Weaver User Interface

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.*

1. Introduction

A graphical user interface is how a program communicates and takes information from the user. Ina typical program we could have menus, buttons, window pop-ups and other typical graphical elements. In games the user intrface usually is conceptually simpler, made by some menus and some viewing elements with information about the game state. For example, a life counter in the corner of the screen.

The fact is that in a computer game we cannot assume almost nothing about the appearence of the user interface. A strategy game could have lots of menus appearing when the user click in some unit. A platform game could have only information about equipped items and life counter. The appearence and behaviour of such elements can change a lot depending on the game style.

The sole universal characteristic that we will assume for our interface elements is that they appear over the ame scenario, they are not obscured by objects that are part of our game world. Typically we will render the interfaces after rendering our game world. And elements from the game world cannot interact with interfaces. Interfaces are not part of the world that the game simulates, they are just elements that give to the player information that would be otherwise difficult to express.

Our objective here will be creating an AP where the user can create a new interface using the function:

Section: Function Declaration (interface.h):

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

Where the first argument is a file that will be open and interpreted to get information about our interface texture (it could be an image filename). The second argument is the file containing the OpenGL shader. Both arguments can be NULL¿ A null shader means that we will use some default shader. A null file means that the interface will have no texture. The other arguments are the initial position and size for our interface.

We could define shaders more easily for them if we pass some additional library with GLSL functions. This can be done passing to this function a string with source code for function definitions in GLSL:

Section: Function Declaration (interface.h) (continuation):

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

When we have a interface, we can move it with the function:

Section: Function Declaration (interface.h) (continuation):

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

We assume that the position of an interface is the coordinate of its center. The axis x and y represents the horizontal and vertical position. The axis z determines which interface will be in the front if they occupy the same place in the space.

We can also rotate it with the function:

Section: Function Declaration (interface.h) (continuation):

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

The function above gets as argument how much the interface will be rotated in radians comparing with its notmal orientation.

We can also resize the interface with the function below:

Section: Function Declaration (interface.h) (continuation):

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

The function below renders in the screen or drawing area all the previously created interfaces without needing to list all them:

Section: Function Declaration (interface.h) (continuation):

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

The time parameter is necessary for when the function needs to know how much time in microseconds passed between the function invocation. This is useful to reder correctly interfaces with animated textures.

But rendering all previous interfaces may not be what the user wants, Sometimes, a person want to render just the interfaces created after some point of the history. For this, the function below creates a marking in our interface history. After this, when we ask to render all the interfaces, only the interfaces created after this marking will be rendered:

Section: Function Declaration (interface.h) (continuation):

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

But what if we want to render some of the interfaces created before the last history marking? In this case, we can create e a new interface that in fact is just a link to a previously created interface:

Section: Function Declaration (interface.h) (continuation):

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

To interact with all the previously created interfaces until the last marking, we use the function below. It executes the programmed actions, if they exist, for each interface if the user puts or removes the mouse over them or if the user clicks in some interface:

Section: Function Declaration (interface.h) (continuation):

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

But how can we erase and destroy some interface when we do not need it anymore? For this we use the following function that erases all previously created interfaces after the last marking in the history. This also erases the last marking, returning the state to what it was just before the creation of that marking. The interfaces before that marking will be rendered again.

Section: Function Declaration (interface.h) (continuation):

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

All this requires that we manage our interface history, its shaders and the markings. And this requires memory allocation and disallocation. There are two kinds of allocations that we could use: for permanent things that will be kept in the memory for a long time and for temporary things that will be disallocated soon. We will store functions to allocate and disallocate in the two cases:

Section: Local Variables (interface.c):

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

By default we will use the allocation and disallocation functions from the standard library. But the user can choose custom functions to replace them. Besides these four functions, we will also use custom functions imediatelly before and after loading a new interfac. These functions will be initialized by NULL, but the user later can change them to other functions:

Section: Local Variables (interface.c):

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

The idea is that all custom functions are chosen during the API initialization:

Section: Function Declaration (interface.h) (continuation):

```
#include <stdlib.h> // Define  'size_t' type
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),
                      ...);
```

It is possible to initializa the disallocation functions with `NULL`. This means that we will not disallocate what we allocate. This could be useful in some scenarios where the memory manager uses its own garbage collector and do not want interferences.

The function above accepts a variable number of arguments. First it gets pointers to where we store our window width and height. They will be stored here:

Section: Local Variables (interface.c):

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

The window size is followed by custom functions as described before. What cames after the six initial functions is a NULL-terminated list of arguments composed by a string followed by some function that create a new interface given a file. The string represents a filename extension (for example, “gif”, “jpg” or others). The function after the file extension have the following type:

Section: Local Macros (interface.c):

```
typedef void pointer_to_interface_function(void *(*)(size_t), void (*)(void *),
                                           void *(*)(size_t), void (*)(void *),
                                           void (*)(void), void (*)(void),
                                           char *, struct user_interface *);
```

It gets as arguments the functions for allocation and disallocation, a filename and a pointer to the interface that should be filled with information from the file. It is expected that each of these functions can interpret correctly the files with extensions given previously and fill the given interface with textures extracted from the file. This way, we pass to the user the responsability of provinding functions able to create interfaces from different file formats.

As we have an initialization function, we will need also a finalization function:

Section: Function Declaration (interface.h) (continuation):

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

And this ends the description of our API functions.

