Weaver Interface for WAVE Format

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

Abstract: *This article presents the implementation of support for the WAVE audio format in the Weaver Game Engine. In accordance with the engine's modular design, proper support for the WAVE format requires the definition of an extraction function that reads the audio file, obtains auxiliary functions for memory allocation and deallocation, and populates a sound structure with an OpenAL audio buffer.*

1. Introduction

The WAVE format is one of the many types of the so-called RIFF format. Every RIFF file represents a generic packed data sequence. The WAVE format, specifically, is designed to encapsulate audio data.

Our goal in this article is to define a loading function with the following signature:

Section: Function Declaration (wave.h):

```
void _extract_wave(void *(*permanent_alloc)(size_t),
                   void (*permanent_free)(void *),
                   void *(*temporary_alloc)(size_t),
                   void (*temporary_free)(void *),
                   void (*before_loading_interface)(void),
                   void (*after_loading_interface)(void),
                   char *source_filename, void *interface);
```

A function with this signature is used in the Weaver Game Engine to interpret audio files. Its purpose is to populate the structure pointed to by the last parameter. The definition of this structure can be accessed by including the appropriate header from the Weaver Game Engine:

Section: Local Headers (wave.c):

```
#include "interface.h"
```

Specifically, the structure is defined as follows:

```
struct sound {
  (...)
  bool _loaded_sound; // Has the sound been loaded?
  ALuint buffer;      // References buffer containing audio data
};
```

Our task is to fill the OpenAL buffer field within this structure. Once we are done, we must set the boolean variable above to true. This informs the game engine that the sound is ready for use.

The buffer is populated by the loading function when it reads the `source_filename` file, which must be in the WAVE format.

The loading function is also responsible for calling the `before_loading_interface` function before performing any other actions. The internal behavior of this function is not relevant to us—it is used by the Weaver Game Engine to perform operations of its own. Similarly, after the file is loaded—or even if loading fails due to an error—we must call the `after_loading_interface` function. Again, its effects are not our concern. If either of these function pointers is NULL, they should simply be ignored and not executed.

The remaining parameters received by the loading function are for data allocation and deallocation. We must distinguish between temporary and permanent allocations. Permanent allocations are meant for data that must remain valid after the extraction function returns. Temporary allocations are only needed during file reading. Each allocation function has a corresponding deallocation function. However, the deallocation function may be NULL—in such cases, we will not invoke it, assuming that some form of automatic memory management is in place.

Because the allocation and deallocation functions may be needed not only by the extraction function but also by various auxiliary routines, we will store them in static global variables. If the values were not initialized, we will assume that we should use the allocation and deallocation functions from standard library.

Section: Local Variables (wave.c):

```
static void *(*permanent_alloc)(size_t) = malloc;
static void *(*temporary_alloc)(size_t) = malloc;
static void (*permanent_free)(void *) = free;
static void (*temporary_free)(void *) = free;
```

1.1. Literate Programming

Our API will be written using the literate programming technique, proposed by Knuth on [Knuth, 1984]. It consist in writting a computer program explaining didactically in a text what is being done while presenting the code. The program is compiled extracting the computer code directly from the didactical text. The code shall be presented in a way and order such that it is best for explaining for a human. Not how it would be easier to compile.

Using this technique, this document is not a simple documentation for our code. It is the code by itself. The part that will be extracted to be compiled can be identified by a gray background. We begin each piece of code by a title that names it. For example, immediately before this subsection we presented a series of function declarations. And how one could deduct by the title, most of them will be positioned in the file `interface.h`.

