Interface de Usuário Weaver

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

Abstract: *This article contains the implementation of the user interface used by Weaver Game Engine. The code presented here are intended to be used when creating buttons, text, menus and other user interface elements. It basically manages shaders, create elements that can be moved, rotated, clicked and can react to mouse hovering. The API presented here are intended to be flexible, the user can extend it and change its behavious registering new functions.*

Resumo: *Este artigo contém a implementação da interface de usuário do Moto de Jogos Weaver. O código apresentado aqui é projetado para ser usado ao criar botões, texto, menus e outros elementos de interface de usuário. O código deste artigo basicamente gerencia "shaders" OpenGL e cria elementos que podem ser movidos, rotacionados, clicados e podem reagir quando o mouse passa sobre eles. A API apresentada aqui é projetada para ser flexível, o usuário pode ampliá-la e mudar seu comportamento registrando novas funções.*

1. Introducão

Uma interface de usuário é como um programa se comunica com o usuário e obtém informações dele. Em um programa típico temos menus, botões, janelas de pop-up e outros elementos de interface típico. Mas em jogos de cmputador a intrface de usuário tende a ser muito mais simples conveitualmente, geralmente formada por alguns menus e por elementos de visualização que dão informações sobre o estado do jogo. POr exemplo, um número no canto da tela indicando quantas vidas o jogador tem.

O fato é que no caso de um jogo de computador, não podemos assumir nada sobre a aparência dos menus, sobre o que eles vã fazer e sobre o que é cada elemento de interface. Um jogo de estratégia poderá ter muuitos menus quando alguém clica com o mouse em uma unidade. Um jogo de plataforma pode ter apenas informações sobre itens equipados e número de vidas. A aparência de tais elementos pode variar enormemente dependendo do estilo do jogo.

A única característica universal que iremos assumir para os elementos de interface com o usuário é que eles sempre devem aparecer sobre o cenário do jogo, não deve ser obscurecido por objetos que fazem parte do mundo do jogo. Tipicamente renderizaremos as interfaces de usuário depois de termos renderizado o mundo de nosso jogo. Além disso, eles não irão interagir com elementos do mundo do jogo. Eles não fazem parte diretamente do mundo que o jogo simula, eles são apenas elementos que dão informações adicionais para o jogador sobre coisas deste mundo, que de outra forma seriam difíceis de serem expressadas.

O nosso objetivo aqui será criar uma API onde o usuário pode criar uma nova interface de usuário invocando a função:

Seção: Declaração de Função (interface.h):

```
struct user_interface *_Wnew_interface(char *filename, char *shader_filename,
                                       float x, float y, float z, float width,
                                       float height);
```

Onde o primeiro argumento é o nome de um arquivo que será aberto e interpretado descrevendo a textura da interface (pode ser um arquivo com uma imagem, por exemplo) e o segundo argumento é um nome de arquivo contendo o shader. Ambos os argumentos podem ser NULL. Um shader nulo significa que um shader padrão será usado. Um arquivo vazio significa que nenhuma textura será preenchida mas usaremos o shader para desenhar a interface na tela. Os demais argumentos são a posição e tamanho inicial da interface na tela.

Podemos definir seus shaders mais facilmente se fornecermos nossa própria biblioteca com definições de funções para o código GLSL. Para isso, podemos passar uma string com código de funções para a função abaixo:

Seção: Declaração de Função (interface.h) (continuação):

```
void _Wset_interface_shader_library(char *source);
```

Uma vez que tenhamos uma interface, poderemos movê-la com a função:

Seção: Declaração de Função (interface.h) (continuação):

```
void _Wmove_interface(struct user_interface *i, float x, float y, float z);
```

Assumimos que a posição de uma interface é a coordenada em pixels de seu centro. O eixo x e y representa a posição horizontal e vertical. O eixo z determina quais interfaces aparecerão na frente se estiverem ocupando as mesmas posições no espaço.

Podemos também rotacioná-la com a função:

Seção: Declaração de Função (interface.h) (continuação):

```
void _Wrotate_interface(struct user_interface *i, float rotation);
```

A função acima recebe o quanto a imagem será rotacionada em relação à orientação padrão usando radianos como medida.

Podemos redimencionar a interface com a função abaixo:

Seção: Declaração de Função (interface.h) (continuação):

```
void _Wresize_interface(struct user_interface *i,
                        float new_width, float new_height);
```

A função abaixo renderiza na tela todas as interfaces disponíveis, sem precisarmos manualmente indicá-las:

Seção: Declaração de Função (interface.h) (continuação):

```
void _Wrender_interface(unsigned long long time);
```

O parâmetro de tempo passado é para que a função tenha noção de quanto tempo em microsegundos se passaram. Isso é útil para renderizar corretamente interfaces com texturas animadas.

Mas renderizar todas as interfaces criadas pode não ser o desejado pelo usuário. Pode ser que queiramos renderizar somente as últimas interfaces criadas à partir de um ponto da história. Para isso, a função abaixo cria uma marcação no nosso histórico de criação de interfaces. E toda vez que pedimos para renderizar, somente as interfaces criadas depois da marcação serão rendrizadas:

Seção: Declaração de Função (interface.h) (continuação):

```
void _Wmark_history_interface(void);
```

Mas e se queremos renderizar somente algumas das interfaces criadas antes da marcação? Neste caso, podemos simplismente criar uma nova interface que na verdade é uma ligação para uma anterior:

Seção: Declaração de Função (interface.h) (continuação):

```
struct user_interface *_Wlink_interface(struct user_interface *i);
```

Já para interagir com todas as interfaces criadas no passado até a última marcação, usamos a função abaixo. Ela irá executar as ações programadas quando um usuário passa o mouse sobre uma interface, retira o mouse sobre uma interface ou clica nela:

Seção: Declaração de Função (interface.h) (continuação):

```
void _Winteract_interface(int mouse_x, int mouse_y, bool left_click,
                          bool middle_click, bool right_click);
```

Mas como apagar iterfaces uma vez que não precisamos mais delas? Para isso fornecemos a função abaixo que apaga todas as interfaces criadas anteriormene até a última marcação na história. Ela também apaga a última marcação existente, voltando o estado de nossa biblioteca até como eraimediatamente antes de tal marcação ser criada. As interfaces qu existirem antes de tal marcação, se existirem, serão novamente renderizadas.

Seção: Declaração de Função (interface.h) (continuação):

```
void _Wrestore_history_interface(void);
```

Tudo isso vai requerer que gerenciemos nosso histórico de interfaces, suas marcações e seus shaders. E isso vai requerer que aloquemos e desaloquemos memória. Há dois tipos de alocações que podemos fazer: coisas permanentes que ficarão alocadas por possivelmente um bom tempo e coisas temporárias que serão rapidamente desalocadas. Vamos então armazenar a função de alocação e desalocação para estes dois casos:

Seção: Variáveis Locais (interface.c):

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

Por padrão, usaremos simplesmente a função de alocação e desalocação da biblioteca padrão. Mas o usuário pode personalizar cada uma destas funções. Além delas, vamos também estabelecer funções personalizáveis que serão executadas imediatamente antes e depois que formos carregar uma nova interface. Inicialmente tais funções serão nulas, mas elas podem ser depois ajustadas pelo usuário:

Seção: Variáveis Locais (interface.c):

```
static void (*before_loading_interface)(void) = NULL;
static void (*after_loading_interface)(void) = NULL;
```

A ideia é que todas as funções persnalizáveis serão definidas durante a inicialização da nossa API:

Seção: Declaração de Função (interface.h) (continuação):

```
#include <stdlib.h> // Define tipo 'size_t'
void _Winit_interface(int *window_width, int *window_height,
                      void *(*permanent_alloc)(size_t),
                      void (*permanent_free)(void *),
                      void *(*temporary_alloc)(size_t),
                      void (*temporary_free)(void *),
                      void (*before_loading_interface)(void),
                      void (*after_loading_interface)(void),
                      ...);
```

É possível inicializar as funções de desalocação como `NULL`. Isso significa que nós não iremos desalocar o que foi alocado. Isso pode ser útil caso seu gerenciador de memória gerencie de alguma forma a coleta de lixo e não quer ter interferência no processo.

A função acima também aceita receber um número variável de argumentos. Primeiro dois ponteiros indicando onde podemos consultar de forma atualizada a largura e altura de nossa janela. Eles serão armazenados aqui:

Seção: Variáveis Locais (interface.c):

```
static int *window_width = NULL, *window_height = NULL;
```

Depois, as seis funções personalizáveis vistas acima. Depois delas, os argumentos adicionais serão uma lista terminada em NULL de uma string seguida de uma função geradora de interfaces. A string representa uma extensão (por exemplo "gif", "jpg" ou outras) e a função que a sucede recebe como argumento um nome de arquivo, as funções de alocação e desalocação e um ponteiro para a interface que deve ser atualizada e preenchida de acordo com o conteúdo do arquivo. Espera-se que a função consiga abrir e interpretar o arquivo e gerar uma nova interface à partir dele. Desta forma, deixamos para o usuário a responsabilidade de fornecer as funções que criam interfaces à partir de arquivos de diferentes formatos.

Como apresentamos uma função de inicialização, vamos precisar também da função de finalização:

Seção: Declaração de Função (interface.h) (continuação):

```
void _Wfinish_interface(void);
```

E isso termina a nossa descrição de todas as funções que suportaremos.

1.1. Programação Literária

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

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

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

Arquivo: `src/interface.h`:

```
#ifndef __WEAVER_INTERFACE
#define __WEAVER_INTERFACE
#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: Inclui Cabeçalhos Gerais (interface.h)>
            <Seção a ser Inserida: Macros Gerais (interface.h)>
          <Seção a ser Inserida: Estrutura de Dados (interface.h)>
          <Seção a ser Inserida: Declaração de Função (interface.h)>
#ifdef __cplusplus
}
#endif
```

```
#endif
```

O código acima mostra a burocracia padrão para definir um cabeçalho para nossa API em C. As duas primeiras linhas mais a última são macros que garantem que esse cabeçalho não será inserido mais de uma vez em uma mesma unidade de compilação. As linhas 3, 4, 5, assim como a penúltima, antepenúltima e a antes da antepenúltima tornam o cabeçalho adequado a ser inserido em código C++. Essas linhas apenas avisam que o que definirmos ali deve ser encarado como código C. Por isso o compilador está livre para fazer otimizações sabendo que não usaremos recursos da linguagem C++, como sobrecarga de operadores. Logo em seguida, inserimos um cabeçalho que nos permite declarar o tipo booleano. E tem também uma parte em vermelha. Note que uma delas é "Declaração de Função (interface.h)", o mesmo nome apresentado no trecho de código mostrado quando descrevemos nossa API antes dessa subseção. Isso significa que aquele código visto antes será depois inserido ali. As outras partes em vermelho representam código que ainda iremos definir nas seções seguintes.

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

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

```
#include "interface.h"
<Seção a ser Inserida: Cabeçalhos Locais (interface.c)>
<Seção a ser Inserida: Macros Locais (interface.c)>
<Seção a ser Inserida: Estrutura de Dados Locais (interface.c)>
<Seção a ser Inserida: Variáveis Locais (interface.c)>
<Seção a ser Inserida: Funções Auxiliares Locais (interface.c)>
<Seção a ser Inserida: Definição de Funções da API (interface.c)>
```

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

1.2. Suportando Múltiplas Threads

A maior parte do código a ser definido neste documento é portável. O único requisito assumido é que um contexto OpenGL está ativo e funcionando. Contudo, há uma parte não-portável que deve ser definida dependendo do sistema em que estamos: o suporte a mutex.

Um mutex é uma estrutura de dados abstrata usada para controlar acesso de múltiplos processos a um recurso em comum. Eles serão tratados de forma diferente dependendo do sistema operacional e ambiente. Devido à seu caráter não-portável, eles serão introduzidos aqui, separados do restante do código.

