Interface de Janela Weaver

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

Abstract: *This article contains the implementation of a portable Windowing API, which can be used to create a single window in Windows, Linux, BSD, or a canvas in Web Assembly running in a web browser. You can set a fullscreen mode, change to windowed mode, resize it, use OpenGL commands and get input from mouse and keyboard. All this can be achieved by the portable API defined in this work.*

Resumo: *Este artigo contém a implementação de uma interface de uso de janelas. Ela pode ser usada para criar uma única janela no Windows, Linux, BSD, ou então um "canvas" rodando Web Assembly em um navegador de Internet. Você pode ativar um modo de tela cheia, mudar para um modo em janela, mudar o tamanho da janela, usar comandos OpenGL e obter entrada do mouse e teclado. Tudo isso pode ser obtido pela interface de aplicação portável definida neste trabalho.*

Índice

1. Introducão

Um programa de computador gráfico precisa de um espaço no qual ele pode

desenhar na tela. Em alguns ambientes, como videogames, por exemplo, cada programa em execução simplesmente tem controle de toda a tela automaticamente sem que seja necessário reservá-lo ou pedi-lo para um Sistema Operacional. Por outro lado, quando um programa executa em um computador com algum ambiente gráfico moderno, é necessário pedir para que uma região chamada "janela" seja criada. Nela o programa passa a ter controle sobre o seu conteúdo e pode desenhar na região.

Além de criar uma janela, é importante que tenhamos a capacidade de entrar e sair do modo tela-cheia se estivermos em ambiente que permite isso. E se estivermos em modo de janela, mudar o tamanho da janela.

1.1. Programação Literária e Notação

Este artigo utiliza a técnica de "Programação Literária" para desenvolver a API de gerador de números aleatórios. Esta técnica foi apresentada em [Knuth, 1984] e tem por objetivo desenvolver *softwares* de tal forma que um programa de computador a ser compilado é exatamente igual a um documento escrito para pessoas detalhando e explicando o código. O presente documento não é algo independente do código, mas sim consiste no próprio código-fonte do projeto. Ferramentas automáticas são utilizadas para extrair o código deste documento, colocá-lo na ordem correta e produzir o código que é passado para o compilador.

Por exemplo, neste artigo serão definidos dois arquivos diferentes: `window.c` e `window.h`, os quais podem ser inseridos estaticamente em qualquer projeto, ou compilados como uma biblioteca compartilhada. O conteúdo de `window.h` é:
(P)

Arquivo: `src/window.h`**:**

```
#ifndef WEAVER_WINDOW
#define WEAVER_WINDOW
#ifdef __cplusplus
extern "C" {
#endif
#include <stdbool.h> // Define tipo 'bool'
#if !defined(_WIN32)
#include <sys/param.h> // Necessário no BSD, mas causa problema no Windows
#endif
                <Seção a ser Inserida: Cabeçalho OpenGL>
                  <Seção a ser Inserida: Define Macros>
            <Seção a ser Inserida: Estruturas de Dados de Janela>
                <Seção a ser Inserida: Declarações de Janela>
#ifdef __cplusplus
}
#endif
#endif
```

As duas primeiras linhas assim como a última são macros que impedem que garantem que as funções e variáveis declaradas ali serão inseridas no máximo uma só vez em cada unidade de compilação. Também colocamos macros para checar se estamos compilando o código como C ou C++. Se estivermos em C++, avisamos o compilador que estamos definindo tudo como código C e garantimos que não vamos modificar nada usando sobrecarga de operadores. O código poderá assim ser armazenado de maneira mais compacta.

As partes vermelhas no código acima mostram que código será inserido ali no futuro.

Cada trecho de código tem um título, que no caso acima é `src/window.h`. O título indica onde o código será inserido. No caso acima, o código irá para um arquivo. Em trechos de código futuros, haverá diversos títulos, inclusive um com exatamente o nome das partes em vermelho do código acima. Se o título de um trecho de código é igual um trecho em vermelho a ser inserido, é

ali que tal trecho de código deve ser posicionado no processo de compilação.

Como um segundo exemplo de código, também declararemos aqui que quando estamos em modo de depuração (ou seja, quando a macro `W_DEBUG_WINDOW` está definida) iremos precisar das declarações de funções de entrada e saída padrão, já que nosso código se tornará mais verboso:

Seção: Cabeçalhos:

```
#if defined(W_DEBUG_WINDOW)
#include <stdio.h>
#endif
```

Note que ainda não informamos onde exatamente o código denominado "Cabeçalho" será colocado. Faremos isso posteriormente.

1.2. Funções de API a serem Definidas

Neste artigo iremos definir as seguintes funções:

Seção: Declarações de Janela:

```
bool _Wcreate_window(struct _Wkeyboard *keyboard, struct _Wmouse *mouse);
```

Esta é a função que irá criar uma nova janela. Por padrão, uma janela em tela cheia. Se a macro `W_WINDOW_NO_FULLSCREEN` estiver definida, ao invés disso, ele criará uma janela que não está em modo de tela-cheia (se suportado pelo sistema).

Se a macro `W_DEBUG_WINDOW` estiver definida, esta função também imprimirá na tela informação sobre o ambiente gráfico. Sua resolução, por exemplo, ou possivelmente outras informações que possam ser relevantes.

A função irá inicializar as estruturas de dado do mouse e teclado.

Em caso de erro, a função retornará falso.

Seção: Declarações de Janela:

```
bool _Wdestroy_window(void);
```

Esta função vai fechar a janela aberta, liberando qualquer recurso que tenha sido alocado pela função anterior ao criar janela. A função deve ser invocada sempre após a janela já ter sido criada. Em caso de erro, retorna falso.

Seção: Declarações de Janela:

```
bool _Wrender_window(void);
```

Esta função irá efetivamente renderizar na tela todos os comandos OpenGL pendentes desde a última renderização. Retornará falso em caso de erro. Esta função provavelmente será chamada no fim de cada iteração de um laço principal.

Seção: Declarações de Janela:

```
bool _Wget_screen_resolution(int *resolution_x, int *resolution_y);
```

Esta função obtém a resolução da tela e a armazena nos ponteiros passados. Se houver mais de uma, retornará aquela que o sistema identifica como sendo a principal. Se um erro ocorrer, ela retornará falso e a resolução será armazenada como zero.

Seção: Declarações de Janela:

```
bool _Wget_screen_dpi(int *dpi_x, int *dpi_y);
```

Esta função irá obter a densidade de pixels da tela, medido em DPI (pontos por polegada). Se um erro ocorrer, ela retornará falso e os valores passados como argumento serão inicializados como zero.

Seção: Declarações de Janela:

```
bool _Wget_window_size(int *width, int *height);
```

Esta função obtém o tamanho em pixels da janela atual e armazena a informação nos ponteiros passados como argumento. Se nós não temos nenhuma janela, ou em caso de erro, a função retorna falso e armazena zero nos ponteiros. Já em caso de sucesso, ela retorna verdadeiro e armazena o resultado correto nos ponteiros.

Seção: Declarações de Janela:

```
void _Wget_window_input(unsigned long long current_time,
```

```
                        struct _Wkeyboard *keyboard,
                        struct _Wmouse *mouse);
```

Esta função periodicamente deve ser chamada para atualizar o estado do teclado e mouse. O primeiro argumento é o tempo atual medido em alguma unidade de tempo. Os próximos argumentos são estruturas que representam teclado e mouse a serem atualizadas.

Seção: Declarações de Janela (continuação):

```
void _Wflush_window_input(struct _Wkeyboard *keyboard,
                          struct _Wmouse *mouse);
```

Esta função limpa o estado de nosso teclado e mouse. Ela deve ser chamada toda vez que pararmos de ler periodicamente o mouse e teclado com chamadas a `_Wget_window_input`. O resultado de chamar esta função é que reiniciamos o estado de nosso mouse e teclado, parando de considerar qualquer tecla como pressionada ou solta.

Seção: Declarações de Janela (continuação):

```
bool _Wis_fullscreen(void);
```

Esta função retorna se estamos em modo de tela-cheia ou não. O resultado é indefinido se uma janela não foi criada.

Seção: Declarações de Janela (continuação):

```
void _Wtoggle_fullscreen(void);
```

Esta função alterna entre o modo de tela cheia e modo em janela. Dependendo do ambiente ela pode falhar.

Seção: Declarações de Janela (continuação):

```
bool _Wresize_window(int width, int height);
```

Esta função muda o tamanho da janela se ela existir e se não estivermos no modo de tela-cheia. Caso a operação falhe devido a estas condições não estiverem satisfeitas, a função retorna falso.

Seção: Declarações de Janela (continuação):

```
void _Wset_resize_function(void (*func)(int, int, int, int));
```

Esta função registra uma função para que seja executada toda vez que ocorrer uma mudança no tamanho de nossa janela. A função recebe como argumentos a largura e altura anterior da janela e como dois últimos argumentos a largura e altura da janela após o redimencionamento.

2. Criando e Gerenciando a Janela

Nesta seção definiremos funções e códigos que envolvem criar uma janela e criar um contexto OpenGL compatível com OpenGL ES 2.0 nelas.

2.1. Obtendo a Resolução e Densidade de Pontos da Tela

Antes de criar nossa janela, é importante checar qual a resolução da tela e da janela. Se estivermos iniciando em modo de tela cheia, este será o tamanho da nossa janela. Isso é feito de maneira diferente dependendo do sistema operacional.

Vamos também criar uma variável estática que funcionará como um cache que memorizará o último tamanho da janela em pixels que obtemos. Memorizar o tamanho da janela é importante porque precisamos deste valor para depois transformar as coordenadas da posição do mouse que iremos medir. Praticamente todos os ambientes informam as coordenadas na janela usando como origem o canto superior esquerdo. Contudo, a API Weaver prefere usar como origem o canto inferior esquerdo para estar mais de acordo com a convenção matemática. Para fazer a transformação de coordenada, é importante memorizar o tamanho da janela para não termos que medi-la toda hora:

Seção: Variáveis Locais:

```
static int window_size_y = 0, window_size_x = 0;
```

Além da resolução, outra medida que pode ser relevante é a quantidade de pontos por polegada, que mede a densidade de pixels por área da nossa tela. O chamado valor de DPI pode ser consultado

por algumas aplicações, mas não é algo que precisamos nos preocupar em ter armazenado em cache, já que ele é um valor que não muda e raramente precisará ser consultado.

2.1.1. Obtendo a Resolução e Densidade de Pontos da Tela no X

O Sistema de Janelas X, também conhecido como X11, é um sistema de janelas presente em muitos Sistemas Operacionais, como Linux, BSD, e até mesmo no MacOX X, onde ele está presente para garantir compatibilidade com programas mais antigos desenvolvidos antes de seu sistema de janelas atual. Iremos começar com a implementação de nossa função no X11 por ser o sistema de janelas mais amplamente presente.

O X11 funciona em uma arquitetura de cliente-servidor. Quando criamos um programa gráfico, criamos um cliente que se comunica com o servidor X usando *sockets*. Todas as operações como a criação de janelas, redimencionar a janela e mais, são feitas com o cliente pedindo para que elas sejam feitas para o servidor, que executa os pedidos se possível.

Iremos usar o X11 sempre que não estivermos usando o Windows (que não o fornece) e nem estivermos compilando Web Assembly (navegadores de Internet também não o implementam). Antes de usar o X11, precisamos inserir seu cabeçalho relevante:

Seção: Cabeçalhos:

```
#if defined(__linux__) || defined(BSD)
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#endif
```

No X11, como temos que nos comunicar com um servidor, assumimos que temos uma variável com algum tipo de estrutura de dados com informação sobre a nossa conexão. Esta variável é chamada de "`display`".

Usando tal conexão, que assumimos ser uma variável estática em `window.c`, ler a resolução da tela é feito com a função abaixo:

Seção: Funções da API:

```
#if defined(__linux__) || defined(BSD)
bool _Wget_screen_resolution(int *resolution_x, int *resolution_y){
  bool keep_alive = true; //Devemos manter a conexão ativa?
  // A primeira coisa a ser feita é chamar esta função para preparar o X11
  // para código multi-thread:
  XInitThreads();
  // Se não temos uma conexcão ativa, criamos ela:
  if(display == NULL){
    display = XOpenDisplay(NULL);
    keep_alive = false;
  }
  // Obtemos a tela padrão:
  int screen = DefaultScreen(display);
  // E perguntamos a resolução
  *resolution_x = DisplayWidth(display, screen);
  *resolution_y = DisplayHeight(display, screen);
  // Se não havia uma conexão ativa ao servidor X antes de invocar a função,
  // fechamos a conexão que abrimos para obter a resolução:
  if(!keep_alive){
    XCloseDisplay(display);
    display = NULL;
  }
  // Se tudo deu certo podemos retornar verdadeiro:
  return true;
}
#endif
```

Claro, temos que assumir a existência da seguinte variável estática:

Seção: Variáveis Locais:

```
#if defined(__linux__) || defined(BSD)
static Display *display = NULL; //Conexão com servidor e info sobre tela
#endif
```

Já com relação à densidade de pontos da tela, medidos em DPI, devemos primeiro checar a resolução da tela, e em seguida usamos as funções `DisplayWidthMM` e `DisplayHeightMM` para obter o tamanho da tela em milímetros. Calcular o número de pontos por polegada é feito apenas convertendo os milímetros para polegada e dividindo o número de pixels por tal valor, tanto horizontalmente como verticalmente.

Seção: Funções da API (continuação):

```
#if defined(__linux__) || defined(BSD)
bool _Wget_screen_dpi(int *dpi_x, int *dpi_y){
  int resolution_x, resolution_y;
  double xres, yres;
  if(!_Wget_screen_resolution(&resolution_x, &resolution_y)){
    *dpi_x = *dpi_y = 0;
    return false;
  }
  xres = ((((double) resolution_x) * 25.4) /
          ((double) DisplayWidthMM(display, 0)));
  yres = ((((double) resolution_y) * 25.4) /
          ((double) DisplayHeightMM(display, 0)));
  *dpi_x = (int) (xres + 0.5);
  *dpi_y = (int) (yres + 0.5);
  return true;
}
#endif
```

2.1.2. Obtendo a Resolução e Densidade de Pontos da Tela no Web Assembly

Se estivermos executando em Web Assembly, então assumimos estar em um Navegador de Internet. A nossa "janela" será um "canvas" HTML. E nesse caso para obter o tamanho da tela, temos que recorrer à invocação de Javascript. Fazemos isso com ajuda da API do Emscripten:

Seção: Funções da API (continuação):

```
#if defined(__EMSCRIPTEN__)
bool _Wget_screen_resolution(int *resolution_x, int *resolution_y){
  *resolution_x = EM_ASM_INT({
    return window.screen.width * window.devicePixelRatio;
  });
  *resolution_y = EM_ASM_INT({
    return window.screen.height * window.devicePixelRatio;
  });
  return true;
}
#endif
```

Em ambiente WebAssembly, obter a densidade de pixels da tela é algo um tanto impreciso. Os navegadores de Internet costumam armazenar um valor em ponto flutuante como um atributo do objeto representando a janela chamado `devicePixelRatio`. Ele mede a relação entre o tamanho de um pixel para o CSS e um píxel físico. Um valor de 1 tipicamente representa um dispositivo com 96 DPI. O valor pode ser interpretado como um multiplicador sobre o valor base de 96DPI. O Emscripten fornece a função `emscripten_get_device_pixel_ratio` como forma de obter

tal valor de forma mais fácil:

Seção: Funções da API (continuação):

```
#if defined(__EMSCRIPTEN__)
bool _Wget_screen_dpi(int *dpi_x, int *dpi_y){
  *dpi_x = emscripten_get_device_pixel_ratio() * 96.0;
  *dpi_y = *resolution_x;
  return true;
}
#endif
```

2.1.3. Obtendo a Resolução e Densidade de Pontos da Tela no Windows

Finalmente, no Windows, isso é feito simplesmente chamando funções da API que nos dão esta informação. No caso da resolução:

Seção: Funções da API (continuação):

```
#if defined(_WIN32)
bool _Wget_screen_resolution(int *resolution_x, int *resolution_y){
  *resolution_x = GetSystemMetrics(SM_CXSCREEN);
  *resolution_y = GetSystemMetrics(SM_CYSCREEN);
  return true;
}
#endif
```

E no caso da densidade de pontos:

Seção: Funções da API (continuação):

```
#if defined(_WIN32)
bool _Wget_screen_dpi(int *dpi_x, int *dpi_y){
  HMONITOR monitor;
  monitor = MonitorFromWindow(window, MONITOR_DEFAULTTOPRIMARY);
  if(GetDpiForMonitor(monitor, MDT_EFFECTIVE_DPI, dpi_x, dpi_y) != S_OK){
    *dpi_x = *dpi_y = 0;
    return false;
  }
  return true;
}
#endif
```

Na função acima assumimos a existência de uma variável chamada `window` que armazena o identificador da janela atual. Este identificador será definido e inicializado posteriormente para o Windows.

2.2. Criando uma Janela

Agora descreveremos como criar uma janela nos diferentes tipos de ambiente que suportamos: ambiente gráfico X11, Windows e rodando em navegador web com Web Assembly.

2.2.1. Criando uma Janela no X

Os passos para criar uma janela no X11 são:

0. Avisamos a biblioteca X que pode ser que múltiplas threads tentem se comunicar com ele. É raro haver motivos para se fazer isso, mas é útil se preparar só para o caso de acontecer. Essa precisa ser a primeira coisa a ser feita antes de usar outras funções da biblioteca.

1. Abrimos uma conexão com o servidor. Se isso for bem-sucedido, o servidor nos revela várias informações relevantes sobre a tela.

2. Lemos a resolução da tela com a função que definimos na seção 2.1.

3. Enviamos uma nova mensagem para o servidor pedindo que a janela seja criada. Em princípio criaremos uma janela com o máximo de tamanho permitido dada a resolução da tela.

4. Memorizamos o valor inicial da altura e largura da janela.

Estes passos são implementados por meio das seguintes funções e macros:

Seção: X11: Criar Janela:

```
#if defined(__linux__) || defined(BSD)
int screen_resolution_x, screen_resolution_y; // Resolução da tela
/* Passo 0: */
XInitThreads();
/* Passo 1: */
display = XOpenDisplay(NULL);
if(display == NULL){
#if defined(W_DEBUG_WINDOW)
  fprintf(stderr, "ERROR: Failed to connect to X11 server.\n");
#endif
  return false; // Não conseguiu se conectar
}
/* Passo 2: */
_Wget_screen_resolution(&screen_resolution_x, &screen_resolution_y);
/* Passo 3: */
window = XCreateSimpleWindow(display, // Conexão com o X11
                             DefaultRootWindow(display), // Janela-mãe
                             0, 0, // Posição da janela criada
                             screen_resolution_x, // Largura
                             screen_resolution_y, // Altura
                             0, 0, // Espessura e cor da borda
                             0); // Cor padrão da janela
/* Passo 4: */
window_size_x =  screen_resolution_x;
window_size_y =  screen_resolution_y;
#endif
```

Este código assume que temos as seguintes variável declarada para armazenar e memorizar a janela criada:

Seção: Variáveis Locais (continuação):

```
#if defined(__linux__) || defined(BSD)
static Window window;    // Estrutura da janela criada
#endif
```

O código que temos até agora cria a janela. Mas não a desenha na tela. Não desenhar ela na tela automaticamente permite que nós ajustemos atributos da janela antes que ela seja finalmente exibida.

A primeira coisa que precisaremos ajustar é que queremos que a janela seja em tela-cheia por padrão. Faremos isso pedindo para o gerenciador de janelas não interferir na criação da janela, colocando bordas ou tentando limitar seu tamanho que iremos ajustar. Mas faremos isso só se realmente formos iniciar no modo tela-cheia:

Seção: X11: Criar Janela (continuação):

```
#if defined(__linux__) || defined(BSD)
#if !defined(W_WINDOW_NO_FULLSCREEN)
{
  XSetWindowAttributes attributes;
  XMoveWindow(display, window, 0, 0);
  attributes.override_redirect = true;
  XChangeWindowAttributes(display, window, CWOverrideRedirect,
                          &attributes);
}
```

```
#endif
#endif
```

Mas e se estivermos fora do modo de tela cheia e o usuário definiu macros que dizem que o tamanho padrão da janela deve ser diferente de ocupar a tela inteira? Neste caso, precisaremos redimencionar a janela antes de desenhá-la na tela pela primeira vez. As macros que controlarão o tamanho da janela quando não estamos em tela cheia são `W_WINDOW_RESOLUTION_X` e `W_WINDOW_RESOLUTION_Y`. Se elas valerem zero ou menos, isso significa que o tamanho deve ser igual o da resolução da tela. Caso contrário, seu valor representará o tamanho em pixels da janela. Mas isso só se aplica quando não estamos em tela cheia:

Seção: X11: Criar Janela (continuação):

```
#if defined(__linux__) || defined(BSD)
#if defined(W_WINDOW_NO_FULLSCREEN)
{
#if W_WINDOW_SIZE_X > 0
  window_size_x = W_WINDOW_SIZE_X;
#else
  window_size_x = screen_resolution_x;
#endif
#if W_WINDOW_SIZE_Y > 0
  window_size_y = W_WINDOW_SIZE_Y;
#else
  window_size_y = screen_resolution_y;
#endif
  XResizeWindow(display, window, window_size_x, window_size_y);
}
#endif
#endif
```

Vamos também fixar o tamanho da janela para o atual para impedir que ela de alguma forma seja redimencionada:

Seção: X11: Criar Janela (continuação):

```
#if defined(__linux__) || defined(BSD)
{
  XSizeHints hints;
  hints.flags = PMinSize | PMaxSize;
#if defined(W_WINDOW_NO_FULLSCREEN) && W_WINDOW_SIZE_X > 0
  hints.min_width = hints.max_width = W_WINDOW_SIZE_X;
#else
  hints.min_width = hints.max_width = screen_resolution_x;
#endif
#if defined(W_WINDOW_NO_FULLSCREEN) && W_WINDOW_SIZE_Y > 0
  hints.min_height = hints.max_height = W_WINDOW_SIZE_Y;
#else
  hints.min_height = hints.max_height = screen_resolution_y;
#endif
  XSetWMNormalHints(display, window, &hints);
}
#endif
```

O recurso acima requer o seguinte cabeçalho:

Seção: Headers (continuação):

```
#if defined(__linux__) || defined(BSD)
#include <X11/Xutil.h>
```

```
#endif
```

Outra coisa relevante a ajustar é em que tipo de eventos nosso programa quer prestar atenção quando estiver executando. Exemplo de evento que não consideraremos interessante: o usuário move a janela pela tela. Exemplo de evento interessante: o usuário pressiona um botão enquanto nossa janela está ativa.

A lista de eventos que considerearemos importantes o bastante para que nosso programa seja notificado é: janela é criada ou destruída, usuário aperta ou solta botão de teclado, usuário aperta ou solta botão do mouse e usuário move o mouse. Se não pedirmos para sermos informados disso, nenhum evento será informado para nosso programa e ele não saberá quando o usuário interage com ele por meio de mouse e teclado.

