



Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #11

Sound & Music

This document is part of the book “Dragonfly – Program a Game Engine from Scratch”, (Version 5.0). Information online at: <http://dragonfly.wpi.edu/book/>

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

4.14 Audio

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

4.14.1 Simple and Fast Multimedia Library – Audio

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

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

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

Listing 4.149: SFML playing a sound

```

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

```

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

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



Listing 4.150: SFML playing music

```

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

```

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

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

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

```

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

```

4.14.2 Dragonfly Audio

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

4.14.2.1 The Sound Class

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



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

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

Listing 4.152: Sound.h

```

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

```

4.14.2.2 The Music Class

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



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

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

Listing 4.153: Music.h

```

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

```



4.14.2.3 Extending the ResourceManager for Audio

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

Listing 4.154: ResourceManager extensions to support audio

```

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

```

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

Listing 4.155: ResourceManager `loadSound()`

```

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

```



```

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

```

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

Listing 4.156: Resource Manager unloadound()

```

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

```

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



Listing 4.157: ResourceManager getSound()

```

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

```

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

4.14.3 Using Audio

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

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

4.14.4 Development Checkpoint #11!

Continue *Dragonfly* development to support audio. Steps:

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