No Linux e BSD, o Mutex é definido pela biblioteca `pthread` e segue a nomenclatura típica dela. No Windows um Mutex é chamado de "seção critica". No Web Assembly não usaremos Mutex, pois o cóigo não usará múltiplas threads.

Seção: Macros Gerais (interface.h):

```
#if defined(__linux__) || defined(BSD)
#define _MUTEX_DECLARATION(mutex) pthread_mutex_t mutex
#define _STATIC_MUTEX_DECLARATION(mutex) static pthread_mutex_t mutex
#elif defined(_WIN32)
#define _MUTEX_DECLARATION(mutex) CRITICAL_SECTION mutex
#define _STATIC_MUTEX_DECLARATION(mutex) static CRITICAL_SECTION mutex
#elif defined(__EMSCRIPTEN__)
#define _MUTEX_DECLARATION(mutex)
#define _STATIC_MUTEX_DECLARATION(mutex)
#endif
```

Isso significa que no Linux e BSD nós precisamos inserir o cabeçalho da biblioteca `pthread`. No Windows, basta inserir o cabeçalho padrão do windows.

Seção: Inclui Cabeçalhos Gerais (interface.h):

```
#if defined(__linux__) || defined(BSD)
#include <pthread.h>
#elif defined(_WIN32)
#include <windows.h>
#endif
```

No nosso código vamos precisar inicializar cada Mutex que declararmos. Para isso, usaremos a seguinte macro:

Seção: Macros Locais (interface.c):

```
#if defined(__linux__) || defined(BSD)
#define MUTEX_INIT(mutex) pthread_mutex_init(mutex, NULL);
#elif defined(_WIN32)
#define MUTEX_INIT(mutex) InitializeCriticalSection(mutex);
#elif defined(__EMSCRIPTEN__)
#define MUTEX_INIT(mutex)
#endif
```

Para finalizar um Mutex quando não precisarmos mais, usamos a seguinte macro:

Seção: Macros Locais (interface.c):

```
#if defined(__linux__) || defined(BSD)
#define MUTEX_DESTROY(mutex) pthread_mutex_destroy(mutex);
#elif defined(_WIN32)
#define MUTEX_DESTROY(mutex) DeleteCriticalSection(mutex);
#elif defined(__EMSCRIPTEN__)
#define MUTEX_DESTROY(mutex)
#endif
```

Tendo um Mutex, há duas operações que podemos fazer com ele. A primeira é requerer o uso do Mutex. Neste momento, se algum outro processo está usando ele, iremos esperar até que o Mutex esteja livre novamente:

Seção: Macros Locais (interface.c):

```
#if defined(__linux__) || defined(BSD)
#define MUTEX_WAIT(mutex) pthread_mutex_lock(mutex);
#elif defined(_WIN32)
#define MUTEX_WAIT(mutex) EnterCriticalSection(mutex);
#elif defined(__EMSCRIPTEN__)
#define MUTEX_WAIT(mutex)
#endif
```

E finalmente, depois de deixarmos de usar o recurso guardado pelo Mutex, podemos liberar ele com o código abaixo:

Seção: Macros Locais (interface.c):

```
#if defined(__linux__) || defined(BSD)
#define MUTEX_SIGNAL(mutex) pthread_mutex_unlock(mutex);
#elif defined(_WIN32)
#define MUTEX_SIGNAL(mutex) LeaveCriticalSection(mutex);
#elif defined(__EMSCRIPTEN__)
#define MUTEX_SIGNAL(mutex)
#endif
```

2. Estruturas de Dados

Nesta seção descreveremos as quatro principais estruturas de dados que gerenciaremos nesta API: os shaders, as interfaces de usuário, as ligações para interfaces

já criadas e as marcações de tempo.

2.1. Estrutura de Dados de Shader

Para renderizarmos qualquer coisa na placa de vídeo, é necessário que sejam definidas as regas para a renderização. Em que posição iremos colocar o objeto renderizado? Qual a cor que ele terá? Qual a textura? A renderização irá mudar com o tempo ou dependendo do ângulo de visualização? Para responder isso, usam-se programas especiais que são compilados e executados na placa de vídeo. Tais programas são os shaders.

Mas além do programa em si, para cada shader precisamos armazenar algumas variáveis adicionais. Shaders possuem suas próprias variáveis que podem ser modificadas e ajustadas antes deles serem executados. Para cada variável modificável, temos que armazenar o endereço no programa de tais variáveis. Só com esta informação podemos ajustá-las.

Um shader então terá a seguinte forma:

Seção: Estrutura de Dados Locais (interface.c):

```
struct shader {
  int type;
  void *next; // Ponteiro de lista encadeada
  GLuint program; // O programa em si
  /* Aqui começam as variáveis modificáveis de shader */
  GLint attribute_vertex_position; // Posição de cada vértice
  GLint uniform_foreground_color, uniform_background_color; // Cor de
frente/fundo
  GLint uniform_model_view_matrix; // Matriz com tamanho, rotação e translação
  GLint uniform_interface_size; // Tamanho em píxels do objeto renderizado
  GLint uniform_time; // Contador de tempo para texturas animadas
  GLint uniform_integer; // Inteiro arbitrário que pode ser passado
  GLint uniform_texture1; // A textura do objeto a ser renderizado
};
```

Como usamos declarações do OpenGL, como `GLuint`, vamos inserir o cabeçalho OpenGL:

Seção: Inclui Cabeçalhos Gerais (interface.h) (continuação):

```
#if defined(__linux__) || defined(BSD) || defined(__EMSCRIPTEN__)
#include <EGL/egl.h>
#include <GLES2/gl2.h>
#endif
#if defined(_WIN32)
#pragma comment(lib, "Opengl32.lib")
#include <windows.h>
#include <GL/gl.h>
#endif
```

As duas primeiras variáveis na estrutura de dados existem porque iremos armazenar os shaders, assim como outros tipos de estruturas de dados em uma lista encadeada. O tipo serve para identificar o quê exatamente está em cada posição da lista e o ponteiro para o próximo elemento é o que monta a nossa lista encadeada ao ligar cada elemento ao seguinte.

No caso de uma estrutura de shader, o tipo, primeira variável, terá sempre o valor de `TYPE_SHADER`. Todos os tipos diferentes que podem ser armazenados na lista encadeada são:

Seção: Macros Locais (interface.c):

```
#define TYPE_INTERFACE 1 // Uma interface, ou seja, objeto a ser renderizado
```

```
#define TYPE_LINK      2 // Ligação para interface anterior
#define TYPE_MARKING   3 // Marcação no histórico de interfaces criadas
#define TYPE_SHADER    4 // Shader para renderizar interface
```

A variável `program` é a representação do programa de shader compilado. E tudo o que vem depois são variáveis indicando onde cada uma das diferentes variáveis configuráveis de nosso programa de shader pode ser encontrada. Nem todas as variáveis que aparecem necessariamente serão usadas por todos os shaders, mas estas são as variáveis que nossa API suporta. Há variáveis com informações sobre o vértices a serem renderizado, tamanho, posição, cores, textura e algumas coisas mais.

2.2. A Estrutura da Interface

A principal estrutura de dados deste documento é a que armazena informações sobre a interface de usuário. A descrição dela é:

Seção: Estrutura de Dados (interface.h):

```
struct user_interface{
  int type;
  void *next; // Ponteiro de lista encadeada
  float x, y, z;
  float rotation;
  GLfloat _transform_matrix[16];
  float height, width;
  float background_color[4], foreground_color[4];
  int integer;
  bool visible;
  struct shader *shader_program;
  _MUTEX_DECLARATION(interface_mutex);
  /* Funções de interação*/
  bool _mouse_over;
  void (*on_mouse_over)(struct user_interface *);
  void (*on_mouse_out)(struct user_interface *);
  void (*on_mouse_left_down)(struct user_interface *);
  void (*on_mouse_left_up)(struct user_interface *);
  void (*on_mouse_middle_down)(struct user_interface *);
  void (*on_mouse_middle_up)(struct user_interface *);
  void (*on_mouse_right_down)(struct user_interface *);
  void (*on_mouse_right_up)(struct user_interface *);
  /* Atributos abaixo devem ser preenchidos por função de carregamento: */
  GLuint *_texture1;
  bool _loaded_texture;
  bool animate;
  unsigned number_of_frames;
  unsigned current_frame;
  unsigned *frame_duration;
  unsigned long _t;
  int max_repetition;
};
```

Vamos agora descrever o que significa cada elemento desta estrutura. O tipo e o ponteiro para o próximo elemento já foram descritos e são os mesmos que para a estrutura de dados de shaders.

Depois destas duas variáveis, armazenamos as informações sobre posição e tamanho da interface. Temos a sua posição em pixels (eixo x e y), seu "índice-z" (eixo z) que determina a ordem em que as interfaces serão desenhadas e quais aparecerão

na frente das outras, a rotação (em radianos) da interface e a altura e largura (em pixels). Todos estes valores podem ser lidos pelo usuário, mas não devem ser modificados diretamente sem usar as funções adequadas para isso e que srão definidas nas próximas seções. Isso porque para corretamente mudar a posição, tamanho e rotação da interface, precisamos atualizar a sua matriz de transfrmação (logo abaixo), que é como o OpenG e nossos shaders realmente interpretam tais informações. E com relação ao eixo z, se atualizamos ele, também temos que atualizar uma outra lista a ser definida que controla a ordem em que as interfaces são desenhadas na tela.

O atributo de cor de frente e de fundo representa cores no formato RGBA onde cada valor individua é entre 0 e 1. Basicamente tais cores serão sempre passadas para o shader de renderização. Mas não necessariamente o shader irá usar tal informação.

Da mesma forma, o atributo inteiro que vem logo em seguida também é um valor inteiro a ser passado para o shader. Não necessariamente ele será usado, isso irá depender do shader.

O próximo atributo indica se a interface está visível ou não. Se não estiver, ela não será renderizada.

Em seguida há um ponteiro para a estrutura de dados de shader associada à nossa interface. Que mostra como ela será renderizada.

E por fim, é inserida uma declaração de um mutex para que múltiplos processos possam manipular simultaneamente nossa interface.

Em seguida temos as funções de interação que se forem diferentes de `NULL`, serão executadas quando o mouse passa sobre a interface, quando deixa a interface ou quando algum botão do mouse é pressionado ou solto sobre elas. E as interfaces tem também uma variável booleana que indica se o mouse está sobre elas ou não.

Os atributos que vem à seguir desta parte inicialmente sempre começam como sendo 0, falso ou nulo. É de responsabilidade da função de carregamento preencher tais valores enquanto executa. A função de carregamento é aquela que é responsável por ler um arquivo de determinada extensão e gerar corretamente a textura da interface.Funções de carregamento são informadas na inicialização desta API.

Primeiro temos um ponteiro para a textura OpenGL que é passada para o shader. Pode não ter somente uma textura, mas várias. Quando por exemplo temos uma interface obtida através de um GIF animado, por exemplo. Cada frame de animação será uma textura diferente.

Caso exista mais de uma textura, a próxima variável booleana determina se a interface deve ser animada ou não. A variável pode ser modificada à vontade durante a execução do programa para fazer com que a animação pause ou continue.

Os dois próximos atributos representam o número total de frames da animação (será sempre 1 quando não for uma interface animada) e qual o frame atual em que estamos (o primeiro frame é o 0).

Em seguida encontramos um ponteiro que apontará para um vetor alocado que conterá a duração de cada frame em unidades de tempo. Se a interface possuir somente um ou nenhum frame, este ponteiro poderá estar apontando para NULL.

