A API 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
#include "../../conf/conf.h"
#include <stdlib.h> // Always useful. Declares 'size_t'.
                    <Seção a ser Inserida: Estrutura Global>
                   <Seção a ser Inserida: Cabeçalhos Weaver>
                    <Seção a ser Inserida: Macros Weaver>
#ifdef __cplusplus
  }
#endif
#endif
```

Notar que incluímos no cabeçalho o arquivo de configuração `conf.h` responsável por controlar e configurar o comportamento de nosso motor.

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: Cabeçalhos Internos>
           <Seção a ser Inserida: API Weaver: Variáves Estáticas>
              <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 incluimos o cabeçalho necessário para usá-lo) e nos demais sistemas usamos uma estrutura de valor de tempo de alta resolução.

Seção: Cabeçalhos Weaver:

```
#if defined(_WIN32)
#include <windows.h>
LARGE_INTEGER _last_time;
#else
#include <sys/time.h>
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;
  prev.QuadPart = _last_time.QuadPart;
  QueryPerformanceCounter(&_last_time);
  return (_last_time.QuadPart - prev.QuadPart);
}
#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 física e lógica executamos uma ou mais iterações de nosso laço de renderização. 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 nas subseções seguintes o que exatamente deverá existir em cada uma das macros que devem estar presentes em todo laço principal.

3.1. Inicialização do Loop.

Isso é o que fará a macro `LOOP_INIT`:

Primeiro devemos checar variáveis que determinam se devemos encerrar o laço. Se estivermos, mas ainda houverem recursos sendo carregados (imagens, vídeos, shaders, sons), apenas retornamos da função. Se não houver nada sendo carregado, mas ainda não executamos a finalização deste laço, pulamos para executar a finalização. Se não há nada a ser carregado e já executamos a finalização, aí sim encerramos de vez o laço. Depois checamos se esta função está sendo chamada pela primeira vez. Em caso afirmativo, apenas continuamos a execução. Em caso 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 ser uma 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) e se já executamos a finalização. Vamos declarar ambas as variáveis:

Seção: Cabeçalhos Weaver (continuação):

```
#include <stdbool.h>
bool _running_loop, _loop_begin, _loop_finalized;
```

E vamos inicializar elas:

Seção: API Weaver: Inicialização (continuação):

```
_running_loop = false;
_loop_begin = false;
_loop_finalized = 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`. O tamanho máximo de nome de um laço que podemos armazenar pode ser personalizado com a macro `W_MAX_LOOP_NAME`.

Seção: Variáveis Weaver (continuação):

```
// Dentro da estrutura W:
#if !defined(W_MAX_LOOP_NAME)
#define W_MAX_LOOP_NAME 64
#endif
unsigned pending_files;
char loop_name[W_MAX_LOOP_NAME];
```

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

Seção: API Weaver: Inicialização (continuação):

```
W.pending_files = 0;
W.loop_name[0] = '\0';
```

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){                                                \
    if(W.pending_files)                                              \
      return;                                                        \
    if(!_loop_finalized){                                            \
      _loop_finalized = true;                                        \
      goto _LOOP_FINALIZATION;                                       \
    }                                                               \
    _exit_loop();                                                    \
  }                                                                  \
  if(!_loop_begin)                                                    \
    goto _END_LOOP_INITIALIZATION;                                    \
  snprintf(W.loop_name, W_MAX_LOOP_NAME, "%s", __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.

3.2. Corpo do Loop.

Isso é o que fará a macro `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):

```
unsigned long _lag;
```

E o inicializamos:

Seção: API Weaver: Inicialização (continuação):

```
_lag = 0;
```

Existirão variáveis que poderão ser lidas pelo usuário com informações de tempo. Uma delas (`W.t`) conterá a quantidae de microssegundos desde que o jogo inicializou. Outra delas (`W.dt`) conterá o intervalo entre execuções do motor de física e da lógica do jogo. Ambas as variáveis precisam ser declaradas na estrutura `W`:

Seção: Variáveis Weaver (continuação):

