts basic instructions to do so. Note the UnitTestManager is a singleton, like game engine managers. The for method calls setup before and after testing functions to call.

Listing 6.1: Setup UTM

```
0 UnitTestManager &utm = UnitTestManager::getInstance();
1 utm.setSetupFunc(&testSetup);
2 utm.setCleanupFunc(&testCleanup);
```

```
PASS  test [0]: testObject_setGetMethods
PASS  test [1]: testLogManager_bytesWritten
PASS  test [2]: testManager_startStop
FAIL  test [3]: testGameManager_getSetGameOver
PASS  test [4]: testClock_deltaAndSplit
Summary: 4 of 5 tests passed
```

Figure 6.2: UTM - Unit Test Manager example output.

```
utm.setBeforeFunc(&testBefore);
utm.setAfterFunc(&testAfter);
```

Next, the individual tests to run are loaded – examples are shown in Listing 6.2. Each call to `registerTestFunction()` provides the string name of the function, followed by a pointer to the function itself.

Listing 6.2: Setup UTM tests

```
utm.registerTestFunction("testObject_setGetMethods",
                         &testObject_setGetMethods);
utm.registerTestFunction("testLogManager_bytesWritten",
                         &testLogManager_bytesWritten);
utm.registerTestFunction("testManager_startStop",
                         &testManager_startStop);
utm.registerTestFunction("testGameManager_getSetGameOver",
                         &testGameManager_getSetGameOver);
utm.registerTestFunction("testClock_deltaAndSplit",
                         &testClock_deltaAndSplit);
```

Finally, the tests are ready to be run. Listing 6.3 shows sample code that runs either all the tests or just a specific test, depending upon the number of arguments passed in from the command line. Remember, `argv[1]` refers to the first command line argument (after the program name itself) and `atoi()` converts a C-style string string to an integer.

Listing 6.3: Run UTM tests

```
  int passed;
  if (strcmp(argv[1], "all") == 0)
    passed = utm.run(UTM_ALL_TESTS);
  else
    passed = utm.run(atoi(argv[1]));
```

For Dragonfly, an appropriate test before would setup the LogManager log level (assuming logging level is implemented – see Section 4.12 on page 63), startup the game manager, and set output flushing to `true`.

lst:sample-utm-testbefore

Listing 6.4: Sample UTM function - testBefore()

```
// Setup conditions before individual tests.
void testBefore(string test_name) {

  LM.setLogLevel(20);
```

```
4
5   // Start up the game manager.
6   GM.startUp();
7
8   // Make sure LogManager flushes in case of crashes.
9   LM.setFlush(true);
10 }
```

An appropriate test after would shutdown the GameManager and then rename the logfile ("dragonfly.log") to be a different name, one per test, for reference.

Listing 6.5: Sample UTM function - testAfter()

```
0  // Cleanup conditions after individual tests.
1  void testAfter(string test_name) {
2
3    // Shut down the game manager.
4    GM.shutDown();
5
6    // If there is a Dragonfly logfile, rename it to be log/test_name.log.
7    if (access("dragonfly.log", F_OK) != -1) {
8      string new_name = LOG_DIR;
9      new_name += test_name;
10     new_name += ".log";
11     if (rename("dragonfly.log", new_name.c_str()) == -1)
12       fprintf(stderr, "testAfter(%s) rename() error: %s\n",
13               test_name.c_str(), strerror(errno));
14   }
15 }
```

The actual tests themselves (those provided in Listing 6.2) need to be defined. That's the real work, needing to be done for each class and method. Listing 6.6 provides an example of a test function that test the Clock class' `delta()` and `split()` methods, verifying that they report about the right time for a second of sleep. After setting up the clock, sleeping and then calling `split()`, Line 16 checks if the time reported is about 1 second. If not, the test fails and `false` is returned. Otherwise, the test passes and `true` is returned. Note, the macro `__FUNCTION__` on line 20 returns the name of the function currently being invoked as a C-style string.

