Interface Weaver para Formato WAVE

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.*

Resumo: *Este artigo apresenta a implementação do suporte ao formato de áudio WAVE no motor de jogos Weaver. Em conformidade com o design modular do motor, o suporte adequado ao formato WAVE requer a definição de uma função de extração que leia o arquivo de áudio, obtenha funções auxiliares para alocação e desalocação de memória e preencha uma estrutura de som com um buffer de áudio do OpenAL.*

1. Introducão

O formato WAVE é um dos muitos tipos existentes do chamado formato RIFF. Todo arquivo no formato RIFF representa uma sequência de dados genérica empacotada. O WAVE em específico, é feito para empacotar arquivos de áudio.

Nosso objetivo neste artigo é definir a função de carregamento com a assinatura abaixo:

Seção: Declaração de Função (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);
```

Uma função com essa assinatura é usada no motor de jogos Weaver para interpretar arquivos de áudio. O objetivo da função é preencher a estrutura indicada pelo último ponteiro recebido como parâmetro. A descrição desta estrutura pode ser importada incluindo o cabeçalho específico do motor de jogos Weaver:

Seção: Cabeçalhos Locais (wave.c):

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

A saber, a estrutura tem a seguinte forma:

```
struct sound {
  (...)
  bool _loaded_sound; // O som já foi carregado?
  ALuint buffer; // Referencia buffer com conteúdo sonoro
};
```

Nossa missão será preencher o buffer OpenAL presente nesta estrutura. E quando terminarmos, devemos mudar para "verdadeiro" o valor da variável booleana acima. Isso indica para o motor de jogos que o som já pode ser usado.

O buffer é preenchido pela função de carregamento quando ela lê o arquivo `source_filename`, a qual deverá ser um arquivo no formato WAVE.

A função de carregamento também tem a obrigação de executar a função `before_loading_interface` antes de começar qualquer outra atividade. O que tal função faz, não nos diz respeito. O motor de jogos Weaver realiza ações relevantes para ele quando a função é executada. Depois que o arquivo é carregado, ou mesmo se ele não puder ser carregado devido a um erro, também temos a obrigação de executar a função `after_loading_interface`, cujo efeito não nos diz respeito. Caso qualquer uma destas funções seja NULL, então ela não precisa ser executada.

Os outros argumentos recebidos pela função de extração são para alocação e desalocação de dados. Devemos distinguir uma alocação entre temporária e permanente. Uma alocação permanente é para dados que precisam estar ativos mesmo depois que a função de extração executar. Já a alocação temporária é para dados necessários somente enquanto lemos o arquivo. Cada função de alocação tem uma de desaloicação correspondente. Mas a função de desalocação pode ser NULL. Em tal caso, nós não a usaremos e apenas assumiremos existir algum tipo de coleta automática de lixo.

Como as funções de alocação e desalocação poderão ser necessárias em muitas outras funções auxiliares, não apenas a de extração, vamos armazenar elas em variáveis estáticas globais. Caso os valores não tenham sido inicializados, usaremos as funções padrão de alocação e desalocação da biblioteca padrão.

Seção: Variáveis Locais (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. Programação Literária

Nossa API será escrita usando a técnica de Programação Literária, proposta por Knuth em [KNUTH, 1984]. Ela consiste em escrever um programa de computador explicando didaticamente em texto o que se está fazendo à medida que apresenta o código. Depois, o programa é compilado através de programas que extraem o código diretamente do texto didático. O código deve assim ser apresentado da forma que for mais adequada para a explicação no texto, não como for mais adequado para o computador.

Seguindo esta técnica, este documento não é uma simples documentação do nosso código. Ele é por si só o código. A parte que será extraída e compilada posteriormente pode ser identificada como sendo o código presente em fundo cinza. Geralmente começamos cada trecho de código com um título que a nomeia. Por exemplo, imediatamente antes desta subseção nós apresentamos uma série de declarações. E como pode-se deduzir pelo título delas, a maioria será posteriormente posicionada dentro de um arquivo chamado `interface.h`.

Podemos apresentar aqui a estrutura do arquivo `wave.h`:

Arquivo: `src/wave.h`:

```
#ifndef __WEAVER_INTERFACE_WAVE
#define __WEAVER_INTERFACE_WAVE
#ifdef __cplusplus
extern "C" {
#endif
#include <stdbool.h> // Define tipo 'bool'
#include <stdlib.h>
            <Seção a ser Inserida: Declaração de Função (wave.h)>
#ifdef __cplusplus
}
#endif
#endif
```

O código acima mostra a burocracia padrão para definir um cabeçalho para nossa API em C. As duas primeiras linhas mais a última são macros que garantem que esse cabeçalho não será inserido mais de uma vez em uma mesma unidade de compilação. As linhas 3, 4, 5, assim como a penúltima, antepenúltima e a antes da antepenúltima tornam o cabeçalho adequado a ser

inserido em código C++. Essas linhas apenas avisam que o que definirmos ali deve ser encarado como código C. Por isso o compilador está livre para fazer otimizações sabendo que não usaremos recursos da linguagem C++, como sobrecarga de operadores. Logo em seguida, inserimos um cabeçalho que nos permite declarar o tipo booleano. E tem também uma parte em vermelha. Note que uma delas é "Declaração de Função (wave.h)", o mesmo nome apresentado no trecho de código mostrado quando descrevemos nossa API antes dessa subseção. Isso significa que aquele código visto antes será depois inserido ali. As outras partes em vermelho representam código que ainda iremos definir nas seções seguintes.

Caso queira observar o que irá no arquivo `wave.c` associado a este cabeçaho, o código será este:

Arquivo: `src/wave.c`**:**

```
#include "wave.h"
#include <AL/al.h>
#include <AL/alc.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
        <Seção a ser Inserida: Cabeçalhos Locais (wave.c)>
        <Seção a ser Inserida: Variáveis Locais (wave.c)>
    <Seção a ser Inserida: Declaração de Funções Locais (wave.c)>
      <Seção a ser Inserida: Funções Auxiliares Locais (wave.c)>
    <Seção a ser Inserida: Definição de Funções da API (wave.c)>
```

Todo o código que definiremos e explicaremos a seguir será posicionado nestes dois arquivos. Além deles, nenhum outro arquivo será criado.

2. O Formato RIFF-WAVE

Iremos criar uma função auxiliar chamada `interpret_wave` que é quem irá fazer o grosso do trabalho para interpretar o formato WAVE e inicializar à partir dele um buffer OpenAL. A função será chamada dentro da função de extração desta forma:

Seção: Definição de Funções da API (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();
}
```

Para entender o que a função `interpret_wave` deve fazer, vamos descrever um pouco como funciona o formato WAVE, ou melhor, RIFF-WAVE. Todo arquivo WAVE é também um arquivo no formato RIFF. Um arquivo RIFF é um conjunto de um ou mais "pedaços", ou "chunks". Felizmente no caso específico do formato WAVE, as coisas são mais simples. Temos um único pedaço que compõe o arquivo.

Cada pedaço é composto por três partes: 4 bytes que devem ser caracteres ASCII identificando o tipo do pedaço, 4 bytes identificando o tamanho dos dados do pedaço, e mais uma sequência de bytes com o tamanho indicado representando os dados. Os dados também podem ter bytes finais de preenchimento caso eles sejam um pouco menores que o tamanho indicado. Números devem ser lidos usando a notação "little endian". Para os arquivos WAVE, o primeiro e único pedaço do arquivo tem como identificador os caracteres "RIFF".

Com estas informações, já podemos começar a ler os 8 primeiros bytes do arquivo:

Seção: Funções Auxiliares Locais (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;
    // Lendo '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;
    }
    // Lendo tamanho do pedaço principal:
    ret = fread(buf, 1, 4, fp);
    if(ret != 4){
      perror(filename);
      return false;
    }
```

<Seção a ser Inserida: Interpreta Arquivo WAVE>