The API also can have its behaviour changed setting the macro `W_FORCE_LANDSCAPE`. If we have an environment where the width is greater than the height, setting this macro will change nothing. But if the window height is greater than the window width and this macro is set, then we will rotate 90 degrees counter-clockwise our axis x and y. This is useful to create more consistent user interfaces in mobile devices. In this case, we can always assume that we will have more space horizontally than vertically and if necessary the user can rotate its device.

1.1. Literate Programming

Our API will be written using the literate programming technique, proposed by Knuth on [Knuth, 1984]. It consist in writting a computer program explaining didactically in a text what is being done while presenting the code. The program is compiled extracting the computer code directly from the didactical text. The code shall be presented in a way and order such that it is best for explaining for a human. Not how it would be easier to compile.

Using this technique, this document is not a simple documentation for our code. It is the code by itself. The part that will be extracted to be compiled can be identified by a gray background. We begin each piece of code by a title that names it. For example, immediately before this subsection we presented a series of function declarations. And how one could deduct by the title, most of them will be positioned in the file `interface.h`.

We can show the structure of the file `interface.h`:

File: `src/interface.h`:

```
#ifndef __WEAVER_INTERFACE
#define __WEAVER_INTERFACE
#ifdef __cplusplus
extern "C" {
#endif
#include <stdbool.h> // Define  'bool' type
#if !defined(_WIN32)
#include <sys/param.h> // Needed on BSD, but does not exist on Windows
#endif
            <Section to be inserted: Include General Headers (interface.h)>
                <Section to be inserted: General Macros (interface.h)>
                <Section to be inserted: Data Structures (interface.h)>
              <Section to be inserted: Function Declaration (interface.h)>
#ifdef __cplusplus
}
#endif
#endif
```

The cde above shows the default boureaucracy to define a header for our C API. The two first lines and the last one are macros that ensure that this header will not be inserted more than once in a single compiling unit. The lines 3, 4, 5 and the three lines before the

last one make the header adequate to be used in C++ code. This tells the compiler that we are using C code and that therefore, the compiler is free to use optimizations assuming that we will not use C++ exclusive techniques, like operator overloading. Next we include a header that will let us to use boolean variables. And there are some parts in red. Note that one of them is called "Declarao de Funo (interface.h)", the same title used in most of the code declared previously. This means that all the previous code with this title will be inserted in that position inside this file. The other parts in red represent code that we will define in the next sections.

If you want to know how is the `interface.c` file related with this header, its structure is:

File: `src/interface.c`:

```
#include "interface.h"
<Section to be inserted: Local Headers (interface.c)>
<Section to be inserted: Local Macros (interface.c)>
<Section to be inserted: Local Data Structures (interface.c)>
<Section to be inserted: Local Variables (interface.c)>
<Section to be inserted: Auxiliary Local Functions (interface.c)>
<Section to be inserted: API Functions Definition (interface.c)>
```

All the code presented in this document will be placed in one of these two files. Besides them, no other file will be created.

1.2. Supporting Multiple Threads

Most code that will be defined in this document is portable. The only assumed requeriment is that an OpenGL context was already created and is active. However, there is a single non-portable part that we should define differently depending on the system and environment: the mutex support.

A mmutex is a data sctructure that controls the access for multiple proccesses to a single common resource. They are treated differently depending on the operating system and environment. Because of its non-portability, we will introduce them here, separated from the remaining of the code.

On Linux and BSD a Mutex is defined using the library `pthread` and follows its naming convention. On Windows, a Mutex is called a "critical section". On Web Assembly, for now, we will not define them because we are not allowing multiple threads in this environment.

Section: General Macros (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
```

This means that on Linux and BSD, we need to include headers for `pthread` library. On Windows, we just include the default Windows header:

Section: Include General Headers (interface.h):

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

```
#include <windows.h>
#endif
```

In our code we will need to initializa each declared Mutex. For this, we will ise the following macros:

Section: Local Macros (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
```

To finalize and delete the Mutex, we use the following macros:

Section: Local Macros (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
```

Once we have a Mutex, there are two operations that we can do. The first is ask for Mutex usage. If some other proccess is already using it, this means that we wait until the Mutex is available:

Section: Local Macros (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
```

And finally, if we finished to use the resource guarded by the Mutex, we signal that other proccesses can now use it:

Section: Local Macros (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. Data Structures

In this section we will describe the four main data structures managed by this API: the shaders, user interfaces, links for previous user interfaces and marks in the history.

2.1. Shader Data Structure

To render any object in the video card, we need to specify the rendering rules. In what position should we render the object? What is its color? And what about the texture? The rendering should change depending on time or the viewing angle? To answer these questions, we use special computer programs that are executed in the video card. Such programs are the shaders.

But besides the program, we need to store for each shader additional variables. Shader programs check a list of variables while executing and we can change the value of such variables before each execution of the shader program. For each modifiable variable, we need to store the address of such variable in the shader program. Only knowing this information we can change their values.

A shader data structure has the following structure:

Section: Local Data Structures (interface.c):

```
struct shader {
  int type;
  void *next; // Ponteiro de lista encadeada
  GLuint program; // O programa em si
  /* Aqui comeam as variveis modificveis de shader */
  GLint uniform_foreground_color, uniform_background_color; // Cor de frente/fundo
  GLint uniform_model_view_matrix; // Matriz com tamanho, rotao e translao
  GLint uniform_interface_size; // Tamanho em pxels do objeto renderizado
  GLint uniform_mouse_coordinate; // Mouse coordinate (interface referential)
  GLint uniform_time; // Contador de tempo
  GLint uniform_integer; // Inteiro arbitrrio que pode ser passado
  GLint uniform_texture1; // A textura do objeto a ser renderizado
};
```

As we are using OpenGL data types, like `GLuint`, we should include the OpenGL headers:

Section: Include General Headers (interface.h) (continuation):

```
#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
```

The two first variables in the data structure exist because we will store each shader data structure in a linked list that also can store other types of data structure. The type stores what type of data structure we have in the list and thepointer for the next element is what creates the linked list, linking each element to the next one.

