O Programa Weaver

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

Abstract: *This article describes using literary programming the Weaver API. Weaver is a game engine and this API are how programmers interact with the engine in their game projects. Besides the API, this article also covers how the configuration file is interpreted and how game loops should be managed in a game project. The API is portable code which should work under OpenBSD, Linux, Windows and Web Assembly environments.*

Resumo: *Este artigo utiliza programação literária para descrever a API Weaver. Weaver é um motor de jogos e esta API é como os programadores interagem com ela em seus projetos. Além da API, este artigo também cobre como o arquivo de configuração é interpretado e como os laços de execução do jogo devem ser organizados. A API é código portável que deve funcionar sob ambientes OpenBSD, Linux, Windows e Web Assembly.*

1. Introdução

1.1. Organização de Arquivos

Quando um usuário digita `weaver PROJETO` na linha de comando, um diretório com um novo projeto Weaver e criado. Dentro deste diretório, o arquivo que contém o loop principal está em `src/game.c` e nele encontramos:

```
#include "game.h"

void main_loop(void){ // Um laço principal do jogo
 LOOP_INIT: // Código de inicialização

 LOOP_BODY: // Código executado a cada iteração
    if(W.keyboard[W_ANY])
        Wexit_loop();
 LOOP_END: // Código executado na finalização
    return;
}

int main(void){
  Winit(); // Initializes Weaver
  Wloop(main_loop); // Enter a new game loop
  return 0;
}
```

E dentro de `src/game.h`, nós encontramos:

```
#ifndef _game_h_
```

```
#define _game_h_

#include "weaver/weaver.h"
#include "includes.h"

struct _game_struct{
  // Você pode personalizar esta estrutura declarando variáveis aqui.
  // Mas não mude seu nome. E acesse ela por meioda variável W.game
  int whatever; // <- Variável só pra prevenir erro em certo compilador
} _game;

void main_loop(void);

#endif
```

Notar que neste arquivo existe uma estrutura que pode ser personalizada pelo usuário e cumpre o papel de centralizar algumas variáveis com estados globais. O tipo de coisa que para um jogo deve ser preservada ao salvarmos o progresso de um jogador. Ou que deve ser acessível para plugins. Ou ainda que deve ser transmitida para informar o estado do jogo a clientes conectados à rede. Segundo o comentário acima, esta estrutura deve ser referenciada pela variável `W.game`, o que já nos indica que a API será organizada de modo a fornecer um `struct` chamado `W` onde serão centralizados os recursos da API.

O arquivo `includes.h` é apenas um cabeçalho que inclui em um projeto todos os cabeçalho de módulos criados pelo usuário (cada módulo é um arquivo de código C e um cabeçalho).

Todo o código da API deve então estar presente ou ser incluída por macros dentro dos arquivos `weaver.c` e `weaver.h`. A organização do `weaver.h` é:

Arquivo: `project/src/weaver/weaver.h`**:**

```
#ifndef _weaver_h_
#define _weaver_h_
#ifdef __cplusplus
  extern "C" {
#endif
                    <Seção a ser Inserida: Estrutura Global>
                  <Seção a ser Inserida: Cabeçalhos Weaver>
                    <Seção a ser Inserida: Macros Weaver>
#ifdef __cplusplus
  }
#endif
#endif
```

1.2. A Estrutura W

A tal "estrutura global" referenciada nos códigos acima é o `struct` chamado `W`. Já mencionamos no comentário que colocaremos nele o `struct _game_struct _game` que definimos em `game.h`. Podemos então começar a definir tal estrutura:

Seção: Estrutura Global:

```
// Esta estrutura conterá todas as variáveis e funções definidas pela
// API Weaver:
extern struct _weaver_struct{
  struct _game_struct *game;
```

```
                <Seção a ser Inserida: Variáveis Weaver>
  //              <Seção a ser Inserida: Funções Weaver>
} W;
```

Notar que além de `W.game`, existirão outras variáveis presentes dentro desta estrutura. Basicamente iremos centralizar dentro dela todas as funções públicas da nossa API. Só não estarão nela funções que começam com "_", e que são consideradas internas e não deveriam ser usadas por programadores utilizando a API. Desta forma deixamos bem delimitado o que faz parte da API e também evitamos poluir com nomes o "namespace" global de programas em C.