Seção: X11: Criar Janela (continuação):

```
#if defined(__linux__) || defined(BSD)
XSelectInput(display, window, StructureNotifyMask | KeyPressMask |
                              KeyReleaseMask | ButtonPressMask |
                              ButtonReleaseMask | PointerMotionMask);
#endif
```

Outra coisa importante é definir o nome da janela que será criada. Geralmetne essa informação é apresentada de alguma forma pelo gerenciador de janelas. Podemos deixar que o usuário escolha isso ajustando a macro `W_WINDOW_NAME`:

Seção: X11: Criar Janela (continuação):

```
#if defined(__linux__) || defined(BSD)
XStoreName(display, window, W_WINDOW_NAME);
#endif
```

Se esta macro não estiver definida, deixamos apenas uma string vazia:

Seção: Define Macros:

```
#if !defined(W_WINDOW_NAME)
#define W_WINDOW_NAME ""
#endif
```

Agora vamos configurar o OpenGL ES. Como isso é suficientemente trabalhoso, colocamos os passos de como fazer isso na próxima sessão:

Seção: X11: Criar Janela (continuação):

```
#if defined(__linux__) || defined(BSD)
```

<Seção a ser Inserida: **X11: Configurar OpenGL ES**>

```
#endif
```

Uma vez que tenhamos ajustado as configurações de nossa janela, podemos enfim começar a desenhar ela. Para isso enviamos uma requisição para o servidor X e ficamos esperando em um laço até que recebamos o evento de que a janela foi criada (já que pedimos para sermos avisados deste evento passando a flag `StructureNotifyMask` para a função `XSelectInput` anteriormente). O código para isso é:

Seção: X11: Criar Janela (continuação):

```
#if defined(__linux__) || defined(BSD)
XMapWindow(display, window);
{
  XEvent e;
  do{
    XNextEvent(display, &e);
  } while(e.type != MapNotify);
}
XSetInputFocus(display, window, RevertToParent, CurrentTime);
#endif
```

2.2.2. Criando uma Janela com Web Assembly

Um dos ambientes mais diferentes nos quais nossa API pode executar será navegadores de Internet após ter o código compilado para Web Assembly. Neste caso, não há janelas verdadeiras, o espaço no qual poderemos desenhar na tela e teremos controle será um "canvas" de HTML. Isso faz com que não tenhamos que nos preocupar com a possibilidade do usuário tentar redimencionar a janela, por exemplo. Mas ainda deveremos permitir ajustes no tamanho do nosso "canvas" além de continuarmos precisando prestar atenção no tamanho da tela.

Aqui iremos manipular a nossa área de desenho combinando duas coisas: a biblioteca SDL, fornecida como interface para realizar ações gráficas pelo ambiente Emscripten e também código Javascript que poderemos executar para ajudar.

Primeiro vamos inserir o cabeçalho do Emscripten com os cabeçalhos SDL e também cabeçalhos adicionais necessários por algumas macros que iremos usar:

Seção: Cabeçalho OpenGL:

```
#if defined(__linux__) || defined(BSD)
#include <X11/X.h>
#endif
#if defined(__EMSCRIPTEN__)
#include <GLES3/gl3.h>
#include <SDL/SDL.h>
#include <emscripten.h>
#endif
```

Agora vamos ler a resolução da tela com a função definida na Subsubseção 2.1.2:

Seção: Web Assembly: Criar Janela:

```
#if defined(__EMSCRIPTEN__)
int screen_resolution_x, screen_resolution_y;
_Wget_screen_resolution(&screen_resolution_x, &screen_resolution_y);
#endif
```

A próxima coisa que temos a fazer é inicializar o sub-sistema de vídeo da biblioteca SDL. Fazer isso é simplesmente chamar a função de inicialização:

Seção: Web Assembly: Criar Janela (continuação):

```
#if defined(__EMSCRIPTEN__)
SDL_Init(SDL_INIT_VIDEO);
#endif
```

Agora vamos efetivamente criar a janela, o que na prática ajusta o tamanho do canvas HTML onde iremos desenhar e o inicializa. O "canvas" HTML deverá ter o tamanho da janela, a menos que exista definida a macro `W_WINDOW_NO_FULLSCREEN` com valores positivos para `W_WINDOW_RESOLUTION_X` e `W_WINDOW_RESOLUTION_Y`, caso em que usaremos esses valores como tamanho. Também iremos nos certificar de que o canvas está realmente visível, pois ele pode ter sido oculto (é o que fazemos com ele se executarmos a função de fechar janela).

Seção: Web Assembly: Criar Janela (continuação):

```
#if defined(__EMSCRIPTEN__)
{
  fullscreen_mode = true;
  double pixel_ratio = EM_ASM_DOUBLE({
    return window.devicePixelRatio;
  });
  window_size_x = EM_ASM_INT({
    return window.innerWidth * window.devicePixelRatio;;
  });
  window_size_y = EM_ASM_INT({
    return window.innerHeight * window.devicePixelRatio;;
```

```
  });
#if defined(W_WINDOW_NO_FULLSCREEN)
  fullscreen_mode = false;
#if defined(W_WINDOW_SIZE_X) && W_Window_Size_X > 0
  window_size_x = W_Window_Size_X;
#endif
#if defined(W_WINDOW_SIZE_Y) && W_Window_Size_Y > 0
  window_size_y = W_Window_Size_Y;
#endif
#endif
  // Ajusta versão do OpenGL
  SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION,
                      W_WINDOW_OPENGL_MAJOR_VERSION);
  SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION,
                      W_WINDOW_OPENGL_MINOR_VERSION);
  // Garante que o canvas estará visível
  EM_ASM(
    var el = document.getElementById("canvas");
    el.style.display = "initial";
  );
  window = SDL_SetVideoMode(window_size_x / pixel_ratio,
                            window_size_y / pixel_ratio, 0, SDL_OPENGL);
  if(fullscreen_mode){
    EM_ASM(
      var el = document.getElementById("canvas");
      el.style.position = "absolute";
      el.style.top = "0px";
      el.style.left = "0px";
      el.style.width = window.innerWidth + "px";
      el.style.height = window.innerHeight + "px";
      el.width = (window.innerWidth * window.devicePixelRatio);
      el.height = (window.innerHeight * window.devicePixelRatio);
    );
  }
  if(window == NULL)
    return false;
  glViewport(0, 0, window_size_x, window_size_y);
}
#endif
```

No código acima, depois de criar a "janela" (que é um "canvas") também especificamos o tamanho de nossa área de exibição, ou "viewport". Na maioria das vezes não é necessário ajustá-lo quando a área em que iremos renderizar é igual à de nossa janela. Contudo, quando o código acima é executado em ambiente móvel, como em celulares onde o pixel lógico é diferente do pixel físico da tela, o valor padrão da tela de exibição pode vir errado. Por isso é importante ajustar manualmente nestes casos.

Outra coisa necessária para o código acima é uma variável estática booleana que mantém se estamos em modo tela-cheia ou não:

Seção: Variáveis Locais (continuação):

```
#if defined(__EMSCRIPTEN__)
static bool fullscreen_mode = false;
#endif
```

A variável é necessária caso estejamos rodando em um navegador porque neste caso, somos nós e não um gerenciador de janelas externo que precisa fazer alguns ajustes na nossa "janela"

para deixá-la em tela cheia. Além disso, também tentamos ajustar nossa janela para a tela-cheia mesmo que o navegador de Internet não nos deixe entrar em modo de tela-cheia: neste caso ainda deixamos a nossa área ficar com o tamanho da tela e ocupar o topo do documento para que o usuário possa ainda entrar em tela-cheia manualmente. Então somente no caso de ambiente Web Assembly, nós memorizamos em uma variável se devemos estar ou não em tela-cheia.

O que chamamos de "janela" neste caso é considerada uma superfície SDL mapeada para um canvas HTML:

Seção: Variáveis Locais (continuação):

```
#if defined(__EMSCRIPTEN__)
static SDL_Surface *window;
#endif
```

2.2.3. Criando uma Janela no Windows

Como nos códigos anteriores, começamos lendo a resolução da tela:

Seção: Windows: Criar Janela:

```
#if defined(_WIN32)
int screen_resolution_x, screen_resolution_y;
_Wget_screen_resolution(&screen_resolution_x, &screen_resolution_y);
#endif
```

A próxima coisa a fazer é definir uma classe para a janela que iremos criar. Primeiro vamos precisar dar um nome arbitrário para ela no formato de uma string qualquer. Iremos chamá-la de "WeaverWindow":

Seção: Variáveis Locais (continuação):

```
#if defined(_WIN32)
static const char *class_name = "WeaverWindow";
#endif
```

Agora precisamos para nossa classe uma função que irá tratar todos os sinais e mensagens enviada para nossa janela. Mensagens são enviadas e devem ser tratadas quando a janela é criada, destruída, redimencionada, exposta na tela, etc. Na dúvida sempre podemos repassar cada mensagem para a função padrão `DefWindowProc`, mas algumas coisas nós mesmos teremos que definir. O formato da função que tratará as mensagens recebidas pela janela é:

Seção: Funções da API (continuação):

```
#if defined(_WIN32)
LRESULT CALLBACK WindowProc(HWND window, UINT msg, WPARAM param1, LPARAM param2){
  switch(msg){
        <Seção a ser Inserida: Windows: Trata Mensagens para Janela>
    default:
      return DefWindowProc(window, msg, param1, param2);
  }
}
#endif
```

Mas em quais casos iremos tratar as mensagens ao invés de apenas passá-las adiante para o `DefWindowProc`? Um dos casos é quando a janela receber uma mensagem para ser fechada:

Seção: Windows: Trata Mensagens para Janela:

```
case WM_DESTROY:
  PostQuitMessage(0);
  return 0;
  break;
```

Agora temos que criar uma classe para a janela que iremos criar. Ela deve apenas ter um nome único que não conflite com nomes padrão usados pelo sistema. Também temos que passar

na criação da classe um identificador do programa que executamos (que obtemos com `GetModuleHandle` e a função que irá lidar com mensagens e sinais recebidos pela janela (no caso, o padrão `DefWindowProc`).

Seção: Windows: Criar Janela (continuação):

```
#if defined(_WIN32)
if(!already_created_a_class){
  ATOM ret;
  WNDCLASS window_class;
  memset(&window_class, 0, sizeof(WNDCLASS));
  window_class.lpfnWndProc = WindowProc;
  window_class.hInstance = GetModuleHandle(NULL);
  window_class.lpszClassName = class_name;
  window_class.hbrBackground = CreateSolidBrush(RGB(0, 0, 0)); // Janela preta
  window_class.hCursor = LoadCursor(NULL, IDC_ARROW); // Cursor normal
  ret = RegisterClass(&window_class);
  if(ret == 0){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Failed to register Window Class. SysError: %d\n",
            GetLastError());
#endif
    return false;
  }
  already_created_a_class = true;
}
#endif
```

Por conveniência usamos a função `memset` para inicializar a estrutura da classe da janela, já que a maior parte de seus elementos pode ser mantida como zero, que é o padrão. Por isso, inserimos o cabeçalho abaixo:

Seção: Cabeçalhos (continuação):

```
#if defined(_WIN32)
#include <string.h>
#endif
```

O código acima presume que temos declarada a seguinte variável que armazena se nossa classe já foi criada:

Seção: Variáveis Locais (continuação):

```
#if defined(_WIN32)
static bool already_created_a_class = false;
#endif
```

Após termos a classe da janela, criamos a janela com o código abaixo:

Seção: Windows: Criar Janela (continuação):

```
#if defined(_WIN32)
{
  RECT size;
  window_size_x = screen_resolution_x;
  window_size_y = screen_resolution_y;
  SystemParametersInfoA(SPI_GETWORKAREA, 0, &size, 0);
  DWORD fullscreen_flag = WS_POPUP;
#if defined(W_WINDOW_NO_FULLSCREEN)
  fullscreen_flag = WS_OVERLAPPED | WS_CAPTION;
  window_size_x = size.left - size.right;
```

```
  window_size_y = size.bottom - size.top;
#if defined(W_WINDOW_SIZE_X) && W_WINDOW_SIZE_X > 0
  window_size_x = W_WINDOW_SIZE_X;
#endif
#if defined(W_WINDOW_SIZE_Y) && W_WINDOW_SIZE_Y > 0
  window_size_y = W_WINDOW_SIZE_Y;
#endif
#endif
  window = CreateWindowEx(0, class_name,
                          W_WINDOW_NAME,
                          fullscreen_flag | WS_VISIBLE,
                          size.left, size.top, window_size_x,
                          window_size_y,
                          NULL, NULL,
                          GetModuleHandle(NULL),
                          NULL);
  if(window == NULL){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Failed creating window. SysCode: %d\n",
            GetLastError());
#endif
    return false;
  }
}
#endif
```

Iremos armazenar o identificador da janela criada na seguinte variável:

Seção: Variáveis Locais (continuação):

```
#if defined(_WIN32)
static HWND window;
#endif
```

Antes de exibir a janela na tela, vamos configurar o OpenGL para funcionar nela:

Seção: Windows: Criar Janela (continuação):

```
#if defined(_WIN32)
            <Seção a ser Inserida: Windows: Configurar OpenGL>
#endif
```

Agora pedimos para que o sistema passe a exibir a janela e esperamos em um laço até o sistema avisar por meio de uma mensagem que a janela foi criada:

Seção: Windows: Criar Janela (continuação):

```
#if defined(_WIN32)
{
  MSG msg;
  ShowWindow(window, SW_NORMAL);
  do{
    GetMessage(&msg, NULL, 0, 0);
  } while(msg.message == WM_CREATE);
}
#endif
```

2.2.4. Definindo a Função de Criação de Janelas para Todo Ambiente

Nas subsubseções anteriores, definimos código para a criação de janela em diferentes tipos de

ambiente. E cada um dos códigos apresentado ficava dentro de macros condicionais para ser executado somente no ambiente correto.Agora podemos unir todos estes trechos de código anteriores em uma só função:

Seção: Funções da API:

```
bool _Wcreate_window(struct _Wkeyboard *keyboard, struct _Wmouse *mouse){
  if(already_have_window == true)
    return false;
                    <Seção a ser Inserida: X11: Criar Janela>
                <Seção a ser Inserida: Web Assembly: Criar Janela>
                  <Seção a ser Inserida: Windows: Criar Janela>
                  <Seção a ser Inserida: Inicialização de Teclado>
  _Wflush_window_input(keyboard, mouse);
  already_have_window = true;
  return true;
}
```

A variável que armazena se a janela já está criada será declarada aqui:

Seção: Variáveis Locais (continuação):

```
static bool already_have_window = false;
```

2.3. Configurando o OpenGL

Nossa API deve suportar pelo menos o OpenG ES 2.0. Na maioria dos ambientes isso não é um problema. Mas no Windows, por exemplo, nós não temos garantias de que o sistema suporta OpenGL ES. Em tais casos, devemos ativar o OpenGL 4, mas iremos suportar não a sua API inteira, mas apenas a funções que estão prsentes no OpenGL ES 2.0.

2.3.1. Configurando OpenGL ES no X11

No X11, a interface por meio da qual programamos usando OpenGL ES se chama EGL e suas funções e macros são declaradas no seguinte cabeçalho:

Seção: Cabeçalho OpenGL (continuação):

```
#if defined(__linux__) || defined(BSD)
#include <EGL/egl.h>
#include <GLES3/gl3.h>
#include <EGL/eglext.h>
#endif
```

Agora precisamos criar uma estrutura que armazena as informações sobre nossa interface gráfica para o OpenGL ES. Assim como com o servidor X, isso é armazenado em uma estrutura chamada `display`. E podemos obter ela à partir do `display` da biblioteca X:

Seção: X11: Configurar OpenGL ES:

```
egl_display = eglGetPlatformDisplay(EGL_PLATFORM_X11_KHR, display,
                                    NULL);
if(egl_display == EGL_NO_DISPLAY){
#if defined(W_DEBUG_WINDOW)
  fprintf(stderr, "ERROR: Could not create EGL display.\n");
#endif
  return false;
}
eglInitialize(egl_display, NULL, NULL);
```

Essa variável é declarada aqui:

Seção: Variáveis Locais (continuação):

```
#if defined(__linux__) || defined(BSD)
```

```
static EGLDisplay *egl_display;
#endif
```

Agora obtemos uma configuração possível para o contexto OpenGL ES a ser criado. Primeiro especificamos uma série de exigências e em seguida obtemos da biblioteca uma configuração possível:

Seção: X11: Configurar OpenGL ES (continuação):

```
{
  bool ret;
  int number_of_configs_returned;
  int requested_attributes[] = {
    // Devemos suportar desenhar em janelas e em texturas:
    EGL_SURFACE_TYPE,  EGL_WINDOW_BIT | EGL_PBUFFER_BIT,
    // Devemos suportar ao menos 1 bit para a cor vermelha:
    EGL_RED_SIZE, 1,
    // Devemos suportar ao menos 1 bit para a cor verde:
    EGL_GREEN_SIZE, 1,
    // Devemos suportar ao menos 1 bit para a cor azul:
    EGL_BLUE_SIZE, 1,
    // Devemos suportar ao menos 1 bit para o canal alfa:
    EGL_ALPHA_SIZE, 1,
    // Devemos suportar ao menos 1 bit para a profundidade:
    EGL_DEPTH_SIZE, 1,
    EGL_NONE
  };
  ret = eglChooseConfig(egl_display, requested_attributes,
                        &egl_config, 1, &number_of_configs_returned);
  if(ret == EGL_FALSE){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Could not create valid EGL config.\n");
#endif
    return false;
  }
}
```

A estrutura que armazena a configuração que usaremos é declarada aqui:

Seção: Variáveis Locais (continuação):

```
#if defined(__linux__) || defined(BSD)
EGLConfig egl_config;
#endif
```

Assim como o EGL precisa de sua própria estrutura de display, ele também precisará de uma estrutura própria para armazenar informações sobre a janela na qual iremos desenhar. Podemos inicializar a estrutura EGL para a janela à partir da estrutura de janela do X:

Seção: X11: Configurar OpenGL ES (continuação):

```
egl_window = eglCreateWindowSurface(egl_display, egl_config, window,
                                    NULL);
if(egl_window == EGL_NO_SURFACE){
#if defined(W_DEBUG_WINDOW)
  fprintf(stderr, "ERROR: Could not create EGL window.\n");
#endif
  return false;
}
```

A janela EGL é declarada aqui:

Seção: Variáveis Locais (continuação):

```
#if defined(__linux__) || defined(BSD)
static EGLSurface egl_window;
#endif
```

Agora criaremos o contexto OpenGL ES. Deixaremos que o usuário escolha qual a versão do OpenGL por meio das macros `W_WINDOW_OPENGL_MAJOR_VERSION` e `W_WINDOW_OPENGL_MINOR_VERSION`. Usando tal informação, o código abaixo cria o contexto:

Seção: X11: Configurar OpenGL ES (continuação):

```
{
  int context_attribs[] = {
    EGL_CONTEXT_MAJOR_VERSION, W_WINDOW_OPENGL_MAJOR_VERSION,
    EGL_CONTEXT_MINOR_VERSION, W_WINDOW_OPENGL_MINOR_VERSION,
    EGL_NONE
  };
  egl_context = eglCreateContext(egl_display, egl_config,
                                 EGL_NO_CONTEXT, context_attribs);
  if(egl_context == EGL_NO_CONTEXT){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Could not create EGL context.\n");
#endif
    return false;
  }
  eglMakeCurrent(egl_display, egl_window, egl_window, egl_context);
}
```

O contexto OpenGL é declarado aqui:

Seção: Variáveis Locais (continuação):

```
#if defined(__linux__) || defined(BSD)
static EGLContext egl_context;
#endif
```

2.3.2. Configurando OpenGL no Web Assembly

Nenhuma configuração adicional é necessária. Nós já suportamos OpenGL porque passamos uma flag que pediu o suporte quando criamos uma "janela" na Subsubseção 2.2.2.

Note que a versção do OpenGL neste caso é o WebGL, mas esta versão é compatível com o OpenGL ES.

2.3.3. Configurando OpenGL no Windows

Para usar o OpenGL no Windows, primeiro precisamos avisamos o compilador das bibliotecas necessárias para não precisar declará-las durante o processo de ligação do programa e em seguida inserimos o cabeçalho padrão e o do OpenGL:

Seção: Cabeçalho OpenGL (continuação):

```
#if defined(_WIN32)
#pragma comment(lib, "Opengl32.lib")
#pragma comment(lib, "Shell32.lib")
#pragma comment(lib, "User32.lib")
#pragma comment(lib, "Gdi32.lib")
#include <windows.h>
#include <GL/gl.h>
#endif
```

Também precisaremos de uma estrutura com informações sobre o dispositivo em que iremos desenhar. No nosso caso, uma janela:

Seção: Windows: Configurar OpenGL:

```
device_context = GetDC(window);
```

Esta estrutura é declarada aqui junto com o contexto OpenGL a ser inicializado:

Seção: Variáveis Locais (continuação):

```
#if defined(_WIN32)
static HGLRC wgl_context;
static HDC device_context;
#endif
```

Além disso, configurar o OpenGL no Windows é uma tarefa um bocado trabahosa. Pra começar, a função que cria contexto OpenGL definida por padrão, a `wglCreateContext` pode criar um contexto muito primitivo, sem suporte à funções mais recentes OpenGL e não há opção para configurá-la com muitos dos parâmetros necessários para usar recursos mais novos. Mas existe definida na prática uma outra função que cria contexto: `wglCreateContextAttribsARB`, a qual permite que façamos coisas básicas como pedir por uma versão específica do OpenGL com suporte à funções mais modernas e a mais parâmetros.

O problema é que a função `wglCreateContextAttribsARB` não faz parte da API padrão e é considerada uma extensão. Então, para podermos criar um contexto OpenGL adequado, precisamos carregar esta função primeiro. Por outro lado, para usar a função que carrega extensões, um contexto OpenGL já deve estar criado.