```
unsigned long long t;
unsigned long dt;
```

A primeira das variáveis, obviamente deve ser inicializada como zero. A segunda deve ser inicializada como tendo o mesmo valor que uma macro `W_TIMESTEP` que pode ser definida pelo usuário para que assim ele controle a granularidade de execução do código mais pesado do motor de física e lógica do jogo. Se esta macro não for definida, iremos assumir um valor de 40000 microssegundos. Isso significa uma freuência de 25 Hz de execução do motor de física (25 vezes por segundo).

Seção: API Weaver: Inicialização (continuação):

```
#if !defined(W_TIMESTEP)
#define W_TIMESTEP 40000
#endif
W.dt = W_TIMESTEP;
W.t = 0;
```

O código da engine de física e lógica interna do jogo deve ficar encapsulado em uma função chamada `_update`:

Seção: Cabeçalhos Weaver (continuação):

```
void _update(void);
```

Por hora ainda não iremos definir o código a ser executado nesta função:

Seção: API Weaver: Base (continuação):

```
void _update(void){
            <Seção a ser Inserida: Código a executar todo loop>
```

```
}
```

Mas com as definições que fizemos já podemos definir a nossa macro que marca o começo do código do laço principal:

Seção: Cabeçalhos Weaver (continuação):

```
#define LOOP_BODY                                                    \
  _loop_begin =  false;                                              \
  if(_loop_begin)                                                    \
    goto _BEGIN_LOOP_INITIALIZATION;                                 \
_END_LOOP_INITIALIZATION:                                            \
  _lag += _update_time();                                            \
  while(_lag >= W.dt){                                               \
    _update();                                                       \
_LABEL_0
```

Notar que a macro acima abre um laço com um `while`, mas não o encerra. Ele deverá ser encerrado pelo código inserido pela macro que delimita o fim do corpo do laço principal. A qual também deverá decrementar a variável `_lag` para que este não seja um laço infinito.

3.3. Finalização do Loop.

Isso é o que fará a macro `LOOP_END`:

Primeiro para fornecer uma condição de parada para o laço começado na macro anterior, decrementaremos de `_lag` o valor de `W.dt`. Em seguida, incrementaremos `W.t` com tal quantidade de microssegundos. Só então encerramos o bloco do laço dentro do qual estávamos. Fora deste laço, seguimos realizando aquilo que deve ser feito independente de estarmos executando o motor de física e de lógica de jogo ou não. Como são sempre atividades relacionadas à renderização na tela, isso estará dentro de uma função chamada `_render`. Em seguida, simplesmente retornamos. Se estamos executando o código aqui, é porque estávamos terminando de executar o corpo do link. Depois do retorno colocamos um desvio para um rótulo que nunca será executado e apenas nos protege de aviso do compilador. E enfim colocamos um rótulo imediatamente antes da finalização e que pode ser atingido somente por um desvio se detectarmos que o laço acabou e precisamos rodar a finalização.

A única coisa nova que temos aqui então é a função de renderização:

Seção: Cabeçalhos Weaver (continuação):

```
void _render(void);
```

Por hora ainda não iremos definir o código a ser executado nesta função:

Seção: API Weaver: Base (continuação):

```
void _render(void){
                <Seção a ser Inserida: Código de renderização>
}
```

E agora colocamos o código da macro:

Seção: Cabeçalhos Weaver (continuação):

```
#define LOOP_END                                                     \
    _lag -=  40000;                                                  \
    W.t +=  40000;                                                   \
  }                                                                  \
  _render();                                                         \
  return;                                                            \
  goto _LABEL_0;                                                     \
_LOOP_FINALIZATION
```

3.4. Entrando em Laço Principal.

Frequentemente estaremos trocando de laços principais ao longo de um jogo. Mas nem sempre isso significa uma substituição completa. Alguns laços principais ocorrem dentro de outros laços principais. Por exemplo, quando acessamos um menu de configurações durante um jogo. Ou quando em um RPG clássico por turnos mudamos para o modo de combate após um encontro aleatório.

Podemos então formar uma pilha de laços principais, onde ao sairmos do último laço voltamos para o que está empilhado imediatamente abaixo dele ao invés de sairmos do jogo.

Sendo assim, existem duas formas de entrar em um laço principal. Uma delas, por meio da função que definiremos chamada `Wloop` e a segunda por meio da `Wsubloop`. A primeira envolve substituirmos o laço principal atual por um novo. A segunda enolve apenas criar um laço principal dentro do laço atual. A primeira é algo que só poderemos fazer se não houverem arquivos pendentes sendo carregados (possivelmente haverá uma tela de carregamento neste caso). Por isso apenas nos asseguramos disso por mieo de um truque com macros:

Seção: Cabeçalhos Weaver (continuação):