We show below the structure of the file `wave.h`:

File: `src/wave.h`:

```
#ifndef __WEAVER_INTERFACE_WAVE
#define __WEAVER_INTERFACE_WAVE
#ifdef __cplusplus
extern "C" {
#endif
#include <stdbool.h> // Define  'bool' type
#include <stdlib.h>
              <Section to be inserted: Function Declaration (wave.h)>
#ifdef __cplusplus
}
#endif
#endif
```

The code above shows the default boilerplate for defining a header in our C API. The first two lines and the last one are macros that ensure the header is not included more than once in a single compilation unit. Lines 3, 4, 5, and the three lines before the last one make the header compatible with C++ code. These lines tell the compiler that we are using C code and, therefore, it can apply optimizations assuming that no C++-specific features—such as operator overloading—will be used.

Next, we include a header that allows us to use boolean variables. You may also notice some parts highlighted in red. One of them is labeled "Function Declaration (wave.h)", the same title used for most of the code declared earlier. This means that all previously defined code blocks with that title will be inserted at this point in the file. The other red-highlighted parts represent code that we will define in the following sections.

If you want to understand how the `wave.c` file relates to this header, its structure is as follows:

File: `src/wave.c`:

```
#include "wave.h"
```

```
#include <AL/al.h>
#include <AL/alc.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
                    <Section to be inserted: Local Headers (wave.c)>
                   <Section to be inserted: Local Variables (wave.c)>
              <Section to be inserted: Local Functions Declaration (wave.c)>
               <Section to be inserted: Auxiliary Local Functions (wave.c)>
               <Section to be inserted: API Functions Definition (wave.c)>
```

All the code presented in this document will be placed in one of these two files. No other files will be created.

2. The RIFF-WAVE Format

We will create an auxiliary function named `interpret_wave`, which will perform most of the work required to interpret the WAVE format and initialize an OpenAL buffer from it. This function will be invoked within the extraction function as follows:

Section: API Functions Definition (wave.c):

```
void _extract_wave(void *(*perm_alloc)(size_t),
                   void (*perm_free)(void *),
                   void *(*temp_alloc)(size_t),
                   void (*temp_free)(void *),
                   void (*before_loading_interface)(void),
                   void (*after_loading_interface)(void),
                   char *source_filename, void *interface){
  struct sound *snd;
  if(before_loading_interface != NULL)
    before_loading_interface();
  snd = (struct sound *) interface;
  permanent_alloc = perm_alloc;
  permanent_free = perm_free;
  temporary_alloc = temp_alloc;
  temporary_free = temp_free;
  alGenBuffers(1, &(snd -> buffer));
  if(interpret_wave(source_filename, snd -> buffer))
    snd -> _loaded_sound = true;
  else{
    alDeleteBuffers(1, &(snd -> buffer));
    snd -> buffer = 0;
  }
  if(after_loading_interface != NULL)
    after_loading_interface();
}
```

To understand what the `interpret_wave` function should do, we need to briefly explain how the WAVE—or more precisely, RIFF-WAVE—format works. Every WAVE file is also a RIFF file. A RIFF file is composed of one or more "chunks". Fortunately, in the specific case of the WAVE format, things are simpler: the entire file is made up of a single chunk.

Each chunk consists of three parts: 4 bytes containing ASCII characters that identify the chunk type, 4 bytes indicating the size of the chunk data, and then a sequence of bytes of the specified size representing the actual data. The data section may also include padding bytes if the actual content is slightly shorter than the declared size. All numeric values must be read using little-endian byte order.

In WAVE files, the first and only chunk in the file must begin with the ASCII identifier “RIFF”. With this information, we can now proceed to read the first 8 bytes of the file:

Section: Auxiliary Local Functions (wave.c):

```
bool interpret_wave(char *filename, ALuint buffer){
  FILE *fp;
  fp = fopen(filename, "r");
  if(fp == NULL){
    perror(filename);
    return false;
  }
  {
    unsigned char buf[4];
    size_t ret;
    // Reading 'RIFF':
    ret = fread(buf, 1, 4, fp);
    if(ret != 4){
      perror(filename);
      return false;
    }
    if(buf[0] != 'R' || buf[1] != 'I' || buf[2] != 'F' || buf[3] != 'F'){
      fprintf(stderr, "ERROR: %s not a RIFF file.\n", filename);
      return false;
    }
    // Reading data size in main chunk:
    ret = fread(buf, 1, 4, fp);
    if(ret != 4){
      perror(filename);
      return false;
    }
                    <Section to be inserted: Interpret WAVE File>
  }
  fclose(fp);
  return true;
}
```

There are two special chunk types in the RIFF format: RIFF and LIST. We can safely ignore the latter. However, the former is crucial, since—as we've seen—every WAVE file consists of a large chunk of type RIFF.