A forma de resolver isso é primeiro criar um contexto OpenGL básico e primitivo suportado pela API, depois carregar a função de criação de contexto moderno, criar o novo contexto, associar o contexto moderno à uma janela e carregar como extensões as funções que precisamos. Mas o problema é que não é possível carregar mais de um contexto OpenGL por janela. Então, para fazer isso, precisamos usar uma janela descartável e temporária na qual criaremos o contexto temporário e descartável:

Seção: Windows: Configurar OpenGL (continuação):

```
{
    <Seção a ser Inserida: Windows: Criar uma Janela Temporária>
    <Seção a ser Inserida: Windows: Criar um Contexto Temporário>
    <Seção a ser Inserida: Windows: Carregar Funções OpenGL Iniciais>
    <Seção a ser Inserida: Windows: Remover Contexto e Janela Temporários>
}
```

Primeiro vamos criar nossa janela descartável. Como só usaremos ela para inicializar o OpenGL, não há necesidade de criar ela seguindo todas as especificações de nossa janela verdadeira:

Seção: Windows: Criar uma Janela Temporária:

```
HWND dummy_window;
{
  WNDCLASS dummy_window_class;
  memset(&dummy_window_class, 0, sizeof(WNDCLASS));
  dummy_window_class.lpfnWndProc = WindowProc;
  dummy_window_class.hInstance = GetModuleHandle(NULL);
  dummy_window_class.lpszClassName = "DummyWindow";
  // Esta função pode falhar se a classe já está registrada. Isso ocorre
  // quando a função que cria janelas é invocada mais de uma vez. Apenas
  // ignoramos os erros ao invocar a função abaixo:
  RegisterClass(&dummy_window_class);
  SetLastError(0);
  dummy_window = CreateWindowEx(0, dummy_window_class.lpszClassName, "Dummy",
                                0, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                                CW_USEDEFAULT, 0, 0,
```

```
                                    dummy_window_class.hInstance, 0);
  if(dummy_window == NULL){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Failed creating window. SysCode: %d\n",
            GetLastError());
#endif
    return false;
  }
}
```

Agora vamos criar o contexto OpenGL temporário. Primeiro começamos obtendo o contexto de dispositivo e configurando o formato de pixel dele:

Seção: Windows: Criar um Contexto Temporário:

```
HGLRC dummy_context;
HDC dummy_device_context = GetDC(dummy_window);
{
  PIXELFORMATDESCRIPTOR pixel_format;
  int chosen_pixel_format;
  memset(&pixel_format, 0, sizeof(WNDCLASS));
  pixel_format.nSize = sizeof(PIXELFORMATDESCRIPTOR); // Tamanho da estrutura
  pixel_format.nVersion = 1; // Número de versão
  pixel_format.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                         PFD_DOUBLEBUFFER | PFD_DRAW_TO_BITMAP;
  pixel_format.iPixelType = PFD_TYPE_RGBA;
  pixel_format.cColorBits = 24; // 24 bits para profundidade de cor
  pixel_format.cDepthBits = 32; // 32 bits para buffer de profundidade
  pixel_format.iLayerType = PFD_MAIN_PLANE;
  chosen_pixel_format = ChoosePixelFormat(dummy_device_context, &pixel_format);
  if(chosen_pixel_format == 0){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Failed to choose a pixel format. SysError: %d\n",
            GetLastError());
#endif
    return false;
  }
  if(! SetPixelFormat(dummy_device_context, chosen_pixel_format, &pixel_format)){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Failed to set a pixel format. SysError: %d\n",
            GetLastError());
#endif
    return false;
  }
  // ...
```

Após termos configurado o formato de pixel que queremos, podemos obter o contexto OpenGL temporário que queríamos:

Seção: Windows: Criar um Contexto Temporário (continuação):

```
  // ...
  dummy_context = wglCreateContext(dummy_device_context);
  if(dummy_context == NULL){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Failed creating dummy OpenGL context. SysError:
%d\n",
            GetLastError());
```

```
#endif
    return false;
  }
  if(! wglMakeCurrent(dummy_device_context, dummy_context)){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Failed setting dummy OpenGL context. SysError: %d\n",
            GetLastError());
#endif
    return false;
  }
}
```

Agora temos que carregar as funções que queremos. Carregar uma função existente, mas que não está declarada e acessível por ser considerada uma extensão, é feito pela função definida abaixo que usa `wglGetProcAddress` para obter um ponteiro para a função desejada:

Seção: Funções Locais:

```
#if defined(_WIN32)
static void *load_function(const char *name){
  void *ret = wglGetProcAddress(name);
  if(ret == NULL || ret == (void *) -1 || ret == (void *) 0x1 ||
     ret == (void *) 0x2 || ret == (void *) 0x3){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Function '%s' not supported.\n", name);
#endif
    return NULL;
  }
  return ret;
}
#endif
```

Note que como indicado pelo código acima, a função `wglGetProcAddress` na prática pode indicar erro ou falha de carregamento retornando 5 valores diferentes. Embora somente o retorno de `NULL` seja realmente documentado como correto.

As duas funções que precisamos carregar aqui é uma função com mais recusos para escolher um formato de pixel (como o que escolhemos antes, mas com suporte a mais parâmetros) e uma funcção para criar um contexto OpenGL (como o que foi criado, mas também com mais recursos).

Para carregar novas funções, primeiro precisamos declarar ponteiros com a posição de memória onde a nova função carregada será armazenada. No caso das duas novas funções que queremos, declaramos o ponteiro delas no cabeçalho `window.h` com o código abaixo:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern BOOL (__stdcall *wglChoosePixelFormatARB)(HDC, const int *, const FLOAT *,
                                                 UINT, int *, UINT *);
extern HGLRC (*wglCreateContextAttribsARB)(HDC, HGLRC, const int *);
#endif
```

Também colocamos a declaração real no arquivo `window.c`:

Seção: Variáveis Globais:

```
#if defined(_WIN32)
BOOL (__stdcall *wglChoosePixelFormatARB)(HDC, const int *, const FLOAT *, UINT,
                                          int *, UINT *);
HGLRC (*wglCreateContextAttribsARB)(HDC, HGLRC, const int *);
#endif
```

Uma vez que tenhamos a declaração dos ponteiros, podemos inicializá-los carregando para eles

as funções nas quais estamos interessados:

Seção: Windows: Carregar Funções OpenGL Iniciais:

```
wglChoosePixelFormatARB = (BOOL (__stdcall *)(HDC, const int *, const FLOAT *,
                                              UINT, int *, UINT *))
                          load_function("wglChoosePixelFormatARB");
if(wglChoosePixelFormatARB == NULL) return false;
wglCreateContextAttribsARB = (HGLRC (*)(HDC, HGLRC, const int *))
                             load_function("wglCreateContextAttribsARB");
if(wglCreateContextAttribsARB == NULL) return false;
```

E finalmente, após termos carregado as duas funções acima, não temos mais nenhuma necessidade da janela e do contexto temporários:

Seção: Windows: Remover Contexto e Janela Temporários:

```
wglMakeCurrent(dummy_device_context, 0);
wglDeleteContext(dummy_context);
ReleaseDC(dummy_window, dummy_device_context);
DestroyWindow(dummy_window);
```

Agora estamos prontos para escolher o formato de pixel (a configuração do OpenGL) da forma moderna com nossa nova função:

Seção: Windows: Configurar OpenGL (continuação):

```
{
  PIXELFORMATDESCRIPTOR pixel_format_descriptor;
  const int pixel_format_attributes[] = {
    WGL_DRAW_TO_WINDOW_ARB, GL_TRUE,
    WGL_SUPPORT_OPENGL_ARB, GL_TRUE,
    WGL_DOUBLE_BUFFER_ARB, GL_TRUE,
    WGL_ACCELERATION_ARB, WGL_FULL_ACCELERATION_ARB,
    WGL_PIXEL_TYPE_ARB, WGL_TYPE_RGBA_ARB,
    WGL_COLOR_BITS_ARB, 32,
    WGL_DEPTH_BITS_ARB, 24,
    WGL_STENCIL_BITS_ARB, 8,
    0 };
  int pixel_format_index = 0;
  UINT number_of_formats = 0;
  if(!wglChoosePixelFormatARB(device_context, pixel_format_attributes, NULL, 1,
                              &pixel_format_index,
                              (UINT *) (&number_of_formats))){
#if defined(W_DEBUG_WINDOW)
     fprintf(stderr, "ERROR: 'wglChoosePixelFormatARB' failed.\n");
#endif
     return false;
  }
  if(number_of_formats == 0){
#if defined(W_DEBUG_WINDOW)
     fprintf(stderr,
             "ERROR: no pixel format returned by 'wglChoosePixelFormatARB'.\n");
#endif
     return false;
  }
  DescribePixelFormat(device_context, pixel_format_index,
                      sizeof(pixel_format_descriptor), &pixel_format_descriptor);
  if(!SetPixelFormat(device_context, pixel_format_index,
                     &pixel_format_descriptor)){
```

```
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: 'SetPixelFormat' failed.\n");
#endif
    return false;
  }
}
```

Agora vamos especificar que queremos criar um contexto OpenGL cuja versão do OpenGL é definida pelas macros que usamos para escolher a versão. E em seguida usamos nossa função de criação de contexto moderno:

Seção: Windows: Configurar OpenGL (continuação):

```
{
  const int opengl_attributes[] = {
    WGL_CONTEXT_MAJOR_VERSION_ARB, W_WINDOW_OPENGL_MAJOR_VERSION,
    WGL_CONTEXT_MINOR_VERSION_ARB, W_WINDOW_OPENGL_MINOR_VERSION,
    WGL_CONTEXT_FLAGS_ARB, WGL_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB,
    0 };
  wgl_context = wglCreateContextAttribsARB(device_context, 0, opengl_attributes);
  if(wgl_context == NULL){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: 'wglCreateContextAttribsARB' failed.\n");
#endif
    return false;
  }
  if(!wglMakeCurrent(device_context, wgl_context)){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: 'wglMakeCurrent' failed.\n");
#endif
    return false;
  }
}
```

Durante a criação de nosso contexto OpenGL verdadeiro, fizemos uso de uma série de macros que por padrão não estão definidas. Como o uso delas é local, declaramos elas no cabeçalho do `window.c`:

Seção: Cabeçalhos (continuação):

```
#define WGL_TYPE_RGBA_ARB                       0x202B
#define WGL_PIXEL_TYPE_ARB                      0x2013
#define WGL_COLOR_BITS_ARB                      0x2014
#define WGL_DEPTH_BITS_ARB                      0x2022
#define WGL_STENCIL_BITS_ARB                    0x2023
#define WGL_ACCELERATION_ARB                    0x2003
#define WGL_DOUBLE_BUFFER_ARB                   0x2011
#define WGL_CONTEXT_FLAGS_ARB                   0x2094
#define WGL_DRAW_TO_WINDOW_ARB                  0x2001
#define WGL_SUPPORT_OPENGL_ARB                  0x2010
#define WGL_FULL_ACCELERATION_ARB               0x2027
#define WGL_CONTEXT_MAJOR_VERSION_ARB           0x2091
#define WGL_CONTEXT_MINOR_VERSION_ARB           0x2092
#define WGL_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB 0x0002
```

Mas queremos suportar com nossa API as funções presentes no OpenGL ES 2.0. E tais funções também não fazem parte da API padrão oferecida pelo WGL no Windows. O que faremos então é carregar tais funções como extensões. Por exemplo, começando com as funções relacionadas à criação e gerenciamento de shaders, vamos declarar seus ponteiros:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern GLuint (__stdcall *glCreateShader)(GLenum shaderType);
extern void (__stdcall *glShaderSource)(GLuint, GLsizei, const GLchar *const*,
                                        const GLint *);
extern void (__stdcall *glCompileShader)(GLuint);
extern void (__stdcall *glReleaseShaderCompiler)(void);
extern void (__stdcall *glDeleteShader)(GLuint);
#endif
```

Em seguida posicionamos os ponteiros como variáveis globais em nosso arquivo `window.c`:

Seção: Variáveis Globais:

```
#if defined(_WIN32)
GLuint (__stdcall *glCreateShader)(GLenum shaderType);
void (__stdcall *glShaderSource)(GLuint, GLsizei, const GLchar *const*,
                                 const GLint *);
void (__stdcall *glCompileShader)(GLuint);
void (__stdcall *glReleaseShaderCompiler)(void);
void (__stdcall *glDeleteShader)(GLuint);
#endif
```

Em seguida carregamos para cada um dos ponteiros a função correspondente usando a função definida um pouco acima de nossa declaração de ponteiros:

Seção: Windows: Configurar OpenGL (continuação):

```
glCreateShader = (GLuint (__stdcall *)(GLenum)) load_function("glCreateShader");
if(glCreateShader == NULL)
  return false;
glShaderSource = (void (__stdcall *)(GLuint, GLsizei, const GLchar *const*,
                                     const GLint *))
                 load_function("glShaderSource");
if(glShaderSource == NULL)
  return false;
glCompileShader = (void (__stdcall *)(GLuint)) load_function("glCompileShader");
if(glCompileShader == NULL)
  return false;
glReleaseShaderCompiler = (void (__stdcall *)(void))
                          load_function("glReleaseShaderCompiler");
if(glReleaseShaderCompiler == NULL)
  return false;
glDeleteShader = (void (__stdcall *)(GLuint)) load_function("glDeleteShader");
if(glDeleteShader == NULL)
  return false;
```

Quando usamos `glCreateShader`, precisamos passar uma destas macros como argumento para escolher o tipo de shader criado:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_VERTEX_SHADER          0x8B31
#define GL_FRAGMENT_SHADER        0x8B30
#endif
```

O tipo `GLchar` também precisa ser criado no Windows:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
```

```
typedef char  GLchar;
#endif
```

Após compilar um shader, a ação típica desempenhada é checar se a compilação fo bem-sucedida. Isso é feito usando funções que fazem consultas com relação à shaders. Em particular, a função `glGetShaderiv`. Declaramos abaixo o ponteiro das funções relacionadas à consultas a shaders:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern GLboolean (__stdcall *glIsShader)(GLuint);
extern void (__stdcall *glGetShaderiv)(GLuint, GLenum, GLint *);
extern void (__stdcall *glGetAttachedShaders)(GLuint, GLsizei, GLsizei *,
                                              GLuint *);
extern void (__stdcall *glGetShaderInfoLog)(GLuint, GLsizei, GLsizei *, GLchar
*);
extern void (__stdcall *glGetShaderSource)(GLuint, GLsizei, GLsizei *, GLchar *);
extern void (__stdcall *glGetShaderPrecisionFormat)(GLenum, GLenum, GLint *,
                                                    GLint *);
extern void (__stdcall *glGetVertexAttribfv)(GLuint, GLenum, GLfloat *);
extern void (__stdcall *glGetVertexAttribiv)(GLuint, GLenum, GLint *);
extern void (__stdcall *glGetVertexAttribPointerv)(GLuint, GLenum, void **);
extern void (__stdcall *glGetUniformfv)(GLuint, GLint, GLfloat *);
extern void (__stdcall *glGetUniformiv)(GLuint, GLint, GLint *);
#endif
```

E posicionamos os ponteiros aqui::

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
GLboolean (__stdcall *glIsShader)(GLuint);
void (__stdcall *glGetShaderiv)(GLuint, GLenum, GLint *);
void (__stdcall *glGetAttachedShaders)(GLuint, GLsizei, GLsizei *, GLuint *);
void (__stdcall *glGetShaderInfoLog)(GLuint, GLsizei, GLsizei *, GLchar *);
void (__stdcall *glGetShaderSource)(GLuint, GLsizei, GLsizei *, GLchar *);
void (__stdcall *glGetShaderPrecisionFormat)(GLenum, GLenum, GLint *, GLint *);
void (__stdcall *glGetVertexAttribfv)(GLuint, GLenum, GLfloat *);
void (__stdcall *glGetVertexAttribiv)(GLuint, GLenum, GLint *);
void (__stdcall *glGetVertexAttribPointerv)(GLuint, GLenum, void **);
void (__stdcall *glGetUniformfv)(GLuint, GLint, GLfloat *);
void (__stdcall *glGetUniformiv)(GLuint, GLint, GLint *);
#endif
```

Carregamos cada uma das funções a seus respectivos ponteiros com o código:

Seção: Windows: Configurar OpenGL (continuação):

```
glIsShader = (GLboolean (__stdcall *)(GLuint)) load_function("glIsShader");
if(glIsShader == NULL) return false;
glGetShaderiv = (void (__stdcall *)(GLuint, GLenum, GLint *))
                  load_function("glGetShaderiv");
if(glGetShaderiv == NULL) return false;
glGetAttachedShaders = (void (__stdcall *)(GLuint, GLsizei, GLsizei *, GLuint *))
                        load_function("glGetAttachedShaders");
if(glGetAttachedShaders == NULL) return false;
glGetShaderInfoLog = (void (__stdcall *)(GLuint, GLsizei, GLsizei *, GLchar *))
                        load_function("glGetShaderInfoLog");
if(glGetShaderInfoLog == NULL)  return false;
```

```
glGetShaderSource = (void (__stdcall *)(GLuint, GLsizei, GLsizei *, GLchar *))
                        load_function("glGetShaderSource");
if(glGetShaderSource == NULL) return false;
glGetShaderPrecisionFormat = (void (__stdcall *)(GLenum, GLenum, GLint *,
                                                 GLint *))
                                load_function("glGetShaderPrecisionFormat");
if(glGetShaderPrecisionFormat == NULL) return false;
glGetVertexAttribfv = (void (__stdcall *)(GLuint, GLenum, GLfloat *))
                          load_function("glGetVertexAttribfv");
if(glGetVertexAttribfv == NULL) return false;
glGetVertexAttribiv = (void (__stdcall *)(GLuint, GLenum, GLint *))
                         load_function("glGetVertexAttribiv");
if(glGetVertexAttribiv == NULL) return false;
glGetVertexAttribPointerv = (void (__stdcall *)(GLuint, GLenum, void **))
                               load_function("glGetVertexAttribPointerv");
if(glGetVertexAttribPointerv == NULL) return false;
glGetUniformfv = (void (__stdcall *)(GLuint, GLint, GLfloat *))
                     load_function("glGetUniformfv");
if(glGetUniformfv == NULL) return false;
glGetUniformiv = (void (__stdcall *)(GLuint, GLint, GLint *))
                     load_function("glGetUniformiv");
if(glGetUniformiv == NULL) return false;
```

Quando a função `glGetShaderiv` é usada, podemos selecionar qual informação sobre o shader estamos consultando passando uma das macros abaixo:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_SHADER_TYPE          0x8B4F
#define GL_DELETE_STATUS        0x8B80
#define GL_COMPILE_STATUS       0x8B81
#define GL_INFO_LOG_LENGTH      0x8B84
#define GL_SHADER_SOURCE_LENGTH 0x8B88
#endif
```

Quando a função `glGetShaderPrecisionFormat` é usada para consultar a precisão de algum tipo, o tipo a ser consultado é definido passando uma das seguintes macros:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_LOW_FLOAT    0x8DF0
#define GL_MEDIUM_FLOAT 0x8DF1
#define GL_HIGH_FLOAT   0x8DF2
#define GL_LOW_INT      0x8DF3
#define GL_MEDIUM_INT   0x8DF4
#define GL_HIGH_INT     0x8DF5
#endif
```

Quando a função `glGetVertexAttribfv` ou `glGetVertexAttribiv` é usada para obter informações sobre vértices, o tipo de informação desejada é informada passando como argumento uma destas macros abaixo:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING 0x889F
#define GL_VERTEX_ATTRIB_ARRAY_ENABLED        0x8622
#define GL_VERTEX_ATTRIB_ARRAY_SIZE           0x8623
```

```
#define GL_VERTEX_ATTRIB_ARRAY_STRIDE         0x8624
#define GL_VERTEX_ATTRIB_ARRAY_TYPE           0x8625
#define GL_VERTEX_ATTRIB_ARRAY_NORMALIZED     0x886A
#define GL_CURRENT_VERTEX_ATTRIB              0x8626
#endif
```

Já quando usamos a função `glGetVertexzAttribPointerv`, precisamos passar como um de seus argumentos a macro abaixo:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_VERTEX_ATTRIB_ARRAY_POINTER 0x8645
#endif
```

Vamos definir também esta macro que serve para consultar qual implementação do GLSL temos:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_SHADING_LANGUAGE_VERSION 0x8B8C
#endif
```

Uma vez que criamos e compilamos shaders, geralmente é desejado criar um programa, ligar os shaders a ele e passar a usá-lo. Para permitir isso, vamos declarar o ponteiro para cada uma das funções responsáveis por lidar com programas:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern GLuint (__stdcall *glCreateProgram)(void);
extern void (__stdcall *glAttachShader)(GLuint, GLuint);
extern void (__stdcall *glDetachShader)(GLuint, GLuint);
extern void (__stdcall *glLinkProgram)(GLuint);
extern void (__stdcall *glUseProgram)(GLuint);
extern void (__stdcall *glDeleteProgram)(GLuint);
#endif
```

Após declarar, os ponteiros são efetivamente posicionados aqui:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
GLuint (__stdcall *glCreateProgram)(void);
void (__stdcall *glAttachShader)(GLuint, GLuint);
void (__stdcall *glDetachShader)(GLuint, GLuint);
void (__stdcall *glLinkProgram)(GLuint);
void (__stdcall *glUseProgram)(GLuint);
void (__stdcall *glDeleteProgram)(GLuint);
#endif
```

E inicializamos os ponteiros com as funções adequadas:

Seção: Windows: Configurar OpenGL (continuação):

```
glCreateProgram = (GLuint (__stdcall *)(void)) load_function("glCreateProgram");
if(glCreateProgram == NULL) return false;
glAttachShader = (void (__stdcall *)(GLuint, GLuint))
                  load_function("glAttachShader");
if(glAttachShader == NULL) return false;
glDetachShader = (void (__stdcall *)(GLuint, GLuint))
                 load_function("glDetachShader");
if(glDetachShader == NULL) return false;
```

```
glLinkProgram = (void (__stdcall *)(GLuint)) load_function("glLinkProgram");
if(glLinkProgram == NULL) return false;
glUseProgram = (void (__stdcall *)(GLuint)) load_function("glUseProgram");
if(glUseProgram == NULL) return false;
glDeleteProgram = (void (__stdcall *)(GLuint)) load_function("glDeleteProgram");
if(glDeleteProgram == NULL) return false;
```

Terminada a geração de um programa GLSL, geralmente o usuário irá checar se deu tudo certo na criação do programa. E para isso é importante prepararmos as funções que fazem consultas a programas:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern GLboolean (__stdcall *glIsProgram)(GLuint);
extern void (__stdcall *glGetProgramiv)(GLuint, GLenum, GLint *);
extern void (__stdcall *glGetProgramInfoLog)(GLuint, GLsizei, GLsizei *,
                                             GLchar *);
extern void (__stdcall *glValidadeProgram)(GLuint);
#endif
```

Posicionamos os ponteiros reais aqui:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
GLboolean (__stdcall *glIsProgram)(GLuint);
void (__stdcall *glGetProgramiv)(GLuint, GLenum, GLint *);
void (__stdcall *glGetProgramInfoLog)(GLuint, GLsizei, GLsizei *, GLchar *);
void (__stdcall *glValidadeProgram)(GLuint);
#endif
```

E os inicializamos com as funções reais:

Seção: Windows: Configurar OpenGL (continuação):

```
glIsProgram = (GLboolean (__stdcall *)(GLuint)) load_function("glIsProgram");
if(glIsProgram == NULL) return false;
glGetProgramiv = (void (__stdcall *)(GLuint, GLenum, GLint *))
                    load_function("glGetProgramiv");
if(glGetProgramiv == NULL) return false;
glGetProgramInfoLog = (void (__stdcall *)(GLuint, GLsizei, GLsizei *, GLchar *))
                          load_function("glGetProgramInfoLog");
if(glGetProgramInfoLog == NULL) return false;
glValidadeProgram = (void (__stdcall *)(GLuint))
                     load_function("glValidateProgram");
if(glValidadeProgram == NULL) return false;
```

Quando usamos `glGetProgramiv` para obter informação sobre um programa, passamos como argumento uma destas macros a seguir para selecionar qual informação queremos saber:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_DELETE_STATUS               0x8B80
#define GL_LINK_STATUS                 0x8B82
#define GL_VALIDATE_STATUS             0x8B83
#define GL_INFO_LOG_LENGTH             0x8B84
#define GL_ATTACHED_SHADERS            0x8B85
#define GL_ACTIVE_ATTRIBUTES           0x8B89
#define GL_ACTIVE_ATTRIBUTE_MAX_LENGTH 0x8B8A
#define GL_ACTIVE_UNIFORMS             0x8B86
```

```
#define GL_ACTIVE_UNIFORM_MAX_LENGTH    0x8B87
#endif
```

Já para obter e escolher aributos de vértices dentro de um shader, usaremos funções que serão ligadas aos seguintes ponteiros:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern void (__stdcall *glGetActiveAttrib)(GLuint, GLuint, GLsizei, GLsizei *,
                                           GLint *, GLenum *, GLchar *);
extern GLint (__stdcall *glGetAttribLocation)(GLuint, const GLchar *);
extern void (__stdcall *glBindAttribLocation)(GLuint, GLuint, const GLchar *);
#endif
```

Que serão posicionados aqui:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
void (__stdcall *glGetActiveAttrib)(GLuint, GLuint, GLsizei, GLsizei *, GLint *,
                                    GLenum *, GLchar *);
GLint (__stdcall *glGetAttribLocation)(GLuint, const GLchar *);
void (__stdcall *glBindAttribLocation)(GLuint, GLuint, const GLchar *);
#endif
```