A variável `_t` será usada para realizar a contagem de tempo internamente no caso de termos uma interface animada. Deve iniciar como zero quando a interface é inicializada e sua textura termina de ser carregada. Depois seguiremos incrementando o contador automaticamente para assim sabermos quando o frame atual precisa ser modificado ou não, a depender da duração de cada frame.

Por fim, o último atributo é, para o caso de termos uma interface animada, quantas vezes devemos repetir até o final sua animação. O valor de 0 significa que ela deverá ser repetida para sempre. Um valor positivo coloca u limite no número de repetições. Quando repetirmos a animação pela última vez permitida, ela ficará estacionada no último frame sem voltar novamente para o primeiro.

2.3. Marcações no Histórico

Como mençionado na Introdução, a lógica de quais interfaces estarão acessíveis em um dado momento terá relação com as marcações feitas no nosso histórico de interfaces. Todas as interfaces criadas após a última marcação estão acessíveis, mas aquelas que forem mais antigas que isso não poderão ser acessadas. Elas não serão renderizadas e nem pode-se interagir com elas. Podem ser criadas várias marcações. E a marcação mais recente existente pode ser destruída, o que destrói as interfaces criadas depois dela.

Para que isso seja possível, cada marcação precisa memorizar qual era a marcação anterior a também um ponteiro para o elemento anterior na lista encadeada. Desta forma, quando apagamos uma marcação, podemos substituir a marcação memorizada colocando-a no logar daquela que foi removida e fazer as atualizações devidas de modo mais fácil. Além disso, faremos cada marcação saber também a quantidade de interfaces que foi criada depois dela. É o número de interfaces que estão ativas e existirão enquanto a marcação existir.

Seção: Estrutura de Dados Locais (interface.c) (continuação):

```
struct marking {
  int type;
  void *next; // Ponteiro para próximo na lista encadeada
  void *prev; // Ponteiro para anterior na lista encadeada
  struct marking *previous_marking;
  unsigned number_of_interfaces;
};
```

Note que declaramos essa estrutura de dados internamente no arquivo `interface.c`. Isso porque essa é uma estrutura que será útil somente internamente à nossa API. Não haverá motivos para que o usuário da API precise obter uma marcação destas. Ele irá interagir com a existência delas somente por meio das funções `_Wmark_history_interface` e `_Wrestore_history_interface` descritas na Introdução.

2.4. Ligações para Outras Interfaces

Caso uma interface esteja inacessível por ser mais antiga que a última marcação, é possível criar uma ligação para ela com a função `_Wlink_interface`. A ligação para a interface antiga conta como se fosse uma nova interfae recém-criada e desta forma a interface antiga volta a estar acessível. Criar uma nova ligação significa criar e colocar na lista encadeada a seguinte estrutura:

Seção: Estrutura de Dados Locais (interface.c):

```
struct link {
  int type;
  void *next; // Ponteiro de lista encadeada
  struct user_interface *linked_interface;
};
```

Além das informações necessárias para a lista encadeada como o tipo que identifica a estrutura e o ponteiro que apontará para o elemento da próxima posiçao, a única informação que precisamos armazenar nesta estrutura é um ponteiro para a interface apontada.

3. Inicializacão e Finalizacão da API

O objetivo da inicialização desta API é ajustar todas as funções personalizadas que serão usadas. Já a finalização desaloca as estruturas alocadas necessárias para armazenar algumas destas funções e também ajusta tais funções para valores padrão.

3.1. Inicialização

Aqui nosso objetivo é ajustar as seis funções personalizáveis discutidas na In-

trodução e também receber uma lista de tamanho variável de funções que irão interpretar arquivos com uma dada extensão.

Para armazenar cada uma destas funções que vai na lista de tamanho variável, precisamos da seguinte estrutura:

Seção: Estrutura de Dados Locais (interface.c):

```
struct file_function {
  char *extension;
  void (*load_texture)(void *(*permanent_alloc)(size_t),
                       void (*permanent_free)(void *),
                       void *(*temporary_alloc)(size_t),
                       void (*temporary_free)(void *),
                       void (*before_loading_interface)(void),
                       void (*after_loading_interface)(void),
                       char *source_filename, struct user_interface *target);
};
static unsigned number_of_file_functions_in_the_list = 0;
static struct file_function *list_of_file_functions = NULL;
```

A estrutura basicamente é um par formado por uma função que extrai texturas de um arquivo (e é informada sobre todas as funções personalizadas que devem ser usadas) e por uma extensão de arquivo que ela interpreta. Depois de definir a estrutura, criamos um ponteiro para ela que irá conter uma lista de funções deste tipo conhecidas. O ponteiro começa como uma lista vazia.

Depois que esta lista não estiver mais vazia, isto é, depois da inicialização, nós poderemos percorrê-la para extrair a função correta dada uma extensão por meio da função auxiliar:

Seção: Funções Auxiliares Locais (interface.c):

```
static inline void (*get_loading_function(char *ext))
                           (void *(*permanent_alloc)(size_t),
                            void (*permanent_free)(void *),
                            void *(*temporary_alloc)(size_t),
                            void (*temporary_free)(void *),
                            void (*before_loading_interface)(void),
                            void (*after_loading_interface)(void),
                            char *source_filename, struct user_interface *target){
  unsigned i;
  for(i = 0; i < number_of_file_functions_in_the_list; i ++){
    if(!strcmp(list_of_file_functions[i].extension, ext)){
      return list_of_file_functions[i].load_texture;
    }
  }
  return NULL;
}
```

O código acima é verboso porque é de uma função que retorna um ponteiro para outra função que tem muitos parâmetros. Mas a função acima na verdade tem um único parâmetro: a extensão que deve ser procurada na lista.

Como usamos a função de comparar strings, vamos inserir o cabeçalho padrão com funções envolvendo strings:

Seção: Cabeçalhos Locais (interface.c):

```
#include <string.h>
```

Podemos agora definir a função de inicialização. O que esta função fará será preencher as seis funções personalizáveis básicas, contar quantas outras funções temos que irão interpretar arquivos, alocar o espaço necessário em nossa lista acima

(usando as funções de alocação informadas) e preencher a lista recém-alocada:

Seção: Definição de Funções da API (interface.c):

```
void _Winit_interface(int *window_width_p, int *window_height_p,
                      void *(*new_permanent_alloc)(size_t),
                      void (*new_permanent_free)(void *),
                      void *(*new_temporary_alloc)(size_t),
                      void (*new_temporary_free)(void *),
                      void (*new_before_loading_interface)(void),
                      void (*new_after_loading_interface)(void), ...){
  if(new_permanent_alloc != NULL) /* Parte 1: Pegar 6 Funções + tamanho janela*/
    permanent_alloc = new_permanent_alloc;
  if(new_temporary_alloc != NULL)
    temporary_alloc = new_temporary_alloc;
  permanent_free = new_permanent_free;
  temporary_free = new_temporary_free;
  before_loading_interface = new_before_loading_interface;
  after_loading_interface = new_after_loading_interface;
  window_width = window_width_p;
  window_height = window_height_p;
  {
    int count = -1, i; /* Parte 2: Contar quantas mais funções existem */
    va_list args;
    char *ext;
    va_start(args, new_after_loading_interface);
    do{
      count ++;
      ext = va_arg(args, char *);
      va_arg(args, void (*)(void *(*)(size_t), void (*)(void *),
                            void *(*)(size_t), void (*)(void *),
                            void (*)(void), void (*)(void),
                            char *, struct user_interface *));
    } while(ext != NULL);
    number_of_file_functions_in_the_list = count;
    list_of_file_functions = (struct file_function *)
                               permanent_alloc(sizeof(struct file_function) *
                                               count); // Parte 3: Alocar o resto
    va_start(args, new_after_loading_interface);
    for(i = 0; i < count; i ++){
      list_of_file_functions[i].extension = va_arg(args, char *);
      list_of_file_functions[i].load_texture =
             va_arg(args, void (*)(void *(*)(size_t), void (*)(void *),
                                   void *(*)(size_t), void (*)(void *),
                                   void (*)(void), void (*)(void),
                                   char *, struct user_interface *));
    }
  }
          <Seção a ser Inserida: Inicialização da API de Interface>
}
```

A função acima ficou verbosa graças ao fato de termos um número variável de argumentos contendo ponteiros para funções com mais argumentos. Infelizmente C se torna uma linguagem verbosa nestes casos. O que infelizmente encobre o quão simples foi o que fizemos na inicialização acima.

No fim da função deixamos em vermelho espaço para que operações adicionais

que ainda serão definidas sejam feitas.

O uso de alguns recursos como o acesso aos argumentos por meio da `va_list` requer a inclusão do cabeçalho abaixo:

Seção: Cabeçalhos Locais (interface.c) (continuação):

```
#include <stdarg.h>
```

3.2. Finalização

A função complementar à nossa inicialização é a função de finalização. Se na inicialização terminamos alocando memória para armazenar as funções que interpretam arquivos, na finaliação terminaremos desalocando essa mesma memória. Se na inicialização começamos ajustanto as seis funções personalizadas básicas, na finalização terminaremos retornando tais funções para seus valores padrão.

Seção: Definição de Funções da API (interface.c) (continuação):

```
void _Wfinish_interface(void){
            <Seção a ser Inserida: Finalização da API de Interface>
  if(permanent_free != NULL)
    permanent_free(list_of_file_functions);
  number_of_file_functions_in_the_list = 0;
  permanent_alloc = malloc;
  temporary_alloc = malloc;
  permanent_free = free;
  temporary_free = free;
  before_loading_interface = NULL;
  after_loading_interface = NULL;
}
```

Note que deixamos um espaço acima em vermelho para código adicional que precisarmos colocar na finalização à medida que a API for ficando mais complexa ao definirmos as outras funções nas seções seguintes.

E isso desfaz tudo o que foi feito na inicialização. É possível inicializar e finalizar a API várias vezes sem que isso cause qualquer problema.

4. Shaders

Uma das coisas que precisaremos definir é o shader padrão a ser usado caso o usuário não forneça nenhum personalizado. Isso nos dá uma ótima oportunidade para apresentar em mais detalhes os requisitos para os shaders suportados por esta API e como esperaremos que um shader estejas organizado para que ele seja suportado.

A primeira informação sobre o formato de um shader é qual a linguagem usada para defini-lo. Usaremos a linguagem GLSL, mas existem várias versões diferentes para ela. É a primeira linha de todo código GLSL que define a versão da linguagem. Mas não iremos exigir que o usuário preencha esta informação, nós faremos isso automaticamente.

Para escolher a versão da linguagem GLSL usada em todos os shaders, iremos consultar se existe uma macro definida com o nome `W_GLSL_VERSION`. Essa macro será usada para preencher a primeira linha de todo arquivo GLSL. Caso ela não tenha sido definida e fornecida pelo usuário, então iremos defini-la vom o valor padrão "`#version 100\n`". Este valor padrão significa que o shader usará a linguagem de shader OpenGL ES v1.00. Faremos esse ajuste de macros bem no início de nosso arquivo com a definição das funções:

Seção: Macros Locais (interface.c):

```
#if !defined(W_GLSL_VERSION)
#define W_GLSL_VERSION "#version 100\n"
```

```
#endif
```

Há ao menos dois tipos de shaders que precisaremos definir para cada uma das interfaces diferentes. O primeiro é o shader de vértice, que irá ser processado para cada vértice de nossa interface. O outro é o shader de fragmento que processará cada píxel individual. Mas como indicamos na Inrodução, quando passamos um shader para uma interface, passamos um único arquivo para ela ao invés de dois. Como poderemos representar dois shaders diferentes por meio de um só arquivo?