```
#if !defined(_MSC_VER)
void _Wloop(void (*f)(void)) __attribute__ ((noreturn));
void Wsubloop(void (*f)(void)) __attribute__ ((noreturn));
#else
__declspec(noreturn) void _Wloop(void (*f)(void));
__declspec(noreturn) void Wsubloop(void (*f)(void));
#endif
#define Wloop(a) ((W.pending_files)?(false):(_Wloop(a)))
```

Nenhum dos dois tipos de função irá retornar jamais. Então especificamos isso tanto na convenção de compiladores como Clang e GCC como na do Visual Studio.

Vamos precisar de uma pilha de laços principais, que representaremos por meio de uma sequência de ponteiros para funções. Essa nossa sequência será um vetor com `W_MAX_SUBLOOP` posições. Esta macro poderá ser controlada pelo usuário, mas se não estiver definida, assumiremos que será 3. Além da pilha, vamos precisar de uma variável para nos informar em qual profundidade da pilha estamos no momento (`_number_of_loops`).

A declaração destas duas variáveis será:

Seção: Cabeçalhos Weaver (continuação):

```
#if !defined(W_MAX_SUBLOOP)
#define W_MAX_SUBLOOP 3
#endif
int _number_of_loops;
void (*_loop_stack[W_MAX_SUBLOOP]) (void);
```

E inicializamos a contagem do número de laços como zero:

Seção: API Weaver: Inicialização (continuação):

```
_number_of_loops = 0;
```

Entrar em um novo laço principal por meio de `Wloop` envolve verificar se já estamos antes em um laço. Se for o caso, cancelamos ele. Em seguida, carregamos o novo laço para a pilha e ajustamos o valor da contagem de laços em execução. Atualizamos o nosso valor de contagem de tempo e finalmente executamos o laço. Em ambiente Web Assembly em navegador de Internet isso envolve chamar diretamente uma função que estabelece nossa função escolhida como laço principal. Nos demais, basta executar a função em um `while` comum:

Seção: API Weaver: Base (continuação):

```
void _Wloop(void (*f)(void)){
  if(_number_of_loops > 0){
                <Seção a ser Inserida: Cancela Loop Principal>
    _number_of_loops --;
  }
            <Seção a ser Inserida: Código antes de Loop e Subloop>
          <Seção a ser Inserida: Código antes de Loop, não de Subloop>
  _loop_stack[_number_of_loops] = f;
  _number_of_loops ++;
#if defined(__EMSCRIPTEN__)
  emscripten_set_main_loop(f, 0, 1);
#else
  while(1)
    f();
#endif
}
```

Cancelar um laço principal já existente envolve caso estejamos executando em ambiente Web Assembly invocar a função que interrompe o laço atual:

Seção: Cancela Loop Principal:

```
#if defined(__EMSCRIPTEN__)
emscripten_cancel_main_loop();
#endif
```

Iniciar um novo subloop, ou sublaço, é bastante semelhante. Mas tratamos de maneira diferente o nosso contador de laços, já que ele realmente precisa ser incrementado. E também temos que checar se tivemos um estouro na nossa pilha de laços:

Seção: API Weaver: Definições (continuação):