In a shader data structure, the type variable always will be `TYPE_SHADER`. All possible values, fo all different types of data structures, are:

Section: Local Macros (interface.c):

```
#define TYPE_INTERFACE 1 // Uma interface, ou seja, objeto a ser renderizado
#define TYPE_LINK      2 // Ligao para interface anterior
#define TYPE_MARKING   3 // Marcao no histrico de interfaces criadas
#define TYPE_SHADER    4 // Shader para renderizar interface
```

The variable `program` in the shader data structure is where we will store the compiled shader. And all variables after it are there to indicate where each configurable variable inside

the shader program ca be found. Not all these variables necessarily will exist in all shaders, but these are the variables supported by our API. We have variables that store vertices to be rendered, size, position, color, textures and some other things.

2.2. User Interface Data Structure

The most complex data structure in this document is the one that stores information about user interfaces. The full definition for this data structure is:

Section: Data Structures (interface.h):

```
struct user_interface{
  int type;
  void *next; // Pointer for linked list
  float x, y, _x, _y, z;
  float rotation, _rotation;
  float mouse_x, mouse_y;
  float height, width;
  float background_color[4], foreground_color[4];
  int integer;
  bool visible;
  GLfloat _transform_matrix[16];
  struct shader *shader_program;
  _MUTEX_DECLARATION(interface_mutex);
  /* Interacting functions */
  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 *);
  /* Attributes below should be filled by the loading function: */
  GLuint *_texture1;
  bool _loaded_texture;
  bool animate;
  unsigned number_of_frames;
  unsigned current_frame;
  unsigned *frame_duration;
  unsigned long _t;
  int max_repetition;
};
```

Now we will describe what means each part of this structure. The type and ponter for next element already were described and are the same than what was presented in the shader data structure.

After the type and the pointer, we store the position in pixels (for axis x and y) for our interface and its z-index (axis z). We then store the rotation (in radians) and the width and height (in pixels). All these values can be read, but should not be changed directly. To change its valies, it is necessary to use the designed functions that we will define in the future sections. This is necessary because to correctly change the position, rotation and size, we must update the transform matrix, which is how OpenGL and our shaders really represent these informations. And the z-index must also be updated in anoother list, where we will

store the correct order to render each element.

Notice that in fact we have two variables to store position x and y and two to store the rotation. We do this because when the macro `W_FORCE_LANDSCAPE` is defined, and if the window height is greater than window width, we rotate the axis x and y, and the interface also will be rotated 90 degrees. In this case, the variables `_x`, `_y` and `_rotation` will store the real value, as if the axis were not swapped and the variables `x`, `y` and `rotation` store the values with the rotated axis. When the height is not greater than the width or when the macro is not defined, then both kind of variables will store the same value.

The attributes `mouse_x` and `mouse_y` are reserved to store the mouse coordinates not using as origin the lower left corner of the screen, but the lower left corner of the interface. If the interface is rotated, the coordinate system is also rotated.

reservados para armazenar a coordenada do mouse usando como referencial no o canto interior esquerdo da tela, mas o canto inferior esquerdo da interface. Rotacionar a interface rotaciona tambm a coordenada usada.

The foreground and background color are colors in RGBA format with individual values in the range between 0 and 1. Such values will be passed to the shader, but not necessarily it will use these informations. Likewise, the integer attribute also is data that will be sent to the shader, but the shader can choose to use or not the information.

The next attribute is if the interface is visible or not. If not, it will not be rendered in the screen.

Next we put a pointer to the shader data structure associated with this interface. It has the rules about how to render this interface.

The interacting functions below are usually initialized with `NULL`. If they are not `NULL`, they are executed if the mouse hover over the interface, leave the interface, or when some mouse button is pressed or released over them. We also have a boolean variable that store if the mouse is over this interface or not.

The attributes after the interacting functions are initialized with 0, false and NULL. But while the loading function is interpreting the filename, it should change and update the values. The loading function is one of the functions that should be passed to the initialization function in our API and that interpret files with a given extension to produce the textures and other information for our interfaces.

In this part first we have a pointer for the OpenGL texture for our interface. We can have not a single texture, but several textures. This happens if we open a file that can store more than one image. Or when we open an animated GIF, for example. Each frame in the animation will be a different texture.

If we have more than one texture, the next variable determines if the interface should animate or not. The value can be changed to pause or resume the animation.

The two next attributes are the number of frames in our animation and in which frame we are now. The first frame is number 0.

Next we have a pointer that should have allocated a vector with the duration for each frame. If the number of frames is 1 or 0, thes can point to NULL.

The variable `_t` will count the time elapsed. It should be set to zero after the interface is loaded and its texture is ready. After this, we will increment the variable periodically and this will help us to know when we should change the current frame in an animated interface.

Finally, the next attribute represents in an animated interface how many times we should repeat the animation. If the value is zero, then we will repeat forever. If it is a positive value and we repeated the animation the allowed number of times, then the interface will stop the animation and stay in the last frame.

2.3. Markings in History

As mentioned in the Introduction, the logic that determines which interfaces are accessible in a given moment uses markings done in our interface history. All interfaces created after the last marking areaccessible and the ones created before the marking are not acces-

sible: they will not be rendered and the user cannot interact with them. Several markings can be creted. And the most recent marking that still exist can be destroyed, which also destroy the interfaces created after it.