Isso é graças ao fato de que a linguagem de shader GLSL suporta macros condicionais de pré-processamento, assim como em C. Desta forma, podemos definir macros diferentes se estamos compilando um shader de vértice ou se estamos compilando um shader de fragmento. Isso será feito definindo em cada um dos casos as seguintes macros:

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

```
static const char vertex_shader_macro[] = "#define VERTEX_SHADER\n";
static const char fragment_shader_macro[] = "#define FRAGMENT_SHADER\n";
```

No código-fonte do shader, poderemos então verificar qual das duas macros acima estará definida. Dependendo do caso, simplesmente compilaremos código diferente, da mesma forma como podemos compilar código diferente em um mesmo arquivo para Linux ou Windows em um mesmo arquivo com código C.

Em seguida, precisamos definir a precisão padrão para cada tipo de dados do shader GLSL caso isso não seja declarado explicitamente pelo usuário ao declarar uma variável. Isso é feito usando a palavra-chave `precision`, seguida por um qualificador de precisão (`lowp`, `mediump`, `highp`) e por um tipo e variável. Na dúvida vamos deixar tudo na precisão mais alta. Nas variáveis menos importantes, podemos modificar a precisão para valores menores para obter mais performance.

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

```
static const char precision_qualifier[] = "precision highp float;\n"
                                          "precision highp int;\n";
```

A próxima coisa que será de nosso interesse será inserir bibliotecas GLSL. O usuário poderá definir funções adicionais para facilitar o código GLSL de seus shaders. As bibliotecas ficarão armazenadas na seguinte variável que iniciará como sendo uma string vazia:

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

```
static char *shader_library = "";
```

A variável será modificada por meio da função que vimos na Introdução para incrementar o GLSL com novas funções. Tal função é extremamente simples e somente irá realizar uma atribuição:

Seção: Definição de Funções da API (interface.c) (continuação):

```
void _Wset_interface_shader_library(char *source){
  shader_library = source;
}
```

Mas não poderemos nos esquecer que na função de finalização precisaremos desfazer esta mudança. Do contrário, a API pode ser finalizada e inicializada novamente de modo que a segunda inicialização fique com as mesmas definições de função feitas na primeira:

Seção: Finalização da API de Interface:

```
shader_library = "";
```

Agora vejamos como compilar um shader completo por meio disso. Primeiro vamos inserir o cabeçalho de entrada e saída padrão para podermos imprimir men-

sagens de erro na tela. Os cabeçalhos OpenGL já foram inseridos em `interface.h`.

Seção: Cabeçalhos Locais (interface.c) (continuação):

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

Uma vez que tenhamos o cabeçalho adequado, vamos definir a função, que dado um código-fonte de um shader OpenGL, a compila para um programa de shader completo. Compilar um shader significa criar no OpenGL os dois tipos de shaders (vértice e fragmento), compilar ambos, e ligá-os em um único programa.

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

```
      <Seção a ser Inserida: Funções para Checar Erros de Compilação>
static GLuint compile_shader(const char *source_code){
  GLuint vertex_shader, fragment_shader, program;
  const char *list_of_source_code[5];
  vertex_shader = glCreateShader(GL_VERTEX_SHADER);
  fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
  list_of_source_code[0] = W_GLSL_VERSION;
  list_of_source_code[1] = vertex_shader_macro;
  list_of_source_code[2] = precision_qualifier;
  list_of_source_code[3] = shader_library;
  list_of_source_code[4] = source_code;
  glShaderSource(vertex_shader, 5, list_of_source_code, NULL);
  list_of_source_code[1] = fragment_shader_macro;
  glShaderSource(fragment_shader, 5, list_of_source_code, NULL);
  glCompileShader(vertex_shader);
  if(check_compiling_error(vertex_shader))
    return 0;
  glCompileShader(fragment_shader);
  if(check_compiling_error(fragment_shader))
    return 0;
  program = glCreateProgram();
  glAttachShader(program, vertex_shader);
  glAttachShader(program, fragment_shader);
  glLinkProgram(program);
  if(check_linking_error(program))
    return 0;
  glDeleteShader(vertex_shader);
  glDeleteShader(fragment_shader);
  return program;
}
```

A parte que não é mostrada acima é como fazemos a verificação de que um shader foi compilado com sucesso. Para isso usamos a função abaixo. Ela funciona para os dois tipos de shader. E consiste em consultar se ocorreu um erro de compilação. Em caso afirmativo, ela lê dos logs do OpenGL o que aconteceu de errado e imprime. Note que usamos as funções temporárias de alocação e desalocação para termos espaço para a mensagem de erro. E a função retorna se achou ou não um erro.

Seção: Funções para Checar Erros de Compilação:

```
static bool check_compiling_error(GLuint shader){
  GLint status;
  glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
  if(status == GL_FALSE){
    int info_log_length;
    char *error_msg;
```

```
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &info_log_length);
    error_msg = (char *) temporary_alloc(info_log_length);
    glGetShaderInfoLog(shader, info_log_length, &info_log_length, error_msg);
    fprintf(stderr, "Shader Error: %s\n", error_msg);
    if(temporary_free != NULL)
      temporary_free(error_msg);
    return true;
  }
  return false;
}
```

Checar se houve um erro quando ligamos os dois shaders funciona de maneira semehante. Entretanto, podemos fazer ainda melhor. Podemos tentar validar o shader simulando o seu uso e assim detectar erros adicionais. Contudo, como isso é uma operação demorada e cara, faremos isso só se a macro `W_DEBUG_INTERFACE` esteja definida. Neste caso, assumiremos estar em modo de depuração:

Seção: Funções para Checar Erros de Compilação (continuação):

```
static bool check_linking_error(GLuint program){
  GLint status;
  GLsizei info_log_length;
  char *error_msg;
  glGetProgramiv(program, GL_LINK_STATUS, &status);
  if(status == GL_FALSE){
    glGetProgramiv(program, GL_INFO_LOG_LENGTH, &info_log_length);
    error_msg = (char *) temporary_alloc(info_log_length);
    glGetProgramInfoLog(program, info_log_length, &info_log_length, error_msg);
    fprintf(stderr, "Shader Error: %s\n", error_msg);
    if(temporary_free != NULL)
      temporary_free(error_msg);
    return true;
  }
#if defined(W_DEBUG_INTERFACE)
  glValidateProgram(program);
  glGetProgramiv(program, GL_VALIDATE_STATUS, &status);
  if(status == GL_FALSE){
    glGetProgramiv(program, GL_INFO_LOG_LENGTH, &info_log_length);
    error_msg = (char *) temporary_alloc(info_log_length);
    glGetProgramInfoLog(program, info_log_length, &info_log_length, error_msg);
    fprintf(stderr, "Shader Error: %s\n", error_msg);
    if(temporary_free != NULL)
      temporary_free(error_msg);
    return true;
  }
#endif
  return false;
}
```

E isso termina a descrição de como compilamos um novo shader e imprimimos erros caso hajam problemas no código-fonte.

Podemos agora apresentar o código do shader padrão a ser usado caso o usuário não passe um shader personalizado. O objetivo do shader padrão deve ser apenas mostrar a textura associada à interface caso exista uma textura.

O código-fonte do shader padrão será armazendado em uma constante:

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

```
static const char default_shader_source[] = ""
    <Seção a ser Inserida: Atributos, Uniformes e Variantes de Shader>
"#if defined(VERTEX_SHADER)\n"
            <Seção a ser Inserida: Shader de Vértice Padrão>
"#else\n"
          <Seção a ser Inserida: Shader de Fragmento Padrão>
"#endif\n"
                                        "";
```

O código-fonte serão literais strings aramazenadas na variável constante acima. No caso do shader de vértice, tudo o que faremos por padrão com cada vértice será multiplicar sua posição pela matriz de modelo e visualização que armazenará coisas como seu tamanho, posição e rotação. Além disso, especificamos a posição em nossa textura associada a cada vértice.

Para entender o código, considere que iremos representar toda interface usando sempre a mesma sequência de vértices: $(0,0,0),(1,0,0),(1,1,0),(0,1,0)$. Eles são declarados abaixo:

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

```
static const float interface_vertices[12] = {0.0, 0.0, 0.0,
                                             1.0, 0.0, 0.0,
                                             1.0, 1.0, 0.0,
                                             0.0, 1.0, 0.0};
static GLuint interface_vbo;
```

A ordem em que declaramos eles é importante. Se seguirmos os vértices na ordem em que são declarados, percorremos um quadrado no sentido anti-horário. Essa é a forma de indicarmos que estamos vendo a parte frontal da face de nossa interface, não a parte traseira. Devemos estar preparados para caso o usuário confiure o OpenGL para não mostrar a parte traseira dos vértices, uma otimização bastante comum e necessária em muitos contextos. Especificando esta ordem garante que nossa interface estará visível mesmo neste caso.

Estes vértices serão carregados na placa de vídeo a à partir daí poderão ser acessados por meio do VBO, ou "vertex buffer object" correspondente. Carregar os vértices para a placa de vídeo é feito na inicialização:

Seção: Inicialização da API de Interface (continuação):

```
glGenBuffers(1, &interface_vbo);
glBindBuffer(GL_ARRAY_BUFFER, interface_vbo);
// Enviando os vértices para a placa de vídeo:
glBufferData(GL_ARRAY_BUFFER, sizeof(interface_vertices), interface_vertices,
             GL_STATIC_DRAW);
```

Na finalização devemos destruir o VBO que criamos:

Seção: Finalização da API de Interface (continuação):

```
glDeleteBuffers(1, &interface_vbo);
```

De qualquer forma, isso significa que a interface originalmente será sempre um retângulo que irá cobrir a tela inteira, pois no OpenGL o valor de 1 vai ser sempre tanto a altura como largura da tela. Para transformar esse retângulo fixo em coisas de diferentes tamanhos, posições e rotações nós usamos diferentes matrizes de modelo e visualização e multiplicamos a posição fixa pela matriz que será diferente para cada interface.

Seção: Shader de Vértice Padrão:

```
"void main(){\n"
"  gl_Position = model_view_matrix * vec4(vertex_position, 1.0);\n"
```

```
"  texture_coordinate = vec2(vertex_position.x, vertex_position.y);\n"
"}\n"
```

No shader de fragmento, o que faremos para cada pixel é desenhar o pixel da textura associado à esta posição.

Seção: Shader de Fragmento Padrão:

```
"void main(){\n"
"  vec4 texture = texture2D(texture1, texture_coordinate);\n"
"  gl_FragData[0] = texture;\n"
"}\n"
```

Agora temos que definir para os shaders seus atributos, uniformes e variantes. Os atributos são variáveis somente-leitura especificados para cada vértice. O único atributo que teremos é a posição do vértice, em coordenada (x, y, z). A coordenada x e y irá variar entre 0 e 1 enquanto $z = 0$ como descrito antes. Atributos só podem ser declarados no shader de vértice.

Seção: Atributos, Uniformes e Variantes de Shader:

```
"#if defined(VERTEX_SHADER)\n"
"attribute vec3 vertex_position;\n"
"#endif\n"
```

Uniformes são variáveis que também são passadas para vértices, mas eles não irão nunca mudar entre um vértice e outro e nunca precisarão também ser interpolados. Iremos manter como uniformes a cor de frente e de fundo, ma matriz de modelo e visualização, o tamanho do objeto em pixels, o tempo atual em segundos, o inteiro associado a cada interface e também sua textura:

Seção: Atributos, Uniformes e Variantes de Shader (continuação):

```
"uniform vec4 foreground_color, background_color;\n"
"uniform mat4 model_view_matrix;\n"
"uniform vec2 interface_size;\n"
"uniform float time;\n"
"uniform int integer;\n"
"uniform sampler2D texture1;\n"
```

Por fim, temos as variantes. Estas variáveis podem ser modificadas e preenchidas pelo shader de vértice. Seu valor interpolado será passado parao shader de fragmento. É aqui que declaramos a coordenada em textura que iremos usar.