Também definiremos aqui a estrutura geral de nosso arquivo `weaver.c`:

Arquivo: `project/src/weaver/weaver.c`:

```
#include "weaver.h"
#include "../game.h"
              <Seção a ser Inserida: API Weaver: Definições>
//              <Seção a ser Inserida: API Weaver: Funções>
               <Seção a ser Inserida: API Weaver: Base>
```

Nas definições declaramos novos tipos de estruturas de dados que forem necessárias. A primeira coisa que já podemos definir é a estrutura `W`, a qual é apenas declarada no cabeçalho:

Seção: API Weaver: Definições:

```
struct _weaver_struct W;
```

Na parte de funções definimos as funções a serem usadas. Já a última partedo arquivo contém as funções mais básicas da API. As únicas que não são colocadas dentro da estrutura `W`. Elas são a função de inicialização e a função que encerra o funcionamento do motor.

1.3. Funções de Inicialização e Finalização

Uma das coisas que a função de inicialização faz é inicializar os valores da estrutura `W`:

Seção: Cabeçalhos Weaver:

```
void Winit(void);
```

Seção: API Weaver: Base:

```
void Winit(void){
  W.game = &_game;
              <Seção a ser Inserida: API Weaver: Inicialização>
}
```

A função de finalização deve desalocar qualquer memória pendente, finalizar o uso de recursos, e deve também fechar o programa informando que tudo correu bem se assim realmente ocorreu:

Seção: Cabeçalhos Weaver:

```
void Wexit(void);
```

Seção: API Weaver: Base:

```
void Wexit(void){
  //              <Seção a ser Inserida: API Weaver: Finalização>
  exit(0);
}
```

O uso da função `exit` nos obriga a inserir o cabeçalho:

Seção: Cabeçalhos Weaver:

```
#include <stdlib.h>
```

Os demais códigos que serão executados durante a inicialização e finalização serão descritos ao longo do artigo.

2. Contagem de Tempo

Weaver mede o tempo em microssegundos (10^{-6}s) e armazena a sua contagem de tempo em pelo menos 64 bits de memória. Weaver serve para criar programas que executam dentro de um laço principal. Então além do tempo total em microssegundos desde que o programa inicializou, também armazenamos a diferença de tempo entre a iteração atual do programa e a última.

Então para começar devemos ter um lugar onde devemos armazenar a última medida de tempo que fizemos. Usaremos para isso uma variável global. No Windows usamos um dos tipos específicos para representar inteiros grandes e nos demais sistemas usamos uma estrutura de valor de tempo de alta resolução.

Seção: Cabeçalhos Weaver:

```
#if defined(_WIN32)
LARGE_INTEGER _last_time;
#else
struct timeval _last_time;
#endif
```

A ideia é que esta variável armazene sempre a última medida de tempo. Ela é inicializada com a primeira medida de tempo na inicialização:

Seção: API Weaver: Inicialização:

```
#if defined(_WIN32)
QueryPerformanceCounter(&_last_time);
#else
gettimeofday(&_last_time, NULL);
#endif
```

Após a inicialização, todas as outras atualizações desta variável deverão ser feitas por meio da função declarada abaixo:

Seção: Cabeçalhos Weaver (continuação):

```
unsigned long _update_time(void);
```

Tal função irá ler o tempo atual e armazenar na variável. Ela irá sempre retornar a diferença de tempo em microssegundos entre a leitura atual e a última. Em sistemas Unix faremos isso exatamente da maneira recomendada pelo manual da GNU Glbc de modo a tornar a subtração de tempo mais portável e funcional mesmo que os elementos da estrutura `timeval` sejam armazenadas como "unsigned". A desvantagem é que o código se torna menos claro. O código fica sendo:

Seção: API Weaver: Definições:

```
#if !defined(_WIN32)
unsigned long _update_time(void){
  int nsec;
  unsigned long result;
  struct timeval _current_time;
  gettimeofday(&_current_time, NULL);
  // Aqui temos algo equivalente ao "vai um" do algoritmo da subtração:
  if(_current_time.tv_usec < _last_time.tv_usec){
    nsec = (_last_time.tv_usec - _current_time.tv_usec) / 1000000 + 1;
    _last_time.tv_usec -= 1000000 * nsec;
    _last_time.tv_sec += nsec;
  }
  if(_current_time.tv_usec - _last_time.tv_usec > 1000000){
    nsec = (_current_time.tv_usec - _last_time.tv_usec) / 1000000;
```

```
    _last_time.tv_usec += 1000000 * nsec;
    _last_time.tv_sec -= nsec;
  }
  if(_current_time.tv_sec < _last_time.tv_sec){
    // Overflow
    result = (_current_time.tv_sec - _last_time.tv_sec) * (-1000000);
    // Sempre positivo:
    result += (_current_time.tv_usec - _last_time.tv_usec);
  }
  else{
    result = (_current_time.tv_sec - _last_time.tv_sec) * 1000000;
    result += (_current_time.tv_usec - _last_time.tv_usec);
  }
  _last_time.tv_sec = _current_time.tv_sec;
  _last_time.tv_usec = _current_time.tv_usec;
  return result;
}
#endif
```

Em sistema Windows, já existe uma função que trata o tempo como sendo em microssegundos exatamente no formato que já era usado antes mesmo de portarmos Weaver para Windows na versão beta. Por causa disso, a função se torna muito mais simples:

Seção: API Weaver: Definições:

```
#if defined(_WIN32)
unsigned long _update_time(void){
  LARGE_INTEGER prev = _last_time;
  QueryPerformanceCounter(&_last_time);
  return (_last_time - prev);
}
#endif
```

3. Os loops principais.

Todos os jogos são organizados dentro de loops, ou laços principais. Eles são basicamente um código que fica iterando indefinidamente até que uma condição nos leve a outro loop principal, ou então ao fim do programa.

Como foi mostrado no código inicial do `game.c`, um loop principal deve ser declarado como:

```
void nome_do_loop(void){
 LOOP_INIT: // Código executado na inicialização

 LOOP_BODY: // Código executado a cada iteração
    if(W.keyboard[W_ANY])
        Wexit_loop();
 LOOP_END: // Código executado na finalização
    return;
}
```

Antes de entendermos como devemos entrar em um loop principal corretamente, é importante descrever como este loop é executado. Nota-se que ele possui uma região de inicialização, de execução e finalização demarcada por rótulos escritos em letras maiúsculas.

Interpretar isso é bastante simples. É perfeitamente possível interpretar o código acima como:

```
void nome_do_loop(void){
```

```
  // LOOP_INIT
  for(;;){
    // LOOP_BODY
    if(W.keyboard[W_ANY])
        Wexit_loop();
  }
  // LOOP_END
}
```

Mas embora esta interpretação seja suficientemente adequada em certos contextos, não é assim que este código será traduzido. Não é em todos os ambientes em que é possível executar um loop infinito sem fazer com que a interface do jogo trave. Um exemplo é o ambiente WebAssembly de um navegador de Internet, onde os laços principais de um programa só podem ser executados se forem corretamente declarados como tais e a execução deles ocorre não por meio de um laço infinito, mas pela contínua chamada de uma função de laço principal.

Por causa disso, para tornar o código mais portável devemos encarar toda execução de um loop principal como no código abaixo:

```
for(;;)
  nome_do_loop();
```

E dentro da função de loop principal nós não colocamos um laço explícito. Ao invés disso, nós decidimos qual parte da função deve ser executada com ajuda dos rótulos inseridos, os quais na verdade são macros com lógica adicional embutida junto a alguns comandos `goto` que decidem o que será executado a cada vez que a função é chamada.

Uma consequência disso é que não é possível lidar com variáveis declaradas dentro da inicialização de um loop principal. Elas ,só teriam um valor correto dentro da inicialização, e dentro do corpo do loop principal seu valor seria indefinido. Por exemplo, o seguinte código terá resultado indefinido e talvez não imprima nada na tela:

```
// ERRADO
void loop(void){
LOOP_INIT:
  int var = 5;
LOOP_BODY:
  if(var == 5)
    printf("var == 5\n");
LOOP_END:
}
```

Já o seguinte código iria imprimir na saída padrão a cada iteração do loop:

```
// CERTO
static int var;

void loop(void){
LOOP_INIT:
  var = 5;
LOOP_BODY:
  if(var == 5)
    printf("var == 5\n");
LOOP_END:
}
```

Outra coisa que deve ser levada em conta é que as macros utilizadas escondem outro detalhe importante: não é apenas um laço principal que executamos, mas existem dois simultâneos. Um laço principal executa com uma frequência fixa: o laço que cuida da física e da lógica do jogo. Outro laço, nós apenas fazemos executar o mais rápido que der no hardware atual: o laço responsável por renderizar coisas na tela.

