Dragonfly

Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #2

Clock & GameManager

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 4.14. Each iteration of the game loop is called a “step” (or a “tick”, as in the tick of a 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 4.14: The game loop

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

Note that the loop in Listing 4.14 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 4.15. In this version of the game loop, one step of the loop is expected to take a fixed amount of time – a `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 4.15: 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 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 lower than 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, providing resolutions in the nanoseconds. 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`, when compiling).

In order to measure the time the game loop takes (everything between line 1 "Get input" and line 4 "Swap" in Listing 4.15), the method in Listing 4.16 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 4.16: Method to measure elapsed time

```
Record before time
Do processing stuff // e.g., get input, update ...
```

```
  Record after time
  Compute elapsed time // after - before
```

For Linux, Listing 4.17 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, but it could easily be adjusted to seconds or milliseconds. For compilation, the system header file `<time.h>` is needed for the timing routines and `-lrt` is needed 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 4.17: Measuring elapsed time in Linux

```
#include <time.h>  // Compile with -lrt

struct timespec before_ts, after_ts;
clock_gettime(CLOCK_REALTIME, &before_ts); // Start timing.
// Do stuff...
clock_gettime(CLOCK_REALTIME, &after_ts);  // Stop timing.

// Compute elapsed time in microseconds.
long long before_msec = before_ts.tv_sec*1000000 + before_ts.tv_nsec/1000;
long long after_msec = after_ts.tv_sec*1000000 + after_ts.tv_nsec/1000;
long long 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 4.18. A call to `gettimeofday()` fills a `struct timeval` with the number of seconds and microseconds.

Listing 4.18: Measuring elapsed time in Mac OS

```
#include <sys/time.h>

struct timeval before_tv, after_tv;

gettimeofday(&before_tv, NULL); // Start timing.
// Do stuff...
gettimeofday(&after_tv, NULL); // Stop timing.

// Compute elapsed time in microseconds.
long int before_msec = before_tv.tv_sec*1000000 + before_tv.tv_usec;
long int after_msec = after_tv.tv_sec*1000000 + after_tv.tv_usec;
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 4.19. A call to `GetSystemTime()` fills a `SYSTEMTIME` structure with the number of minutes, seconds, and milliseconds.

Listing 4.19: Measuring elapsed time in Windows

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

```

SYSTEMTIME before_st, after_st;
GetSystemTime(&before_st);
// Do stuff...
GetSystemTime(&after_st);

// Compute elapsed time in microseconds.
long int before_msec = (before_st.wDay * 24 * 60 * 60 * 1000000)
                     + (before_st.wHour * 60 * 60 * 1000000)
                     + (before_st.wMinute * 60 * 1000000)
                     + (before_st.wSecond * 1000000)
                     + (before_st.wMilliseconds * 1000);
long int after_msec = (after_st.wDay * 24 * 60 * 60 * 1000000)
                    + (after_st.wHour * 60 * 60 * 1000000)
                    + (after_st.wMinute * 60 * 1000000)
                    + (after_st.wSecond * 1000000)
                    + (after_st.wMilliseconds * 1000);
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 4.20 provides the header file for the Clock class.[4] The clock functions as a sort of "stopwatch", so the time is stored in the variable `previous_time`, initialized to the current time when a Clock object is instantiated. 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 4.17, 4.18, or 4.19 (as appropriate to the development platform), as a reference.

Listing 4.20: Clock.h

```
// The clock, for timing (such as in the game loop).

class Clock {

 private:
  long long m_previous_time; // Previous time delta() called (in microsec).

 public:
  // Sets previous_time to current time.
  Clock();

  // Return time elapsed since delta() was last called, -1 if error.
  // Resets previous time.
  // Units are microseconds.
  long long delta();

  // Return time elapsed since delta() was last called, -1 if error.
```

[4]Note, the conditional `#ifdef` directives described in Section 4.3.4 are not shown.

```
  // Does not reset previous time.
  // Units are microseconds.
  long long split() const;
};
```

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 4.15). 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, high-resolution sleeping can be done with `nanosleep()` which sleeps for a given number of nanoseconds.[5] 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 4.21 shows a call to `nanosleep()` for 20 milliseconds.

Listing 4.21: nanosleep() example for Linux and Mac

```
// Sleep for 20 milliseconds.
struct timespec sleep_time;
sleep_time.tv_sec = 0;
sleep_time.tv_nsec = 20000000;
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 when 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 72). This functions must be linked in via the `Winmm.lib` library.

