Dragonfly

Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #11

Sound & Music

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) *sound effects*, which are typically small (fitting in the computer's main memory) and, for games, are typically played in response to a game action. 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 (i.e., sound effects). 2) *music*, which is typically longer (e.g., an entire song) and, for games, is often played continuously in the background, either during game loading screens or 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 effects 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 4.170.

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 4.170 shows a code fragment to do this. Note, sounds can also be played simultaneously without any issues.

Listing 4.170: SFML playing a sound

```
#include <SFML/Audio.hpp>

sf::SoundBuffer buffer;
if (buffer.loadFromFile("sound.wav") == false)
  // Error!

sf::Sound sound(buffer);
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 4.171.

* **Did you know (#11)?** 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 4.171: SFML playing music

```
#include <SFML/Audio.hpp>

sf::Music music;
if (music.openFromFile("music.wav") == false)
  // Error!

music.play();
```

Looping for both sound and music can be done with `setLooping()`, 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). To illustrate how this constrains use, the code samples in Listing 4.172 provide examples of compile-time errors, if tried.

Listing 4.172: SFML sf::Music not copyable

```
sf::Music music;
sf::Music music_copy = music; // Error!

void makeItSo(sf::Music music_parameter) {
    ...
}
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 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 4.173. The primary attributes provide for a `sf::Sound` (`sound`) and a `sf::SoundBuffer` (`sound_buffer`). Note, the `sf::Sound` attribute is a pointer since the `sf::Sound` allocation needs to allocated buffer upon instantiating. The attribute `m_p_-sound` should be set to `NULL` in the Sound constructor. The method `loadSound()` calls `loadFromFile()`, using the indicated filename and then allocates the `sf::Sound` and sets the sound buffer. See Listing 4.170 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 4.119 on page 157). 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 `setLooping()`. Looping is off by default. To allow the game programmer to manipulate the `sf::Sound` object directly, `getSound()` returns `sound`. The Sound destructor (`~Sound()`) should `delete` the `sf::Sound` object pointed to by `m_p_sound`, if allocated.

Listing 4.173: Sound.h

```
// System includes.
#include <string>
#include <SFML/Audio.hpp>

class Sound {

 private:
  sf::Sound *m_p_sound;            // The SFML sound.
  sf::SoundBuffer m_sound_buffer; // SFML sound buffer associated with
      sound.
  std::string m_label;             // Text label to identify sound.

 public:
  Sound();
  ~Sound();

  // Load sound buffer from file.
  // Return 0 if ok, else -1.
  int loadSound(std::string filename);

  // Set label associated with sound.
  void setLabel(std::string new_label);

  // Get label associated with sound.
  std::string getLabel() const;

  // Play sound.
  // If loop is true, repeat play when done.
  void play(bool loop=false);

  // Stop sound.
  void stop();

  // Pause sound.
  void pause();

  // Return SFML sound.
  sf::Sound getSound() const;
};
```

4.14.2.2 The Music Class

Dragonfly provides a Music class for supporting music, with the header file shown in Listing 4.174. The primary attribute is `sf::Music` (`music`). The method `loadMusic()` calls openFromFile(), using the indicated filename. See Listing 4.171 for examples.

Note, as mentioned above, SFML does not allow copying of sf::Music objects (See Listing 4.172). 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 4.173) and Sprites (see Listing 4.119 on page 157). 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 setLooping(). 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 4.174: Music.h

```cpp
// System includes.
#include <string>
#include <SFML/Audio.hpp>

class Music {

 private:
  Music(Music const&);           // SFML doesn't allow music copy.
  void operator=(Music const&); // SFML doesn't allow music assignment.
  sf::Music m_music;             // The SFML music.
  std::string m_label;           // Text label to identify music.

 public:
  Music();

  // Associate music buffer with file.
  // Return 0 if ok, else -1.
  int loadMusic(std::string filename);

  // Set label associated with music.
  void setLabel(std::string new_label);

  // Get label associated with music.
  std::string getLabel() const;

  // Play music.
  // If loop is true, repeat play when done.
  void play(bool loop=true);

  // Stop music.
  void stop();

  // Pause music.
  void pause();

  // Return pointer to SFML music.
  sf::Music *getMusic();
};
```

4.14.2.3 Extending the ResourceManager for Audio

With Sound and Music in place, the ResourceManager is extended to manage sound and music resources. The needed extensions are shown in Listing 4.175. 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 4.175: ResourceManager extensions to support audio

```
const int MAX_SOUNDS = 50;
const int MAX_MUSICS = 50;

 private:
  Sound m_sound[MAX_SOUNDS];     // Array of sound buffers.
  int m_sound_count;             // Count of number of loaded sounds.
  Music m_music[MAX_MUSICS];     // Array of music buffers.
  int m_music_count;             // Count of number of loaded musics.

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

  // Remove Sound with indicated label.
  // Return 0 if ok, else -1.
  int unloadSound(std::string label);

  // Find Sound with indicated label.
  // Return pointer to it if found, else NULL.
  Sound *getSound(std::string label);

  // Associate file with Music.
  // Return 0 if ok, else -1.
  int loadMusic(std::string filename, std::string label);

  // Remove label for Music with indicated label.
  // Return 0 if ok, else -1.
  int unloadMusic(std::string label);

  // Find Music with indicated label.
  // Return pointer to it if found, else NULL.
  Music *getMusic(std::string label);
```

The `loadSound()` method to load a sound from a file is shown in Listing 4.176. 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 4.176: ResourceManager loadSound()

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

  if m_sound_count is MAX_SOUNDS then
    writeLog("Sound array full.")
    return error
  end if

  if m_sound[m_sound_count].loadSound(filename) is -1 then
    writeLog("Unable to load from file")
    return error
  end if

  // All is well.
  m_sound[m_sound_count].setLabel(label)
  increment m_sound_count
  return ok
```

The complement of `loadSound()` is `unloadSound()`, shown in Listing 4.177. The method loops through the Sounds in the ResourceManager. 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 ResourceManager and an error is returned.

Listing 4.177: ResourceManager unloadound()

```
// Remove Sound with indicated label.
// Return 0 if ok, else -1.
int ResourceManager::unloadSound(std::string label)

  for i = 0 to sound_count-1

    if label is sound[i].getLabel() then

      // Scoot over remaining sounds
      for j = i to sound_count-2
        sound[j] = sound[j+1]
      end for

      decrement sound_count

      return ok

    end if

  end for

  return error // Sound not found.
```

The final method needed by the ResourceManager for sound is `getSound()`, with pseudo code show in Listing 4.178. The method loops through all the Sounds in the ResourceManager. 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 4.178: 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 // Sound 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 "scooted 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 44 and example code to load music is shown in Listing 3.9 on page 45.

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 44) and plays a sound effect when the player fires a bullet (see Listing 3.10 on page 45).

4.14.4 Development Checkpoint #11!

Continue Dragonfly development to support audio. Steps:

1. Make the Sound and Music classes, referring to Listings 4.173 and 4.174, 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 44) 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 4.175 as needed. Write and test methods to `loadSound()`, `unloadSound()`, and `getSound()`. Refer to Listings 4.176, 4.177, and 4.178, as needed.

3. Extend the ResourceManager to support music, referring to Listing 4.175 as needed. Write and test methods to `loadMusic()`, `unloadMusic()`, and `getMusic()`. Base the music support implementation off the corresponding sound support previously implemented.