Seção: Atributos, Uniformes e Variantes de Shader (continuação):

```
"varying mediump vec2 texture_coordinate;\n"
```

Note que nós não usamos todas as variáveis definidas no shader padrão. Durante a compilação muitas delas serão simplesmente ignoradas e descartadas como otimização. Mas declaramos mesmo as que não usamos apenas para informar quais são as variáveis que shaders personalizados definidos pelo usuário pode usar.

Falando em shaders personalizados pelo usuário, eles serão fornecidos pela função `_Wnew_interface` na forma de uma string com o nome do arquivo em que ela estará. Para tal caso, vamos também fornecer uma função que cria um novo shader à partir do conteúdo de um arquivo cujo caminho é passado como argumento:

Seção: Funções Auxiliares Locais (interface.c):

```
static GLuint compile_shader_from_file(const char *filename){
  char *buffer;
  size_t source_size, ret;
  FILE *fp;
  GLuint shader_program;
  fp = fopen(filename, "r");
```

```
  if(fp == NULL)  return 0;
  // Vai pro fim do arquivo para ler o tamanho e volta pro começo:
  fseek(fp, 0, SEEK_END);
  source_size = ftell(fp);
  // Aloca e lê buffer
  buffer = (char *) temporary_alloc(sizeof(char) * (source_size + 1));
  if(buffer == NULL) return 0;
  do{
    rewind(fp);
    ret = fread(buffer, sizeof(char), source_size, fp);
  } while(feof(fp) && !ferror(fp) && ret / sizeof(char) == source_size);
  buffer[source_size] = '\0';
  shader_program = compile_shader(buffer);
  if(temporary_free != NULL) temporary_free(buffer);
  return shader_program;
}
```

Uma última preocupação que teremos aqui é o que renderizar como textura padrão quando um shader não tem textura alguma. Para isso, vamos criar uma textura de apenas 1 pixel branco:

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

```
static GLuint default_texture;
```

Criamos tal textura branca na inicialização. Isso é feito gerando a textura no OpenGL, a associando à textura bidimensional atual e especificando seu único pixel branco.

Seção: Inicialização da API de Interface (continuação):

```
{
  GLubyte pixels[3] = {255, 255, 255};
  glGenTextures(1, &default_texture);
  glBindTexture(GL_TEXTURE_2D, default_texture);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 1, 1, 0, GL_RGB, GL_UNSIGNED_BYTE,
               pixels);
}
```

E na finalização descartamos a textura:

Seção: Finalização da API de Interface (continuação):

```
glDeleteTextures(1, &default_texture);
```

5. A Matriz de Modelo-Visualizacão

Como vimos, toda interface será representada por quatro vértices fixos de tamanho 1. O que modificará na prática o tamanho, posição e rotação de nossos vértices será a matriz de modelo-visualização que usaremos para cada interface.

Para entender a matriz, primeiro lembre-se que quando lidamos com ela no nosso shader de vértice padrão, cujo código mostramos anteriormente, nós convertemos cada vértice como um vértice em 4 dimensões ao invés de duas ou três. O trecho de código `vec4vertex_position, 1.0)` basicamente pega uma coordenada em três dimensões e adiciona uma quarta dimensão sempre igual a 1.

Nós fazemos isso porque somente em quatro dimensões é possível representar todas as operações necessárias como a rotação, translação e escala de nossa interface somente por meio de multiplicação de matrizes. Ou seja, somente se usarmos quatro dimensões tais operações são lineares. Placas de vídeo são bastante rápidas quando estão fazendo esse tipo de operação, enquanto fazer outras operações gera impacto maior na performance.

Antes de chegar ao formato final da matriz modelo-visualização, vamos primeiro observar as suas diferentes partes. Primeiro, vamos assumir que temos um vetor de 4 dimensões que multiplicará nossa matriz. Se estamos apenas interessados na translação de seus pontos, movendo eles sem modificar o tamanho, podemos multiplicá-lo por uma matriz com a forma abaixo:

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1x_0+0y_0+0z_0+x \\ 0x_0+1y_0+0z_0+y \\ 0x_0+0y_0+1z_0+0 \\ 0x_0+0y_0+0z_0+1 \end{bmatrix} = \begin{bmatrix} x_0+x \\ y_0+y \\ z_0 \\ 1 \end{bmatrix}$$

Já para mudar o tamanho de uma interface, tanto a largura como a altura, assumindo que ela está centralizada na origem $(0, 0, 0, 1)$, podemos multiplicar o vetor pela seguinte matriz de escala::

$$\begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} wx_0+0y_0+0z_0+0 \\ 0x_0+hy_0+0z_0+0 \\ 0x_0+0y_0+1z_0+0 \\ 0x_0+0y_0+0z_0+1 \end{bmatrix} = \begin{bmatrix} wx_0 \\ hy_0 \\ z_0 \\ 1 \end{bmatrix}$$

E finalmente, no caso da rotação, para rotacionar θ radianos, nós podemos usar a seguinte matriz que produzirá o resultado correto de acordo com a fórmula trigonométrica da rotação:

$$\begin{bmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ \sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_0cos(\theta)-y_0sin(\theta)+0z_0+0 \\ x_0sin(\theta)+y_0cos(\theta)+0z_0+0 \\ 0x_0+0y_0+1z_0+0 \\ 0x_0+0y_0+0z_0+1 \end{bmatrix} = \begin{bmatrix} x_0cos(theta)-y_0sin(\theta) \\ x_0sin(\theta)+y_0cos(\theta) \\ z_0 \\ 1 \end{bmatrix}$$

Para formar a matriz de modelo-visualização nós iremos multiplicar essas três matrizes dependendo dos valores exator que precisamos ajustar o tamanho, posição e rotação de cada interface. Contudo, a ordem em que a multiplicação de matrizes ocorre faz diferença no resultado. A ordem correta na qual devemos fazer as operações é:

1. Devemos centralizar o quadrado na origem do OpenGL. (Matriz A)
2. Rotacionamos ela o ângulo necessário. (Matriz B)
2. Aumentamos ou diminuimos o tamanho da interface. (Matriz C)
4. Fazemos a translação dela para a posição desejada, levando em conta medimos nossas coordenadas de modo diferente que o OpenGL. (Matrix D)

$$D(C(B(Av))) = (((DC)B)A)v$$

Se usarmos outra ordem, coisas podem dar errado. Por exemplo, ao invés de rotacionar a interface usando o seu centro como eixo, usaríamos o seu canto inferior direito ou alguma outra posição como eixo.

Ao invés de toda vez termos que multiplicar toda vez essas quatro matrizes, podemos calcular agora qual vai ser o formato da matriz que resulta de toda essa multiplicação. Essa matriz será a matriz de modelo-visualização final. Por exemplo, multiplicar as matrizes D e C resulta em:

$$DC = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} w & 0 & 0 & x \\ 0 & h & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Agora se multiplicarmos este resultado pela matriz B, obtemos:

$$DCB = \begin{bmatrix} w & 0 & 0 & x \\ 0 & h & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ \sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} w{\cdot}cos(\theta) & -w{\cdot}sin(\theta) & 0 & x \\ h{\cdot}sin(\theta) & h{\cdot}cos(\theta) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

E finalmente, multiplicamos este resultado pela matriz A, que centraliza em $(0, 0, 0, 1)$ toda interface. Note que originalmente a interface não-multiplicada pela matriz de modelo-visuzlização está centralizada em $(1/2, 1/2, 0, 1)$. Então a matriz

A é simplesmente uma matriz de translação com valores constantes deslocando 1/2 para a esquerda e 1/2 para baixo:

$$\begin{bmatrix} w{\cdot}cos(\theta) & -w{\cdot}sin(\theta) & 0 & x \\ h{\cdot}sin(\theta) & h{\cdot}cos(\theta) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -1/2 \\ 0 & 1 & 0 & -1/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} w{\cdot}cos(\theta) & -w{\cdot}sin(\theta) & 0 & -w/2{\cdot}cos(\theta)+w/2{\cdot}sin(\theta)+x \\ h{\cdot}sin(\theta) & h{\cdot}cos(\theta) & 0 & -h/2{\cdot}sin(\theta)-h/2{\cdot}cos(\theta)+y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

O formato acima então é a forma final da matriz de modelo-visualização de toda interface que iremos armazenar.

Entretanto, os valores (x, y) e (w, h) acima são valores nas coordenadas OpenGL. Mas nós medimos as coordenadas de forma difeente. Para nós a origem não é o centro da tela, mas o canto esquerdo inferior da tela. Par nós, a altura e largura da nossa área é o valor em píxels, e não sempre o mesmo valor constante de 2 como para o OpenGL. Então a função que irá preencher esta matriz precisa também realizar a conversão.

No caso, se temos um ponteiro para uma interface chamado `i`, a conversão segue as seguintes regras:

Seção: Conversão de Coordenadas e Tamanho:

```
x = 2.0 * (i -> x) / (*window_width) - 1.0;
y = 2.0 * (i -> y) / (*window_height) - 1.0;
w = 2.0 * (i -> width) / (*window_width);
h = 2.0 * (i -> height) / (*window_height);
```

Usando a conversão acima, a função abaixo preenche a matriz de modelo-visualização pela primeira vez:

Seção: Funções Auxiliares Locais (interface.c):

```
static void initialize_model_view_matrix(struct user_interface *i){
  GLfloat x, y, w, h;
        <Seção a ser Inserida: Conversão de Coordenadas e Tamanho>
  GLfloat cos_theta = cos(i -> rotation);
  GLfloat sin_theta = sin(i -> rotation);
  /* Primeira Coluna */
  i -> _transform_matrix[0] = w * cos_theta;
  i -> _transform_matrix[1] = h * sin_theta;
  i -> _transform_matrix[2] = 0.0;
  i -> _transform_matrix[3] = 0.0;
  /* Segunda Coluna */
  i -> _transform_matrix[4] = -w * sin_theta;
  i -> _transform_matrix[5] = h * cos_theta;
  i -> _transform_matrix[6] = 0.0;
  i -> _transform_matrix[7] = 0.0;
  /* Terceira Coluna */
  i -> _transform_matrix[8] = 0.0;
  i -> _transform_matrix[9] = 0.0;
  i -> _transform_matrix[10] = 1.0;
  i -> _transform_matrix[11] = 0.0;
  /* Quarta Coluna */
  i -> _transform_matrix[12] = -w/2 * cos_theta + w/2 * sin_theta + x;
  i -> _transform_matrix[13] = -h/2 * sin_theta - h/2 * cos_theta + y;
  i -> _transform_matrix[14] = 0.0;
  i -> _transform_matrix[15] = 1.0;
}
```

O uso das funções de seno e cosseno requer que coloquemos o cabeçalho de funções matemáticas:

Seção: Cabeçalhos Locais (interface.c) (continuação):

```
#include <math.h>
```

6. Gerenciando as Estruturas de Dados

Vimos que as estruturas de dados que nossa API irá gerenciar é uma estrutura para interfaces, outra para ligação para interface existente, uma terceira para fazer marcação no histórico e por fim, uma lista encadeada que manterá a sequência e o histórico de criação para os três tipos de interface anteriores.

A nossa lista encadeada pode ser acessada e percorrida através de dois ponteiros. O primeiro apontará para a última estrutura criada e o segundo apontará para a última marcação no hstórico criada. Também iremos usar um mutex para garantir que somente uma thread possa modificar a lista encadeada:

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

```
static void *last_structure = NULL;
static struct marking *last_marking = NULL;
_STATIC_MUTEX_DECLARATION(linked_list_mutex);
```

O mutex que controla acesso à lista encadeada precisa ser criado na inicialização:

Seção: Inicialização da API de Interface (continuação):

```
MUTEX_INIT(&linked_list_mutex);
```

E ele será destruído na finalização:

Seção: Finalização da API de Interface (continuação):

```
MUTEX_DESTROY(&linked_list_mutex);
```

6.1. Criando e Destruindo Estruturas de Shader

Na Seção 4 nós mostramos o código-fonte do shader padrão e também mostramos funções que fazem a compilação do código-fonte de um shader. Mas nós não mostramos como aquelas funções são intergradas na função que constrói a estrutura de shader. Faremos isso nesta seção.