E eles são inicializados aqui:

Seção: Windows: Configurar OpenGL (continuação):

```
glGetActiveAttrib = (void (__stdcall *)(GLuint, GLuint, GLsizei, GLsizei *,
                                        GLint *, GLenum *, GLchar *))
                    load_function("glGetActiveAttrib");
if(glGetActiveAttrib == NULL) return false;
glGetAttribLocation = (GLint (__stdcall *)(GLuint, const GLchar *))
                          load_function("glGetAttribLocation");
if(glGetAttribLocation == NULL) return false;
glBindAttribLocation = (void (__stdcall *)(GLuint, GLuint, const GLchar *))
                             load_function("glBindAttribLocation");
if(glBindAttribLocation == NULL) return false;
```

O tipo de um atributo de vértice, que é retornado por `glGetActiveAttrib` pode ser:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_FLOAT      0x1406
#define GL_FLOAT_VEC2 0x8B50
#define GL_FLOAT_VEC3 0x8B51
#define GL_FLOAT_VEC4 0x8B52
#define GL_FLOAT_MAT2 0x8B5A
#define GL_FLOAT_MAT3 0x8B5B
#define GL_FLOAT_MAT4 0x8B5C
#endif
```

E finalmente, as últimas funções relacionadas aos shaders são as responsáveis por lidar com variáveis uniformes:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern GLint (__stdcall *glGetUniformLocation)(GLuint, const GLchar *);
extern void (__stdcall *glGetActiveUniform)(GLuint, GLuint, GLsizei, GLsizei *,
                                            GLint *, GLenum *, GLchar *);
```

```
extern void (__stdcall *glUniform1f)(GLint, GLfloat);
extern void (__stdcall *glUniform2f)(GLint, GLfloat, GLfloat);
extern void (__stdcall *glUniform3f)(GLint, GLfloat, GLfloat, GLfloat);
extern void (__stdcall *glUniform4f)(GLint, GLfloat, GLfloat, GLfloat, GLfloat);
extern void (__stdcall *glUniform1i)(GLint, GLint);
extern void (__stdcall *glUniform2i)(GLint, GLint, GLint);
extern void (__stdcall *glUniform3i)(GLint, GLint, GLint, GLint);
extern void (__stdcall *glUniform4i)(GLint, GLint, GLint, GLint, GLint);
extern void (__stdcall *glUniform1fv)(GLint, GLsizei, const GLfloat *);
extern void (__stdcall *glUniform2fv)(GLint, GLsizei, const GLfloat *);
extern void (__stdcall *glUniform3fv)(GLint, GLsizei, const GLfloat *);
extern void (__stdcall *glUniform4fv)(GLint, GLsizei, const GLfloat *);
extern void (__stdcall *glUniform1iv)(GLint, GLsizei, const GLint *);
extern void (__stdcall *glUniform2iv)(GLint, GLsizei, const GLint *);
extern void (__stdcall *glUniform3iv)(GLint, GLsizei, const GLint *);
extern void (__stdcall *glUniform4iv)(GLint, GLsizei, const GLint *);
extern void (__stdcall *glUniformMatrix2fv)(GLint, GLsizei, GLboolean,
                                            const GLfloat *);
extern void (__stdcall *glUniformMatrix3fv)(GLint, GLsizei, GLboolean,
                                            const GLfloat *);
extern void (__stdcall *glUniformMatrix4fv)(GLint, GLsizei, GLboolean,
                                            const GLfloat *);
#endif
```

Estes 21 ponteiros para funções são posicionados aqui:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
GLint (__stdcall *glGetUniformLocation)(GLuint, const GLchar *);
void (__stdcall *glGetActiveUniform)(GLuint, GLuint, GLsizei, GLsizei *, GLint *,
                                     GLenum *, GLchar *);
void (__stdcall *glUniform1f)(GLint, GLfloat);
void (__stdcall *glUniform2f)(GLint, GLfloat, GLfloat);
void (__stdcall *glUniform3f)(GLint, GLfloat, GLfloat, GLfloat);
void (__stdcall *glUniform4f)(GLint, GLfloat, GLfloat, GLfloat, GLfloat);
void (__stdcall *glUniform1i)(GLint, GLint);
void (__stdcall *glUniform2i)(GLint, GLint, GLint);
void (__stdcall *glUniform3i)(GLint, GLint, GLint, GLint);
void (__stdcall *glUniform4i)(GLint, GLint, GLint, GLint, GLint);
void (__stdcall *glUniform1fv)(GLint, GLsizei, const GLfloat *);
void (__stdcall *glUniform2fv)(GLint, GLsizei, const GLfloat *);
void (__stdcall *glUniform3fv)(GLint, GLsizei, const GLfloat *);
void (__stdcall *glUniform4fv)(GLint, GLsizei, const GLfloat *);
void (__stdcall *glUniform1iv)(GLint, GLsizei, const GLint *);
void (__stdcall *glUniform2iv)(GLint, GLsizei, const GLint *);
void (__stdcall *glUniform3iv)(GLint, GLsizei, const GLint *);
void (__stdcall *glUniform4iv)(GLint, GLsizei, const GLint *);
void (__stdcall *glUniformMatrix2fv)(GLint, GLsizei, GLboolean, const GLfloat *);
void (__stdcall *glUniformMatrix3fv)(GLint, GLsizei, GLboolean, const GLfloat *);
void (__stdcall *glUniformMatrix4fv)(GLint, GLsizei, GLboolean, const GLfloat *);
#endif
```

E agora temos que inicializar todos estes ponteiros com suas funções:

Seção: Windows: Configurar OpenGL (continuação):

```
glGetUniformLocation = (GLint (__stdcall *)(GLuint, const GLchar *))
                            load_function("glGetUniformLocation");
if(glGetUniformLocation == NULL) return false;
glGetActiveUniform = (void (__stdcall *)(GLuint, GLuint, GLsizei, GLsizei *,
                                         GLint *, GLenum *, GLchar *))
                     load_function("glGetActiveUniform");
if(glGetActiveUniform == NULL) return false;
glUniform1f = (void (__stdcall *)(GLint, GLfloat)) load_function("glUniform1f");
if(glUniform1f == NULL) return false;
glUniform2f = (void (__stdcall *)(GLint, GLfloat, GLfloat))
               load_function("glUniform2f");
if(glUniform2f == NULL) return false;
glUniform3f = (void (__stdcall *)(GLint, GLfloat, GLfloat, GLfloat))
                        load_function("glUniform3f");
if(glUniform3f == NULL) return false;
glUniform4f = (void (__stdcall *)(GLint, GLfloat, GLfloat, GLfloat, GLfloat))
                        load_function("glUniform4f");
if(glUniform4f == NULL) return false;
glUniform1i = (void (__stdcall *)(GLint, GLint)) load_function("glUniform1i");
if(glUniform1i == NULL) return false;
glUniform2i = (void (__stdcall *)(GLint, GLint, GLint))
load_function("glUniform2i");
if(glUniform2i == NULL) return false;
glUniform3i = (void (__stdcall *)(GLint, GLint, GLint, GLint))
               load_function("glUniform3i");
if(glUniform3i == NULL) return false;
glUniform4i = (void (__stdcall *)(GLint, GLint, GLint, GLint, GLint))
                 load_function("glUniform4i");
if(glUniform4i == NULL) return false;
glUniform1fv = (void (__stdcall *)(GLint, GLsizei, const GLfloat *))
                 load_function("glUniform1fv");
if(glUniform1fv == NULL) return false;
glUniform2fv = (void (__stdcall *)(GLint, GLsizei, const GLfloat *))
                 load_function("glUniform2fv");
if(glUniform2fv == NULL) return false;
glUniform3fv = (void (__stdcall *)(GLint, GLsizei, const GLfloat *))
                 load_function("glUniform3fv");
if(glUniform3fv == NULL) return false;
glUniform4fv = (void (__stdcall *)(GLint, GLsizei, const GLfloat *))
                 load_function("glUniform4fv");
if(glUniform4fv == NULL) return false;
glUniform1iv = (void (__stdcall *)(GLint, GLsizei, const GLint *))
                 load_function("glUniform1iv");
if(glUniform1iv == NULL) return false;
glUniform2iv = (void (__stdcall *)(GLint, GLsizei, const GLint *))
                 load_function("glUniform2iv");
if(glUniform2iv == NULL) return false;
glUniform3iv = (void (__stdcall *)(GLint, GLsizei, const GLint *))
                 load_function("glUniform3iv");
if(glUniform3iv == NULL) return false;
glUniform4iv = (void (__stdcall *)(GLint, GLsizei, const GLint *))
                 load_function("glUniform4iv");
if(glUniform4iv == NULL) return false;
```

```
glUniformMatrix2fv = (void (__stdcall *)(GLint, GLsizei, GLboolean,
                                         const GLfloat *))
                     load_function("glUniformMatrix2fv");
if(glUniformMatrix2fv == NULL) return false;
glUniformMatrix3fv = (void (__stdcall *)(GLint, GLsizei, GLboolean,
                                         const GLfloat *))
                     load_function("glUniformMatrix3fv");
if(glUniformMatrix3fv == NULL) return false;
glUniformMatrix4fv = (void (__stdcall *)(GLint, GLsizei, GLboolean,
                                         const GLfloat *))
                     load_function("glUniformMatrix4fv");
if(glUniformMatrix4fv == NULL) return false;
```

As variávei uniformes podem ter o mesmo tipo que atributos de vértice, mas podem ter também alguns destes novos tipos:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_INT         0x1404
#define GL_INT_VEC2    0x8B53
#define GL_INT_VEC3    0x8B54
#define GL_INT_VEC4    0x8B55
#define GL_BOOL        0x8B56
#define GL_BOOL_VEC2   0x8B57
#define GL_BOOL_VEC3   0x8B58
#define GL_BOOL_VEC4   0x8B59
#define GL_SAMPLER_2D  0x8B5E
#define GL_SAMPER_CUBE 0x8B60
#endif
```

A última coisa que precisamos para renderizar imagens simples é enviar vértices para a placa de vídeo. Uma das formas de fazer isso é enviar ponteiro para os vértices na placa de vídeo ao invés de enviá-los diretamente para acesso mais rápido. Essa opção pode ser feita com as funções abaixo:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern void (__stdcall *glVertexAttrib1f)(GLuint, GLfloat);
extern void (__stdcall *glVertexAttrib2f)(GLuint, GLfloat, GLfloat);
extern void (__stdcall *glVertexAttrib3f)(GLuint, GLfloat, GLfloat, GLfloat);
extern void (__stdcall *glVertexAttrib4f)(GLuint, GLfloat, GLfloat, GLfloat,
                                          GLfloat);
extern void (__stdcall *glVertexAttrib1fv)(GLuint, GLfloat *);
extern void (__stdcall *glVertexAttrib2fv)(GLuint, GLfloat *);
extern void (__stdcall *glVertexAttrib3fv)(GLuint, GLfloat *);
extern void (__stdcall *glVertexAttrib4fv)(GLuint, GLfloat *);
extern void (__stdcall *glVertexAttribPointer)(GLuint, GLint, GLenum, GLboolean,
                                               GLsizei, const void *);
extern void (__stdcall *glEnableVertexAttribArray)(GLuint);
extern void (__stdcall *glDisableVertexAttribArray)(GLuint);
#endif
```

Posicionamos os ponteiros como variáveis globais:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
void (__stdcall *glVertexAttrib1f)(GLuint, GLfloat);
```

```
void (__stdcall *glVertexAttrib2f)(GLuint, GLfloat, GLfloat);
void (__stdcall *glVertexAttrib3f)(GLuint, GLfloat, GLfloat, GLfloat);
void (__stdcall *glVertexAttrib4f)(GLuint, GLfloat, GLfloat, GLfloat, GLfloat);
void (__stdcall *glVertexAttrib1fv)(GLuint, GLfloat *);
void (__stdcall *glVertexAttrib2fv)(GLuint, GLfloat *);
void (__stdcall *glVertexAttrib3fv)(GLuint, GLfloat *);
void (__stdcall *glVertexAttrib4fv)(GLuint, GLfloat *);
void (__stdcall *glVertexAttribPointer)(GLuint, GLint, GLenum, GLboolean,
                                        GLsizei, const void *);
void (__stdcall *glEnableVertexAttribArray)(GLuint);
void (__stdcall *glDisableVertexAttribArray)(GLuint);
#endif
```

E as inicializamos:

Seção: Windows: Configurar OpenGL (continuação):

```
glVertexAttrib1f = (void (__stdcall *)(GLuint, GLfloat))
                     load_function("glVertexAttrib1f");
if(glVertexAttrib1f == NULL) return false;
glVertexAttrib2f = (void (__stdcall *)(GLuint, GLfloat, GLfloat))
                      load_function("glVertexAttrib2f");
if(glVertexAttrib2f == NULL) return false;
glVertexAttrib3f = (void (__stdcall *)(GLuint, GLfloat, GLfloat, GLfloat))
                      load_function("glVertexAttrib3f");
if(glVertexAttrib3f == NULL) return false;
glVertexAttrib4f = (void (__stdcall *)(GLuint, GLfloat, GLfloat, GLfloat,
                                       GLfloat))
                      load_function("glVertexAttrib4f");
if(glVertexAttrib4f == NULL) return false;
glVertexAttrib1fv = (void (__stdcall *)(GLuint, GLfloat *))
                       load_function("glVertexAttrib1fv");
if(glVertexAttrib1fv == NULL) return false;
glVertexAttrib2fv = (void (__stdcall *)(GLuint, GLfloat *))
                       load_function("glVertexAttrib2fv");
if(glVertexAttrib2fv == NULL) return false;
glVertexAttrib3fv = (void (__stdcall *)(GLuint, GLfloat *))
                       load_function("glVertexAttrib3fv");
if(glVertexAttrib3fv == NULL) return false;
glVertexAttrib4fv = (void (__stdcall *)(GLuint, GLfloat *))
                       load_function("glVertexAttrib4fv");
if(glVertexAttrib4fv == NULL) return false;
glVertexAttribPointer = (void (__stdcall *)(GLuint, GLint, GLenum, GLboolean,
                         GLsizei, const void *))
                              load_function("glVertexAttribPointer");
if(glVertexAttribPointer == NULL) return false;
glEnableVertexAttribArray = (void (__stdcall *)(GLuint))
                               load_function("glEnableVertexAttribArray");
if(glEnableVertexAttribArray == NULL) return false;
glDisableVertexAttribArray = (void (__stdcall *)(GLuint))
                                load_function("glDisableVertexAttribArray");
if(glDisableVertexAttribArray == NULL) return false;
```

Os atributos de vétices podem ter tipos específicos. Além de tipos que já foram definidos (`GL_FLOAT`), é necessário definir também:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_FIXED          0x140C
#endif
```

Agora definiremos os ponteiros para as funções relacionadas a objetos do tipo buffer. Os ponteiros são:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern void (__stdcall *glGenBuffers)(GLsizei, GLuint *);
extern void (__stdcall *glDeleteBuffers)(GLsizei, const GLuint *);
extern void (__stdcall *glBindBuffer)(GLenum, GLuint);
extern void (__stdcall *glBufferData)(GLenum, GLsizeiptr, const void *, GLenum);
extern void (__stdcall *glBufferSubData)(GLenum, GLintptr, GLsizeiptr,
                                         const void *);
extern void (__stdcall *glIsBuffer)(GLuint);
extern void (__stdcall *glGetBufferParameteriv)(GLenum, GLenum, GLint *);
#endif
```

E a declaração:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
void (__stdcall *glGenBuffers)(GLsizei, GLuint *);
void (__stdcall *glDeleteBuffers)(GLsizei, const GLuint *);
void (__stdcall *glBindBuffer)(GLenum, GLuint);
void (__stdcall *glBufferData)(GLenum, GLsizeiptr, const void *, GLenum);
void (__stdcall *glBufferSubData)(GLenum, GLintptr, GLsizeiptr, const void *);
void (__stdcall *glIsBuffer)(GLuint);
void (__stdcall *glGetBufferParameteriv)(GLenum, GLenum, GLint *);
#endif
```

A inicialização:

Seção: Windows: Configurar OpenGL (continuação):

```
glGenBuffers = (void (__stdcall *)(GLsizei, GLuint *))
                load_function("glGenBuffers");
if(glGenBuffers == NULL) return false;
glDeleteBuffers = (void (__stdcall *)(GLsizei, const GLuint *))
                   load_function("glDeleteBuffers");
if(glDeleteBuffers == NULL) return false;
glBindBuffer = (void (__stdcall *)(GLenum, GLuint))load_function("glBindBuffer");
if(glBindBuffer == NULL) return false;
glBufferData = (void (__stdcall *)(GLenum, GLsizeiptr, const void *, GLenum))
                load_function("glBufferData");
if(glBufferData == NULL) return false;
glBufferSubData = (void (__stdcall *)(GLenum, GLintptr, GLsizeiptr, const void
*))
                   load_function("glBufferSubData");
if(glBufferSubData == NULL) return false;
glIsBuffer = (void (__stdcall *)(GLuint)) load_function("glIsBuffer");
if(glIsBuffer == NULL) return false;
glGetBufferParameteriv = (void (__stdcall *)(GLenum, GLenum, GLint *))
                          load_function("glGetBufferParameteriv");
if(glGetBufferParameteriv == NULL) return false;
```

Quando escolhemos ativar um buffer com `glBindBuffer`, ele pode ser dos seguintes tipos:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_ARRAY_BUFFER         0x8892
#define GL_ELEMENT_ARRAY_BUFFER 0x8893
#endif
```

Quando passamos dados em um buffer com `glBufferData`, podemos escolher dentre os seguintes modos de uso para os dados:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_STATIC_DRAW  0x88E4
#define GL_STREAM_DRAW  0x88E0
#define GL_DYNAMIC_DRAW 0x88E8
#endif
```

Quando escolhemos pedir informações sobre um buffer com `glGetBufferParameteriv`, a informação pedida pode ser:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_BUFFER_SIZE  0x8764
#define GL_BUFFER_USAGE 0x8765
#endif
```

E as novas funções requerem também que definamos os tipos de dados abaixo que devem ser grandes o bastante para armazenar ponteiros, mesmo não necessariamente sendo ponteiros. A diferença entre eles é somente ter ou não ter sinal.

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
// Isso inclui um signed size_t:
#include <BaseTsd.h>
typedef size_t GLsizeiptr;
typedef SSIZE_T GLintptr;
#endif
```

As duas funções da API referentes à janela de exibição (“viewport”) e recortte (“clipping”) são:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern void (__stdcall *glDepthRangef)(GLclampf, GLclampf);
#endif
```

Suas declarações:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
void (__stdcall *glDepthRangef)(GLclampf, GLclampf);
#endif
```

E inicialização:

Seção: Windows: Configurar OpenGL (continuação):

```
glDepthRangef = (void (__stdcall *)(GLclampf, GLclampf))
                    load_function("glDepthRangef");
if(glDepthRangef == NULL) return false;
```

O tipo `GLclampf` representa um número em ponto flutuante que deve estar no intervalo entre

0 e 1. Podemos representá-los como números em ponto flutuante mesmo:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
typedef float GLclampf;
#endif
```

Agora vamos preparar as funções do OpenGL relacionadas à textura:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern void (__stdcall *glActiveTexture)(GLenum);
extern void (__stdcall *glCompressedTexImage2D)(GLenum, int, GLenum, GLsizei,
                                                GLsizei, int, GLsizei, void *);
extern void (__stdcall *glCompressedTexSubImage2D)(GLenum, int, int, int,
GLsizei,
                                                   GLsizei, GLenum, GLsizei,
                                                   void *);
extern void (__stdcall *glGenerateMipmap)(GLenum);
#endif
```

E as declaramos no arquivo `window.c`:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
void (__stdcall *glActiveTexture)(GLenum); // TEXTURE0..TEXTURE1...
void (__stdcall *glCompressedTexImage2D)(GLenum, int, GLenum, GLsizei,
                                         GLsizei, int, GLsizei, void *);
void (__stdcall *glCompressedTexSubImage2D)(GLenum, int, int, int, GLsizei,
                                            GLsizei, GLenum, GLsizei,
                                            void *);
void (__stdcall *glGenerateMipmap)(GLenum);
#endif
```

Vamos agora inizializar estas 17 funções para que possam ser usadas:

Seção: Windows: Configurar OpenGL (continuação):

```
glActiveTexture = (void (__stdcall *)(GLenum)) load_function("glActiveTexture");
if(glActiveTexture == NULL) return false;
glCompressedTexImage2D = (void (__stdcall *)(GLenum, int, GLenum, GLsizei,
                          GLsizei, int, GLsizei, void *))
                               load_function("glCompressedTexImage2D");
if(glCompressedTexImage2D == NULL) return false;
glCompressedTexSubImage2D = (void (__stdcall *)(GLenum, int, int, int, GLsizei,
                            GLsizei, GLenum, GLsizei, void *))
                                 load_function("glCompressedTexSubImage2D");
if(glCompressedTexSubImage2D == NULL) return false;
glGenerateMipmap = (void (__stdcall *)(GLenum))
load_function("glGenerateMipmap");
if(glGenerateMipmap == NULL) return false;
```

As funções acima também requerem que as seguintes macros sejam definidas para serem usadas nas enumerações:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_RGB                            0x1907
#define GL_RGBA                           0x1908
```

```
#define GL_ALPHA                          0x1906
#define GL_TEXTURE0                       0x84C0
#define GL_TEXTURE1                       0x84C1
#define GL_TEXTURE2                       0x84C2
#define GL_TEXTURE3                       0x84C3
#define GL_TEXTURE4                       0x84C4
#define GL_TEXTURE5                       0x84C5
#define GL_TEXTURE6                       0x84C6
#define GL_TEXTURE7                       0x84C7
#define GL_TEXTURE8                       0x84C8
#define GL_TEXTURE9                       0x84C9
#define GL_TEXTURE10                      0x84CA
#define GL_TEXTURE11                      0x84CB
#define GL_TEXTURE12                      0x84CC
#define GL_TEXTURE13                      0x84CD
#define GL_TEXTURE14                      0x84CE
#define GL_TEXTURE15                      0x84CF
#define GL_TEXTURE16                      0x84D0
#define GL_TEXTURE17                      0x84D1
#define GL_TEXTURE18                      0x84D2
#define GL_TEXTURE19                      0x84D3
#define GL_TEXTURE20                      0x84D4
#define GL_TEXTURE21                      0x84D5
#define GL_TEXTURE22                      0x84D6
#define GL_TEXTURE23                      0x84D7
#define GL_TEXTURE24                      0x84D8
#define GL_TEXTURE25                      0x84D9
#define GL_TEXTURE26                      0x84DA
#define GL_TEXTURE27                      0x84DB
#define GL_TEXTURE28                      0x84DC
#define GL_TEXTURE29                      0x84DD
#define GL_TEXTURE30                      0x84DE
#define GL_TEXTURE31                      0x84DF
#define GL_LUMINANCE                      0x1909
#define GL_TEXTURE_2D                     0x0DE1
#define GL_UNSIGNED_BYTE                  0x1401
#define GL_TEXTURE_WRAP_S                 0x2802
#define GL_TEXTURE_WRAP_T                 0x2803
#define GL_LUMINANCE_ALPHA                0x190A
#define GL_TEXTURE_MAG_FILTER             0x2800
#define GL_TEXTURE_MIN_FILTER             0x2801
#define GL_UNSIGNED_SHORT_5_6_5           0x8363
#define GL_UNSIGNED_SHORT_4_4_4_4         0x8033
#define GL_UNSIGNED_SHORT_5_5_5_1         0x8034
#define GL_MAX_TEXTURE_IMAGE_UNITS        0x8872
#define GL_TEXTURE_CUBE_MAP_POSITIVE_X    0x8515
#define GL_TEXTURE_CUBE_MAP_POSITIVE_Y    0x8517
#define GL_TEXTURE_CUBE_MAP_POSITIVE_Z    0x8519
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_X    0x8516
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_Y    0x8518
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_Z    0x851A
#define GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS 0x8B4C
#define GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS 0x8B4D
```

