Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #15

Dragonfly

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 9.0). Information online at: `http://dragonfly.wpi.edu/book/`

4.17.3 Gracefully Shutting Down (optional)

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

4.17.3.1 Closing the Game Window (optional)

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

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

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

This provides a close button in the SFML game window that a user can click. When the user does so, this sends a `sf::Event::Closed` event to the window that can be detected along with other keyboard and mouse input in the InputManager. Needed extensions to the InputManager `getInput()` are shown in Listing 4.225. Basically, within the `while(event)` loop, the event type `sf::Event::Closed` is looked for (in addition to the keyboard and mouse events – see Listing 4.92 on page 130). If it is found, Dragonfly is shut down by calling GameManager `setGameOver()`. The header file `GameManager.h` needs to be `#include`d.

Listing 4.225: InputManager extensions to getInput() to window close

```
// Get input from the keyboard and mouse.
// Pass event along to all Objects.
void InputManager::getInput() const

  // Check past window events.
  while event do

    // Special case - see if Window closed.
    if event.type is sf::Event::Closed then
      GameManager  setGameOver()
      return
    end if
    ...
```

4.17.3.2 Catching Ctrl-C (optional)

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

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

Most programs interpret the `ctrl-c` as an indication to terminate – Dragonfly is no exception and it will end when `ctrl-c` is pressed. However, by recognizing a signal, a process has a chance to take some actions before being terminated. In particular, for Dragonfly, this means it can call `shutDown()` for each Manager, closing the logfile and reverting and settings back to standard mode before the process exits. Not responding to `ctrl-c` in this way means that the process terminates abruptly, possibly leaving the system in settings changed and leaving unwritten data that is still in memory out of the logfile.

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

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

First, the signal handler function is defined. Since the signal handler can be invoked from anywhere, it cannot be a method of any class, but must be a global function. This could suggest it be placed in `utility.cpp`, but since it only calls GameManager `shutDown()`, it should be placed in `GameManager.cpp`. The definition is shown in Listing 4.226 where, as stated earlier, the GameManager is shutdown and the process exits via the `exit()` system call.

Listing 4.226: Function doShutDown() in GameManager.cpp

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

The second step is to tell the operating system to call the signal handler, `doShutDown()`, when it receives a `ctrl-c`. This is done by adding code to `startUp()`, shown in Listing 4.227. A `#include` of `signal.h` is needed for the system calls. In `startUp()`, a variable of type `struct sigaction` is created (here, named `action`) to hold the parameters for the system call. In this case, the main parameter to specify is the name of the handler, `doShutDown`. Note, `doShutdown()` must be defined above `startUp()` in `GameManager.cpp` or a function prototype must appear before `startUp()` – not doing this will result in a compiler error complaining about `doShutDown()` being undefined. The call `sigemptyset()` initializes the set of signals to empty, and setting `sa_flags` to 0 indicates there are no spe-

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

Listing 4.227: GameManager extensions to support handling ctrl-c (Linux and Mac)

```
...
#include <signal.h>
...
// Startup all GameManager services.
int GameManager::startUp()
  ...
  // Catch ctrl-C (SIGINT) and shutdown.
  struct sigaction action
   action.sa_handler = doShutDown   // The signal handler.
  sigemptyset(&action.sa_mask)     // Clear signal set.
  action.sa_flags = 0              // No special modification to behavior.
  sigaction(SIGINT, &action, NULL) // Enable the handler.
  ...
```

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

4.17.4 Random Numbers (optional)

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

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

$$X_n = ((5 \times X_{n-1}) + 1) \; mod \; 16 \tag{4.1}$$

Assume X_0 is 5. Then $X_1 = ((5 \times 5) + 1)$ mod 16, or $X_1 = 26$ mod 16, or $X_1 = 10$. Doing the same computation for $n = 1, 2, ...$ gives the numbers 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14... It is difficult to look at this pattern and guess what number is next – to the eye (and to the game player) it looks random. If the sequence was restarted, say, the next time

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

Tip 23! Random seeds. For most games, players expect games have different behavior each time they are run. Otherwise, for example, a bad guy might always start by "turning right" in a maze game. Developers, however, mostly do *not* want different behavior from run to run, particularly when debugging – note that "reproducing the problem consistently" is the first step in debugging![a] In such cases, providing the same random seed provides the same not-so-random behavior, making it easier to reproduce a game's behavior from run to run. In addition, multiplayer games, where separate computers may be computing "random" events, are often started with the same random seed in order for all players to see the same behavior.