A nossa API não tem nenhuma função definida para gerar um novo shader por si só. Ao invés disso, nós iremos criar um novo shader por meio de uma função interna toda vez que formos criar uma nova interface e o usuário passar um shader personalizado. Ou seja, quando o usuário da API invoca `_Wnew_interface` com o segundo parâmetro, que é o do arquivo com código-fonte de shader, diferente de `NULL`.

Também precisamos gerar a estrutura para o shader padrão, cujo código apresentamos na Seção 4.

O shader padrão será armazenado no seguinte ponteiro:

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

```
struct shader *default_shader;
```

Vamos definir agora uma função interna que irá gerar uma nova estrutura de shader. Ela recebe como argumento a string com nome de arquivo onde está o código-fonte do shader. A função retorna o ponteiro para a estrutura gerada e também a insere na lista encadeada. Exceto se o nome de arquivo passado for `NULL`. Neste caso, nós apenas geramos uma estrutura à partir do código-fonte do shader padrão definido na Seção 4. Ele não é armazenado na lista encadeada.

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

```
static struct shader *new_shader(char *shader_source){
  struct shader *new = (struct shader *) permanent_alloc(sizeof(struct shader));
  if(new != NULL){
```

```
    new -> type = TYPE_SHADER;
    new -> next = NULL;
    if(shader_source == NULL)
      new -> program = compile_shader(default_shader_source);
    else
      new -> program = compile_shader_from_file(shader_source);
    new -> attribute_vertex_position = glGetAttribLocation(new -> program,
                                                           "vertex_position");
    new -> uniform_foreground_color =  glGetUniformLocation(new -> program,
                                                            "foreground_color");
    new -> uniform_background_color =  glGetUniformLocation(new -> program,
                                                            "background_color");
    new -> uniform_model_view_matrix = glGetUniformLocation(new -> program,
                                                            "model_view_matrix");
    new -> uniform_interface_size = glGetUniformLocation(new -> program,
                                                         "interface_size");
    new -> uniform_time = glGetUniformLocation(new -> program, "time");
    new -> uniform_integer = glGetUniformLocation(new -> program, "integer");
    new -> uniform_texture1 = glGetUniformLocation(new -> program, "texture1");
    if(shader_source != NULL){ // Insere na lista encadeada:
      MUTEX_WAIT(&linked_list_mutex); // Preparando mutex
      if(last_structure != NULL)
        ((struct user_interface *) last_structure)-> next = (void *) new;
      last_structure = (void *) new;
      MUTEX_SIGNAL(&linked_list_mutex);
    }
  }
  return new;
}
```

Basicamente o código acima usa a função da Seção 4 para compilar o shader, e uma vez que tem o shader compilado, armazena na estrutura de shader a posição de cada uma das variáveis do shader compilado que podem ser modificadas antes de executar o shader.

Lembrando que o shader padrão precisa ser compilado e criado na Inicialização. O código para isso é:

Seção: Inicialização da API de Interface (continuação):

```
default_shader = new_shader(NULL);
```

Para destruir um shader criado, só precisamos informar que não usaremos mais o programa de shader compilado e que ele deve ser apagado. Em seguida, desalocamos a estrutura de shader que o armazenava:

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

```
static void destroy_shader(struct shader *shader_struct){
  glDeleteProgram(shader_struct -> program);
  if(permanent_free != NULL)
    permanent_free(shader_struct);
}
```

O shader padrão deve ser destruído na finalização da API:

Seção: Finalização da API de Interface (continuação):

```
destroy_shader(default_shader);
```

6.2. Criando e Destruindo Interface

Vimos que a criação de novas interfaces se dará pela função `_Wnew_interface`. Esta função deverá alocar a nova interface, executar as funções adequadas para inicializá-la e incluí-la na lista encadeada atualizando os ponteiros de maneira adequada. A função será definida como:

Seção: Definição de Funções da API (interface.c) (continuação):

```
struct user_interface *_Wnew_interface(char *filename, char *shader_filename,
                                       float x, float y, float z, float width,
                                       float height){
  struct user_interface *new_interface;
  void (*loading_function)(void *(*permanent_alloc)(size_t),
                           void (*permanent_free)(void *),
                           void *(*temporary_alloc)(size_t),
                           void (*temporary_free)(void *),
                           void (*before_loading_interface)(void),
                           void (*after_loading_interface)(void),
                           char *source_filename, struct user_interface *target);
  int i;
  new_interface = permanent_alloc(sizeof(struct user_interface));
  if(new_interface != NULL){
    new_interface -> type = TYPE_INTERFACE;
    new_interface -> next = NULL;
    new_interface-> x = x;
    new_interface -> y = y;
    new_interface -> z = z;
    new_interface -> rotation = 0;
    new_interface -> width = width;
    new_interface -> height = height;
    for(i = 0; i < 4; i ++){
      new_interface -> background_color[i] = 0.0;
      new_interface -> foreground_color[i] = 0.0;
    }
    new_interface -> integer = 0;
    new_interface -> visible = true;
    initialize_model_view_matrix(new_interface);
    if(shader_filename != NULL)
      new_interface -> shader_program = new_shader(shader_filename);
    else
      new_interface -> shader_program = default_shader;
    new_interface -> _texture1 = NULL;
    if(filename != NULL) // Ainda tem que carregar a textura:
      new_interface -> _loaded_texture = false;
    else // Sem textura para ser carregada:
      new_interface -> _loaded_texture = true;
    new_interface -> animate = false;
    new_interface -> number_of_frames = 0;
    new_interface -> current_frame = 0;
    new_interface -> frame_duration = NULL;
    new_interface -> _t = 0;
    new_interface -> max_repetition = 0;
    MUTEX_INIT(&(new_interface -> interface_mutex));
    new_interface -> _mouse_over = false;
    new_interface -> on_mouse_over = NULL;
```

```
    new_interface -> on_mouse_out = NULL;
    new_interface -> on_mouse_left_down = NULL;
    new_interface -> on_mouse_left_up = NULL;
    new_interface -> on_mouse_middle_down = NULL;
    new_interface -> on_mouse_middle_up = NULL;
    new_interface -> on_mouse_right_down = NULL;
    new_interface -> on_mouse_right_up = NULL;
    MUTEX_WAIT(&linked_list_mutex); // Inserindo na lista encadeada
    if(last_structure != NULL)
      ((struct user_interface *) last_structure)-> next = (void *) new_interface;
    last_structure = (void *) new_interface;
    last_marking -> number_of_interfaces ++;
    MUTEX_SIGNAL(&linked_list_mutex);
    if(filename != NULL){ // Get and run loading function:
      char *ext;
      for(ext = filename; *ext != '\0'; ext ++);
      for(; *ext != '.' && ext != filename; ext --);
      if(*ext == '.'){
        ext ++;
        loading_function = get_loading_function(ext);
        if(loading_function != NULL)
          loading_function(permanent_alloc, permanent_free, temporary_alloc,
                           temporary_free, before_loading_interface,
                           after_loading_interface, filename, new_interface);
      }
    }
  }
  return new_interface;
}
```

Apesar de longa, tudo o que a função acima faz é alocar a interface com a função de alocação permanente configurada, inicializar os valores iniciais, inseri-la na lista encadeada e executar a função adequada para carrega a textura dependendo da extensão do arquivo.

Destruir uma interface significa checar algumas de suas variáveis específicas para vcer se tem um valor diferente de `NULL`. Se for o caso, isso significa que a função de carregamento alocou e preencheu alguns de seus valores. Como por exemplo, as variáveis que armazenam texturas, que armazenam o tempo para cada frame de animação se for uma textura animada. A função de destruição de interfaces desaloca primeiro essas coisas antes de desalocar a estrutura de interface em si. Ela também deve finalizar o mutex da interface.

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

```
static void destroy_interface(struct user_interface *interface_struct){
  if(interface_struct -> _texture1 != NULL){
    glDeleteTextures(interface_struct -> number_of_frames,
                     interface_struct -> _texture1);
    if(permanent_free != NULL)
      permanent_free(interface_struct -> _texture1);
  }
  if(interface_struct -> frame_duration != NULL && permanent_free != NULL)
    permanent_free(interface_struct -> frame_duration);
  MUTEX_DESTROY(&(interface_struct -> interface_mutex));
  if(permanent_free != NULL)
    permanent_free(interface_struct);
```

```
}
```

6.3. Criando e Destruindo Ligação para Interface Existente

Como vimos, além de criar uma nova interface podemos apenas criar uma ligação para uma interface existente, que ela irá se comportar como uma interface recém-criada para fins de interação. Uma ligação será apenas um ponteiro para outra interface em nossa lista encadeada. E elas são criadas com a função `_Wlink_interface`. A definição da função é:

Seção: Definição de Funções da API (interface.c) (continuação):

```
struct user_interface *_Wlink_interface(struct user_interface *i){
  struct link *new_link = permanent_alloc(sizeof(struct link));
  if(new_link == NULL)
    return NULL;
  new_link -> type = TYPE_LINK;
  new_link -> next = NULL;
  new_link -> linked_interface = i;
  MUTEX_WAIT(&linked_list_mutex); // Inserindo na lista encadeada
  if(last_structure != NULL)
    ((struct user_interface *) last_structure)-> next = (void *) new_link;
  last_structure = (void *) new_link;
  last_marking -> number_of_interfaces ++;
  MUTEX_SIGNAL(&linked_list_mutex);
  return i;
}
```

Destruir uma interface de ligação significa apenas executar a função de desalocação e por isso não criaremos uma função auxiliar para isso.

6.4. Criando Marcação no Histórico de Interfaces

Para criar uma nova marcação no histórico de interfaces, o usuário deve usar a função `_Wmark_history_interface`. Depois da marcação, todas as interfaces criadas antes ficarão inacessíveis até que a marcação seja removida.

Seção: Definição de Funções da API (interface.c) (continuação):

```
void _Wmark_history_interface(void){
  struct marking *new_marking = permanent_alloc(sizeof(struct marking));
  if(new_marking != NULL){
    new_marking -> type = TYPE_MARKING;
    new_marking -> next = NULL;
    new_marking -> previous_marking = last_marking;
    new_marking -> number_of_interfaces = 0;
    MUTEX_WAIT(&linked_list_mutex); // Inserindo na lista encadeada
    new_marking -> prev = last_structure;
    if(last_structure != NULL)
      ((struct user_interface *) last_structure)-> next = (void *) new_marking;
    last_structure = (void *) new_marking;
    last_marking = new_marking;
    MUTEX_SIGNAL(&linked_list_mutex);
  }
}
```

Como é importante existir uma última marcação no hstórico para termos uma contagem de quantas interfaces estão ativas em um dado momento, vamos criar a primeira marcação durante a inicialização da API. Esta primeira marcação nunca será apagada, exceto na finalização da API. Mesmo que o usuário tente removê-la

antes por meio de funções.

Seção: Inicialização da API de Interface (continuação):

```
_Wmark_history_interface();
```

Seção: Finalização da API de Interface (continuação):

```
// Apaga todas as marcações, menos a primeira:
while(last_marking -> previous_marking != NULL){
  _Wrestore_history_interface();
}
// Apaga as interfaces após a primeira marcação:
_Wrestore_history_interface();
// Apaga a primeira marcação:
if(permanent_free != NULL)
  permanent_free(last_marking);
last_marking = NULL;
last_structure = NULL;
```

A função `_Wrestore_history_interface` é uma das funções de API que ainda não definimos. Ela destrói e remove todas as estruturas criadas após a última marcação no histórico e a última marcação. Exceto no caso da primeira marcação que nunca é removida. E por causa disso, no código acima, esta primeira marcação é removida manualmente com um `permanenet_free`.