Listing 6.6: Example Unit Test - Clock delta() and split()

```
0  // Note: only 1 second granularity.  Needs
1  // additional tests with finer clock granularity.
2  bool testClock_deltaAndSplit(void) {
3
4    // Make a clock object for testing.
5    df::Clock clock;
6
7    // Make calls to delta() and split().
8    clock.delta();
9    sleep(1);
10   int t = (int) clock.split() / 1000000; // About 1 second.
11
12   // Print time to logfile for debugging.
13   LM.writeLog("split time t1 is %d", t);
```

```
14
15    // See if reported time is as expected.
16    if (t != 1)
17      return false;
18
19    // If we get here, test has passed.
20    LM.writeLog("%s passed.", __FUNCTION__);
21    return true;
22  }
```

Similar tests could be written for each class and method, returning `true` whenever the test passed and `false` if not. The UTM framework, with at least some of the sample code above, then allows easy invocation of the tests.

In summary, writing unit tests takes time and skill. Likewise, interpreting unit test failures takes time and skill. Fortunately, like many skills (including coding), it improves with practice. Writing tests goes hand-and-hand with finding and fixing bugs, a skill that also gets better with practice.

6.2 Particles

Many modern game engines provide for particle systems that can be used by game programmers for compelling background and environmental effects. While game programmers can do without such systems since most particle effects can be achieved through artistic animation (e.g., creating a Sprite with the effect), particle systems provide a measure of convenience for the game programmer for two main reasons: 1) they do not need to make the art; and 2) particle systems make the effects random, whereas the same animation played over and over can look repetitive.

The core for the Dragonfly particle system is the Particle class, shown in Listing 6.7. Basically, a Particle is a derived class of Object that is SPECTRAL. Unlike most Objects that use text-based Sprites, a Particle uses an SFML shape – in this case, a circle – which it draws itself in a custom `draw()` method. An additional attribute is how long the Particle will live (`age`, in game loop ticks). The Particle handles step events so it can age itself (i.e., get older) and expire when it is time. The colors are not stored as attributes, but are used to set the color of the SFML shape in the Particle constructor.

Listing 6.7: Particle.h

```
0  // System includes.
1  #include <SFML/Graphics/CircleShape.hpp>
2
3  // Engine include.
4  #include "Object.h"
5
6  class Particle : public Object {
7
8   private:
9    int age; // Age to live (in ticks).
10   sf::CircleShape shape;
11
12  public:
```

```
  // Create particle with size (pixels), age (in ticks), opacity
  // (0-255) and rgb color.
  Particle(float size, int age, unsigned char opacity,
           unsigned char r, unsigned char g, unsigned char b);

  // Set age.
  void setAge(int new_age);

  // Get age.
  int getAge() const;

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

  // Draw particle.
  virtual int draw() override;
};
```

The Particle is handled by the game engine as are all other game objects. However, since a Particle has a defined `draw()`, it draws an SFML shape at its position instead of an animated Sprite. Also, when the Particle `eventHandler()` ages the particle (decrements `age` during step events), it checks the age and, when it reaches 0 it destroys itself.

A single Particle is simple, but its visual power comes from creating a lot of them. The particle system provides utility functions to help create a lot of particles. A shortened version of a Dragonfly utility function `addParticles()` is shown in Listing 6.8.

Listing 6.8: Utility addParticles() – general

```
// Add particles.  Each parameter has average and spread.
//  count - number to add
//  position - location
//  size - size (pixels)
//  speed - speed (spaces/tick)
//  age - age (ticks)
//  opacity - how "see through" [0-255, 0 is transparent]
//  r, g, b - color in RGB values
// Return 0 if ok, else -1.
int addParticles(...)
  for i = 0 to count
    randomize opacity
    randomize age
    randomize size
    randomize color
    create Particle with color, size, opacity, age
    set position
    randomize speed
    randomize direction
  end for
```

Clusters of similar types of particles can achieve a variety of effects. Downward moving grey particles spread across screen can look like rain. Sideways moving white particles of different sizes and speeds can look like a starfield. Randomly expanding bright red particles can look like sparks. So, versions of `addParticles()` create clusters of particles, shown

with their headers only in Listing 6.9.