[a] See Section 5.1.1 on page 247 for details.

In order to make the "random" sequence easy to use, two functions are provided – one to set the seed, and one to return the next number in the sequence. An example of a `rand()` function to generate random numbers is shown in Listing 4.228. The variable `g_next` is declared globally since it needs to be maintained from call to call. By default, `g_next` is initialized to a value in seconds since 1970 (see `time()` described in Section 4.3.2 on page 56), a starting value that will seem random to the player and vary from game run to game run. If the starting value (or actually at any time) needs to be explicitly controlled by the programmer, the function `srand()` sets `g_next` explicitly to the value `seed`.

Listing 4.228: Pseudo-random number functions

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

While aspiring programmers could research good random number generating algorithms and code their own `rand()` and `srand()` functions, standard library functions already exist that do a good job. Namely, `srand()` can be used set the seed and `rand()` can be used

to return the next random number in the sequence. The random function returns an arbitrarily large integer, but typically a game programmer wants a random number bounded to something smaller. For example, a dice roll would be in the range 1–6. In order to map, say, `rand()` to the range 1–6, the game programmer calls `(rand() % 6) + 1`.

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

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

Second, as shown in Listing 4.229, a new `startUp()` method is provided, this one allowing the game programmer to set the seed explicitly (by calling `srand(seed)`).[22] The rest of the method is exactly the same as the normal GameManager `startUp()` method.

Listing 4.229: GameManager extension to support random seeds

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

4.18 Development Checkpoint #15 – Dragonfly!

(Optional)

Finish off your Dragonfly development!

1. Implement the `doShutDown()` function based on Listing 4.226. Test by having a game object call `doShutDown()` explicitly and verify in the logfiles that the engine shuts down properly.

2. Extend the GameManager to support handling closing the game window and `ctrl-c` (Linux), following guidelines from Listing 4.225 and Listing 4.227, respectively. Be sure to error as appropriate, writing informative errors to the logfile as needed. Test with a simple "game" that does not terminate, but can be ended by closing and/or `ctrl-c`. Verify these actions are actually caught via messages in the logfile.

3. Extend the GameManager to support random seeds, based on Listing 4.229. Test with a simple "game" that generates random numbers. Verify that the numbers produced are the same each run by using a fixed, explicit seed.

After completing the above steps (and all the previous Development Checkpoints), you will have a fully functional, full featured game engine! Features include everything from Dragonfly Naiad (Section 4.11 on page 149), plus:

[22] The type `time_t` is the type returned by `time()`, but can be thought of as an integer.

- Velocity support for game objects.
- Objects with bounding boxes, enabling multi-character-sized objects.
- Collisions for bounding boxes, not just single characters.
- Objects with sprites, giving them animated frames.
- Support for audio, including sound effects and music.
- View objects for display elements, each with a value and HUD-type location.
- Camera control for the game world, enabling "views" over a subset of the world with explicit camera control and support for the camera following a specific object.
- (Optional) Objects register for interest in events (e.g., step events), getting notification for only those events.
- (Optional) Scene graph support for more efficient queries.
- (Optional) An inactive feature for game objects to temporarily not have the engine act upon them.
- (Optional) Invisible objects that are not drawn but otherwise interact with the world normally.
- (Optional) Ability to gracefully close the game through closing the window and/or catching `ctrl-c`.
- (Optional) Automatic random number seeding, with the ability of the game programmer to control initial seeds.

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

Have fun!