```
  }
  fclose(fp);
  return true;
}
```

Existem dois tipos de pedaços que são especiais no formato RIFF: o de tipo RIFF e o LIST. Não precisamos nos preocupar com o segundo. Mas com o primeiro sim, pois como vimos, todo o arquivo no formato WAVE é composto por um grande pedaço do tipo RIFF. O que estes tipos tem em comum é que a região de dados deles é ocupada por muitos sub-pedaços, seguindo a mesma regra de que cada um deles começa com um tipo formado por 4 caracteres ASCII, um tamanho e uma região de dados. Entretanto, antes de começar os subpedaços, a região contém 4 bytes contendo caracteres ASCII com um rótulo. No nosso caso, o rótulo é sempre "WAVE". Vamos

interpretar isso:

Seção: Interpreta Arquivo WAVE:

```
// Lendo '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;
}
```

E depois desta parte, vem a leitura dos sub-pedaços:

Seção: Interpreta Arquivo WAVE (continuação):

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

A função `read_chunk` será responsável por ler um único pedaço e, baseando-se no conteúdo, modificar a estrutura `audio_data` que terá tudo o que é necessário para gerar o nosso áudio. Enquanto ela conseguir ler um pedaço inteiro, ela deve retornar verdadeiro. Ao encontrar erro ou o fim do arquivo, ela retorna falso.

Os atributos da estrutura `audio_data` serão definidos posteriormente. Mas ela será declarada aqui:

Seção: Variáveis Locais (wave.c) (continuação):

```
struct audio_data{
        <Seção a ser Inserida: struct audio"'data: Variáveis da Estrutura>
};
```

Vamos agora para a função que lê os sub-pedaços. O padrão especifica seis diferentes tipos de subpedaços para o formato WAVE. Mas na prática somente os dois tipos obrigatórios são amplamente empregados: o de tipo "fmt " com informações do formato que o áudio está armazenado e o "data" com os dados do áudio em si. A especificação prevê que qualquer sub-pedaço que não seja compreendido, deve ser ignorado.

Nossa função que interpreta os sub-pedaços é definida assim:

Seção: Declaração de Funções Locais (wave.c):

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

Seção: Funções Auxiliares Locais (wave.c) (continuação):

```
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; // Fim do arquivo, sem erro
    else{
      perror(filename); // Erro
      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 que há três tipos diferentes de funções, cada uma para interpretar um dos tipos para os sub-pedaços. Vamos começar com a mais simples de todas, a que lê um sub-pedaço cujo formato nos é desconhecido. Iremos simplesmente ignorar todo o seu conteúdo. Precisamos apenas ler os próximos 4 bytes com o tamanho dos dados do sub-pedaço e ignorar todos os valores seguintes até o final do sub-pedaço:

Seção: Declaração de Funções Locais (wave.c):

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

Seção: Funções Auxiliares Locais (wave.c) (continuação):

```
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]);
  printf("size: %lu\n", size);
  if(fseek(fp, size, SEEK_CUR) != 0){
    perror(filename);
    return false;
  }
  return true;
}
```

O próximo subpedaço que devemos ler é o de formato. Identificado pela string “fmt ” (com um espaço final para que fique com 4 bytes). A especificação do formato WAVE diz que este subpedaço é obrigatório e deve sempre vir antes do pedaço que contém os dados de áudio em si.

O formato WAVE na verdade não é um só. Podem haver muitos tipos deles, cada um com sua própria característica. O formato que iremos suportar aqui é o mais comum, que representa a imensa maioria dos exemplares que se encontra distribuídos pela Internet. O “Microsoft Pulse Code Modulation”, ou formato PCM da Microsoft.

Quando estamos diante deste formato, a primeira coisa que encontramos no subpedaço de formato, após seu tamanho, é o número 1 ocupando 2 bytes. Esta é a assinatura deste tipo de formato. Se encontrarmos algo diferente, estamos diante de outro formato não suportado e retornamos um erro.

O próximo valor de 2 bytes lido é a quantidade de canais de áudio. Valores mais comuns são 1 (som monoaural) e 2 (som estéreo).

Em seguida, ocupando 4 bytes, vem a taxa de amostragem por segundo em cada canal. O som com qualidade comparável a um CD de áudio teria o número 44100. A qualidade de um telefone é de 8000.