Idealmente para cada iteração do laço de renderização executamos uma ou mais iterações de nosso laço de física e lógica. Isso significa que podemos renderizar com uma frequência maior que executamos a iteração responsável por realmente mover objetos, detectar colisões e ler entrada do usuário. Para que a cada vez nós não renderizemos exatamente a mesma imagem, o que derrotaria o propósito de fazermos isso, nós interpolamos a posição dos objetos da tela de acordo com seus valores de aceleração, velocidade e posição.

Garantindo que a nossa física e lógica do jogo execute sempre em intervalos constantes, nós garantimos o determinismo necessário para podermos sincronizar partidas por meio de redes como a Internet. E renderizando os objetos na tela o mais rápido que podemos com ajuda de interpolação nos dá a experiência visual mais suave e natural que for possível.

Uma referência e maiores detalhes de como implementar isso pode ser encontrado em [Fiedler 2004]. Nossa implementação será como mostrado na referência, com exceção de que nosso código será muito menos transparente por ter que estar contido dentro de macros sem usar loops explícitos.

Vamos agora definir o que exatamente deverá existir em cada uma das macros que devem estar presentes em todo laço principal:

1) `LOOP_INIT`: Primeiro devemos checar variáveis que determinam se devemos encerrar o laço e se ainda estamos carregando recursos do laço (imagens, vídeos, shaders, sons). Se temos que sair e não estamos com qualquer recurso pendente que ainda está sendo carregado, apenas chamamos uma função para sair do laço (ela irá chamar de volta o laço-pai anterior ao laço atual ou encerrar o programa se ele não existir). Se temos que sair mas ainda não terminamos de carregar algum recurso, apenas encerramos a função. Ela será chamada novamente e não irá fazer nada até que os recursos terminem de ser carregados para que possamos sair. Depois checamos se esta função está sendo chamada pela primeira vez. Em caso afirmativo, apenas continuamos a execução. Em case negativo, fazemos um desvio incondicional para não termos que executar novamente o código de inicialização. Por fim, se não fizemos o desvio, faremos com que a variável `W.loop_name` passe a apontar par auma string com o nome do laço principal atual.

Saber se devemos continuar executando ou não um laço é algo que pode ser controlado por uma variável global, não sendo nem necessário se preocupar com semáforos. Afinal, somente um laço irá executar em um dado momento. O mesmo pode ser feito com a variável que determina se estamos na primeira execução de um laço (ou o começo de um laço). Vamos declarar ambas as variáveis:

Seção: Cabeçalhos Weaver (continuação):

```
bool _running_loop, _loop_begin;
```

E vamos inicializar elas:

Seção: API Weaver: Inicialização (continuação):

```
_running_loop = false;
_loop_begin = false;
```

Saber se ainda estamos carregando arquivos (ou melhor, quantos arquivos pendentes ainda estamos carregando) ou o nome do laço em que estamos são duas informações que são úteis não só para a lógica interna do motor, mas também para o seu usuário. Saber se o laço ainda não terminou de carregar é útil para fornecer uma tela de carregamento. Saber durante a execução o nome do laço em que estamos é útil tanto para depuração como para podermos carregar recursos diferentes dependendo do laço em que estamos. Por causa disso, ambas as variáveis devem ser declaradas na estrutura `W`:

Seção: Variáveis Weaver (continuação):

```
// Dentro da estrutura W:
unsigned pending_files;
char *loop_name;
```

Na inicialização ajustamos tais variáveis como 0 e `NULL` respectivamente:

Seção: API Weaver: Inicialização (continuação):

```
W.pending_files = 0;
```

```
W.loop_name = NULL;
```

A função que usaremos para sair do laço será esta:

Seção: Cabeçalhos Weaver (continuação):

```
#if !defined(_MSC_VER)
void _exit_loop(void) __attribute__ ((noreturn));
#else
__declspec(noreturn) void _exit_loop(void);
#endif
```

Mas não iremos definir ela ainda. Pelo cabeçalho nota-se que é uma função que nunca retorna, tendo isso especificado tanto pela convenção do GCC e Clang como pela convenção do Visual Studio. Isso porque o que ela fará é apenas chamar o código do laço anterior, ou sair do programa dependendo do caso.