What makes these chunk types special is that their data region contains multiple subchunks, each following the same pattern: a 4-byte ASCII identifier for the subchunk type, a 4-byte size field, and a data section.

However, before the list of subchunks begins, the data region starts with 4 ASCII bytes that serve as a label. In the case of WAVE files, this label is always ”WAVE”. Let's interpret this part of the file:

Section: Interpret WAVE File:

```
// Reading 'WAVE':
ret = fread(buf, 1, 4, fp);
if(ret != 4){
  perror(filename);
  return false;
}
if(buf[0] != 'W' || buf[1] != 'A' || buf[2] != 'V' || buf[3] != 'E'){
  fprintf(stderr, "ERROR: %s not a WAVE file.\n", filename);
  return false;
}
```

After this part, we read the sub-chunks:

Section: Interpret WAVE File (continuation):

```
struct audio_data audio;
memset(&audio, 0, sizeof(struct audio_data));
while(read_chunk(fp, filename, &audio));
```

The `read_chunk` function will be responsible for reading a single chunk and, based on its contents, updating the `audio_data` structure, which will hold all the necessary information to generate our audio. As long as it successfully reads a complete chunk, it should return true. If it encounters an error or reaches the end of the file, it must return false.

The attributes of the `audio_data` structure will be defined later. For now, we declare it as follows:

Section: Local Variables (wave.c) (continuation):

```
struct audio_data{
              <Section to be inserted: struct audio“‘data: Struct Variables>
};
```

We now move on to the function that reads the subchunks. The WAVE specification defines six different subchunk types. However, in practice, only two mandatory types are commonly used: the “fmt” subchunk, which contains information about the audio format, and the “data” subchunk, which holds the actual audio data.

The specification states that any unrecognized subchunk should be safely ignored.

Our function for interpreting subchunks is defined as follows:

Section: Local Functions Declaration (wave.c):

```
bool read_chunk(FILE *fp, char *filename, struct audio_data *);
```

Section: Auxiliary Local Functions (wave.c) (continuation):

```
bool read_chunk(FILE *fp, char *filename, struct audio_data *audio){
  size_t ret;
  // Lendo o tipo do 'chunk' nos 4 primeiros caracteres
  unsigned char type[4];
  ret = fread(type, 1, 4, fp);
  if(ret != 4){
    if(feof(fp))
      return false; // End of file, not an error
    else{
      perror(filename); // Error
      return false;
    }
  }
  if(type[0] == 'f' && type[1] == 'm' && type[2] == 't' && type[3] == ' ')
    return read_fmt_chunk(fp, filename, audio);
  else if(type[0] == 'd' && type[1] == 'a' && type[2] == 't' && type[3] == 'a')
    return read_data_chunk(fp, filename, audio);
  else
    return read_unknown_chunk(fp, filename, audio);
}
```

Note that there are three different types of functions, each responsible for interpreting a specific type of subchunk. We will start with the simplest one—the function that handles an unknown subchunk type. In this case, we will simply skip its contents.