Próximo valor de 4 bytes é a quantidade de bytes por segundo que devem ser transferidos para tocar o som. Este valor existe para auxiliar a calcular o tamanho do buffer de tocadores de áudio. No nosso caso, queremos armazenar o áudio inteiro na memória, então não precisaremos de tal valor. Da mesma forma, os 2 bytes seguintes tem relação ao alinhamento de blocos na memória e

é um valor mais relevante para ajudar a criar buffers de reprodução quando o som não é carregado inteiramente para a memória.

Finalmente, terminamos com 2 próximos bytes informando a quantidade de bits que cada amostra de som tem.

Sendo assim, os valores que são relevantes para nós são o número de canais, a taxa de amostragem e quantos bits cada amostra ocupa. Tais valores devem ser armazenados na nossa estrutura de áudio:

Seção: struct audio_data: Variáveis da Estrutura:

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

E abaixo segue a declaração e implementação da função que lê o subpedaço de formato:

Seção: Declaração de Funções Locais (wave.c) (continuação):

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

Seção: Funções Auxiliares Locais (wave.c) (continuação):

```
bool read_fmt_chunk(FILE *fp, char *filename, struct audio_data *audio){
  unsigned char buffer[4];
  size_t ret;
  // Ignorando o tamanho (WAVE em Microsoft PCM sempre tem o mesmo tamanho)
  if(fseek(fp, 4, SEEK_CUR) != 0){
    perror(filename);
    return false;
  }
  // Lendo o tipo de áudio: deve ser 1 para 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;
  }
  // Lendo o número de canais de áudio
  ret = fread(buffer, 1, 2, fp);
  if(ret != 2){
    perror(filename);
    return false;
  }
  audio -> channel = buffer[0] + 256 * buffer[1];
  // Lendo a taxa de amostragem
  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;
  // Ignorando os próximos 6 bytes:
```

```
  if(fseek(fp, 6, SEEK_CUR) != 0){
    perror(filename);
    return false;
  }
  // Lendo a quantidade de bits por amostra:
  ret = fread(buffer, 1, 2, fp);
  if(ret != 2){
    perror(filename);
    return false;
  }
  audio -> bits_per_sample = buffer[0] + 256 * buffer[1];
  return true;
}
```

Por fim, vamos ao subpedaço de de dados. Nele é onde encontramos em estado bruto as amostragens da onda sonora que corresponde ao nosso som. O que temos que fazer posteriormente é copiar as amostragens para um buffer no OpenAL. Mas antes devemos armazenar elas na nossa estrutura de áudio:

Seção: struct audio_data: Variáveis da Estrutura (continuação):

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

Criamos na estrutura uma variável booleana que deve ser ajustada para verdadeira quando o buffer for preenchido. Isso irá evitar que façamos várias inicializações e comecemos a perder dados caso encontremos um arquivo incorreto que possui vários subpedaços de dados. Em tais casos, somente o primeiro subpedaço será lido e os demais poderão ser ignorados. Isso nunca ocorrerá com arquivos bem-formados, mas devemos nos precaver contra arquivos com problemas.

A função que processa o sub-pedaço de dados é:

Seção: Declaração de Funções Locais (wave.c) (continuação):

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

Seção: Funções Auxiliares Locais (wave.c) (continuação):

```
bool read_data_chunk(FILE *fp, char *filename, struct audio_data *audio){
  unsigned char buffer[4];
  size_t size, ret;
  // Lendo o tamanho
  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){
    // Se os dados já forma lidos, ignorar este sub-pedaço
    if(fseek(fp, size, SEEK_CUR) != 0){
      perror(filename);
      return false;
    }
    return true;
  }
```

```
  // Se os dados não foram lidos, alocar o buffer e copiar o conteúdo do
  // sub-caminho:
  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;
}
```

Com isso, já escrevemos todo o código que interpreta arquivos WAVE para uma estrutura de dados de áudio. Resta agora passarmos o conteúdo da estrutura de dados para o nosso buffer OpenAL:

Seção: Interpreta Arquivo WAVE (continuação):

```
{
  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;
}
```

E isso conclui todas as operações necessárias para interpretar um arquivo WAVE.

Referências

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