Após descrever tudo isso, podemos enfim definir a macro de início de laço:

Seção: Cabeçalhos Weaver (continuação):

```
#define LOOP_INIT                                                    \
  if(!_running_loop && !W.pending_files)                              \
    _exit_loop();                                                     \
  if(!_running_loop)                                                  \
   goto _LOOP_FINALIZATION;                                           \
  if(!_loop_begin)                                                    \
    goto _END_LOOP_INITIALIZATION;                                    \
  W.loop_name = __func__;                                             \
  _BEGIN_LOOP_INITIALIZATION
```

Terminamos com o identificador `_BEGIN_LOOP_INITIALIZATION`, o qual será o verdadeiro nome do rótulo que existirá por trás de nossa macro.

2) `LOOP_BODY`: Ao chega nesta macro, devemos ajustar como falsa a informação de que é nossa primeira execução do laço, pois assim não iremos executar a inicialização novamente. Em seguida, aproveitamos para colocar um desvio incondicional por trás de um `if` que garanta que ele nunca seja executado para o rótulo que termina a macro anterior. Esse desvio nunca irá ocorrer, mas isso previne que o compilador reclame que o rótulo que encerra a última macro não é usado. Em seguida, crimaos o verdadeiro rótulo que marca o fim da inicialização e o começo da execução do corpo do laço. Neste começo de corpo do laço nós medimos o tempo que passou desde o último laço e armazenamos em um acumulador. Se este acumulador tiver um valor maior que o período de tempo entre execuções do código de física e lógica, então executamos o código presente no laço e o código associado à física. Caso contrário, só ignoramos tudo e vamos para a finalização onde apenas renderizamos na tela. Caso tenha passado um longo tempo entre cada iteração de laço, executamos mais de uma vez o código do corpo do laço.

O acumulador que usamos para saber se devemos executar a lógica e a física do jogo será chamado de `_lag`. Declaramos ele globalmente:

Seção: Cabeçalhos Weaver (continuação):

```
long _lag;
```

E o inicializamos:

Seção: API Weaver: Inicialização (continuação):

```
_lag = 0;
```

E podemos entrar em um novo loop principal, ou iniciar o primeiro loop principal por meio de uma chamada `Wloop(nome_do_loop)`.

Com relação à função `Wloop`, a primeira coisa com a qual devemos nos preocupar é se não existe algum arquivo ou recurso pendente que deve ser obtido antes de iniciarmos um novo loop.

Se existir, não poderemos passar para o novo loop. Por causa disso, a função `Wloop` na verdade é uma macro que verifica isso antes de chamar a verdadeira função de loop:

Seção: Macros Weaver (continuação):

```
#define Wloop(a) ((W.pending_files)?(false):(_Wloop(a)))
```

A estrutura `W` deve então possuir esta variável que armazena se temos arquivos pendentes ou não:

Seção: Variáveis Weaver:

```
// Isso vai dentro da estrutura W:
unsigned int pending_files;
```

E ela deve ser inicializada como 0, incrementar se começamos a obter um recurso relevante assíncrono e decrementar quando terminamos.

Seção: API Weaver: Inicialização:

```
W.pending_files = 0;
```

Com relação à verdadeira função de loop, ela é declarada da seguinte forma:

Seção: Cabeçalhos Weaver (continuação):

```
#if !defined(_MSC_VER)
void _Wloop(void (*f)(void)) __attribute__ ((noreturn));
#else
__declspec(noreturn) void _Wloop(void (*f)(void));
#endif
```

Notar que a função de loop nunca retornará e a macro condicional serve para especificar isso tanto em uma notação que o GCC e Clang entendem como na notação do Visual Studio. Qualquer código presente imediatamente após ela será ignorado, a menos que falhemos ao iniciar um novo loop devido a algum recurso assíncrono pendente.

Seção: Funções Weaver:

<Seção a ser Inserida: API Weaver: Finalização>

Seção: API Weaver: Funções:

<Seção a ser Inserida: API Weaver: Inicialização>

Referências

Fiedler, G. (2004) "Fix Your Timestep!", acessado em `https://gafferongames.com/post/fix_your` em 2020.