To do that, we just need to read the next 4 bytes, which indicate the size of the subchunks data, and then ignore all subsequent bytes up to the end of the subchunk:

Section: Local Functions Declaration (wave.c) (continuation):

```
bool read_unknown_chunk(FILE *fp, char *filename, struct audio_data *);
```

Section: Auxiliary Local Functions (wave.c) (continuation):

```
bool read_unknown_chunk(FILE *fp, char *filename, struct audio_data *audio){
  unsigned long size;
  unsigned char buffer[4];
  size_t ret;
  ret = fread(buffer, 1, 4, fp);
  if(ret != 4){
    perror(filename);
    return false;
  }
  size = buffer[0] + (256 * buffer[1]) + (256 * 256 * buffer[2]) +
         (256 * 256 * 256 * buffer[3]);
  if(fseek(fp, size, SEEK_CUR) != 0){
    perror(filename);
    return false;
  }
  return true;
}
```

The next subchunk we need to read is the format subchunk, identified by the string "fmt " (with a trailing space to make up 4 bytes). According to the WAVE specification, this subchunk is mandatory and must always appear before the subchunk containing the actual audio data.

The WAVE format is not a single, fixed format–there are multiple variants, each with its own characteristics. The format we will support here is the most common one: Microsoft Pulse Code Modulation, or Microsoft PCM, which represents the vast majority of WAVE files found on the Internet.

For this format, the first 2 bytes after the size in the format subchunk must contain the number 1. This serves as the identifier for PCM. If we encounter a different value, it indicates an unsupported format, and we should return an error.

The next 2-byte value specifies the number of audio channels. Common values are 1 for mono and 2 for stereo.

Following that, we have 4 bytes representing the sampling rate (samples per second per channel). For example, CD-quality audio typically uses 44100, while telephone-quality audio might use 8000.

The next 4 bytes represent the average number of bytes per second that must be transferred to play the sound. This value is typically used to help size streaming audio buffers. Since our implementation loads the entire audio into memory, we can ignore it. Similarly, the following 2 bytes relate to block alignment in memory and are also mainly relevant for streaming implementations.

Finally, we conclude with the next 2 bytes, which indicate the number of bits per audio sample.

Therefore, the only values we need to extract and store in our audio structure are the number of channels, the sampling rate and number of bits per sample. These values should be saved into our `audio_data` structure as follows:

Section: struct audio_data: Struct Variables:

```
uint16_t channel, bits_per_sample;
uint32_t sample_rate;
```

Below is the declaration and implementation of the function that reads the format subchunk:

Section: Local Functions Declaration (wave.c) (continuation):

```
bool read_fmt_chunk(FILE *fp, char *filename, struct audio_data *);
```

Section: Auxiliary Local Functions (wave.c) (continuation):

```
bool read_fmt_chunk(FILE *fp, char *filename, struct audio_data *audio){
  unsigned char buffer[4];
  size_t ret;
  // Ignoring the size (WAVE in Microsoft PCM format always has same size)
  if(fseek(fp, 4, SEEK_CUR) != 0){
    perror(filename);
    return false;
  }
  // Reading format: must be 1 for Microsoft PCM
  ret = fread(buffer, 1, 2, fp);
  if(ret != 2){
    perror(filename);
    return false;
  }
  if(buffer[0] != 1 || buffer[1] != 0){
    fprintf(stderr, "ERROR: WAV file %s is not in Microsoft PCM format.\n",
            filename);
    return false;
  }
  // Reading number of channels
  ret = fread(buffer, 1, 2, fp);
  if(ret != 2){
    perror(filename);
    return false;
  }
  audio -> channel = buffer[0] + 256 * buffer[1];
  // Reading sampling rate
  ret = fread(buffer, 1, 4, fp);
  if(ret != 4){
    perror(filename);
    return false;
  }
  audio -> sample_rate = buffer[0] +
                         buffer[1] * 256 +
ΨΨΨ buffer[2] * 256 * 256 +
ΨΨΨ buffer[3] * 256 * 256 * 256;
  // Ignoring next 6 bytes:
  if(fseek(fp, 6, SEEK_CUR) != 0){
    perror(filename);
    return false;
  }
  // Reading number of bits per sample:
  ret = fread(buffer, 1, 2, fp);
  if(ret != 2){
    perror(filename);
    return false;
  }
  audio -> bits_per_sample = buffer[0] + 256 * buffer[1];
  return true;
}
```