```
void Wsubloop(void (*f)(void)){
#if defined(__EMSCRIPTEN__)
    emscripten_cancel_main_loop();
#endif
            <Seção a ser Inserida: Código antes de Loop e Subloop>
          <Seção a ser Inserida: Código antes de Subloop, não de Loop>
  if(_number_of_loops >= W_MAX_SUBLOOP){
    fprintf(stderr, "Error: Max number of subloops achieved.\n");
    fprintf(stderr, "Please, increase W_MAX_SUBLOOP in conf/conf.h"
            " to a value bigger than %d.\n", W_MAX_SUBLOOP);
    exit(1);
  }
  _loop_stack[_number_of_loops] = f;
  _number_of_loops ++;
#if defined(__EMSCRIPTEN__)
  emscripten_set_main_loop(f, 0, 1);
#else
  while(1)
    f();
#endif
}
```

Vamos incluir o cabçalho para podermos imprimir mensagens de erro:

Seção: Cabeçalhos Weaver (continuação):

```
#include <stdio.h>
```

Uma coisa que faremos tanto antes de um laço como de um sublaço é atualizar a variável `_running_loop` que avisa ao motor que realmente estamos executando um laço ao invés de tentar sair dele, ajustamos a variável que diz que estamos entrando no começo de um laço e que portanto precisaremos executar o código de inicialização, a variável que informa que a finalização do laço ainda não foi executada. Estas são as variáveis que controlam o fluxo de execução do laço. Também atualizamos nosso contador de tempo:

Seção: Código antes de Loop e Subloop:

```
_running_loop = true;
_loop_begin = true;
_loop_finalized = false;
_update_time();
```

3.4. Saindo do Laço Principal.

Assim como existem duas funções para entrar em laços principais, existitão duas funções para sair. Uma delas apenas sai do laço principal atual, voltando para o próximo laço principal da pilha se existir. A outra sai de todos os laços existentes e encerra o programa.

A que sai de todos os laços já foi parcialmente definida e é a `Wexit`.

Sair de um laço atual já existente envolve ajustar a variável global que diz que estamos executando o laço para um valor que significa que queremos sair do laço:

Seção: Cabeçalhos Weaver (continuação):

```
#define Wexit_loop() (_running_loop = false)
```

Se você verificar novamente o código inserido pelas macros presentes em laços principais, verá que se esta variável for falsa e não existir nenhum recurso ou arquivo ainda sendo carregado, então será chamada a função `_exit_loop`:

Seção: Cabeçalhos Weaver (continuação):

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

Já o código desta função envolve cancelar o laço atual e checar se existe outro na pilha. Se não existir, o programa se encerra. Se existir, executa código de entrada no novo laço:

Seção: API Weaver: Definições (continuação):

```
void _exit_loop(void){
  if(_number_of_loops <= 1){
    Wexit();
    exit(1); // Esta linha apenas previne aviso na compilação
  }
  else{
            <Seção a ser Inserida: Código após sairmos de Subloop>
    _number_of_loops --;
            <Seção a ser Inserida: Código antes de Loop e Subloop>
#if defined(__EMSCRIPTEN__)
    emscripten_cancel_main_loop();
    emscripten_set_main_loop(_loop_stack[_number_of_loops - 1], 0, 1);
#else
    while(1)
```

```
      _loop_stack[_number_of_loops - 1]();
#endif
  }
}
```

Gerenciamento de Memória

O gerenciamento de memória utiliza um subsistema próprio que fornece suas próprias funções de alocação e gerenciamento de memória. Uma das diferenças do subsistema utilizado e do gerenciamento tradicional por meio das funções `malloc` e `free` é que nele somos obrigados a informar com antecedência qual a quantidade máxima de memória que iremos precisar.

A quantidade máxima deve ser pensada de acordo com as especificações de um projeto. Um jogo feito para um console pode usar como quantidade máxima a quantidade de RAM da especificação do console. Um projeto escrito para rodar em computadores pode começar com um valor pequeno, talvez potência de dois que sempre irá dobrar à medida que o projeto vai sendo escrito e a quantidade inicial não é mais o suficiente, sempre cuidando para que o valor não se torne maior que o suportado pelas máquinas na qual espera-se rodar o programa após ser finalizado.

Espera-se que o usuário informe qual a quantidade máxima de memória definindo a macro `W_MAX_MEMORY` em `conf/conf.h`. Se esta macro não estiver definida, vamos deliberadamente escolher o valor pequeno de 4 MiB, o qual forçará o usuário a ter que explicitar um valor mais adequado exceto nos menores e mais triviais dos projetos:

Seção: Cabeçalhos Weaver (continuação):