To make this possible, each marking must store a pointer to the previous marking and to the previous structure in the linked list. This way, when we erase a marking, we can replace the deleted marking with the marking pointed by it. We also store in each marking the number of interfaces created after it. This is the number of interfaces that will be active while this marking is the last created and will exist while this marking exists.

Section: Local Data Structures (interface.c):

```
struct marking {
  int type;
  void *next; // Ponter to linked list
  void *prev; // Pointer to previous element in the linked list
  struct marking *previous_marking;
  unsigned number_of_interfaces;
};
```

Notice that we declared this as a local data structure in file `interface.c`. We did this because this structure will be useful only internally in our API. The user will not need to manipulate directly markings. The user will interact with this data structure indirectly, when using functions `_Wmark_history_interface` and `_Wrestore_history_interface` described in the Introduction.

2.4. Link to Other Interfaces

If some interface is inaccessible because is older than the last marking in the history, it is possible to create a link to it usinf function `_Wlink_interface`. The linking to the older interface acts as a recently created interface and then the older interface becames accessible again. Creating a link means create and store in the linked list the following struct:

Section: Local Data Structures (interface.c):

```
struct link{
  int type;
  void *next; // Pointer to linked list
  struct user_interface *linked_interface;
};
```

Besides the information necessary to the linked list, like the type and the pointer to the next element, the only information that we need to store in a link is a pointer to the older linked interface.

3. Initializing and Finalizing the API

The purpose of initialization in this API is choosing all the custom functions that wil be used to allocate, disallocate, and to manage interface creation. The finalization disallocates data necessary to store information about some of these functions. It also reset all the cusatom functions to default values.

3.1. Initialization

Here our objecticve is set the six custom functions discussed in the Introduction, and also get a list with variable size with all functions that will be used to interpret the content of files with a given extension.

To store the functions from the list with variable size, we need the following structure:

Section: Local Data Structures (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;
```

The struct is basically a pair composed by a function that extracts textures from a given file (and gets all the custom functions that should be used) and by a file extension string. After defining the structure, we create a pointer to a list of such structures. Initially the pointer represents an empty list.

After, when this list is not empty anymore, in other words, after the initialization, we will be able to check each element in this list and extract the correct loading function using this auxiliary function:

Section: Auxiliary Local Functions (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;
}
```

The code above is verbose because it is a function that returns a function pointer to a function with lots of parameters. But the function above gets only a single argument: an extension to be searched in the list.

As we used a function to compare strings, we should include the standard header involving strings:

Section: Local Headers (interface.c):

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

Now we can define the initialization function. What this function will do os set the six basic custom functions, count how many additional custom functions we have to interpret files, allocate the space necessary in the list above (using the custom allocation functions) and fill the newly allocated list:

Section: API Functions Definition (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 Funes + 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 funes existem */
    va_list args;
    char *ext;
    va_start(args, new_after_loading_interface);
    do{
      count ++;
      ext = va_arg(args, char *);
      va_arg(args, pointer_to_interface_function*);
    } 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, pointer_to_interface_function*);
    }
  }
```

<Section to be inserted: Initializing Interface API>

```
}
```

The above function is very verbose because C is verbose when we have functions with variable number of parameters and these parameters can be pointers to functions with lots of parameters. Despite the verbosity, the initialization above is very simple.

In the end, in red, we marked a place where we will put additional initialization code that we will define in the next sections.

The use of some resources, like accessing arguments by `va_list` requires the following header:

Section: Local Headers (interface.c):

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

3.2. Finalization

The function that complements our initialization is the finalization function. If in the initialization we end allocating memory to store a list of functions that interpret files, in

finalizagion we begin diallocating this memory. If in the initialization we begin setting the six basic custom functions, in the finalization we end setting these functions to the default initial values:

Section: API Functions Definition (interface.c) (continuation):

```
void _Wfinish_interface(void){
                    <Section to be inserted: Finalizing Interface API>
  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;
}
```

Notice that we let additional space in red for additional code that we will eventually need in the finalization when we increase the API complexity defining new functions in the next sections.

And this undoes everything done in the initialization. It is possible to initialize and finalize the API without problems.

4. Shaders

One of the things that we need to define is our default shader to be used if the user do not pass a custom shader when creating an interface. Defining the custom shader gives us a great opportunity to present in more details the requisites for shaders supported by this API. We will require a specific format for shaders.

First, all shader source files must specify in the first line the version for the shading language used. We will use the GLSL language, but this language have different versions. We will not require that the user put this information in all shader sources, we will write this automatically.

To choose the correct version for the GLSL language that will be used in all the shaders, we will check the value in macro `W_GLSL_VERSION`. This macro is what will be written in the first line of each shader source code. The macro can be defined by the user passing adequate compilation flags, or including the macro definition in the code, but if this is not done, then we will use the standard string "`#version 100\n`". This standard value means that we will use the shading language OpenGL ES v1.00. We eill define the macro, if it does not exist, in the beginning of the file that defines our functions:

Section: Local Macros (interface.c):

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

There is at least two different shaders that we need to define for each interface. The first is the vertex shader, that will be processed for each vertex in our interface. The other is the fragment shader, that will process each individual pixel. Each of them have a different source code. But as we defined in the Introduction, when we specify a shader to each interface passing a single filename with its source code. We do not pass two fileames. How can we represent two source codes in the same file?

Well, as the GLSL language support C-like macros, we can do this in the same way that we can create a program in C that will run in both Windows and Linux. We use conditional macros to include conditionally custom source code for each environment. For this, we must

have two different GLSL macros. One that will be defined when we need a vertex shader and the other when we need a fragment shader:

Section: Local Variables (interface.c) (continuation):

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

In the shader source code, we just need to check which macro is defined and we can include code conditionally depending on this.

Next we need to specify the default precision for each variable type if the user do not specify one in variable declaration. This is done using the keyword `precision` followed by the precision qualifier (`lowp`, `mediump`, `highp`) and by a variable type. Here we will define as default the highest possible precision. For less important variables, the user always can lower the precision trying to make the shader more efficient.

Section: Local Variables (interface.c) (continuation):

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

The next thing that we must define is how to support libraries in our GLSL code. They can make shader definition easier. The user can define additional GLSL functions that will be available in all interface shaders storing its source code in the following variable:

Section: Local Variables (interface.c) (continuation):

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

The variable will be changed by the function seen in the Introduction to increment GLSL with new functions. Such function is extremely simple, it justs assign a string with function source code to the variable above:

Section: API Functions Definition (interface.c) (continuation):

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

But we should not forget that when finalizing our API, we need to undo the assignment. Otherwise, when a user finalize and initialize again the API, the second initialization would get the same function definitions assigned in the first initialization:

Section: Finalizing Interface API:

```
shader_library = "";
```

The list of default variables and attributes passed to the shader also must be part of the source code. We will define them later, but for now we show that they will be in this variable:

Section: Local Variables (interface.c) (continuation):

```
static const char shader_variables[] = ""
        <Section to be inserted: Shader Attributes, Uniforms and Varyings>
"";
```

Now we will see how to compile a shader given these informations and a string with its source code. First we will include the standard input/output library to print in the screen error messages if there are errors in the shader source code. The OpenGL headers are already included in `interface.h`.

Section: Local Headers (interface.c) (continuation):

```
#if defined(__linux__) || defined(BSD)
#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
#include <stdio.h>
```

Once we have the necessary headers, we can define the function that given a shader source code, compiles it to a complete shader program. Compiling a shader means creating on OpenGL the two kind of shaders (vertex and fragment), compile both and link them in a single program.

Section: Auxiliary Local Functions (interface.c):

```
<Section to be inserted: Functions to Check Compiling Errors>
static GLuint compile_shader(const char *source_code){
  GLuint vertex_shader, fragment_shader, program;
  const char *list_of_source_code[6];
  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] = shader_variables;
  list_of_source_code[5] = source_code;
  glShaderSource(vertex_shader, 6, list_of_source_code, NULL);
  list_of_source_code[1] = fragment_shader_macro;
  glShaderSource(fragment_shader, 6, 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;
}
```

What we are not showing above is hoe we verify if the shader was compiled and linked successfully. To do this with shader compilation, we use the function below. It checks if a compiling error happened and if so, it reads in the logs what was wrong and print in the screen. Notice that here we are using the temporary allocation and deallocation functions to manage space in the memory for the error message. The function also returns if an error was found or not.

Section: Functions to Check Compiling Errors:

```
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;
}
```

Checking if some error happened during the linking stag is similar. However, after the linking we can make even better. We can try to validate the shader simulating its usage to detect additional errors that could not be detected in syntatic analysis. Howerer, this is an expensive operation, we will do this only if the macro `W_DEBUG_INTERFACE` is defined. In this case, we assume that we are in debug mode.

Section: Functions to Check Compiling Errors (continuation):

```
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;
}
```

And this ends the description of how we compile a new shader and print messages in case of errors in the source code.

Now we can define the default shader source code that will be used if the user do not use a custom shader. The shader objective will be showing the texture associated with the interface.

The source code for the default shader will be stored in the following constant:

Section: Local Variables (interface.c) (continuation):

```
static const char default_shader_source[] = ""
"#if defined(VERTEX_SHADER)\n"
                    <Section to be inserted: Default Vertex Shader>
"#else\n"
                   <Section to be inserted: Default Fragment Shader>
"#endif\n"
"";
```

The source code will be composed by string literals stored in the constant variable above. In the case of the vertex shader, it will multiply each vertex by the model-view matrix that stores information about its size, position and rotation. It will also set which texture coordinate is associated with each vertex.

To understand the code, you need to consider that all interfaces will have the same vertices: $(0,0,0),(1,0,0),(1,1,0),(0,1,0)$. Each vertex have as argument its texture coordinate. They are declared below:

Section: Local Variables (interface.c) (continuation):

```
static const float interface_vertices[20] = {
  0.0, 0.0, 0.0, // First Vertice
  0.0, 0.0,     // Texture coordinate
  1.0, 0.0, 0.0, // Second vertice
  1.0, 0.0,     // Texture coordinate
  1.0, 1.0, 0.0,  // Third Vertice
  1.0, 1.0,      // Texture coordinate
  0.0, 1.0, 0.0, // Fourth vertice
  0.0, 1.0};    // Texture coordinate
static GLuint interface_vbo;
```

The order in which we declare these vertices is important. If we follow these vertices in the declared order, we draw a square in a counter-clockwise order. This is how we declare to OpenGL that we are seeing the front of this interface, not its back. We do this to be prepared for if the user configures OpenGL to do not show the back of geometric figures. This is a very common optimization and a necessary one in lots of contexts. Specifying this order guarantees that the interface will be visible even with this option enabled.

The rotated textures coordinates will be used if the macro `W_FORCE_LANDSCAPE` is defined and if the window height is greater than window width. In this case we swap the interface width and height and also rotate the texture. This rotates our interfaces.

These vertices will be loaded in the video card and after this we could render them asking to render the corresponding VBO ("vertex buffer object"). LOading the vertices in the VBO is done in the Initialization:

Section: Initializing Interface API (continuation):