Finally, we reach the data subchunk. This is where we find the raw waveform samples that represent the actual sound. Later, we will need to copy these samples into an OpenAL buffer. But before that, we must store them in our audio structure:

Section: struct audio_data: Struct Variables (continuation):

```
bool initialized;
size_t buffer_size;
unsigned char *buffer;
```

We added a boolean variable to the structure, which should be set to true once the buffer is filled. This prevents multiple initializations and data loss in the event that we encounter a malformed file containing multiple data subchunks. In such cases, only the first subchunk will be processed, and the others can be safely ignored. This situation should never occur with well-formed files, but we must account for the possibility of corrupted or non-compliant input.

The function that processes the data subchunk is:

Section: Local Functions Declaration (wave.c) (continuation):

```
bool read_data_chunk(FILE *fp, char *filename, struct audio_data *);
```

Section: Auxiliary Local Functions (wave.c) (continuation):

```
bool read_data_chunk(FILE *fp, char *filename, struct audio_data *audio){
  unsigned char buffer[4];
  size_t size, ret;
  // Reading the size
  ret = fread(buffer, 1, 4, fp);
  if(ret != 4){
    perror(filename);
    return false;
  }
  size = buffer[0] +
         buffer[1] * 256 +
Ψ buffer[2] * 256 * 256 +
         buffer[3] * 256 * 256 * 256;
  if(audio -> initialized){
    // If the audio data was already extracted in this file, ignore this chunk
    if(fseek(fp, size, SEEK_CUR) != 0){
      perror(filename);
      return false;
    }
    return true;
  }
  // Otherwise, allocate memory in the buffer and copy the data from this
  // chunk:
  audio -> buffer = (unsigned char *) temporary_alloc(size);
  if(audio -> buffer == NULL){
    fprintf(stderr, "ERROR: No memory to copy audio data from %s.\n",
            filename);
    return false;
  }
  audio -> buffer_size = size;
  ret = fread(audio -> buffer, 1, size, fp);
  if(ret != size){
    perror(filename);
    return false;
  }
  return true;
}
```

With this, we have completed all the code needed to interpret WAVE files into an audio data structure. The final step is to transfer the contents of this structure into our OpenAL buffer:

Section: Interpret WAVE File (continuation):

```
{
  ALenum error, format = 0;
  if(audio.channel == 1){
    if(audio.bits_per_sample == 8)
      format = AL_FORMAT_MONO8;
    else if(audio.bits_per_sample == 16)
      format = AL_FORMAT_MONO16;
  }
  else if(audio.channel == 2){
    if(audio.bits_per_sample == 8)
      format = AL_FORMAT_STEREO8;
    else if(audio.bits_per_sample == 16)
      format = AL_FORMAT_STEREO16;
  }
  else{
    fprintf(stderr, "ERROR: %s has %d channels, but we support at most 2.\n",
            filename, audio.channel);
    return false;
  }
  if(audio.bits_per_sample != 8 && audio.bits_per_sample != 16){
    fprintf(stderr, "ERROR: %s uses %d bits per sample, but we support "
            "only 8 or 16.\n", filename, audio.bits_per_sample);
    return false;
  }
  alBufferData(buffer, format, audio.buffer, audio.buffer_size,
               audio.sample_rate);
  error = alGetError();
  if(error == AL_OUT_OF_MEMORY){
    fprintf(stderr, "ERROR: OpenAL: No memory!\n");
    return false;
  }
  else if(error != AL_NO_ERROR){
    fprintf(stderr, "ERROR: %s: OpenAL: Unexpected error.\n");
    return false;
  }
  temporary_free(audio.buffer);
  return true;
}
```

And that concludes all the operations required to interpret a WAVE file.

References

Knuth, D. E. (1984) “Literate Programming”, The Computer Journal, Volume 27, Issue 2, Pages 97–111.