Listing 6.9: Utility addParticles() - cluster effects

```
// Add particle effect of specific type.
//  type: SMOKE, SPARKS, RINGS, FIREWORKS
//  position - location
//  scale - size of particle effect
//  color - base color
// Return 0 if ok, else -1.
int addParticles(ParticleType type, Vector position, float scale,
                 Color color)

// Add environment particles of specific type.
//  type - type of particle: RAIN, SNOW, STARFIELD
//  scale - scale size (default 1.0)
//  color - dragonfly color to use (default 'built-in')
// Return 0 if ok, else -1.
int addParticles(PrecipitationType type, Direction direction, float scale,
                 Color color)
```

Consider the explosion in the Saucer Shoot tutorial when a Bullet hits a Saucer. This effect is achieved by an Explosion sprite – 8 frames that, when animated, look like a spaceship exploding into a ball of fire then fading to nothing. To create the animation, each frame was created by hand ahead of time, then tweaked to get the right explosion effect. A similar effect can be achieved with the Dragonfly particle system, where a mass of red Particles emerge from a single location, spread out randomly and then fade away. Thus, an explosion effect can be created with one line, shown in Listing 6.10. The first parameter indicates the effect is `SPARKS`, the second the location (e.g., the Bullet at the point of impact with a Saucer), and the third is the scale of the effect.

Listing 6.10: Example call to addParticles() for explosion effect

```
df::addParticles(df::SPARKS, getPosition(), 2.0);
```

Listing 6.11 shows pseudo code for an excerpt of the inner workings of `addParticles()` method. At the top, if the function checks if the particle type desired (a parameter passed in to the function) is `SPARKS`. If so, Particle settings appropriate for "sparks" are set. For example, 50 particles, each 1.5x as big as a normal particle moving 0.08 spaces/second. Such parameters are adjusted based on the `scale` parameter passed in, allowing for general tweaking of the spark effect size (e.g., big spark effect would have a `scale` of 2.0 or even 3.0 while a small spark effect might have a `scale` of 0.5).

Note, on Line 12, the `direction` and `direction_spread` are set to 0 not since the spark particles will not move (they will!), but so that when created, the engine will give the particles a random direction and spread.

Similar settings are done for each particle system type – for example, `SMOKE` starting on Line 22.

Once all parameters are set, the full `addParticles()` function is called in Line 30, which, in turn, creates the individual Particles.

Listing 6.11: Utility addParticles() - sparks example

```
  // SPARKS.
  if type is SPARKS then
    position_spread = 0.0
    number = 50 * scale
    number_spread = 20 * scale
    size = 1.5 * scale
    speed = 0.08 * (1 + scale/4)
    speed_spread = speed
    age = 15 * scale
    age_spread = age
    opacity = 255
    opacity_spread = 75
    direction = Vector(0,0)
    direction_spread = 0
    // Sparks are red
    r = 255
    g = 0
    b = 0
    color_spread = 0
    particle_class = PARTICLE
  end if

  // SMOKE.
  if type is SMOKE then
    ...
  end if

  ...

  // Call full addParticles() function.
  addParticles(
          number, number_spread,
          position, position_spread,
          direction, direction_spread,
          size, size_spread,
          speed, speed_spread,
          age, age_spread,
          opacity, opacity_spread,
          r, g, b, color_spread,
          particle_class)
```

6.3 Lines of Code

> *"Measuring software productivity by lines of code is like measuring progress on an airplane by how much it weighs."* – Bill Gates

The number of lines of code is a poor metric for determining the complexity of a program or for determining the productivity of a programmer. Conversely, it is a wonderful metric because it is so easy to measure and compare. Lines of code can be easily checked in Linux or a Mac by typing `wc -l *.h *.cpp`) in a terminal window. However, that command will count all whitespace lines and comment lines, too – generally, the number of lines of code does not count whitespace or lines containing only comments. The GNU program