Listing 4.22: Sleep() example for Windows

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

Listing 4.23 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 4.23: The game loop with Clock

```
Clock clock
```

[5] There are 1 billion nanoseconds in 1 second.

```
 1 while (game not over) do
 2   clock.delta()
 3
 4   Get input // e.g., keyboard/mouse
 5   Update game world state
 6   Draw current scene to back buffer
 7   Swap back buffer to current buffer
 8
 9   loop_time = clock.split()
10   sleep(TARGET_TIME - loop_time)
11 end while
```

The expectation is that `(TARGET_TIME - loop_time)` is positive, since the `sleep()` call on line 10 of Listing 4.23 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 processing 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 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. The game engine could pass along timing information to update functions and for those functions to use the information accordingly.

For example, in the Saucer Shoot tutorial (Chapter 3), the Hero decrements a counter each step to restrict the rate of fire. If the goal was to keep the rate of fire consistent with the real-world time (e.g., fire one bullet every second), then the game code could use the elapsed time as in the following listing:

```
0   fire_countdown -= ceil(elapsed_time / TARGET_TIME)
```

This would decrease the `fire_countdown` value by more than 1 each step when the elapsed time (`elapsed_time`) was greater than the target loop time (`TARGET_TIME`).

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 at most. However, in some cases, such as when trying to synchronize game state on two different machines in a multi-player game or when tying to keep game time consistent with real-world time (i.e., external clocks), more precision in the total time a game loop takes is required.

If 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 4.24 shows how to put in this adjustment.

Listing 4.24: The game loop with Clock and sleep adjustment

```
Clock clock
while (game not over) do
  clock.delta()

  Get input // e.g., keyboard/mouse
  Update game world state
  Draw current scene to back buffer
  Swap back buffer to current buffer

  loop_time = clock.split()
  intended_sleep_time = TARGET_TIME - loop_time - adjust_time
  clock.delta()
  sleep(intended_sleep_time)

  actual_sleep_time = clock.split()
  adjust_time = actual_sleep_time - intended_sleep_time
  if adjust_time < 0 then
    set adjust_time to 0
  end if

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 4.25.

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 70).

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 70).

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

Listing 4.25: GameManager.h

```
#include "Manager.h"

// Default frame time (game loop time) in milliseconds (33 ms == 30 f/s).
const int FRAME_TIME_DEFAULT = 33;

class GameManager : public Manager {

 private:
  GameManager();                      // Private since a singleton.
  GameManager (GameManager const&);   // Don't allow copy.
  void operator=(GameManager const&); // Don't allow assignment.
  bool game_over;        // True, then game loop should stop.
  int frame_time;        // Target time per game loop, in milliseconds.

 public:
  // Get the singleton instance of the GameManager.
  static GameManager &getInstance();

  // Startup all GameManager services.
  int startUp();

  // Shut down GameManager services.
  void shutDown();

  // Run game loop.
  void run();

  // Set game over status to indicated value.
  // If true (default), will stop game loop.
  void setGameOver(bool new_game_over=true);

  // Get game over status.
  bool getGameOver() const;

  // Return frame time.
  // Frame time is target time for game loop, in milliseconds.
  int getFrameTime() const;
};
```

* **Did you know (#3)?** 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.

Tip 8! Acronyms for Dragonfly managers. Comparing Listing 4.25 with the full GameManager.h header file available online will show an extra line:

```
#define GM df::GameManager::getInstance()
```

This allows programmers, both game engine programmers and game programmers, to access the GameManager singleton via `GM`. For example, a game programmer could write `GM.setGameOver()` instead of `df::GameMangager::getInstance().setGameOver()`. The full version of Dragonfly has similar code in the header file for each manager: "GM" for GameManager, "LM" for LogManager, "RM" for ResourceManager, "IM" for InputManager, "DM" for DisplayManager, and "WM" for WorldManager. While such blanket syntax replacement should be used sparingly (remember, `#define` directives are handled by the pre-processor during compilation), in this case the ability to use the two-letter acronym for the singleton managers makes coding more convenient and code more readable. (Note, there is no semi-colon at the end of the above line.)

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 4.20. 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 4.25. 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.

Tip 9! Measuring elapsed time from a shell. From a command shell, such as a Bash shell in Linux and the Power Shell in Windows, the Linux `time` utility and the `Measure-Command` utility can be used to measure the elapsed time for a running program. For example, in a Linux Bash shell the command would be `time a.out` and in Windows Power Shell the command would be `Measure-Command myprogram.exe`. So, for example, having a game loop iterate 100 times and then exit can be timed via a command shell to verify it takes 3.3 seconds.