```
#ifndef W_MAX_MEMORY
#define W_MAX_MEMORY 4096
#endif
```

Essa quantidade máxima de memória será alocada durante a inicialização e armazenaremos o endereço para ela na variável abaixo:

Seção: API Weaver: Variáves Estáticas:

```
static void *memory_arena;
```

Para podermos usar as funções de nosso subsistema de memória, incluimos o cabeçalho:

Seção: API Weaver: Cabeçalhos Internos:

```
#include "memory.h"
```

E durante a inicialização nós alocamos em nossa arena de memória interna toda a memória que iremos precisar ao longo da execução de nosso projeto:

Seção: API Weaver: Inicialização (continuação):

```
memory_arena = _Wcreate_arena(W_MAX_MEMORY);
```

Na finalização nós desalocamos toda a memória obtida na inicialização. Mas antes disso nós desalocamos qualquer outra coisa que tenha sido alocada na inicialização. A invocação para a função `_Wtrash` faz isso: desaloca tudo da pilha da direita (por isso o argumento 1 abaixo), o local reservado para alocações internas da nossa API:

Seção: API Weaver: Finalização:

```
_Wtrash(memory_arena, 1);
_Wdestroy_arena(memory_arena);
```

Antes de entrar em cada laço principal, usamos a função `_Wmempoint`. Esta função salva o estado atual da memória para podermos voltar até ele depois. Fazendo isso, depois para desalocarmos de uma só vez tudo que foi alocado durante o laço principal torna-se fácil: é só restaurar a arena de memória para o estado salvo.

Como salvar o estado atual requer escrever na memória, vamos precisar saber qual o alinhamento de bytes recomendado para o nosso CPU. Weaver obtém este valor da macro `W_MEMORY_ALIGNMENT` que pode ser definida em `conf/conf.h`. Se ela não estiver definida, nós usamos por padrão o tamanho de um `unsigned long` como palpite.

Seção: Cabeçalhos Weaver (continuação):

```
#ifndef W_MEMORY_ALIGNMENT
#define W_MEMORY_ALIGNMENT (sizeof(unsigned long))
#endif
```

Com isso podemos salvar o estado da pilha esquerda e direita de nossa arena de memória:

Seção: Código antes de Loop e Subloop (continuação):

```
_Wmempoint(memory_arena, W_MEMORY_ALIGNMENT, 0);
_Wmempoint(memory_arena, W_MEMORY_ALIGNMENT, 1);
```

Isso significa que assim que sairmos de um laço, devemos restaurar o estado da arena de memória para como estava quando entramos nele, tanto na pilha de memória da esquerda como da direita:

Seção: Cancela Loop Principal (continuação):

```
_Wtrash(memory_arena, 0);
_Wtrash(memory_arena, 1);
```

E isso basicamente será o coletor de lixo de nosso projeto. Vamos agora definir uma função para que o usuário possa usar nossa função de alocação de memória. Todas as alocações feitas pelo usuário devem ser da pilha esquerda da memória, usando o alinhamento padrão:

Seção: API Weaver: Funções:

```
static void *_alloc(size_t size){
  return _Walloc(memory_arena, W_MEMORY_ALIGNMENT, 0, size);
}
```

Vamos declarar um ponteiro para esta função dentro da estrutura `W` para que a função possa ser invocada como `W.alloc()`:

Seção: Funções Weaver:

```
void *(*alloc)(size_t);
```

E a função estará pronta para ser usada após a inicialização:

Seção: API Weaver: Inicialização (continuação):

```
W.alloc = _alloc;
```

Definições Vazias

As seguintes definições ainda estão em branco e estão aqui para manter o projeto compilando com sucesso. Esta parte não existirá na versão final deste documento e as partes que aparecem aqui vazias estarão posteriormente preenchidas e melhor explicadas.

Seção: Código antes de Subloop, não de Loop:

Seção: Código após sairmos de Subloop:

Seção: Código a executar todo loop:

Seção: Código de renderização:

Seção: Código antes de Loop, não de Subloop:

Seção: Macros Weaver:

Referências

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