6.5. Removendo Marcações e Interfaces

A forma de remover e desalocar interfaces é através da restauração do histórico, removendo a última marcação e voltando ao estado que estávamos quando ela foi criada. Isso remove toda e qualquer interface que tenha sido criada após a marcação removida. Isso é feito usando a função `_Wrestore_history_interface`.

O que esta função faz é remover a última marcação de histórico em nossa lista encadeada, fazendo com que a última marcação volte a ser aquela que existia antes. Se não existir nenhuma outra marcação além da que existe, nós não a removemos. Mas independente disso, nós de qualquer forma percorremos a lista encadeada e removemos todas as interfaces (e ligações) que estiverem depois dela na lista.

A implementação da função é:

Seção: Definição de Funções da API (interface.c) (continuação):

```
void _Wrestore_history_interface(void){
  struct marking *to_be_removed;
  struct user_interface *current, *next;
  MUTEX_WAIT(&linked_list_mutex);
  last_structure = last_marking -> prev;
  if(last_structure != NULL)
    ((struct user_interface *) last_structure) -> next = NULL;
  to_be_removed = last_marking;
  current = (struct user_interface *) to_be_removed -> next;
  // Removendo estruturas criadas após última marcação:
  while(current != NULL){
    next = (struct user_interface *) (current -> next);
    if(current -> type == TYPE_INTERFACE)
      destroy_interface(current);
    else if(current -> type == TYPE_SHADER)
      destroy_shader((struct shader *) current);
    else if(permanent_free != NULL)
      permanent_free(current);
```

```
    current = next;
  }
  // Removendo última marcação se não for a primeira
  if(to_be_removed -> previous_marking != NULL){
    last_marking = to_be_removed -> previous_marking;
    if(permanent_free != NULL)
      permanent_free(to_be_removed);
  }
  else
    to_be_removed -> number_of_interfaces = 0;
              <Seção a ser Inserida: Restauração de Histórico>
  MUTEX_SIGNAL(&linked_list_mutex);
}
```

Basicamente a função acima primeiro remove todas as estruturas de dados criadas após a última marcação. Só depois de fazer isso que a marcação é então removida. Exceto se não houver nenhuma marcação antes, o que significa que devemos mantê-la. O ponteiro para o último elemento da nossa lista encadeada é atualizado logo no começo da função, pois a última marcação armazena qual era o último elemento antes dela ser criada por meio de seu ponteiro para o elemento anterior.

7. Renderizando Interfaces

Para renderizar as interfaces, devemos levar em conta a ordem correta em que cada uma delas deve ser renderizada. Uma das formas mais simples de se lidar com isso é simplesmente desenhar em qualquer ordem e confiar na coordenada z para que o OpenGL saiba o que desenhar na frente por meio de um simples teste de profundidade. Contudo, ao lidar com objetos trasparentes, isso nem sempre gera o resultado correto. Se primeiro desenharmos um objeto transparente e depois um objeto opaco logo atrás dele, partes que deveriam aparecer do objeto atrás por causa da transparêndia do objeto na frente podem não aparecer.

No caso de nossas interfaces, como estamos sempre lidando com objetos bidimensionais que sempre estarão paralelos em relação ao eixo z, o que faremos é apenas manter uma lista ordenada de ponteiros para interfaces. A lista estará ordenada de modo que objetos com valores menores da coordenada z apareçam antes dos objetos com coordenadas de valores maiores. Desta forma, objetos que são desenhados na frente serão desenhados sempre depois de objetos posicionados atrás deles. Basicamente este é o chamado "algoritmo do pintor". Tal como o pintor de um quadro, começaremos sempre desenhando aquilo que estiver mais distante.

Para interfaces que tiverem exatamente a mesma posição na coordenada z, elas poderão ser desenhadas em qualquer ordem.

O ponteiro que aponta para a lista ordenada de interfaces será este:

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

```
static struct user_interface **z_list = NULL;
static unsigned z_list_size = 0;
_STATIC_MUTEX_DECLARATION(z_list_mutex);
```

Na inicialização de nossa interface, devemos sempre inicializar as variáveis acima:

Seção: Inicialização da API de Interface (continuação):

```
MUTEX_INIT(&z_list_mutex);
z_list_size = 0;
z_list = NULL;
```

Ao finalizar a nossa API, se a lista de interfaces foi alocada e portanto tem um valor diferente de nulo, precisamos desalocar e ajustar o valor para nulo novamente:

Seção: Finalização da API de Interface (continuação):

```
MUTEX_DESTROY(&z_list_mutex);
if(z_list != NULL && permanent_free != NULL)
  permanent_free(z_list);
z_list = NULL;
z_list_size = 0;
```

Da mesma forma que na finalização, quando restauramos o histórico de interfaces, removendo uma marcação e todas as interfaces depois dela, teremos que reiniciar nossa lista deixando-a vazia novamente.

Seção: Restauração de Histórico:

```
MUTEX_WAIT(&z_list_mutex);
if(z_list != NULL && permanent_free != NULL)
  permanent_free(z_list);
z_list = NULL;
z_list_size = 0;
MUTEX_SIGNAL(&z_list_mutex);
```

A inicialização desta lista ordenada de interfaces se dará na função de renderização `_Wrender_interface`. Dentro desta função, deveremos checar se o tamanho desta lista é igual ao número de interfaces ativas no momento (que pode ser conferida checando a última marcação de histórico). Se for um valor diferente, isso significa que interfaces novas foram criadas e devemos gerar novamente nossa lista ordenada com as interfaces que temos. O código que faz isso é mostrado abaixo:

Seção: Gerar Lista Ordenada de Interfaces:

```
if(z_list_size != last_marking -> number_of_interfaces){
  void *p;
  int i, j;
  MUTEX_WAIT(&z_list_mutex);
  // Realocando
  if(z_list != NULL && permanent_free != NULL)
    permanent_free(z_list);
  z_list_size = last_marking -> number_of_interfaces;
  z_list = (struct user_interface **)
             permanent_alloc(sizeof(struct user_interface *) * z_list_size);
  // Copiando para lista:
  p = last_marking -> next;
  for(i = 0; i < z_list_size; i ++){
    if(((struct user_interface *) p) -> type == TYPE_INTERFACE)
      z_list[i] = (struct user_interface *) p;
    else
      z_list[i] = ((struct link *) p) -> linked_interface;
    p = ((struct user_interface *) p) -> next;
  }
  // Ordenando lista (insertion sort):
  for(i = 1; i < z_list_size; i ++){
    j = i;
    while(j > 0 && z_list[j - 1] -> z > z_list[j] -> z){
      p = z_list[j];
      z_list[j] = z_list[j - 1];
      z_list[j - 1] = (struct user_interface *) p;
      j = j - 1;
    }
  }
```

```
  MUTEX_SIGNAL(&z_list_mutex);
}
```

Note que este cenário no qual a lista deve ser reconstruída é para ser incomum. Tipicamente ele irá ocorrer na primeira vez que formos renderizar após o programa começar a execução e após restaurarmos histórico. O único cenário no qual a execução deste código será mais frequente é quando novas interfaces forem sendo criadas durante o laço principal. Contudo, esta é uma má prática e por isso iremos assuir que somente raramente o código acima entrará em execução para inicializar a lista. No caso típico, já temos uma lista ordenada com todas as interfaces, e de vez em quando ajustamos a posição de um dos elementos se ele for movido.

Uma vez que tenhamos a lista ordenada, basta renderizarmos individualmente as interfaces na ordem que elas aparecem. Para cada uma delas devemos carregar o shader correto, passar os atributos, uniformes e variantes para ele e pedir para renderizar os vértices compartilhados por todas as interfaces.

Para renderizar as interfaces, invoca-se a função de API `_Wrender_interface`. A função recebe como parâmetro o tempo atual em microsegundos. Mas para aber quanto tempo se passou entre uma renderização e outra, precisamos armazenar a nossa última medida de tempo. Faremos isso na seguinte variável que começará valendo zero quando nenhuma medida de tempo foi feita:

Seção: Variáveis Locais (interface.c):

```
static unsigned long long previous_time = 0;
```

A variável sempre precisa ser inicializada para este valor na inicialização da API. Ou podemos obter valores incorretos se a API for finalizada e inicializada novamente:

Seção: Inicialização da API de Interface (continuação):

```
previous_time = 0;
```

E eis abaixo nossa função que itera sobre a lista ordenada de interface e renderiza cada uma delas, atualizando também o tempo medido. Ela começa primeiro atualizando o tempo, depois carrega os vértices adequados que representam as interfaces indicando como ele está representado e passa para uma iteração sobre cada uma das interfaces ordenadas.

Para cada uma delas, as informações adequadas presentes na estrutura de interface são passadas para cada uniforme e variante do shader. Depois de fazer isso, para saber qual textura renderizar se houver mais de uma, checamos se estamos diante de uma interface com texturas animadas (comos eria no caso de um GIF animado). Se for o caso, com base no tempo passado entre esta e a invocação anterior, atualizamos a contagem de tempo dentro da interface e verificamos se temos que atualizar ou não o frame.Por fim, renderizamos a interface atual e sua textura. Depois de executar o laço inteiro, memorizamos o tempo atual como sendo o tempo anterior para a próxima execução.

A implementação do que foi descrito acima é:

Seção: Definição de Funções da API (interface.c) (continuação):

```
void _Wrender_interface(unsigned long long time){
          <Seção a ser Inserida: Gerar Lista Ordenada de Interfaces>
  {
    int i, elapsed_time;
    if(previous_time != 0)
      elapsed_time = (int) (time - previous_time);
    else
      elapsed_time = 0;
    // Carregando os vértices do VBO:
    glBindBuffer(GL_ARRAY_BUFFER, interface_vbo);
    // Especificando como os dados estão representados no VBO:
```

```
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
                          (void *) 0);
    glEnableVertexAttribArray(0);
    MUTEX_WAIT(&z_list_mutex);
    for(i = 0; i < z_list_size; i ++){
      if(!(z_list[i] -> _loaded_texture) || !(z_list[i] -> visible))
        continue;
      // Escolhendo o shader certo:
      glUseProgram(z_list[i] -> shader_program -> program);
      // Passando os Uniformes:
      glUniform4fv(z_list[i] -> shader_program -> uniform_foreground_color, 4,
                   z_list[i] -> foreground_color);
      glUniform4fv(z_list[i] -> shader_program -> uniform_background_color, 4,
                   z_list[i] -> background_color);
      glUniformMatrix4fv(z_list[i] -> shader_program ->
                          uniform_model_view_matrix, 1, false,
                         z_list[i] -> _transform_matrix);
      glUniform2f(z_list[i] -> shader_program -> uniform_interface_size,
                  z_list[i] -> width, z_list[i] -> height);
      // O shader recebe contagem de tempo em segundos módulo 1 hora
      glUniform1f(z_list[i] -> shader_program -> uniform_time,
                  ((double) (time % 3600000000ull)) / (double) (1000000.0));
      glUniform1i(z_list[i] -> shader_program -> uniform_integer,
                  z_list[i] -> integer);
      // Animating texture
      if(z_list[i] -> animate && z_list[i] -> number_of_frames > 1 &&
         z_list[i] -> max_repetition != 0){
        z_list[i] -> _t += elapsed_time;
        z_list[i] -> current_frame %= z_list[i] -> number_of_frames;
        while(z_list[i] -> _t >
                    z_list[i] -> frame_duration[z_list[i] -> current_frame]){
          z_list[i] -> _t -=
            z_list[i] -> frame_duration[z_list[i] -> current_frame];
          z_list[i] -> current_frame ++;
          z_list[i] -> current_frame %= z_list[i] -> number_of_frames;
        }
      }
      // Rendering:
      if(z_list[i] -> _texture1 != NULL)
        glBindTexture(GL_TEXTURE_2D,
                      z_list[i] -> _texture1[z_list[i] -> current_frame]);
      else
        glBindTexture(GL_TEXTURE_2D, default_texture);
      glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    }
    MUTEX_SIGNAL(&z_list_mutex);
    glBindTexture(GL_TEXTURE_2D, 0);
  }
  previous_time = time;
}
```

8. Movendo, Rotacionando e Redimencionando Interfaces

8.1. Movendo Interfaces

Mover uma interface significa atualizar suas variáveis (x, y, z). Mas além disso, precisamos atualizar também a matriz de modelo-visualização para que tenha novos valores x e y, além de que se acoordenada z mudou, então podemos ter que mudar a posição da interface com outra na lista ordenada de interfaces que usamos para determinar qual desenhamos primeiro. Para tudo isso, precisamos também reservar o uso do Mutex da interface para garantir que duas invocções da função não se sobreponham.