```
#endif
```

As funções OpenGL ES por fragmento são uma série de testes e operações que podem modificar a cor de um pixel antes que ele seja efetivamente desenhado em uma superfíce. Geralmente. se um dos testes falhar, o pixel será descarado. As funções relacionadas a isso são:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern void (__stdcall *glSampleCoverage)(GLclampf, bool);
extern void (__stdcall *glStencilFuncSeparate)(GLenum, GLenum, int, unsigned
int);
extern void (__stdcall *glStencilOpSeparate)(GLenum, GLenum, GLenum, GLenum);
extern void (__stdcall *glBlendEquation)(GLenum);
extern void (__stdcall *glBlendEquationSeparate)(GLenum, GLenum);
extern void (__stdcall *glBlendFuncSeparate)(GLenum, GLenum);
extern void (__stdcall *glBlendColor)(GLclampf, GLclampf, GLclampf, GLclampf);
#endif
```

E a declaração destas funções no `window.c`:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
void (__stdcall *glSampleCoverage)(GLclampf, bool);
void (__stdcall *glStencilFuncSeparate)(GLenum, GLenum, int, unsigned int);
void (__stdcall *glStencilOpSeparate)(GLenum, GLenum, GLenum, GLenum);
void (__stdcall *glBlendEquation)(GLenum);
void (__stdcall *glBlendEquationSeparate)(GLenum, GLenum);
void (__stdcall *glBlendFuncSeparate)(GLenum, GLenum);
void (__stdcall *glBlendColor)(GLclampf, GLclampf, GLclampf, GLclampf);
#endif
```

E a inicialização:

Seção: Windows: Configurar OpenGL (continuação):

```
glSampleCoverage = (void (__stdcall *)(GLclampf, bool))
                        load_function("glSampleCoverage");
if(glSampleCoverage == NULL) return false;
glStencilFuncSeparate = (void (__stdcall *)(GLenum, GLenum, int, unsigned int))
                          load_function("glStencilFuncSeparate");
if(glStencilFuncSeparate == NULL) return false;
glStencilOpSeparate = (void (__stdcall *)(GLenum, GLenum, GLenum, GLenum))
                         load_function("glStencilOpSeparate");
if(glStencilOpSeparate == NULL) return false;
glBlendEquation = (void (__stdcall *)(GLenum)) load_function("glBlendEquation");
if(glBlendEquation == NULL) return false;
glBlendEquationSeparate = (void (__stdcall *)(GLenum, GLenum))
                            load_function("glBlendEquationSeparate");
if(glBlendEquationSeparate == NULL) return false;
glBlendFuncSeparate = (void (__stdcall *)(GLenum, GLenum))
                        load_function("glBlendFuncSeparate");
if(glBlendFuncSeparate == NULL) return false;
glBlendColor = (void (__stdcall *)(GLclampf, GLclampf, GLclampf, GLclampf))
                    load_function("glBlendColor");
if(glBlendColor == NULL) return false;
```

O uso das funções acima requer também a definição das seguintes macros a serem usadas como enumerações:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_ONE                      1
#define GL_ZERO                     0
#define GL_LESS                     0x0201
#define GL_INCR                     0x1E02
#define GL_DECR                     0x1E03
#define GL_KEEP                     0x1E00
#define GL_BACK                     0x0405
#define GL_FRONT                    0x0404
#define GL_EQUAL                    0x0202
#define GL_NEVER                    0x0200
#define GL_ALWAYS                   0x0207
#define GL_LEQUAL                   0x0203
#define GL_GEQUAL                   0x0206
#define GL_INVERT                   0x150A
#define GL_REPLACE                  0x1E01
#define GL_GREATER                  0x0204
#define GL_NOTEQUAL                 0x0205
#define GL_FUNC_ADD                 0x8006
#define GL_INCR_WRAP                0x8507
#define GL_DECR_WRAP                0x8508
#define GL_SRC_ALPHA                0x0302
#define GL_DST_ALPHA                0x0304
#define GL_SRC_COLOR                0x0300
#define GL_DST_COLOR                0x0306
#define GL_FUNC_SUBTRACT            0x800A
#define GL_FRONT_AND_BACK           0x0408
#define GL_CONSTANT_COLOR           0x8001
#define GL_CONSTANT_ALPHA           0x8003
#define GL_SRC_ALPHA_SATURATE¥    0x0308
#define GL_ONE_MINUS_SRC_COLOR      0x0301
#define GL_ONE_MINUS_DST_COLOR      0x0307
#define GL_ONE_MINUS_SRC_ALPHA      0x0303
#define GL_ONE_MINUS_DST_ALPHA      0x0305
#define GL_FUNC_REVERSE_SUBTRACT    0x800B
#define GL_ONE_MINUS_CONSTANT_COLOR 0x8002
#define GL_ONE_MINUS_CONSTANT_ALPHA 0x8004
#endif
```

Com relação à funções que afetam todo o “framebuffer”, temos que preparar estas 2 funções:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern void (__stdcall *glStencilMaskSeparate)(GLenum, unsigned int);
extern void (__stdcall *glClearDepthf)(GLclampf);
#endif
```

A declaração delas em `window.c`:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
void (__stdcall *glStencilMaskSeparate)(GLenum, unsigned int);
void (__stdcall *glClearDepthf)(GLclampf);
#endif
```

E a inicialização:

Seção: Windows: Configurar OpenGL (continuação):

```
glStencilMaskSeparate = (void (__stdcall *)(GLenum, unsigned int))
                            load_function("glStencilMaskSeparate");
if(glStencilMaskSeparate == NULL) return false;
glClearDepthf = (void (__stdcall *)(GLclampf)) load_function("glClearDepthf");
if(glClearDepthf == NULL) return false;
```

O último conjunto de funções faltantes são as funções que gerenciam objetos de framebufer:

Seção: Declarações de Janela (continuação):

```
#if defined(_WIN32)
extern void (__stdcall *glBindFramebuffer)(GLenum, unsigned int);
extern void (__stdcall *glDeleteFramebuffers)(GLsizei, unsigned int *);
extern void (__stdcall *glGenFramebuffers)(GLsizei, unsigned int *);
extern void (__stdcall *glBindRenderbuffer)(GLenum, unsigned int);
extern void (__stdcall *glDeleteRenderbuffers)(GLsizei, const unsigned int *);
extern void (__stdcall *glGenRenderbuffers)(GLsizei, unsigned int *);
extern void (__stdcall *glRenderbufferStorage)(GLenum, GLenum, GLsizei, GLsizei);
extern void (__stdcall *glFramebufferRenderbuffer)(GLenum, GLenum, GLenum,
                                                   unsigned int);
extern void (__stdcall *glFramebufferTexture2D)(GLenum, GLenum, GLenum,
                                                unsigned int, int);
extern void (__stdcall *glCheckFramebufferStatus)(GLenum);
extern boolean (__stdcall *glIsFramebuffer)(unsigned int);
extern void (__stdcall *glGetFramebufferAttachmentParameteriv)(GLenum, GLenum,
                                                               GLenum, int *);
extern boolean (__stdcall *glIsRenderbuffer)(unsigned int);
extern void (__stdcall *glGetRenderbufferParameteriv)(GLenum, GLenum, int *);
#endif
```

A declaração destas 14 funções em `window.c`:

Seção: Variáveis Globais (continuação):

```
#if defined(_WIN32)
void (__stdcall *glBindFramebuffer)(GLenum, unsigned int);
void (__stdcall *glDeleteFramebuffers)(GLsizei, unsigned int *);
void (__stdcall *glGenFramebuffers)(GLsizei, unsigned int *);
void (__stdcall *glBindRenderbuffer)(GLenum, unsigned int);
void (__stdcall *glDeleteRenderbuffers)(GLsizei, const unsigned int *);
void (__stdcall *glGenRenderbuffers)(GLsizei, unsigned int *);
void (__stdcall *glRenderbufferStorage)(GLenum, GLenum, GLsizei, GLsizei);
void (__stdcall *glFramebufferRenderbuffer)(GLenum, GLenum, GLenum,
                                            unsigned int);
void (__stdcall *glFramebufferTexture2D)(GLenum, GLenum, GLenum,
                                         unsigned int, int);
void (__stdcall *glCheckFramebufferStatus)(GLenum);
boolean (__stdcall *glIsFramebuffer)(unsigned int);
void (__stdcall *glGetFramebufferAttachmentParameteriv)(GLenum, GLenum,
                                                        GLenum, int *);
boolean (__stdcall *glIsRenderbuffer)(unsigned int);
void (__stdcall *glGetRenderbufferParameteriv)(GLenum, GLenum, int *);
#endif
```

E a inicialização:

Seção: Windows: Configurar OpenGL (continuação):

```
glBindFramebuffer = (void (__stdcall *)(GLenum, unsigned int))
                        load_function("glBindFramebuffer");
if(glBindFramebuffer == NULL) return false;
glDeleteFramebuffers = (void (__stdcall *)(GLsizei, unsigned int *))
                           load_function("glDeleteFramebuffers");
if(glDeleteFramebuffers == NULL) return false;
glGenFramebuffers = (void (__stdcall *)(GLsizei, unsigned int *))
                        load_function("glGenFramebuffers");
if(glGenFramebuffers == NULL) return false;
glBindRenderbuffer = (void (__stdcall *)(GLenum, unsigned int))
                          load_function("glBindRenderbuffer");
if(glBindRenderbuffer == NULL) return false;
glDeleteRenderbuffers = (void (__stdcall *)(GLsizei, const unsigned int *))
                          load_function("glDeleteRenderbuffers");
if(glDeleteRenderbuffers == NULL) return false;
glGenRenderbuffers = (void (__stdcall *)(GLsizei, unsigned int *))
                          load_function("glGenRenderbuffers");
if(glGenRenderbuffers == NULL) return false;
glRenderbufferStorage = (void (__stdcall *)(GLenum, GLenum, GLsizei, GLsizei))
                          load_function("glRenderbufferStorage");
if(glRenderbufferStorage == NULL) return false;
glFramebufferRenderbuffer = (void (__stdcall *)(GLenum, GLenum, GLenum,
                                                unsigned int))
                            load_function("glFramebufferRenderbuffer");
if(glFramebufferRenderbuffer == NULL) return false;
glFramebufferTexture2D = (void (__stdcall *)(GLenum, GLenum, GLenum,
                                             unsigned int, int))
                          load_function("glFramebufferTexture2D");
if(glFramebufferTexture2D == NULL) return false;
glCheckFramebufferStatus = (void (__stdcall *)(GLenum))
                             load_function("glCheckFramebufferStatus");
if(glCheckFramebufferStatus == NULL) return false;
glIsFramebuffer = (boolean (__stdcall *)(unsigned int))
                    load_function("glIsFramebuffer");
if(glIsFramebuffer == NULL) return false;
glGetFramebufferAttachmentParameteriv = (void (__stdcall *)(GLenum, GLenum,
                                                            GLenum, int *))
                         load_function("glGetFramebufferAttachmentParameteriv");
if(glGetFramebufferAttachmentParameteriv == NULL) return false;
glIsRenderbuffer = (boolean (__stdcall *)(unsigned int))
                        load_function("glIsRenderbuffer");
if(glIsRenderbuffer == NULL) return false;
glGetRenderbufferParameteriv = (void (__stdcall *)(GLenum, GLenum, int *))
                                  load_function("glGetRenderbufferParameteriv");
if(glGetRenderbufferParameteriv == NULL) return false;
```

E as macros necessárias para o uso destas funções:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define GL_RGBA4                                        0x8056
#define GL_RGB565                                       0x8D62
#define GL_RGB5_A1                                      0x8057
```

```
#define GL_FRAMEBUFFER                                  0x8D40
#define GL_RENDERBUFFER                                 0x8D41
#define GL_STENCIL_INDEX8                               0x8D48
#define GL_DEPTH_ATTACHMENT                             0x8D00
#define GL_DEPTH_COMPONENT16                            0x81A5
#define GL_COLOR_ATTACHMENT0                            0x8CE0
#define GL_STENCIL_ATTACHMENT                           0x8D20
#define GL_RENDERBUFFER_WIDTH                           0x8D42
#define GL_RENDERBUFFER_HEIGHT                          0x8D43
#define GL_FRAMEBUFFER_COMPLETE                         0x8CD5
#define GL_RENDERBUFFER_RED_SIZE                        0x8D50
#define GL_RENDERBUFFER_BLUE_SIZE                       0x8D52
#define GL_RENDERBUFFER_GREEN_SIZE                      0x8D51
#define GL_RENDERBUFFER_ALPHA_SIZE                      0x8D53
#define GL_RENDERBUFFER_DEPTH_SIZE                      0x8D54
#define GL_RENDERBUFFER_STENCIL_SIZE                    0x8D55
#define GL_RENDERBUFFER_INTERNAL_FORMAT                 0x8D44
#define GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE           0x8CD0
#define GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME           0x8CD1
#define GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL         0x8CD2
#define GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE 0x8CD3
#endif
```

2.3.4. Escolhendo a Versão do OpenGL

No código definido antes, em todos os ambientes, nós escolhemos como versão do OpenGL aquilo que estivesse definido nas macros:

1) `W_WINDOW_OPENGL_MAJOR_VERSION`

2) `W_WINDOW_OPENGL_MINOR_VERSION`

Vamos agora definir qual será a versão padrão caso o usuário não personalize estas macros.

Caso estejamos rodando o X11, estamos usango o EGL para criar um contexto OpenGL ES. Por padrão iremos pedir então para criar um contexto OpenGL ES 3.0. Isso fará com que todas as funções e recursos do OpenGL ES 2.0 fiquem definidos, além de mais algumas coisas novas da versão 3.0.

Se estivermos rodando em navegador de Internet por meio de Web Assembly, então estaremos usando WebGL. A versão equivalente ao Open GL ES 3.0 é o WebGL 2. Enquanto o primeiro WebGL seria equivalente ao OpenGLES 2.0. Neste caso queremos a versão 2.

Já no Windows, nós não temos garantia de que é possível criar um contexto OpenGL ES, pois isso depende do hardware em que estamos rodando. Sendo assim, pedimos a criação de um contexto OpenGL 4.1, já que é a versão mais garantida de funcionar que contém em sua API as funções mais modernas do OpenGL Es.

Sendo assim, a nossa macro que define a versão OpenGL é:

Seção: Cabeçalhos (continuação):

```
#if defined(_WIN32) && !defined(W_WINDOW_OPENGL_MAJOR_VERSION)
#define W_WINDOW_OPENGL_MAJOR_VERSION 4
#define W_WINDOW_OPENGL_MINOR_VERSION 1
#elif defined(__EMSCRIPTEN__) && !defined(W_WINDOW_OPENGL_MAJOR_VERSION)
#define W_WINDOW_OPENGL_MAJOR_VERSION 2
#define W_WINDOW_OPENGL_MINOR_VERSION 0
#elif !defined(W_WINDOW_OPENGL_MAJOR_VERSION)
#define W_WINDOW_OPENGL_MAJOR_VERSION 3
#define W_WINDOW_OPENGL_MINOR_VERSION 0
#endif
```

2.4. Fechando uma Janela

Fechar um janela faz a janela desaparecer e finaliza qualquer coisa que tenha sido inicializada durante a criação da janela. O que inclui o contexto OpenGL.

2.4.1. Fechando uma Janela no X11

Fechar uma janela no X11 significa invocar a função do X que pede ao servidor para que a janela seja fechada. E além disso, fechar a conexão com o servidor. Isso é feito chamando respectivamente `XDestroyWindow` e `XCloseDisplay`. Também fazemos uma checagem para ver se realmente existe uma janela a serfechada e destruída.

Seção: Funções da API (continuação):

```
#if defined(__linux__) || defined(BSD)
bool _Wdestroy_window(void){
  if(already_have_window == false)
    return false;
  eglMakeCurrent(egl_display, EGL_NO_SURFACE, EGL_NO_SURFACE,
                 EGL_NO_CONTEXT );
  eglDestroySurface(egl_display, egl_window);
  eglDestroyContext(egl_display, egl_context);
  eglTerminate(egl_display);
  XDestroyWindow(display, window);
  XCloseDisplay(display);
  display = NULL;
  already_have_window = false;
  return true;
}
#endif
```

2.4.2. Fechando uma Janela em Web Assembly

Fechar uma janela quando executando dentro de um navegador de Internet graças ao Web Assembly significa finalizar todas as estruturas SDL e esconder o canvas onde estávamos desenhando. Fazemos isso com a seguinte função:

Seção: Funções da API (continuação):

```
#if defined(__EMSCRIPTEN__)
bool _Wdestroy_window(void){
  if(already_have_window == false)
    return false;
  SDL_FreeSurface(window);
  EM_ASM(
    var el = document.getElementById("canvas");
    el.style.display = "none";
  );
  already_have_window = false;
  return true;
}
#endif
```

2.4.3. Fechando uma Janela no Windows

Fechar a janela no Windows significa chamar a função que irá encerrá-la:

Seção: Funções da API (continuação):

```
#if defined(_WIN32)
bool _Wdestroy_window(void){
```

```
  if(already_have_window == false)
    return false;
  wglMakeCurrent(NULL, NULL);
  wglDeleteContext(wgl_context);
  DestroyWindow(window);
  already_have_window = false;
  return true;
}
#endif
```

2.5. Renderizando a Janela

Em um programa gráfico, geralmente temos um laço principal e dentro deste laço, chamamos alguma função para desenhar na tela. Isso significa pedir para que a janela seja atualizada depois de realizar chamadas OpenGL para desenhar. O modo de fazer isso depende do nosso ambiente e API.

2.5.1. Renderizando a Janela no X

No X, como pedimos para o EGL criar uma janela e por padrão ele cria uma janela com buffer duplo, para renderizarmos os comandos OpenGL após eles terem sido feitos, devemos fazer com que o buffer traseiro, onde estávamos desenhando, se torne o buffer dianteiro que é exibido na tela e vice-versa. Fazemos isso invocando a função `eglSwapBuffers`.

Entretanto, além do comando para garantir a renderização na tela, devemos nos preocupar com outro cenário: existe uma miríade de gerenciadores de janela no X, e uma variedade muito grande de ambientes nos quais nossa janela irá rodar. Não temos muito controle de alguns eventos como o fato da nossa janela poder perder o foco de teclado e mouse, mas continuar ocupando a tela toda caso seja uma janela em tela-cheia. Como este é um cenário problemático, pois pode fazer com que não possamos mais interagir com o sistema, devemos garantir que se estamos em tela-cheia e detectarmos que nossa janela perdeu o foco, devemos reassumi-lo. Mas só fazemos isso se a nossa janela no momento estiver realmente visível (ou o X provoca um erro fatal, devemos sempre checar isso antes de mudar o foco para uma janela):

Seção: Funções da API (continuação):

```
#if defined(__linux__) || defined(BSD)
bool _Wrender_window(void){
  if(_Wis_fullscreen()){ // Garante foco em tela-cheia
    Window focused_window;
    int focus_status;
    XGetInputFocus(display, &focused_window, &focus_status);
    if(focused_window != window){
      XWindowAttributes attr;
      XGetWindowAttributes(display, window, &attr);
      if(attr.map_state == IsViewable)
        XSetInputFocus(display, window, RevertToParent, CurrentTime);
    }
  }// Faz a troca de buffers para renderização:
  return eglSwapBuffers(egl_display, egl_window);
}
#endif
```

2.5.2. Renderizando a Janela no Emscripten

A função `emscripten_sleep` seria a que realiza a atualização de uma janela quando estamos em um laço principal. Contudo, usá-la é uma má prática. No Web Assembly, programas nunca devem rodar em um laço infinito, mas ao invés disso uma função deve ser registrada para rodar várias vezes seguidas sem parar ao invés de ser feito um laço infinito como é mais comum. Quando

registramos uma função desta forma, não é necessário usar nenhum comando adicioinal para que a janela seja atualizada, isso é feito automaticamente. Por isso, na nssa função abaixo, a única coisa que fazemos é chamar uma função inócua apenas para garantir que enviamos todos os comandos OpenGL. Mas não usamos nada para pedir para a tela ser atualizada, já que assumimos que o programa não irá seguir a má-prática:

Seção: Funções da API (continuação):

```
#if defined(__EMSCRIPTEN__)
bool _Wrender_window(void){
  glFlush();
  return true;
}
#endif
```

2.5.3. Renderizando a Janela no Windows

No Windows, a função WGL responsável por trocar os buffers da janela e assim tornar visíveis os desenhos feitos é chamada `wglSwapLayerBuffers`:

Seção: Funções da API (continuação):

```
#if defined(_WIN32)
bool _Wrender_window(void){
  return wglSwapLayerBuffers(device_context, WGL_SWAP_MAIN_PLANE);
}
#endif
```

2.6. Obtendo o Tamanho da Janela

Agora iremos definir a função que armazena nos ponteirso passados como argumento o tamanho em pixels da nossa janela.

2.6.1. Obtendo o Tamanho da Janela no X

No X11, a API do Xlib nos fornece a função `XGetGeometry` que fornece informação sobre uma janela ou pixmap (imagem). A função retorna uma grande quantidade de informação em diferentes ponteiros. Mas a informação que importa para nós é apenas a altura e largura da janela:

Seção: Funções da API (continuação):

```
#if defined(__linux__) || defined(BSD)
bool _Wget_window_size(int *width, int *height){
  Window root_window;
  int x, y;
  unsigned int border, depth;
  if(!already_have_window || display == NULL){
    *width = 0;
    *height = 0;
    return false;
  }
  XGetGeometry(display, window, &root_window, &x, &y,
               (unsigned int *) width, (unsigned int *) height, &border, &depth);
  window_size_x = *width;
  window_size_y = *height;
  return true;
}
#endif
```

2.6.2. Obtendo o Tamanho da Janela no Web Assembly

Em um navegador web, nós não temos uma janela, mas um "canvas" HTML. Por convenção, o canvas que usamos como janela tem o ID "cancas". Se queremos saber o seu tamanho, precisamos usar Javascript para obter o elemento e em seguida ler sua altura e largura.

Seção: Funções da API (continuação):