```
glGenBuffers(1, &interface_vbo);
glBindBuffer(GL_ARRAY_BUFFER, interface_vbo);
```

```
// Enviando os vrtices para a placa de vdeo:
glBufferData(GL_ARRAY_BUFFER, sizeof(interface_vertices), interface_vertices,
             GL_STATIC_DRAW);
```

In the Finalization we need to destroy the VBO:

Section: Finalizing Interface API (continuation):

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

Anyway, all interfaces will begin as a rectangle centered in the screen which will occupy all the screeen size, as OpenGL defines "1" as the rendering area width and height. To transform this fix square in different sizes, positions and rotations, we will use different model-view matrices. Each interface will have its own model-view matrix.

Section: Default Vertex Shader:

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

In the fragment shader, for each pixel we will draw the color associated with the texture in a given position:

Section: Default Fragment Shader:

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

Now we need to define in the shaders its attributes, uniforms and varyings. The attributes are read-only data specified for each vertex. The attributes that we will define are the vertex position in format (x, y, z) and texture coordinate in format (x, y). The coordinates x and y will be between 0 and 1 and $z = 0$, as described previously. Attributes are declared only in vertex shaders.

Section: Shader Attributes, Uniforms and Varyings:

```
"#if defined(VERTEX_SHADER)\n"
"attribute vec3 vertex_position;\n"
"attribute vec2 vertex_texture_coordinate;\n"
"#endif\n"
```

Uniforms are variables that also will be passed to vertices, but they will not change between them. Therefore, the fragment shader do not need to interpolate its value. We will store as uniforms the foreground color, the background color, the model-view matrix, the object size in pixels, the current time in seconds, the integer associated with each interface and its texture:

Section: Shader Attributes, Uniforms and Varyings (continuation):

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

Finally we have varying variables. They can be modified and set on the vertex shader and its interpolated value will be given to the fragment shader. Here we put the texture coordinate.

Section: Shader Attributes, Uniforms and Varyings (continuation):

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

Notice that we did not use all the variables defined in the default shader. The unused variables will be discarded during the compilation as an optimization. But we declared them anyway just to list the supported variables that can be used by custom shaders defined by the user.

By the way, talking about custom shaders, they will be sent by the user using the function `_Wnew_interface` and will be specified as a path to a file where the source code is. For this case, we will define a function that creates a new shader program from a filename given as argument:

Section: Auxiliary Local Functions (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 comeo:
  fseek(fp, 0, SEEK_END);
  source_size = ftell(fp);
  // Aloca e l 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;
}
```

The last problem that we will solve in this section is what should we render as default texture when the user supplies no texture for a given interface. For these cases, We will create a texture composed by a single white pixel:

Section: Local Variables (interface.c) (continuation):

```
static GLuint default_texture;
```

We will create this texture during the initialization. This is done in the following code, where we generate the texture in OpenGL, associate it as the current 2D-texture and specify the white pixel explaining how it is represented.

Section: Initializing Interface API (continuation):

```
{
  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);
```

```
}
```

And in the finalization we discard the created texture:

Section: Finalizing Interface API (continuation):

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

5. The Model-View Matrix

As we have seen, all interfaces will be represented by four vertices whose edges have the fixed size "1". What will make each interface have its own size, position and rotation is the model-view matrix that we will use for each interface.

To understand the matrix, first recall that when we use it in our default vertex shader, whose code was already shown, we convert each vertex to a point in 4 dimensions, instead of 3 or 2. The piece of code `vec4vertex_position, 1.0)` basically take the coordinate in three dimensions and add a fourth dimension whose value always will be 1.

We will do it because only in 4 dimensions it is possible to represent all the geometric operations of rotation, translation and scaling as matrix multiplications. Which means that only with 4 dimensions all them are linear operations. Video cards are very fast when computing matrix multiplication, treating these geometric operations in other ways that are not matrix multiplication would create a negative impact in the performance.

Before seeing the final form of the model-view matrix, we will first see each of its different parts isolated. First assume that we have a four-dimensional vector $(x_0, y_0, z_0, 1)$ multiplying our matrix. If we are interested only in translating the vector to other position, we could multiply the translation matrix below by our vector:

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

To change the size of some interface, both the width and the height, assuming that the interface is centered in the origin $(0, 0, 0, 1)$, we can multiply each vector for the following scaling matrix:

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

And finally, to rotate our interfaca θ radians, we could use the following matrix that would produce the right result based on the trigonometric formula for rotation:

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

To form our model-view matrix, we will multiply all these matrices depending of the values that we want for the scale, rotation and translation. However, the order in which we multiply these matrices change the result. The correct order for each operation is:

1. First we should center the interface square in the OpenGL origin. (Matrix A).
2. Then we increase or decrease the size of each side of our interface using its pixel size. (Matrix B).
3. We rotate the interface θ radians. (Matrix C)
4. We resize again the interface, this time adjusting to screen proportion using OpenGL size. (Matrix D)
5. Finally, we translate the interface for its correct position using OpenGL coordinate. (Matrix E)

To make these operations in the correct order, each vetor v should be multiplied in the following order:

$$E(D(C(B(Av)))) = ((((ED)C)B)A)v$$

If we use other order, things can go wrong. For example, instead of rotating the interface using its center as the axis, we would use its lower left corner or some other position as axis.