`cloc`[†] can count and summarize the "true" number of lines of code for many programming languages, including C++. Using `cloc` for the core version of Dragonfly (version A, only the non-optional features in this book implemented) produces:*

```
-------------------------------------------------------------------------------
Language          files          blank        comment           code
-------------------------------------------------------------------------------
C++                  26            570            710           2188
C/C++ Header         27            429            507            751
-------------------------------------------------------------------------------
SUM:                 53            999           1217           2939
```

Sorted from most lines of code to least, the individual Dragonfly `.cpp` files are:

```
-------------------------------------------------------------------------------
File                            blank        comment           code
-------------------------------------------------------------------------------
ResourceManager.cpp                66             79            318
InputManager.cpp                   35             45            273
WorldManager.cpp                   73             89            214
DisplayManager.cpp                 50             57            191
Object.cpp                         64             66            188
ViewObject.cpp                     31             40            178
GameManager.cpp                    44             43            136
Clock.cpp                          10             28             78
Object.h                           40             52             67
Sprite.cpp                         15             21             64
utility.cpp                        14             19             61
DisplayManager.h                   27             36             61
LogManager.cpp                     21             23             61
Vector.cpp                         15             11             48
ViewObject.h                       24             28             47
ResourceManager.h                  22             34             46
ObjectList.cpp                     15             21             44
WorldManager.h                     28             46             44
Manager.cpp                        16             21             43
EventMouse.h                       16             14             36
EventKeyboard.h                    14             13             36
Sound.cpp                          12             16             34
Sprite.h                           20             23             33
EventView.cpp                      10             14             32
Box.cpp                            10             14             31
Frame.cpp                          11             14             31
Music.cpp                          10             15             30
EventCollision.cpp                 10             15             29
GameManager.h                      19             21             28
```

[†] `http://cloc.sourceforge.net/`

* **Did you know (#12)?** The "Helicopter Damselfly" *Megaloprepus Caerulatus* has the largest wingspan of any dragonfly/damselfly, up to 7.5 inches. – "Megaloprepus Caerulatus" *Wikipedia*, 2013.

```
EventMouse.cpp                 9            12            26
LogManager.h                  15            18            25
Music.h                       14            16            23
Sound.h                       16            18            23
ObjectList.h                  16            15            23
EventCollision.h              15            15            23
Vector.h                       9             8            22
Manager.h                     15            18            22
EventView.h                   15            14            22
Box.h                         14            14            21
Frame.h                       14            14            21
EventKeyboard.cpp              7            10            19
InputManager.h                11            13            18
Color.h                        5             8            18
Event.h                       11            11            16
EventStep.cpp                  6            10            16
EventStep.h                   11            10            16
utility.h                     11            18            16
Clock.h                        8            13            13
Event.cpp                      6            10            13
EventOut.h                     7             6            11
EventOut.cpp                   3             6             5
-------------------------------------------------------------
SUM:                         999          1217          2939
-------------------------------------------------------------
```

Table 6.1: Commercial Game Engines Lines of Code

Engine	Game	Game Release	Engine Release	Lines of Code
id Tech	Quake	1996	1999	79k
id Tech 2	Quake II	1997	2001	138k
id Tech 3	Quake III	1999	2005	329k
id Tech 4	Quake IV	2005	2011	586k
Unreal Engine 4	Unreal Tournament	2014	2015	1964k

For comparison, Table 6.1 lists id Software's family of game engines, along with the *Quake* games built using them. Quake is a first person shooter, with support for 3d graphics and a variety of other features that Dragonfly does not support. For a more recent comparison, the bottom line show Epic's latest Unreal Engine (v4), used for *Unreal Tournament* (2014), another first person shooter. Note, the intent is not to compare id Tech or Unreal Engine to Dragonfly nor to minimize the accomplishment of successfully implementing the Dragonfly game engine (in fact, the opposite is true – anyone completing Dragonfly should be proud!). Instead, Table 6.1 is meant to show the size and scope of a commercial game engine that can be completed by teams of programmers over several years.