```
#if defined(__EMSCRIPTEN__)
bool _Wget_window_size(int *width, int *height){
  if(!already_have_window){
    *width = 0;
    *height = 0;
    return false;
  }
  *width = EM_ASM_INT({
    return document.getElementById("canvas").width;
  });
  *height = EM_ASM_INT({
    return document.getElementById("canvas").height;
  });
  window_size_x = *width;
  window_size_y = *height;
  if(*width > 0 && *height > 0)
    return true;
  else{
    *width = 0;
    *height = 0;
    return false;
  }
}
#endif
```

2.6.3. Obtendo o Tamanho da Janela no Windows

No Windows nós usamos uma chamada a `GetWindowRect` para obter o tamanho da janela. o qual é armazenado em uma estrutura `RECT`:

Seção: Funções da API (continuação):

```
#if defined(_WIN32)
bool _Wget_window_size(int *width, int *height){
  BOOL ret;
  RECT rectangle;
  ret = GetWindowRect(window, &rectangle);
  if(ret){
    *width = rectangle.right - rectangle.left;
    *height = rectangle.bottom - rectangle.top;
    window_size_x = *width;
    window_size_y = *height;
    return true;
  }
  else{
    *width = 0;
    *height = 0;
    return false;
  }
}
```

```
#endif
```

2.7. Checando o modo de tela cheia

Aqui iremos definir as funções que verificam se estamos em modo de tela cheia ou não.

2.7.1. Checando modo de tela cheia no X11

No X11, criamos uma janela em tela cheia ajustando um atributo para que ela seja renderizada ocupando toda a tela ignorando qualquer configuração do gerenciador de janelas. Para checarmos se estamos em tela cheia, precisamos checar com a função `XGetWindowAttributes` se a nossa janela tem tal atributo marcado ou não:

Seção: Funções da API (continuação):

```
#if defined(__linux__) || defined(BSD)
bool _Wis_fullscreen(void){
  XWindowAttributes attributes;
  if(!already_have_window || display == NULL)
    return false;
  XGetWindowAttributes(display, window, &attributes);
  return (attributes.override_redirect == true);
}
#endif
```

2.7.2. Checando modo de tela cheia no Web Assembly

Rodando em um navegador de Internet, precisamos conferir por meio de Javascript se estamos rodando em tela cheia ou não. Lembrando que aqui nem sempre conseguiremos entrar em tela cheia, já que isso depende das configurações do navegador. A fução abaixo então será uma forma de verificar se estamos ou não em tal modo de tela cheia.

Seção: Funções da API (continuação):

```
#if defined(__EMSCRIPTEN__)
bool _Wis_fullscreen(void){
  int full = EM_ASM_INT({
    if(document.fullscreenElement === null){
      return 0;
    } else{
      return 1;
    }
  });
  return full;
}
#endif
```

2.7.3. Checando modo de tela cheia no Windows

Detectar a tela cheia no Windows envolve usar a função `SHQueryUserNotificationState` para obter o estado da tela. Se tivermos uma janela em tela-cheia, saberemos disso se obtiveros como resultado `QUNS_BUSY`:

Seção: Funções da API (continuação):

```
#if defined(_WIN32)
bool _Wis_fullscreen(void){
  DWORD window_style = GetWindowLongA(window, GWL_STYLE);
  return (window_style & WS_POPUP);
}
#endif
```

2.8. Alternando entre Tela Cheia e Janela

Aqui iremos definir a função que alterna entre modo tela-cheia e janela nos diferentes ambientes suportados:

Fazer isso corretamente requer que memorizemos qual o último tamanho de nossa janela quando não estamos em tela-cheia. Quando passamos da tela cheia para o modo de janela iremos fazer nossa janela ocupar de volta o mesmo tamanho de antes. O tamanho inicial é configurado por macros, se elas existirem. Neste caso, o valor inicial para a variável que memoriza o tamanho é o conteúdo das macros. Se elas não existirem, apenas armazenamos zero nas variáveis significando que não temos memória alguma de qual deve ser o tamanho da janela fora da tela cheia:

Seção: Variáveis Locais (continuação):

```
#if defined(W_WINDOW_SIZE_X) && defined(W_WINDOW_SIZE_Y)
static unsigned last_window_size_x = W_WINDOW_SIZE_X;
static unsigned last_window_size_y = W_WINDOW_SIZE_Y;
#else
static unsigned last_window_size_x = 0;
static unsigned last_window_size_y = 0;
#endif
```

Além disso, quando entramos, saímos da tela-cheia ou quando mudamos o tamanho da janela, pode ser necessário para o usuário recalcular a posição dos objetos que aparecem na tela. Por causa disso, vamos deixar que o usuário possa definir uma função a ser executada toda vez que isso acontecer. Se nenhuma função for passada pelo usuário, nada adicional será executado. O ponteiro que armazena a função definida pelo usuário, se ela existir, será:

Seção: Variáveis Locais (continuação):

```
static void (*resizing_function)(int, int, int, int) = NULL;
```

Para registrar uma função e armazenar o endereço dela neste ponteiro, invoca-se a seguinte função:

Seção: Funções da API (continuação):

```
void _Wset_resize_function(void (*func)(int, int, int, int)){
  resizing_function = func;
}
```

2.8.1. Alternando entre Tela Cheia e Janela no X11

Alternar entre a tela cheia no X11 significa mudar um atributo da janela o atributo que detrmina se o gerenciador de janelas terá o controle dela para poder colocar decorações na janela, ou se pelo contrário, nós devemos ignorar o gerenciador de janelas e controlar completamente o tamanho e posição da janela, que ficará sem decoradores.

Como isso significa alternar o controle da janela entre o gerenciador de janelas e o próprio servidor X, após mudar este atributo precisamos retirarnossa janela, fazendo-a deixar de existir momentaneamente para em seguida mapeá-la novamente. Assim a janela irá ressurgir com ou sem os decoradores característicos do gerenciador de janela. Caso estejamos entrando na tela cheia, tambem será necessário mover nossa janela e ajustar o tamanho dela para o tamanho da tela.

Seção: Funções da API (continuação):

```
#if defined(__linux__) || defined(BSD)
void _Wtoggle_fullscreen(void){
  int new_size_x, new_size_y, old_size_x, old_size_y;
  bool changing_to_fullscreen = false;
  XSetWindowAttributes attributes;
  if(!already_have_window)
    return;
  _Wget_window_size(&old_size_x, &old_size_y);
  XWithdrawWindow(display, window, 0);
```

```
  if(!_Wis_fullscreen()){
    attributes.override_redirect = true;
    changing_to_fullscreen = true;
    XMoveWindow(display, window, 0, 0);
  }
  else
    attributes.override_redirect = false;
  XChangeWindowAttributes(display, window, CWOverrideRedirect,
                          &attributes);
  XMapWindow(display, window);
  { // Ao mapear a janela novamente temos que esperar isso surtir efeito:
    XEvent e;
    do{
      XNextEvent(display, &e);
    } while(e.type != MapNotify);
  }
  if(changing_to_fullscreen){
    _Wget_screen_resolution(&new_size_x, &new_size_y);
    XMoveResizeWindow(display, window, 0, 0, new_size_x, new_size_y);
      <Seção a ser Inserida: X11: Esperar Redimencionamento Surtir Efeito>
  }
  else{
    new_size_x = ((last_window_size_x > 0)?(last_window_size_x):(800));
    new_size_y = ((last_window_size_y > 0)?(last_window_size_y):(600));
    XResizeWindow(display, window, new_size_x, new_size_y);
      <Seção a ser Inserida: X11: Esperar Redimencionamento Surtir Efeito>
  }
  _Wget_window_size(&new_size_x, &new_size_y);
  glViewport(0, 0, new_size_x, new_size_y);
  if(resizing_function != NULL)
    resizing_function(old_size_x, old_size_y, new_size_x, new_size_y);
}
#endif
```

Na função acima nós enviamos ao servidor X o pedido para que a janela tenha seu tamanho modificado. Entretanto, enfviar o pedido não quer dizer que o pedido é realizado. É importante esperar a modificação para que possamos realmente atualizar corretamente o tamanho da área de desenho do OpenGL e garantir que as próximas funções que tentarem medir o tamanho da janela retorne o valor correto. Por causa dsso, nós temos um trecho de código acima para esperar as modificações surtirem efeito.

Mas não podemos esperar para sempre. Pois pode ser que o gerenciador de janelas se recuse a mudar o tamanho e neste cenário nós nunca terímamos o redimencionamento. Por causa disso, vamos estabelecer o tempo de espera máximo de 0,1 segundos.

O trecho de código que nos permite esperar a mudança de tamanho surtir efeito é:

Seção: X11: Esperar Redimencionamento Surtir Efeito:

```
{ // Espera no máximo 0.1 segundos para que a mudança surta efeito
  struct timeval begin_time, now;
  int new_width, new_height;
  gettimeofday(&begin_time, NULL);
  do{
    _Wget_window_size(&new_width, &new_height);
    gettimeofday(&now, NULL);
  } while((new_width != new_size_x || new_height != new_size_y) &&
          (now.tv_sec - begin_time.tv_sec) * 1000000 +
```

```
          (now.tv_usec - begin_time.tv_usec) < 100000);
}
```

E como usamos a função de obter o tempo, precisamos inserir o seguinte cabeçalho:

Seção: Cabeçalhos (continuação):

```
#if defined(__linux__) || defined(BSD)
#include <sys/time.h>
#endif
```

2.8.2. Alternando entre Tela Cheia e Janela no Web Assembly

Em um navegador de Internet, quando tentamos entrar em tela-cheia, duas coisas podem acontecer: podemos realmente obter uma tela-cheia verdadeira se o navegador de Internet deixar ou podemos não obter permissão. Neste segundo caso nós ainda simulamos uma tela-cheia fazendo nosso canvas HTML ter o mesmo tamanho da tela e ocupar a área de topo do documento. Isso permite que o usuário ainda possa entrar em tela-cheia manualmente nas opções do navegador.

Isso significa que podemos estar em uma tela-cheia verdadeira ou simulada. Por causa disso neste caso temos uma variável estática booleana, a `fullscreen_mode` que nos diz se estamos em tela-cheia (verdadeira ou simulada) ou não. Em contraste, a função `_Wis_fullscreen` apenas nos informa se estamos em tela-cheia verdadeira.

Nossa função que alternará entre modo janela e tela-cheia se baseará então na informação presente na variável estática. E mesmo que não possamos entrar em tela-cheia verdadeira, alternaremos entre o modo de janela e o de tela-cheia simulada:

Seção: Funções da API (continuação):

```
#if defined(__EMSCRIPTEN__)
void _Wtoggle_fullscreen(void){
  int new_size_x, new_size_y, old_size_x, old_size_y;
  if(!already_have_window)
    return;
  _Wget_window_size(&old_size_x, &old_size_y);
  if(fullscreen_mode){
    EM_ASM({
      var el = document.getElementById("canvas");
      el.style.position = "initial";
      el.style.top = "initial";
      el.style.left = "initial";
      if($0 === 0 || $1 == 0){
        el.style.width = "initial";
        el.width = el.clientWidth * window.devicePixelRatio;
        el.style.height = "initial";
        el.height = el.clientHeight * window.devicePixelRatio;
      } else{
        el.style.width = ($0 / window.devicePixelRatio) + "px";
        el.width = $0;
        el.style.height = ($1 / window.devicePixelRatio) + "px";
        el.height = $1;
      }
    }, last_window_size_x, last_window_size_y);
    fullscreen_mode = false;
  } else{
    EM_ASM(
      var el = document.getElementById("canvas");
      el.style.position = "absolute";
      el.style.top = "0px";
```

```
      el.style.left = "0px";
      el.style.width = window.innerWidth + "px";
      el.style.height = window.innerHeight + "px";
      el.width = (window.innerWidth * window.devicePixelRatio);
      el.height = (window.innerHeight * window.devicePixelRatio);
    );
    fullscreen_mode = true;
  }
  _Wget_window_size(&new_size_x, &new_size_y);
  glViewport(0, 0, new_size_x, new_size_y);
  if(resizing_function != NULL)
    resizing_function(old_size_x, old_size_y, new_size_x, new_size_y);
}
#endif
```

2.8.3. Alternando entre Tela Cheia e Janela no Windows

No Windows controlamos a tela-cheia alternando entre os estilos da janela. Podemos ler e escrever informações sobre um estilo de janela por meio de `GetWindowLongPtrA` e `SetWindowLongPtrA`:

Seção: Funções da API (continuação):

```
#if defined(_WIN32)
void _Wtoggle_fullscreen(void){
  int new_size_x, new_size_y, old_size_x, old_size_y;
  DWORD window_style = GetWindowLongPtrA(window, GWL_STYLE);
  if(!already_have_window)
    return;
  _Wget_window_size(&old_size_x, &old_size_y);
  if(window_style & WS_POPUP){ // Estamos em tela-cheia. Entrar em modo janela.
    SetWindowLong(window, GWL_STYLE, WS_OVERLAPPED | WS_CAPTION | WS_VISIBLE);
    if(last_window_size_x > 0 && last_window_size_y > 0)
      SetWindowPos(window, HWND_TOP, 0, 0, last_window_size_x,
last_window_size_y,
                   SWP_FRAMECHANGED | SWP_SHOWWINDOW);
    else
      SetWindowPos(window, HWND_TOP, 0, 0, last_window_size_x,
last_window_size_y,
                   SWP_FRAMECHANGED | SWP_SHOWWINDOW | SWP_NOSIZE);
  } else{ // Estamos em modo janela. Entrar em tela-cheia.
    _Wget_screen_resolution(&new_size_x, &new_size_y);
    SetWindowLongPtr(window, GWL_STYLE, WS_POPUP | WS_VISIBLE);
    SetWindowPos(window, HWND_TOP, 0, 0, new_size_x, new_size_y,
                 SWP_SHOWWINDOW | SWP_FRAMECHANGED);
  }
  _Wget_window_size(&new_size_x, &new_size_y);
  glViewport(0, 0, new_size_x, new_size_y);
  if(resizing_function != NULL)
    resizing_function(old_size_x, old_size_y, new_size_x, new_size_y);
}
#endif
```

2.9. Redimencionando a Janela

Aqui definiremos como redimencionar a nossa janela em diferentes ambientes, caso nossa janela

exista e caso não estejamos em modo de tela-cheia. Em caso de erro, a função retorna falso. Caso contrário, retorna verdadeiro.

2.9.1. Redimencionando a Janela no X11

Redimencionar o X11 significa usar a função `XResizeWindow`, que já viemos usando em outras funções. Mas além disso, precisamos também ajustar um novo tamanho máximo e mínimo de janela como sendo igual ao novo tamanho usando `XSetWMNormalHints`. Fazemos isso para impedir que o usuário possa redimencionar a janela sem que seja por meio da função de redimencionamento. Por fim, após redimencionar a janela, esperamos a confirmação de que a operação foi bem-sucedida lendo o novo tamanho da janela para conferir se a operação foi bem-sucedida. Mas se não tivermos confirmação de que foi tudo bem-sucedido em até 0.5 segundos, desistimos e continuamos de qualquer forma, executando a função personalizada escolhida pelo usuário relacionada ao redimencionamento da janela, caso ela exista.

Seção: Funções da API (continuação):

```
#if defined(__linux__) || defined(BSD)
bool _Wresize_window(int width, int height){
  int old_size_x, old_size_y, new_size_x = width, new_size_y = height;
  if(_Wis_fullscreen() || width <= 0 || height <= 0 || !already_have_window)
    return false;
  _Wget_window_size(&old_size_x, &old_size_y);
  {
    XSizeHints hints;
    hints.flags = PMinSize | PMaxSize;
    hints.min_width = hints.max_width = width;
    hints.min_height = hints.max_height = height;
    XSetWMNormalHints(display, window, &hints);
  }
  last_window_size_x = width;
  last_window_size_y = height;
  XResizeWindow(display, window, width, height);
    <Seção a ser Inserida: X11: Esperar Redimencionamento Surtir Efeito>
  glViewport(0, 0, width, height);
  if(resizing_function != NULL)
    resizing_function(old_size_x, old_size_y, width, height);
  return true;
}
#endif
```

2.9.2. Redimencionando a Janela no Web Assembly

No Web Assembly, para mudar o tamanho de nosso "canvas", caso não estejamos nem em tela-cheia e nem simulando uma tela-cheia, apenas acessamos o nosso "canvas" por meio de seu ID e editamos suas propriedades CSS e atributos de tamanho:

Seção: Funções da API (continuação):

```
#if defined(__EMSCRIPTEN__)
bool _Wresize_window(int width, int height){
  int old_size_x, old_size_y;
  if(fullscreen_mode || width <= 0 || height <= 0 || !already_have_window)
    return false;
  _Wget_window_size(&old_size_x, &old_size_y);
  last_window_size_x = width;
  last_window_size_y = height;
  EM_ASM({
```

```
    var el = document.getElementById("canvas");
    el.style.width = ($0 / window.devicePixelRatio) + "px";
    el.width = $0;
    el.style.height = ($1 / window.devicePixelRatio) + "px";
    el.height = $1;
  }, width, height);
  glViewport(0, 0, width, height);
  if(resizing_function != NULL)
    resizing_function(old_size_x, old_size_y, width, height);
  return true;
}
#endif
```

2.9.3. Redimencionando a Janela no Windows

No Windows mover a janela significa usar a função `SetWindowPos`, o que fazemos somente se não estivermos em modo de tela-cheia. Em seguida, se existir, executamos a função personalizada pelo usuário a ser usada sempre que o tamanho da janela muda:

Seção: Funções da API (continuação):

```
#if defined(_WIN32)
bool _Wresize_window(int width, int height){
  int old_size_x, old_size_y;
  bool ret;
  if(_Wis_fullscreen() || width <= 0 || height <= 0 || !already_have_window)
    return false;
  _Wget_window_size(&old_size_x, &old_size_y);
  last_window_size_x = width;
  last_window_size_y = height;
  ret = SetWindowPos(window, 0, 0, 0, width, height, SWP_NOMOVE | SWP_NOZORDER);
  if(!ret)
    return false;
  glViewport(0, 0, width, height);
  if(resizing_function != NULL)
    resizing_function(old_size_x, old_size_y, width, height);
  return true;
}
#endif
```

3. Gerenciando Entrada

Nesta seção trataremos a detecção de eventos de entrada de um usuário. Como quando ele pressiona uma tecla ou move o mouse.

3.1. Definindo o Teclado e Mouse

Quando nós temos uma janela, podemos detectar entrada gerada pelo usuário quando tal janela tem foco. Podemos como entrada detectar eventos feitos pelo mouse e teclado.

Para nós, um teclado será representado pela seguinte estrutura de dados:

Seção: Estruturas de Dados de Janela:

```
struct _Wkeyboard{
  long key[W_KEYBOARD_SIZE + 1]; // Vetor de teclas: contador de tempo para elas
};
```

No vetor de teclas acima, nós reservamos uma posição para cada tecla diferente com o propósito de contar o tempo que leva para serem pressionadas e soltas. O significado do número armazenado em cada posição depende de seu valor. As opções são:

1) Se armazenamos zero, isso significa que a tecla específica associada à tal posição não está sendo pressionada.

2) Se armazenamos 1, isso significa que a tecla começou a ser pressionada agora.

3) Se armazenamos um número positivo maior que 1, isso significa que a tecla está sendo pressionada, ainda não foi solta e o número representa a quantas unidades de tempo essa tecla está sendo pressionada.

4) Se armazenamos um número negativo, isso significa que a tecla foi solta agora. O oposto do número armazenado representa por quantas unidades de tempo a tecla foi pressionada antes de ser solta.

O número de teclas acompanhadas depende do sistema operacional e ambiente. A macro `W_KEYBOARD_SIZE` representa o número de teclas que suportamos. A estrutura do teclado possui também uma posição adicional para a qual será mapeada qualquer tecla desconhecida. A última posição do vetor nunca armazenará nenhum valor diferente de zero.

O mouse será representado pela seguinte estrutura:

Seção: Estruturas de Dados de Janela (continuação):

```
struct _Wmouse{
  long button[W_MOUSE_SIZE];
  int x, y, dx, dy, ddx, ddy;
};
```

Os botões do mouse seguem as mesmas convenções das teclas de um teclado. Nós podemos detectar se elas estão sendo pressionadas, por quanto tempo, se elas são soltas e por quanto tempo foram pressionadas antes de serem soltas.

As variáveis `x` e `y` representam a posição do ponteiro do mouse em pixels de nossa janela. As variáveis `dx` e `dy` representam a velocidade do ponteiro em pixels por unidade de tempo. As variáveis `ddx` e `ddy` representam a aceleração do ponteiro em pixels por unidade de tempo ao quadrado.

Dadas estas estruturas, precisamos chamar a seguinte função periodicamente para atualizá-las:

Seção: Funções da API (continuação):

```
void _Wget_window_input(unsigned long long current_time,
                        struct _Wkeyboard *keyboard,
                        struct _Wmouse *mouse){
  if(already_have_window == false)
    return;
          <Seção a ser Inserida: Antes de Obter Eventos da Janela>
              <Seção a ser Inserida: Obter Eventos da Janela>
          <Seção a ser Inserida: Depois de Obter Eventos da Janela>
}
```

A função basicamente entra em um laço para ler todos os eventos que a janela recebe. Dependendo do evento, ela atualiza de maneira adequada o estado das estruturas da janela e mouse. Por exemplo, no X11, a leitura de eventos acontece assim:

Seção: Obter Eventos da Janela:

```
#if defined(__linux__) || defined(BSD)
XEvent event;
while(XPending(display)){
  XNextEvent(display, &event);
                  <Seção a ser Inserida: X11: Tratar Eventos>
}
#endif
```

Se estamos em ambiente Web Assembly, usamos a API SDL para ler os eventos:

Seção: Obter Eventos da Janela (continuação):

```
#if defined(__EMSCRIPTEN__)
```

```
SDL_Event event;
while(SDL_PollEvent(&event)){
            <Seção a ser Inserida: Web Assembly: Tratar Eventos>
}
#endif
```

No Windows, os eventos são chamados de "mensagens". E a forma de ler eles é dada abaixo:

Seção: Obter Eventos da Janela (continuação):

```
#if defined(_WIN32)
MSG event;
while(PeekMessage(&event, window, WM_KEYFIRST, WM_KEYLAST, PM_REMOVE)){
          <Seção a ser Inserida: Windows: Tratar Eventos de Teclado>
}
while(PeekMessage(&event, window, WM_MOUSEFIRST, WM_MOUSELAST, PM_REMOVE)){
          <Seção a ser Inserida: Windows: Tratar Eventos de Mouse>
}
#endif
```

No caso do Windows, nós podemos filtrar o tipo de evento que queremos tratar. Então separamos os eventos relacionados ao mouse e teclado, tratando eles exclusivamente.

3.2. Lendo o Teclado

Aqui definiremos como iremos monitorar o teclado em diferentes ambientes para atualizar a estrutura de teclado que definimos.

Os eventos, em todos os ambientes, vao nos avisar se uma tecla é pressionada ou solta. Mas eles não vao nos passar qualquer informação sobre o tempo que elas ficaram pressionadas. Nós temos que armazenar e cuidar desta informação. Para isso, usaremos um vetor que armamazena teclas e o tempo no qual recebemos a informação de que elas foram pressionadas ou soltas. No laço em que lemos os eventos, é esse vetor de teclas que iremos atualizar:

Seção: Variáveis Locais:

```
static struct{
  unsigned key; // Qual tecla foi pressionada?
  unsigned long long time;    // Quando foi pressionada?
} pressed_keys[32];
static unsigned released_keys[32]; // Que teclas foram soltas?
```

Como parte da inicialização de nosso teclado, e também para reiniciar o estado dele, devemos chamar sempre um código que percorre a lista e deixa cada uma das posições marcada como tendo uma tecla de valor zero. Iremos usar uma tecla de valor zero para marcar o fim da nossa lista, exatamente como usa-se um caractere de valor zero para marcar o fim de uma string. Além disso, quando queremos inicializar ou reinicializar o teclado, também temos que percorrer o seu vetor de teclas e ajustar todos para zero. O código de inicialização e reinicialização do teclado é:

Seção: Iniciando ou Reiniciando Teclado:

```
{
  int i;
  for(i = 0; i < 32; i ++){
    pressed_keys[i].key = 0;
    released_keys[i] = 0;
  }
  for(i = 0; i < W_KEYBOARD_SIZE + 1; i ++)
    keyboard -> key[i] = 0;
}
```

3.2.1. Lendo o Teclado no X

No X11, cada tecla de teclado diferente possui um códico único entre 8 e 255. Esse código

representa a tecla física pressionada, mas não tem relação com o símbolo específico associado à tal tecla. O símbolo depende do mapeamento de teclado configurado. Para obter o símbolo associado à tecla pressionada, é necessário traduzir o código para um símbolo (`keycode` para `keysym`). De qualquer forma, isso significa que teremos que acompanhar um número potencial de quase 256 teclas diferentes.

Seção: Define Macros (continuação):

```
#if defined(__linux__) || defined(BSD)
#define W_KEYBOARD_SIZE 256
#endif
```

Para detectar se uma tecla foi pressionada, antes de adicioná-la à lista de teclas pressionadas, usamos o seguinte código:

Seção: X11: Tratar Eventos:

```
if(event.type == KeyPress){
  unsigned key = event.xkey.keycode;
    <Seção a ser Inserida: Adiciona 'key' à Lista de Teclas Pressionadas>
}
```

Para detectar se uma tecla foi solta, antes de podermos removê-la da lista de teclas pressionadas e adicionar à lista de teclas soltas, usamos o código abaixo:

Seção: X11: Tratar Eventos (continuação):

```
if(event.type == KeyRelease){
  unsigned key = event.xkey.keycode;
    <Seção a ser Inserida: Remove 'key' da Lista de Teclas Pressionadas e
Adiciona às Teclas Soltas>
}
```

Embora o código acima trate corretamente os códigos de teclas, um usuário também deve saber o que os códigos representam. Ser informado que a tecla de código 0xff0d foi pressionada significa muito pouco. Saber que a tecla "Enter" foi pressionada nos dá muito mais informação. O usuário deve ser capaz de checar o conteúdo de `keyboard.key[W_ENTER]` sem ter que saber qual o código para a tecla no teclado que está sendo usado. Para isso, na inicialização de teclado, devemos descobrir todos os símbolos que suportamos no teclado para assim darmos a eles nomes adequados. Essa informação depende de como o teclado está mapeado, então só podemos obtê-la durante a execução de um programa. O código para descobrirmos a posição de cada tecla que suportamos é:

Seção: Inicialização de Teclado:

```
#if defined(__linux__) || defined(BSD)
{
  int i;
  // Produza eventos de teclas somente quando eles realmente ocorrerem:
  XkbSetDetectableAutoRepeat(display, true, NULL);
  for(i = 8; i < 256; i ++){
    unsigned long value = XkbKeycodeToKeysym(display, i, 0, 0);
    switch(value){
    case 0: break;
    case XK_Escape: W_ESC = i; break;
    case XK_BackSpace: W_BACKSPACE = i; break;
    case XK_Tab: W_TAB = i; break;
    case XK_Return: W_ENTER = i; break;
    case XK_Up:   W_UP   = i; break; case XK_Down:  W_DOWN =  i; break;
    case XK_Left: W_LEFT = i; break; case XK_Right: W_RIGHT = i; break;
    case XK_0: W_0 = i; break;     case XK_1: W_1 = i; break;
    case XK_2: W_2 = i; break;     case XK_3: W_3 = i; break;
    case XK_4: W_4 = i; break;     case XK_5: W_5 = i; break;
```

```
    case XK_6: W_6 = i; break;      case XK_7: W_7 = i; break;
    case XK_8: W_8 = i; break;      case XK_9: W_9 = i; break;
    case XK_minus: W_MINUS = i; break;    case XK_plus: W_PLUS = i; break;
    case XK_F1: W_F1 = i; break;    case XK_F2: W_F2 = i; break;
    case XK_F3: W_F3 = i; break;    case XK_F4: W_F4 = i; break;
    case XK_F5: W_F5 = i; break;    case XK_F6: W_F6 = i; break;
    case XK_F7: W_F7 = i; break;    case XK_F8: W_F8 = i; break;
    case XK_F9: W_F9 = i; break;    case XK_F10: W_F10 = i; break;
    case XK_F11: W_F11 = i; break; case XK_F12: W_F12 = i; break;
    case XK_Shift_L: W_LEFT_SHIFT = i; break;
    case XK_Shift_R: W_RIGHT_SHIFT = i; break;
    case XK_Control_L: W_LEFT_CTRL = i; break;
    case XK_Control_R: W_RIGHT_CTRL = i; break;
    case XK_Alt_L: W_LEFT_ALT = i; break;
    case XK_Alt_R: W_RIGHT_ALT = i; break;
    case XK_space: W_SPACE = i; break;
    case XK_a: W_A = i; break;    case XK_b: W_B = i; break;
    case XK_c: W_C = i; break;    case XK_d: W_D = i; break;
    case XK_e: W_E = i; break;    case XK_f: W_F = i; break;
    case XK_g: W_G = i; break;    case XK_h: W_H = i; break;
    case XK_i: W_I = i; break;    case XK_j: W_J = i; break;
    case XK_k: W_K = i; break;    case XK_l: W_L = i; break;
    case XK_m: W_M = i; break;    case XK_n: W_N = i; break;
    case XK_o: W_O = i; break;    case XK_p: W_P = i; break;
    case XK_q: W_Q = i; break;    case XK_r: W_R = i; break;
    case XK_s: W_S = i; break;    case XK_t: W_T = i; break;
    case XK_u: W_U = i; break;    case XK_v: W_V = i; break;
    case XK_w: W_W = i; break;    case XK_x: W_X = i; break;
    case XK_y: W_Y = i; break;    case XK_z: W_Z = i; break;
    case XK_Insert: W_INSERT = i; break;
    case XK_Home: W_HOME = i; break;
    case XK_Page_Up: W_PAGE_UP = i; break;
    case XK_Delete: W_DELETE = i; break;
    case XK_End: W_END = i; break;
    case XK_Page_Down: W_PAGE_DOWN = i; break;
    default: break;
    }
  }
}
#endif
```

O uso da função `XkbKeycodeToKeysym` requer o cabeçalho:

Seção: Cabeçalhos (continuação):

```
#if defined(__linux__) || defined(BSD)
#include <X11/XKBlib.h>
#endif
```

Todas estas variáveis que representam posições no nosso vetor de teclas de teclado precisam ser declaradas:

Seção: Declarações de Janela (continuação):

```
extern int W_BACKSPACE, W_TAB, W_ENTER, W_UP, W_DOWN, W_LEFT, W_RIGHT, W_0, W_1,
           W_2, W_3, W_4, W_5, W_6, W_7, W_8, W_9, W_MINUS, W_PLUS, W_F1, W_F2,
           W_F3, W_F4, W_F5, W_F6, W_F7, W_F8, W_F9, W_F10, W_F11, W_F12,
           W_LEFT_SHIFT, W_RIGHT_SHIFT, W_LEFT_ALT, W_RIGHT_ALT, W_LEFT_CTRL,
```

```
        W_RIGHT_CTRL, W_SPACE, W_A, W_B, W_C, W_D, W_E, W_F, W_G, W_H, W_I,
        W_J, W_K, W_L, W_M, W_N, W_O, W_P, W_Q, W_R, W_S, W_T, W_U, W_V, W_W,
        W_X, W_Y, W_Z, W_INSERT, W_HOME, W_PAGE_UP, W_DELETE, W_END,
        W_PAGE_DOWN, W_ESC, W_ANY;
```

E elas são inicializadas como sendo o valor `W_KEYBOARD_SIZE`, reservado para representar teclas desconhecidas. Somente depois da inicialização do teclado as variáveis que representam teclas reconhecidas são mapeadas para sua verdadeira posição.

Seção: Variáveis Globais (continuação):

```
int W_BACKSPACE = W_KEYBOARD_SIZE, W_TAB = W_KEYBOARD_SIZE,
    W_ENTER = W_KEYBOARD_SIZE, W_UP = W_KEYBOARD_SIZE, W_DOWN = W_KEYBOARD_SIZE,
    W_LEFT = W_KEYBOARD_SIZE, W_RIGHT = W_KEYBOARD_SIZE, W_0 = W_KEYBOARD_SIZE,
    W_1 = W_KEYBOARD_SIZE, W_2 = W_KEYBOARD_SIZE, W_3 = W_KEYBOARD_SIZE,
    W_4 = W_KEYBOARD_SIZE, W_5 = W_KEYBOARD_SIZE, W_6 = W_KEYBOARD_SIZE,
    W_7 = W_KEYBOARD_SIZE, W_8 = W_KEYBOARD_SIZE, W_9 = W_KEYBOARD_SIZE,
    W_MINUS = W_KEYBOARD_SIZE, W_PLUS = W_KEYBOARD_SIZE, W_F1 = W_KEYBOARD_SIZE,
    W_F2 = W_KEYBOARD_SIZE, W_F3 = W_KEYBOARD_SIZE, W_F4 = W_KEYBOARD_SIZE,
    W_F5 = W_KEYBOARD_SIZE, W_F6 = W_KEYBOARD_SIZE, W_F7 = W_KEYBOARD_SIZE,
    W_F8 = W_KEYBOARD_SIZE, W_F9 = W_KEYBOARD_SIZE, W_F10 = W_KEYBOARD_SIZE,
    W_F11 = W_KEYBOARD_SIZE, W_F12 = W_KEYBOARD_SIZE,
    W_LEFT_SHIFT = W_KEYBOARD_SIZE, W_RIGHT_SHIFT = W_KEYBOARD_SIZE,
    W_LEFT_ALT = W_KEYBOARD_SIZE, W_RIGHT_ALT = W_KEYBOARD_SIZE,
    W_LEFT_CTRL = W_KEYBOARD_SIZE, W_RIGHT_CTRL = W_KEYBOARD_SIZE,
    W_SPACE = W_KEYBOARD_SIZE, W_A = W_KEYBOARD_SIZE, W_B = W_KEYBOARD_SIZE,
    W_C = W_KEYBOARD_SIZE, W_D = W_KEYBOARD_SIZE, W_E = W_KEYBOARD_SIZE,
    W_F = W_KEYBOARD_SIZE, W_G = W_KEYBOARD_SIZE, W_H = W_KEYBOARD_SIZE,
    W_I = W_KEYBOARD_SIZE, W_J = W_KEYBOARD_SIZE, W_K = W_KEYBOARD_SIZE,
    W_L = W_KEYBOARD_SIZE, W_M = W_KEYBOARD_SIZE, W_N = W_KEYBOARD_SIZE,
    W_O = W_KEYBOARD_SIZE, W_P = W_KEYBOARD_SIZE, W_Q = W_KEYBOARD_SIZE,
    W_R = W_KEYBOARD_SIZE, W_S = W_KEYBOARD_SIZE, W_T = W_KEYBOARD_SIZE,
    W_U = W_KEYBOARD_SIZE, W_V = W_KEYBOARD_SIZE, W_W = W_KEYBOARD_SIZE,
    W_X = W_KEYBOARD_SIZE, W_Y = W_KEYBOARD_SIZE, W_Z = W_KEYBOARD_SIZE,
    W_INSERT = W_KEYBOARD_SIZE, W_HOME = W_KEYBOARD_SIZE,
    W_PAGE_UP = W_KEYBOARD_SIZE, W_DELETE = W_KEYBOARD_SIZE,
    W_END = W_KEYBOARD_SIZE, W_PAGE_DOWN = W_KEYBOARD_SIZE,
    W_ESC = W_KEYBOARD_SIZE, W_ANY = 0;
```

3.2.2. Lendo o Teclado no Web Assembly

O Emscripten fornece para nós a API do SDL para interagir com o teclado. Nesta API, o número de teclas suportadas é definida pela macro `SDL_NUM_SCANCODES`, que na época de escrita deste texto corresponde a 512 teclas diferentes. Provavelmente o valor não mudará no futuro próximo.

Seção: Define Macros (continuação):

```
#if defined(__EMSCRIPTEN__)
#define W_KEYBOARD_SIZE SDL_NUM_SCANCODES
#endif
```

No X11, vimos que há uma diferença entre a tecla física pressionada e o síbolo que ela representa. A mesma diferença aparece aqui. A tecla física no X é chamada de `Keycode` a aqui no SDL é chamada de `Scancode`. O símbolo que ela representa no X era chamado de `Keysym` e aqui é chamada de `Scancode`. Sim, a nomenclatura é confusa pois o `Scancode` significa coisas completamente diferentes nas duas APIs. Usando a nomenclatura do SDL, é assim que detectamos uma tecla sendo pressionada:

Seção: Web Assembly: Tratar Eventos:

```
if(event.type == SDL_KEYDOWN){
  unsigned key = event.key.keysym.scancode;
     <Seção a ser Inserida: Adiciona 'key' à Lista de Teclas Pressionadas>
}
```

Já para tratar o caso de estarmos soltando uma tecla, usamos o código abaixo:

Seção: Web Assembly: Tratar Eventos (continuação):

```
if(event.type == SDL_KEYUP){
  unsigned key = event.key.keysym.scancode;
     <Seção a ser Inserida: Remove 'key' da Lista de Teclas Pressionadas e
Adiciona às Teclas Soltas>
}
```

O código acima é idêntico ao usado no X11, apenas foi necessário mudar os nomes de variáveis seguindo a nomenclatura da estrutura do SDL. Agora precisamos do código que inicializa corretamente o nome de cada uma destas teclas:

Seção: Inicialização de Teclado (continuação):

```
#if defined(__EMSCRIPTEN__)
{
  int i;
  for(i = 0; i < W_KEYBOARD_SIZE; i ++){
    // TODO: When Emscripten implements 'SDL_GetKeyFromScancode', use it:
    unsigned long value = SDL_SCANCODE_TO_KEYCODE(i);
    switch(value){
    case 0: break;
    case SDLK_ESCAPE: W_ESC = i; break;
    case SDLK_BACKSPACE: W_BACKSPACE = i; break;
    case SDLK_TAB: W_TAB = i; break;
    case SDLK_RETURN: W_ENTER = i; break;
    case SDLK_UP:   W_UP   = i; break; case SDLK_DOWN:  W_DOWN =  i; break;
    case SDLK_LEFT: W_LEFT = i; break; case SDLK_RIGHT: W_RIGHT = i; break;
    case SDLK_0: W_0 = i; break;     case SDLK_1: W_1 = i; break;
    case SDLK_2: W_2 = i; break;     case SDLK_3: W_3 = i; break;
    case SDLK_4: W_4 = i; break;     case SDLK_5: W_5 = i; break;
    case SDLK_6: W_6 = i; break;     case SDLK_7: W_7 = i; break;
    case SDLK_8: W_8 = i; break;     case SDLK_9: W_9 = i; break;
    case SDLK_MINUS: W_MINUS = i; break;    case SDLK_PLUS: W_PLUS = i; break;
    case SDLK_F1: W_F1 = i; break;   case SDLK_F2: W_F2 = i; break;
    case SDLK_F3: W_F3 = i; break;   case SDLK_F4: W_F4 = i; break;
    case SDLK_F5: W_F5 = i; break;   case SDLK_F6: W_F6 = i; break;
    case SDLK_F7: W_F7 = i; break;   case SDLK_F8: W_F8 = i; break;
    case SDLK_F9: W_F9 = i; break;   case SDLK_F10: W_F10 = i; break;
    case SDLK_F11: W_F11 = i; break; case SDLK_F12: W_F12 = i; break;
    case SDLK_LSHIFT: W_LEFT_SHIFT = i; break;
    case SDLK_RSHIFT: W_RIGHT_SHIFT = i; break;
    case SDLK_LCTRL: W_LEFT_CTRL = i; break;
    case SDLK_RCTRL: W_RIGHT_CTRL = i; break;
    case SDLK_LALT: W_LEFT_ALT = i; break;
    case SDLK_RALT: W_RIGHT_ALT = i; break;
    case SDLK_SPACE: W_SPACE = i; break;
    case SDLK_a: W_A = i; break;   case SDLK_b: W_B = i; break;
    case SDLK_c: W_C = i; break;   case SDLK_d: W_D = i; break;
    case SDLK_e: W_E = i; break;   case SDLK_f: W_F = i; break;
```

```
    case SDLK_g: W_G = i; break;   case SDLK_h: W_H = i; break;
    case SDLK_i: W_I = i; break;   case SDLK_j: W_J = i; break;
    case SDLK_k: W_K = i; break;   case SDLK_l: W_L = i; break;
    case SDLK_m: W_M = i; break;   case SDLK_n: W_N = i; break;
    case SDLK_o: W_O = i; break;   case SDLK_p: W_P = i; break;
    case SDLK_q: W_Q = i; break;   case SDLK_r: W_R = i; break;
    case SDLK_s: W_S = i; break;   case SDLK_t: W_T = i; break;
    case SDLK_u: W_U = i; break;   case SDLK_v: W_V = i; break;
    case SDLK_w: W_W = i; break;   case SDLK_x: W_X = i; break;
    case SDLK_y: W_Y = i; break;   case SDLK_z: W_Z = i; break;
    case SDLK_INSERT: W_INSERT = i; break;
    case SDLK_HOME: W_HOME = i; break;
    case SDLK_PAGEUP: W_PAGE_UP = i; break;
    case SDLK_DELETE: W_DELETE = i; break;
    case SDLK_END: W_END = i; break;
    case SDLK_PAGEDOWN: W_PAGE_DOWN = i; break;
    default: break;
    }
  }
}
#endif
```

3.2.3. Lendo o Teclado no Windows

No Windows as teclas físicas são chamadas pelo nome de `Scancode` e o símbolo associado a ela é chamado de "código de tecla virtural". O número de diferentes "scancodes" é sempre 256:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define W_KEYBOARD_SIZE 256
#endif
```

Detectar teclas sendo pressionadas é feito de maneira idêntica ao código que já vimos para o X11 e SDL, apenas com modificações menores para abarcar algumas diferenças, e também tendo que mudar o nome das coisas, já que o Windows nomeia de maniera diferente as funções e estruturas:

Seção: Windows: Tratar Eventos de Teclado:

```
if(event.message == WM_KEYDOWN){
  unsigned key = (event.lParam & 0x00ff0000) >> 16;
     <Seção a ser Inserida: Adiciona 'key' à Lista de Teclas Pressionadas>
}
```

Note que uma das diferenças aqui é que para extrair o "scancode" da tecla física, precisamos fazer um pouco de manipulação de bits. Fora isso, o código é o mesmo que já foi visto nas subsubseções anteriores para X11 e SDL. Já o código que detecta se uma tecla foi solta é:

Seção: Windows: Tratar Eventos de Teclado (continuação):

```
if(event.message == WM_KEYUP){
  unsigned key = (event.lParam & 0x00ff0000) >> 16;
     <Seção a ser Inserida: Remove 'key' da Lista de Teclas Pressionadas e
Adiciona às Teclas Soltas>
}
```

E finalmente, inicializamos aqui os nomes das teclas para que eles apontem para a posição correta no nosso vetor de teclas:

Seção: Inicialização de Teclado (continuação):

```
#if defined(_WIN32)
{
```

```
  int i;
  for(i = 0; i < W_KEYBOARD_SIZE; i ++){
    unsigned long value = MapVirtualKey(i, MAPVK_VSC_TO_VK_EX);
    switch(value){
    case 0: break;
    case VK_ESCAPE: W_ESC = i; break;
    case VK_BACK: W_BACKSPACE = i; break;
    case VK_TAB: W_TAB = i; break;
    case VK_RETURN: W_ENTER = i; break;
    case VK_UP:   W_UP   = i; break; case VK_DOWN:  W_DOWN =  i; break;
    case VK_LEFT: W_LEFT = i; break; case VK_RIGHT: W_RIGHT = i; break;
    case '0': W_0 = i; break;     case '1': W_1 = i; break;
    case '2': W_2 = i; break;     case '3': W_3 = i; break;
    case '4': W_4 = i; break;     case '5': W_5 = i; break;
    case '6': W_6 = i; break;     case '7': W_7 = i; break;
    case '8': W_8 = i; break;     case '9': W_9 = i; break;
    case VK_OEM_MINUS: W_MINUS = i; break; case VK_OEM_PLUS: W_PLUS = i; break;
    case VK_F1: W_F1 = i; break;   case VK_F2: W_F2 = i; break;
    case VK_F3: W_F3 = i; break;   case VK_F4: W_F4 = i; break;
    case VK_F5: W_F5 = i; break;   case VK_F6: W_F6 = i; break;
    case VK_F7: W_F7 = i; break;   case VK_F8: W_F8 = i; break;
    case VK_F9: W_F9 = i; break;   case VK_F10: W_F10 = i; break;
    case VK_F11: W_F11 = i; break; case VK_F12: W_F12 = i; break;
    case VK_LSHIFT: W_LEFT_SHIFT = i; break;
    case VK_RSHIFT: W_RIGHT_SHIFT = i; break;
    case VK_LCONTROL: W_LEFT_CTRL = i; break;
    case VK_RCONTROL: W_RIGHT_CTRL = i; break;
    case VK_MENU: W_LEFT_ALT = i; break;
    case VK_RMENU: W_RIGHT_ALT = i; break;
    case VK_SPACE: W_SPACE = i; break;
    case 'A': W_A = i; break;   case 'B': W_B = i; break;
    case 'C': W_C = i; break;   case 'D': W_D = i; break;
    case 'E': W_E = i; break;   case 'F': W_F = i; break;
    case 'G': W_G = i; break;   case 'H': W_H = i; break;
    case 'I': W_I = i; break;   case 'J': W_J = i; break;
    case 'K': W_K = i; break;   case 'L': W_L = i; break;
    case 'M': W_M = i; break;   case 'N': W_N = i; break;
    case 'O': W_O = i; break;   case 'P': W_P = i; break;
    case 'Q': W_Q = i; break;   case 'R': W_R = i; break;
    case 'S': W_S = i; break;   case 'T': W_T = i; break;
    case 'U': W_U = i; break;   case 'V': W_V = i; break;
    case 'W': W_W = i; break;   case 'X': W_X = i; break;
    case 'Y': W_Y = i; break;   case 'Z': W_Z = i; break;
    case VK_INSERT: W_INSERT = i; break;
    case VK_HOME: W_HOME = i; break;
    case VK_PRIOR: W_PAGE_UP = i; break;
    case VK_DELETE: W_DELETE = i; break;
    case VK_END: W_END = i; break;
    case VK_NEXT: W_PAGE_DOWN = i; break;
    default: break;
    }
  }
}
```

```
#endif
```

3.2.4. Código Adicional para Suporte de Teclado

Com o código que foi escrito até agora nós podemos detectar quando uma nova tecla é pressionada ou solta. Tal código não é portável, depende do Sistema Operacional. Mas nós ainda não escrevemos o código que adiciona tal informação à nossa lista de teclas pressionadas e soltas e nem o código que lê informações destas listas para atualizar a estrutura de nosso teclado.