Instead of always having to multiply four matrices, we will compute the format of the final matrix obtained when we multiply them. This final matrix will be our model-view matrix. For example, assuming that w_w is window width and h_w is window height, multiplying matrices E and D, we get:

$$ED = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2/w_w & 0 & 0 & 0 \\ 0 & 2/h_w & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2/w_w & 0 & 0 & x \\ 0 & 2/h_w & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that we are using $2/w_w$ and $2/h_w$ because we assume that window width and height in OpenGL have size 2. Therefore, an object with width equal w_w should have width 2 to occupy the whole screen. The same happens with height.

If we multiply it by matrix C, we get:

$$EDC = \begin{bmatrix} 2/w_w & 0 & 0 & x \\ 0 & 2/h_w & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ \sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2/w_w \cdot cos(\theta) & -2/w_w \cdot sin(\theta) & 0 & x \\ 2/h_w \cdot sin(\theta) & 2/h_w \cdot cos(\theta) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now if we multiply the result by matrix B, considering that h_p is the object height in pixels and w_p is the width in pixels, we obtain:

$$EDCB = \begin{bmatrix} 2/w_w \cdot cos(\theta) & -2/w_w \cdot sin(\theta) & 0 & x \\ 2/h_w \cdot sin(\theta) & 2/h_w \cdot cos(\theta) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w_p & 0 & 0 & 0 \\ 0 & h_p & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2w_p/w_w \cdot cos(\theta) & -2h_p/w_w \cdot sin(\theta) & 0 & x \\ 2w_p/h_w \cdot sin(\theta) & 2h_p/h_w \cdot cos(\theta) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And finally, we multiply this by matrix A, which center the interface in $(0, 0, 0, 1)$. Recall that the interface originally is centered in position $(1/2, 1/2, 0, 1)$. Therefore, matrix A is just a translation matrix with constant values that moves each vertex $1/2$ to the left and $1/2$ down:

$$EDCBA = \begin{bmatrix} 2w_p/w_w \cdot cos(\theta) & -2h_p/w_w \cdot sin(\theta) & 0 & x \\ 2w_p/h_w \cdot sin(\theta) & 2h_p/h_w \cdot cos(\theta) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -1/2 \\ 0 & 1 & 0 & -1/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} 2w_p/w_w \cdot cos(\theta) & -2h_p/w_w \cdot sin(\theta) & 0 & x + h_p/w_w \cdot sin(\theta) - w_p/w_w \cdot cos(\theta) \\ 2w_p/h_w \cdot sin(\theta) & 2h_p/h_w \cdot cos(\theta) & 0 & y - w_p/h_w \cdot sin(\theta) - h_p/h_w \cdot cos(\theta) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The format above is the final form of our model-view matrix for each interface.

However, the values (x, y) above are in OpenGL coordinates. But in this API we measure differently the coordinates. In our measures, the origin of ou coordinate is the lower left corner of the screen, not in the center of screen like in OpenGL. And we measure positions in pixels, not using the constant size 2 for the screen, like in OpenGL. A function that fills the model-view matrix must first convert our coordinates and sizes to OpenGL format.

Assuming that we have a pointer `i` to a given interface, this can be done with the following code:

Section: Converting Coordinates and Sizes:

```
x = 2.0 * (i -> _x) / (*window_width) - 1.0;
y = 2.0 * (i -> _y) / (*window_height) - 1.0;
```

And using the conversion rules above, we can fill the model-view matrix for the first time with the function below:

Section: Auxiliary Local Functions (interface.c):

```
static void initialize_model_view_matrix(struct user_interface *i){
  GLfloat x, y;
            <Section to be inserted: Converting Coordinates and Sizes>
  GLfloat cos_theta = cos(i -> _rotation);
  GLfloat sin_theta = sin(i -> _rotation);
  /* Primeira Coluna */
  i -> _transform_matrix[0] = (2 * i -> width / (*window_width)) *
    cos_theta;
  i -> _transform_matrix[1] = (2 * i -> width / (*window_height)) *
    sin_theta;
  i -> _transform_matrix[2] = 0.0;
  i -> _transform_matrix[3] = 0.0;
  /* Segunda Coluna */
  i -> _transform_matrix[4] = -(2 * i -> height / (*window_width)) *
     sin_theta;
  i -> _transform_matrix[5] = (2 * i -> height / (*window_height)) *
    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] = x +
    (i -> height / (*window_width)) * sin_theta -
    (i -> width / (*window_width)) * cos_theta;
  i -> _transform_matrix[13] = y -
    (i -> width / (*window_height)) * sin_theta -
    (i -> height / (*window_height)) * cos_theta;
  i -> _transform_matrix[14] = 0.0;
  i -> _transform_matrix[15] = 1.0;
}
```

Using sine and cosine functions requires the header for mathematical operations:

Section: Local Headers (interface.c) (continuation):

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

6. Managing Data Structures

As was explained and defined before, the data structures that our API will manage will be a struct for interfaces, another to create linking to interfaces, a third one to create markings in the history and finally, a linked list to store the previous structures.

Our liked list will be accessed using two different pointers. The first one will point to the last structure created and inserted in the list. The second will point to the last history marking inserted in the list. We also will define a mutex to ensure that only one thread can update the linked list at a given moment:

Section: Local Variables (interface.c) (continuation):

```
static void *last_structure = NULL;
```

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

The mutex that wil control access to the linked list needs to be created during the initialization:

Section: Initializing Interface API (continuation):

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

We will put the code to finalize this mutex later, when we define the code thatfinalize the entire linked list.