A função que move interfaces em nossa API é `_Wmove_interface` e definimos ela assim:

Seção: Definição de Funções da API (interface.c) (continuação):

```
void _Wmove_interface(struct user_interface *i,
                      float new_x, float new_y, float new_z){
  GLfloat x, y, w, h;
  GLfloat cos_theta = cos(i -> rotation);
  GLfloat sin_theta = sin(i -> rotation);
  MUTEX_WAIT(&(i -> interface_mutex));
  i -> x = new_x;
  i -> y = new_y;
        <Seção a ser Inserida: Conversão de Coordenadas e Tamanho>
  i -> _transform_matrix[12] = -w/2 * cos_theta + w/2 * sin_theta + x;
  i -> _transform_matrix[13] = -h/2 * sin_theta - h/2 * cos_theta + y;
  if(new_z != i -> z){ // Atualizando lista ordenada de interfaces
    int j;
    i -> z = new_z;
    MUTEX_WAIT(&z_list_mutex);
    for(j = 0; j < z_list_size; j ++){
      if(z_list[j] == i){
        while(j > 0 && i -> z < z_list[j - 1] -> z){
          z_list[j] = z_list[j - 1];
          z_list[j - 1] = i;
          j --;
        }
        while(j < z_list_size - 1 && i -> z > z_list[j + 1] -> z){
          z_list[j] = z_list[j + 1];
          z_list[j + 1] = i;
          j ++;
        }
      }
    }
    MUTEX_SIGNAL(&z_list_mutex);
  }
  MUTEX_SIGNAL(&(i -> interface_mutex));
}
```

8.2. Rotacionando Interfaces

Rotacionar interfaces envolve apenas reservar o mutex da interface, atualizar sua variável de rotação e também atualizar sua matriz de modelo-visualização. O código para isso segue abaixo:

Seção: Definição de Funções da API (interface.c) (continuação):

```
void _Wrotate_interface(struct user_interface *i, float rotation){
  GLfloat x, y, w, h;
```

```
  GLfloat cos_theta = cos(rotation);
  GLfloat sin_theta = sin(rotation);
  MUTEX_WAIT(&(i -> interface_mutex));
  i -> rotation = rotation;
        <Seção a ser Inserida: Conversão de Coordenadas e Tamanho>
  i -> _transform_matrix[0] = w * cos_theta;
  i -> _transform_matrix[1] = h * sin_theta;
  i -> _transform_matrix[4] = -w * sin_theta;
  i -> _transform_matrix[5] = h * cos_theta;
  i -> _transform_matrix[12] = -w/2 * cos_theta + w/2 * sin_theta + x;
  i -> _transform_matrix[13] = -h/2 * sin_theta - h/2 * cos_theta + y;
  MUTEX_SIGNAL(&(i -> interface_mutex));
}
```

8.3. Redimensionando Interfaces

Redimensionar interfaces, assim como rotacioná-las, envolve apenas reservar um mutex, atualizar as variáveis de tamanho e atualizar a matriz de modelo-visualização. Entretanto, também devemos tomar cuidado com as variáveis de tamanho máximo e mínimo da interface, lembrando que um tamanho máximo ou mínimo não-positivo significa que tal limite não existe.

A função que redimenciona interfaces é:

Seção: Definição de Funções da API (interface.c) (continuação):

```
void _Wresize_interface(struct user_interface *i,
                        float new_width, float new_height){
  GLfloat x, y, w, h;
  GLfloat cos_theta = cos(i -> rotation);
  GLfloat sin_theta = sin(i -> rotation);
  MUTEX_WAIT(&(i -> interface_mutex));
  i -> width = new_width;
  i -> height = new_height;
        <Seção a ser Inserida: Conversão de Coordenadas e Tamanho>
  i -> _transform_matrix[0] = w * cos_theta;
  i -> _transform_matrix[1] = h * sin_theta;
  i -> _transform_matrix[4] = -w * sin_theta;
  i -> _transform_matrix[5] = h * cos_theta;
  i -> _transform_matrix[12] = -w/2 * cos_theta + w/2 * sin_theta + x;
  i -> _transform_matrix[13] = -h/2 * sin_theta - h/2 * cos_theta + y;
  MUTEX_SIGNAL(&(i -> interface_mutex));
}
```

8. Interagindo com Interfaces

Dadas as interfaces ativas, pode-se interagir com elas passando o cursor do mouse delas, clicando com o botão direito do mouse nelas, clicando com o esquerdo ou clicando com o botão do meio. Para gerenciar isso usamos a função `_Winteract_interface` que nos informa o que o mouse está fazendo e automaticamente executa as funções associadas com cada interação para cada interface.

Para isso ser possível, temos que memorizar o estado anterior dos botões do mouse, não apenas lidar com o estado atual. Por causa disso, vamos usar as seguintes variáveis estáticas para memorizar o último estado dos botões:

Seção: Variáveis Locais (interface.c):

```
static bool mouse_last_left_click = false, mouse_last_middle_click = false,
  mouse_last_right_click = false;
```

Inicializamos tais variáveis com o valor falso na inicialização de nossa API:

Seção: Inicialização da API de Interface (continuação):

```
mouse_last_left_click = false;
mouse_last_middle_click = false;
mouse_last_right_click = false;
```

Sabendo o estado anterior do mouse e o estado atual, somos capazes de dizer não apenas se um botão está sendo pressionado, mas também se ele começou a ser pressionado agora ou a pressão sobre ele está apenas continuando. Saberemos não apenas que um botão não está sendo pressionado, mas também se ele foi solto em algum momento entre o frame anterior e o atual. Só assim seremos capazes de executar corretamente as funções de interação de interface.

Em um dado frame, só podemos interagir com uma única interface, dado que só temos um cursor do mouse. Mas se mais de uma interface ocupar a mesma posição, como saberemos com qual interagir. Ora, felizmente armazenamos em uma lista ordenada as interfaces na ordem em que elas são desenhadas na tela. Devemos então priorizar a interface que é desenhada depois, pois é ela que o usuário está vendo sob o cursor do mouse.

Então, para interagir com interfaces, percorremos em ordem reversa a lista ordenada de interfaces até acharmos aquela que está diretamente sob o mouse. É com ela que iremos interagir. Só temos que também ao percorrer a lista, marcar que o mouse não está sobre qualquer uma outra. E se for o caso, executar a função `on_mouse_out` da interface. Só então, checamos a interação com a interface certa e executamos a função adequada se existir.

O código da função que faz isso é:

Seção: Definição de Funções da API (interface.c) (continuação):

```
void _Winteract_interface(int mouse_x, int mouse_y, bool left_click,
                          bool middle_click, bool right_click){
  int i;
  struct user_interface *previous = NULL, *current = NULL;
  MUTEX_WAIT(&z_list_mutex);
  for(i = z_list_size - 1; i >= 0; i --){
    float x, y;
      <Seção a ser Inserida: Converter Coordenadas do Mouse para x e y>
    if(current == NULL && z_list[i] -> x - (z_list[i] -> width / 2) < x &&
       z_list[i] -> x + (z_list[i] -> width / 2) > x &&
       z_list[i] -> y - (z_list[i] -> height / 2) < y &&
       z_list[i] -> y + (z_list[i] -> height / 2) > y)
      current = z_list[i];
    else{
      if(z_list[i] -> _mouse_over){
         z_list[i] -> _mouse_over = false;
         previous = z_list[i];
       }
    }
  }
  MUTEX_SIGNAL(&z_list_mutex);
  if(previous != NULL && previous -> on_mouse_out != NULL){
    previous -> on_mouse_out(previous);
  }
  if(current != NULL){
    if(current -> _mouse_over == false){
      current -> _mouse_over = true;
```

```
      if(current -> on_mouse_over != NULL)
        current -> on_mouse_over(current);
    }
    if(left_click && !mouse_last_left_click && current -> on_mouse_left_down)
      current -> on_mouse_left_down(current);
    else if(!left_click && mouse_last_left_click && current -> on_mouse_left_up)
      current -> on_mouse_left_up(current);
    if(middle_click && !mouse_last_middle_click &&
       current -> on_mouse_middle_down)
      current -> on_mouse_middle_down(current);
    else if(!middle_click && mouse_last_middle_click &&
            current -> on_mouse_middle_up)
      current -> on_mouse_middle_up(current);
    if(right_click && !mouse_last_right_click && current -> on_mouse_right_down)
      current -> on_mouse_right_down(current);
    else if(!right_click && mouse_last_right_click &&
            current -> on_mouse_right_up)
      current -> on_mouse_right_up(current);
  }
  mouse_last_left_click = left_click;
  mouse_last_middle_click = middle_click;
  mouse_last_right_click = right_click;
}
```

A parte que omitimos acima é como obtemos a coordenada (x, y) que usamos para comparar com a posição de cada interface à partir das coordenadas de mouse que recebemos como argumento. A coordenada (x, y), muitos casos é exatamente a mesma coordenada de mouse que recebemos como argumento. Mas se a interface estiver rotacionada, então precisamos fazer um ajuste para poder medir corretamente.

Se o mouse estiver bem na ponta inferior direita de interface retangular, e subitamente ela for rotacionada um pouco, a interface não estará mais sob o mouse. Para levar em conta esse tipo de rotação, a forma mais fácil é ignorarmos que a interface está rotacionada e rotacionarmos as coordenadas do mouse na direção oposta da rotação da interface e usando o centro da interface como eixo de rotação. Se a coordenada transformada ainda estiver dentro dos limites retangulares da interface, isso significa que o mouse realmente está sobre a nossa interface.

O código que gera a coordenada (x, y) então apenas checa se a interface está rotacionada. Se não estiver, usamos como valores as próprias coordenadas do mouse, sem precisar fazer cálculos adicionais de conversão. Se estiver, então obtemos (x, y) fazendo a rotação descrita na coordenada do mouse:

Seção: Converter Coordenadas do Mouse para x e y:

```
if(z_list[i] -> rotation == 0.0){
  x = mouse_x;
  y = mouse_y;
}
else{
 float cos_theta = cos(- (z_list[i] -> rotation));
 float sin_theta = sin(- (z_list[i] -> rotation));
 x = (mouse_x - z_list[i] -> x) * cos_theta -
       (mouse_y - z_list[i] -> y) * sin_theta;
 y = (mouse_x - z_list[i] -> x) * sin_theta +
       (mouse_y - z_list[i] -> y) * cos_theta;
 x +=  z_list[i] -> x;
 y +=  z_list[i] -> y;
```

```
}
```

Referências

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