Primeiro de tudo, vamos definir como iremos adicionar uma nova tecla à lista de teclas pressionadas. Lembre-se que a esta altura, em todos os Sistemas Operacionais e ambientes, já escrevemos o código que armazena a informação de qual tecla foi pressionada na variável `key`. E lembre-se que nossa lista de teclas pressionadas representa o fim da lista por uma tecla nula armazenada. Sabendo disso, adicionar uma tecla na lista significa iterar sobre ela até achar uma teclas nula e fazer a substituição:

Seção: Adiciona 'key' à Lista de Teclas Pressionadas:

```
{
  int i;
  for(i = 0; i < 32; i ++){
    if(pressed_keys[i].key == key) // Já tava pressionada
      break;
    if(pressed_keys[i].key == 0){ // Começou a ser pressionada agora
      pressed_keys[i].key = key;
      pressed_keys[i].time = current_time;
      // Atualizando teclado:
      keyboard -> key[key] = 1;
      break;
    }
  }
  if(i == 32) continue; // Ignoring: too many keypresses
}
```

Agora para removermos uma tecla da lista de teclas pressionadas e adicioná-las à lista de teclas soltas, primeiro iteramos sobre s teclas pressionadas até acharmos a que queremos remover. Em seguida movemos todas as teclas das posições seguintes para uma posição anterior, o que efetivamente apaga a tecla desejada da lista. Depois disso, iteramos sobre a lista de teclas soltas até achar uma posição vaga e inserimos nossa nova tecla ali:

Seção: Remove 'key' da Lista de Teclas Pressionadas e Adiciona às Teclas Soltas:

```
{
  int i;
  long stored_time = -1;
  for(i = 0; i < 32; i ++){ // Removing from pressed keys
    if(pressed_keys[i].key == key){
      int j;
      stored_time = pressed_keys[i].time;
      for(j = i; j < 31; j ++){
        pressed_keys[j].key = pressed_keys[j + 1].key;
        pressed_keys[j].time = pressed_keys[j + 1].time;
        if(pressed_keys[j].key == 0)
          break;
      }
      pressed_keys[31].key = 0;
      break;
    }
  }
```

```
  for(i = 0; i < 32; i ++){ // Adding to released keys:
    if(released_keys[i] == 0)
      released_keys[i] = key;
  }
  if(i == 32) // Too many key releases, ignore and clean keyboard
    keyboard -> key[key] = 0;
  else{
    // Updating keyboard struct:
    keyboard -> key[key] = - (long) (current_time - stored_time);
    if(keyboard -> key[key] == 0)
      keyboard -> key[key] = -1; // Press/release in the same frame
  }
}
```

Mas como usar a informação da lista de teclas pressionadas e soltas para atualizar nossa estrutura de teclado? Primeiro, a cada frame, depois de atualizarmos a lista, devemos iterar sobre nossa lista de teclas pressionadas. E para cada uma das teclas, devemos atualizar a contagem de tempo que elas estão sendo pressionadas na nossa estrutura de teclado:

Seção: Depois de Obter Eventos da Janela:

```
{
  int i;
  // Update the information if any key is being pressed:
  keyboard -> key[W_ANY] = (pressed_keys[0].key != 0);
  for(i = 0; i < 32; i ++){
    if(pressed_keys[i].key == 0)
      break;
    if(current_time > pressed_keys[i].time)
      keyboard -> key[pressed_keys[i].key] = (current_time -
pressed_keys[i].time);
  }
}
```

Agora o contador de tempo de teclas pressionadas vai atualizar a cada frame, nao só ficar sempre no valor de 1.

Para quando a tecla é solta, nós já escrevemos código que torna negativo o valor presente no contador de tempo. Mas não escrevemos o código que faz o contador voltar a ser zero no frame seguinte a ele ser solto. Para isso, antes de lermos a entrada da janela e atualizarmos a lista de teclas pressionadas, devemos iterar sobre a lista de teclas que foram soltas no último frame e para cada uma delas ajustar o número de cada tecla para zero e esvaziar a lista de teclas soltas:

Seção: Antes de Obter Eventos da Janela:

```
{
  int i;
  for(i = 0; i < 32; i ++){
    if(released_keys[i] == 0)
      break;
    keyboard -> key[released_keys[i]] = 0;
    released_keys[i] = 0;
  }
}
```

3.3. Lendo o Mouse

Como com as teclas do teclado, também precisamos armazenar uma lista de botões que estão sendo pressionados ou soltos. Mas o número de botões é muito menor. Nossas listas irão apenas armazenar no máximo 4 botões. É pouco realista dar significado a ações que envolvem pressionar mais do que quatro botões de teclado:

Seção: Variáveis Locais (continuação):

```
static struct{
  unsigned button; // Which button was pressed?
  unsigned long long time;       // When it was pressed?
} pressed_buttons[4];
static unsigned released_buttons[4]; // Which buttons were released?
```

Mas ao contrário do teclado, um mouse pode se mover. E se queremos saber não apenas a posição do mouse, mas sua velocidade e aceleração, devemos armazenar como informação adicional a sua última velocidade. Vamos usar estas variáveis para isso:

Seção: Variáveis Locais (continuação):

```
static int last_mouse_dx = 0, last_mouse_dy = 0;
```

Por fim, quando lemos pela primeira vez a posição do mouse, não temos uma leitura anterior de sua posição e velocidade. Então não faz sentido calcularmos neste momento nem a velocidade e nem a aceleração. Já depois de passar o primeiro frame após a inicialização, temos a posição anterior e a atual, o que nos permite obter um valor correto para a velocidade. Mas ainda não podemos saber qual é a aceleração. Somente no segundo frame após a inicialização que podemos calcular todas estas informações. Até lá, é interessante manter como zero valores que não temos como calcular. Por causa disso, vamos usar uma variável adicional que indica se o mouse ainda está inicializando. O valor 3 indica que não foi inicializada e não sabemos nem a posição. O valor 2 indica que ele esta inicializando agora e não temos como saber a velocidade. O valor 1 indica que nã há ainda como obter aceleração. E o valor 0 indica que a inicialização terminou. A variável que armazenará isso é:

Seção: Variáveis Locais (continuação):

```
static int mouse_initialization = 3;
```

Quando inicializamos o mouse, iremos tornar vazia nossa lista de botões pressionados e soltos e também inicializamos como zero todos os botões, bem como a posição, velocidade e aceleração.

Seção: Inicializando ou Reinicializando o Mouse:

```
{
  int i;
  for(i = 0; i < 4; i ++){
    pressed_buttons[i].button = 0;
    released_buttons[i] = 0;
  }
  for(i = 0; i < W_MOUSE_SIZE; i ++)
    mouse -> button[i] = 0;
  mouse -> x = mouse -> y = mouse -> dx = mouse -> dy = mouse -> ddx = mouse ->
ddy = 0;
  last_mouse_dx = last_mouse_dy = 0;
  mouse_initialization = 3;
                <Seção a ser Inserida: Obtém Posição do Mouse>
}
```

3.3.1. Lendo o Mouse no X

No X11, o número de botões suportados em um mouse é 5, e eles são numerados entre 1 e 5. Iremos associar ao número zero o significado de "qualquer botão foi pressionado". Para um mouse, nós não precisamos nos preocupar com a existência de diferentes mapeamentos que dão significados diferentes para cada botão. Assim, podemos definir o número de botões existentes e suas respectivas posições no vetor de botões por meio das seguintes macros:

Seção: Define Macros (continuação):

```
#if defined(__linux__) || defined(BSD)
#define W_MOUSE_SIZE 6
```

```
#define W_MOUSE_LEFT   Button1
#define W_MOUSE_MIDDLE Button2
#define W_MOUSE_RIGHT  Button3
#define W_MOUSE_X1     Button4
#define W_MOUSE_X2     Button5
#endif
```

Sobre o pressionar de botões, para detectá-los devemos detectar eventos to tipo `ButtonPress`. E então ler neles a informação de qual botão do mouse foi pressionado, adicionando-o para nossa lista de botões pressionados:

Seção: X11: Tratar Eventos (continuação):

```
if(event.type == ButtonPress){
  unsigned button = event.xbutton.button;
    <Seção a ser Inserida: Adiciona 'button' à Lista de Botões Pressionados>
}
```

Para detectar um botão sendo solto, devemos checar por eventos do tipo `ButtonRelease` e ler no evento qual dos possíveis cinco botões foram soltos. Em seguida, nós removemos tal botão da lista de botões pressionados e o colocamos na lista de botões soltos:

Seção: X11: Tratar Eventos (continuação):

```
if(event.type == ButtonRelease){
  unsigned button = event.xbutton.button;
    <Seção a ser Inserida: Remove 'botton' da Lista de Botões Pressionados e
Adiciona à Botões Soltos>
}
```

E também temos que detectar movimentos do mouse. Para isso devemos esperar por eventos de movimento do mouse:

Seção: X11: Tratar Eventos (continuação):

```
if(event.type == MotionNotify){
  int x, y;
  x = event.xmotion.x;
  y = (window_size_y - 1) - event.xmotion.y;
          <Seção a ser Inserida: Atualiza Posição do Mouse 'x' e 'y'>
}
```

Note que temos que fazer uma transformação na coordenada y, pois o Xlib considera a posição zero o topo da janela, enquanto aqui tratamos como zero a parte inferior da janela.

Além disso, caso o mouse nunca tenha sido movido, nós ainda temos que saber qual a sua posição inicial. Para estes casos, usamos o código abaixo que chama a função `XQueryPointer`. Esta função retorna uma grande quantidade de informação sobre a posição do mouse relativo não so à nossa janela atual, mas também relacionado à janela acima e à janelas abaixo na hierarquia. Nós ignoramos a maioria de tais informações, exceto pela posição relativa à nossa janela atual.

Seção: Obtém Posição do Mouse:

```
#if defined(__linux__) || defined(BSD)
{
  int x, y;
  Window root_return, child_return;
  int root_x_return, root_y_return;
  unsigned mask_return;
  XQueryPointer(display, window, &root_return, &child_return,
                &root_x_return, &root_y_return, &x, &y, &mask_return);
  // Transformando coordenada 'y':
  y = (window_size_y - 1) - y;
          <Seção a ser Inserida: Atualiza Posição do Mouse 'x' e 'y'>
```

```
}
#endif
```

3.3.2. Lendo o Mouse no Web Assembly

Usando a API SDL fornecida pelo Emscripten, também suportamos um total de cinco botões no Web Assembly, cada um deles numerado entre 1 e 5. Também podemos aqui usar a posição zero para representar "qualquer tecla":

Seção: Define Macros (continuação):

```
#if defined(__EMSCRIPTEN__)
#define W_MOUSE_SIZE 6
#define W_MOUSE_LEFT   SDL_BUTTON_LEFT
#define W_MOUSE_MIDDLE SDL_BUTTON_MIDDLE
#define W_MOUSE_RIGHT  SDL_BUTTON_RIGHT
#define W_MOUSE_X1     SDL_BUTTON_X1
#define W_MOUSE_X2     SDL_BUTTON_X2
#endif
```

Detectamos que um botão do mouse foi pressionado com o código abaixo:

Seção: Web Assembly: Tratar Eventos (continuação):

```
if(event.type == SDL_MOUSEBUTTONDOWN){
  unsigned button = event.button.button;
    <Seção a ser Inserida: Adiciona 'button' à Lista de Botões Pressionados>
}
```

Detectamos que o botão foi solto com o código abaixo:

Seção: Web Assembly: Tratar Eventos (continuação):

```
if(event.type == SDL_MOUSEBUTTONUP){
  unsigned button = event.button.button;
    <Seção a ser Inserida: Remove 'botton' da Lista de Botões Pressionados e
Adiciona à Botões Soltos>
}
```

E detectamos o movimento do mouse abaixo:

Seção: Web Assembly: Tratar Eventos (continuação):

```
if(event.type == SDL_MOUSEMOTION){
  int x, y;
  x = event.motion.x;
  y = (window_size_y - 1) - event.motion.y;
          <Seção a ser Inserida: Atualiza Posição do Mouse 'x' e 'y'>
}
```

Já para obter a posição inicial do mouse, esmo que ele não tenhasido movido, usamos o código abaixo:

Seção: Obtém Posição do Mouse (continuação):

```
#if defined(__EMSCRIPTEN__)
{
  int x, y;
  SDL_GetMouseState(&x, &y);
  // Transformando coordenada 'y':
  y = (window_size_y - 1) - y;
          <Seção a ser Inserida: Atualiza Posição do Mouse 'x' e 'y'>
}
#endif
```

3.3.3. Lendo o Mouse no Windows

No Windows nós não tems números que por padrão são associados a cada um dos diferentes botões do mouse. Ao invés disso, cada botão gera um evento diferente ao invés de gerar um evento de botão pressionado genérico. Mas nós ainda precisamos associar cada botão a um número que será sua posição no vetor de botões de nossa estrutura de mouse. Então nós criamos nossa própria associação entre botões e números com as macros abaixo:

Seção: Define Macros (continuação):

```
#if defined(_WIN32)
#define W_MOUSE_SIZE 6
#define W_MOUSE_LEFT   1
#define W_MOUSE_MIDDLE 2
#define W_MOUSE_RIGHT  3
#define W_MOUSE_X1     4
#define W_MOUSE_X2     5
#endif
```

Como o Windows cria eventos de tipos diferentes para cada um dos botões, e além disso trata como se fôsse um único botão o X1 e X2, apenas diferenciando-os por meio de um dos bits da mensagem, temos trabalho adicional para tratar os eventos de clique de mouse:

Seção: Windows: Tratar Eventos de Mouse:

```
if(event.message == WM_LBUTTONDOWN){
  unsigned button = W_MOUSE_LEFT;
    <Seção a ser Inserida: Adiciona 'button' à Lista de Botões Pressionados>
}
else if(event.message == WM_MBUTTONDOWN){
  unsigned button = W_MOUSE_MIDDLE;
    <Seção a ser Inserida: Adiciona 'button' à Lista de Botões Pressionados>
}
else if(event.message == WM_RBUTTONDOWN){
  unsigned button = W_MOUSE_RIGHT;
    <Seção a ser Inserida: Adiciona 'button' à Lista de Botões Pressionados>
}
else if(event.message == WM_XBUTTONDOWN){
  unsigned button = W_MOUSE_X2;
  if((event.wParam >> 16) & 0x0001){
    unsigned button = W_MOUSE_X1;
  }
    <Seção a ser Inserida: Adiciona 'button' à Lista de Botões Pressionados>
}
```

Para detectar que um dos botões clicados foi solto, também temos que verificar um evento diferente para cada botão tratando os botões X1 e X2 de maneira diferente dos demais:

Seção: Windows: Tratar Eventos de Mouse (continuação):

```
if(event.message == WM_LBUTTONUP){
  unsigned button = W_MOUSE_LEFT;
    <Seção a ser Inserida: Remove 'botton' da Lista de Botões Pressionados e
Adiciona à Botões Soltos>
}
else if(event.message == WM_MBUTTONUP){
  unsigned button = W_MOUSE_MIDDLE;
    <Seção a ser Inserida: Remove 'botton' da Lista de Botões Pressionados e
Adiciona à Botões Soltos>
}
else if(event.message == WM_RBUTTONUP){
```

```
  unsigned button = W_MOUSE_RIGHT;
    <Seção a ser Inserida: Remove 'botton' da Lista de Botões Pressionados e
Adiciona à Botões Soltos>
}
else if(event.message == WM_XBUTTONUP){
  unsigned button = W_MOUSE_X2;
  if((event.wParam >> 16) & 0x0001){
    unsigned button = W_MOUSE_X1;
  }
    <Seção a ser Inserida: Remove 'botton' da Lista de Botões Pressionados e
Adiciona à Botões Soltos>
}
```

E assim é como detectamos o evento no qual o mmouse se move:

Seção: Windows: Tratar Eventos de Mouse (continuação):

```
if(event.message == WM_MOUSEMOVE){
  int x, y;
  x = (event.lParam & 0xffff);
  y = (window_size_y - 1) - (event.lParam >> 16);
          <Seção a ser Inserida: Atualiza Posição do Mouse 'x' e 'y'>
}
```

Como em todos os outros ambientes,mudamos a coordenada para que ela fique relatia ao canto inferior esquerdo da tela, não o canto superior esquerdo como é mais comum.

E para obter a posição inicial do mouse, mesmo que ele nunca tenha se movido, usamos o código:

Seção: Obtém Posição do Mouse (continuação):

```
#if defined(_WIN32)
{
  int x, y;
  POINT point;
  // Obtém coordenadas em relação à tela:
  GetCursorPos(&point);
  // Converte para coordenada em relação à janela
  ScreenToClient(window, &point);
  // Armazena a coordenada, transformando a coordenada 'y':
  x = point.x;
  y = (window_size_y - 1) - point.y;
          <Seção a ser Inserida: Atualiza Posição do Mouse 'x' e 'y'>
}
#endif
```

3.3.4. Código Adicional para Suporte de Mouse

Depois de definirmos nas Subseções anteriores o código específico para cada ambiente para detectarmos cliques e movimentos do mouse, agora iremos lidar com o código comum a todos os ambientes.

Primeiro, iremos explicitar o código que adiciona um botão do mouse pressionado à nossa lista de botões pressionados. Lembre-se que a lista é um vetor de até 4 elementos onde marcamos qualquer fim prematuro da lista com um botão nulo:

Seção: Adiciona 'button' à Lista de Botões Pressionados:

```
{
  int i;
  for(i = 0; i < 4; i ++){
    if(pressed_buttons[i].button == button) // Já tava pressionado
```

```
      break;
    if(pressed_buttons[i].button == 0){ // Começou a ser pressionado agora
      pressed_buttons[i].button = button;
      pressed_buttons[i].time = current_time;
      // Atualizando mouse:
      mouse -> button[button] = 1;
      break;
    }
  }
  if(i == 4) continue; // Ignorando: muitos botões pressionados
}
```

Assim como podemos adicionar botões à esta lista, vamos também removê-los com o código abaixo. E ao remover um botão da lista de botões pressionados, o adicionamos necessariamente à lista de botões soltos:

Seção: Remove 'botton' da Lista de Botões Pressionados e Adiciona à Botões Soltos:

```
{
  int i;
  long stored_time = -1;
  for(i = 0; i < 4; i ++){ // Removendo da lsta de botões pressionados
    if(pressed_buttons[i].button == button){
      int j;
      stored_time = pressed_buttons[i].time;
      for(j = i; j < 3; j ++){
        pressed_buttons[j].button = pressed_buttons[j + 1].button;
        pressed_buttons[j].time = pressed_buttons[j + 1].time;
        if(pressed_buttons[j].button == 0)
          break;
      }
      pressed_buttons[3].button = 0;
      break;
    }
  }
  for(i = 0; i < 4; i ++){ // Adicionando à lista de botões soltos:
    if(released_buttons[i] == 0)
      released_buttons[i] = button;
  }
  if(i == 4) // Muitos botões soltos, ignore e limpe o mouse:
    mouse -> button[button] = 0;
  else{
    // Atualizando estrutura do mouse:
    mouse -> button[button] = - (long) (current_time - stored_time);
    if(mouse -> button[button] == 0)
      mouse -> button[button] = -1; //Apertou/soltou no mesmo frame
  }
}
```

Assim como tivemos que fazer no teclado, depois de atualizarmos a lista de botões pressionados, temos que atualizar o vetor de botões do mouse, atualizando informação de por quanto tempo cada um está sendo pressionado:

Seção: Depois de Obter Eventos da Janela (continuação):

```
{
  int i;
```

```
  // Atualiza se existe qualquer botão sendo pressionado:
  mouse -> button[W_ANY] = (pressed_buttons[0].button != 0);
  for(i = 0; i < 4; i ++){
    if(pressed_buttons[i].button == 0)
      break;
    if(current_time > pressed_buttons[i].time)
      mouse -> button[pressed_buttons[i].button] =
        (current_time - pressed_buttons[i].time);
  }
}
```

Assim como antes de atualizarmos a lista de botões pressionados, temos que esvaziar a lista de botões soltos e marcamos no mouse que aquele boão não está mais sendo pressionado:

Seção: Antes de Obter Eventos da Janela:

```
{
  int i;
  for(i = 0; i < 4; i ++){
    if(released_buttons[i] == 0)
      break;
    mouse -> button[released_buttons[i]] = 0;
    released_buttons[i] = 0;
  }
}
```

E agora mostramos o código para atualizar a posição do mouse, bem como sua velocidade e aceleração (a depender do quanto avançamos na inicialização do mouse). Nós rodamos este código ao detectar o movimento do mouse e também quando lemos pela primeira vez a posição inicial do mouse.

Seção: Atualiza Posição do Mouse 'x' e 'y':

```
{
```

<Seção a ser Inserida: Corrige Posição do Mouse 'x' e 'y' se Forçamos Modo Paisagem>

```
  if(mouse_initialization < 3){ // Se sabemos a posição prévia
    mouse -> dx = (x - mouse -> x);
    mouse -> dy = (y - mouse -> y);
  }
  mouse -> x = x;
  mouse -> y = y;
  if(mouse_initialization < 2){ // Se sabemos a velocidade prévia
    mouse -> ddx = (mouse -> dx - last_mouse_dx);
    mouse -> ddy = (mouse -> dy - last_mouse_dy);
  }
  last_mouse_dx = mouse -> dx;
  last_mouse_dy = mouse -> dy;
  if(mouse_initialization > 0)
    mouse_initialization --;
}
```

Acima nós também podemos precisar corrigir a posição do mouse se a macro `W_FORCE_LANDSCAPE` estiver definida. Neste caso, se a altura de nossa janela for maior que a largura, vamos inverter os eixos x e y. O que significa que vamos girar em 90 graus os eixos x e y:

Seção: Corrige Posição do Mouse 'x' e 'y' se Forçamos Modo Paisagem:

```
#if defined(W_FORCE_LANDSCAPE)
if(window_size_y > window_size_x){
  int tmp = window_size_x - x;
```

```
  x = y;
  y = tmp;
}
#endif
```

3.4. Funções Adicionais de Entrada

Tanto quando vamos ler a nossa entrada pela primeira vez, como quando algo nos interrompe e ficamos um tempo sem atender a novos eventos de entrada, uma grande quantidade de eventos de movimento de mouse e teclas pressionadas pode se acumular e vir ao mesmo tempo. Nestes casos, o melhor a fazer é só ignorar esse fluxo inicial, já que não seremos capazes de tratá-lo corretamente. É para isso que usamos a seguinte função:

Seção: Funções da API (continuação):

```
void _Wflush_window_input(struct _Wkeyboard *keyboard,
                          struct _Wmouse *mouse){
  // Lemos todos os eventos acumulados:
  _Wget_window_input(~0x0, keyboard, mouse);
  // Esvaziamos tudo que temos sobre o teclado:
          <Seção a ser Inserida: Iniciando ou Reiniciando Teclado>
  // E fazemos o mesmo para o mouse
        <Seção a ser Inserida: Inicializando ou Reinicializando o Mouse>
}
```

4. Estrutura Final do Arquivo

O arquivo com o código-fonte de todas as funções definidas neste artigo terá a seguinte forma: (P)

Arquivo: `src/window.c`:

```
#include "window.h"
#include <stdio.h>
                    <Seção a ser Inserida: Cabeçalhos>
                  <Seção a ser Inserida: Funções Locais>
                 <Seção a ser Inserida: Variáveis Globais>
                  <Seção a ser Inserida: Variáveis Locais>
                  <Seção a ser Inserida: Funções da API>
```