6.1. Creating and Destroying Shader Data Structures

In Section 4 we presented the default shader source code and how to compile shaders. But we did not integrate that functions and data structures with the shader data structure. We will do it here in this Section.

Our API have no exorted function to create new shaders. Instead of this, it will create new shaders when the user creates a new interface and associate it with a custom shader passing a filename with shader source code. That is, when the user uses the API function `_Wnew_interface` with the second parameter, that specifies a filename with shader source code, different than `NULL`.

We also need to create an structure for our default shader, whose source code is in Section 4. The default shader structure address will be stored in this pointer:

Section: Local Variables (interface.c) (continuation):

```
struct shader *default_shader;
```

Now we will define the internal function that generates new shader structures. This function will get as argument a string with the filename that stores the shader source code. The function returns a pointer for the new shader structure and also insert such data structure in the linked list. Except if we pass `NULL` as the filename. In this case, we create a new shader structure using the default shader source code defined in Section 4. And we do not insert it in the linked list.

Section: Auxiliary Local Functions (interface.c) (continuation):

```
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);
    // Binding attributes
    glBindAttribLocation(new -> program, 0, "vertex_position");
    glBindAttribLocation(new -> program, 1, "vertex_texture_coordinate");
    // Getting uniforms
    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_mouse_coordinate = glGetUniformLocation(new -> program,
                                                    "mouse_coordinate");
    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;
}
```

Basically the code above uses the functions defined in Section 4 to compile the shader and after compiling it, gets the address where we can find the compiled shader variables that can be changed before executing the shader. These addresses are stored in the shader data structure.

Now we can initialize the default shader in the Initialization:

Section: Initializing Interface API (continuation):

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

To destroy a shader, we need to inform that the compiler shader program will not be executed anymore and should be deleted. Then, we disallocate the allocated memory for the struct:

Section: Auxiliary Local Functions (interface.c) (continuation):

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

The default shader should be destroyed in the API finalization:

Section: Finalizing Interface API (continuation):

```
destroy_shader(default_shader);
```

6.2. Creating and Destroying Interfaces

As was described in the Introduction, all new interfaces will be created using function `_Wnew_interface`. This function will allocate the new interface, execute the adequate functions to initialize it and will include it in the linked list. The function definition is:

Section: API Functions Definition (interface.c) (continuation):

```
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 = new_interface-> _x = x;
    new_interface -> y = new_interface-> _y = y;
    new_interface -> rotation = new_interface -> _rotation = 0;
#if defined(W_FORCE_LANDSCAPE)
   if(*window_height > *window_width){
      new_interface-> _x = *window_width - y;
      new_interface -> _y = x;
      new_interface -> _rotation += M_PI_2;
   }
#endif
    new_interface -> z = z;
    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) // Still need to load texture:
      new_interface -> _loaded_texture = false;
    else // No texture to be loaded:
      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;
}
```

Despite being a long function, what it does is just allocate the interface with the function set as our permanent allocator, initialize the variables in the interface, insert it in the linked list and run the correct function to load the texture, depending on the filename extension.

Destroy an interface means checking if some of its variables have values different than NULL. If so, this means that these variables were allocated by the loading function andd should also be disallocated. Some variables of this kind is the texture variable and the list of frame duration for animated textures. The destructor function should first disallocate these variables, and then disallocate the entire struct:

Section: Auxiliary Local Functions (interface.c) (continuation):

```
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. Creating and Destroying Link to an Existing Interface

As was explained, instead of creating a new interface, we can create a link to an existing

interface. This makes that interface acts as a newly created interface regarding functions that interact with it. A linking will be just a pointer to another interface in the linked list, created using `_Wlink_interface`. The function that creates it is defined below:

Section: API Functions Definition (interface.c) (continuation):

```
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;
}
```

Destroying a link is done just running the disallocation function. Because of this, we will not define an auxiliary function to perform this action.

6.4. Creating Markings in the Interface History

To create a new marking in our interface history, the user should call the function `_Wmark_history_interface`. After this marking, all interfaces created before it will became inaccessible until the marking is removed.

Section: API Functions Definition (interface.c) (continuation):

```
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);
  }
}
```

As it is important to have a last marking in the history to keep track of the number of active interfaces, we create the first marking in the initialization. And it will be destroyed only in the finalization. If the user uses the API function to remove the first marking before the finalization, we will ignore the action.

Section: Initializing Interface API (continuation):

```
_Wmark_history_interface();
```

Section: Finalizing Interface API (continuation):

```
// Erases all markings, except the first one:
while(last_marking -> previous_marking != NULL){
  _Wrestore_history_interface();
}
// Erase the interfaces after the first marking:
_Wrestore_history_interface();
// Erases the first marking:
if(permanent_free != NULL)
  permanent_free(last_marking);
last_marking = NULL;
last_structure = NULL;
MUTEX_DESTROY(&linked_list_mutex);
```

The function `_Wrestore_history_interface` is an API function that we will define in the next Section. It destroys and removes all structures created after the last marking. Then it removes and destroy the last marking. Except that it never will destroy the first marking. Because of this, in the code above we destroy manually the first marking with `permanenet_free`.

6.5. Removing Markings and Interfaces

We will remove and disallocate interfaces using the function to restore the history to a previous point. The function will remove all interfaces created after the last marking and will remove the last marking. This will be made with function `_Wrestore_history_interface`.

After this function, the last marking will be removed and we will remember the previous marking that existed before it. If there is no other marking before the last one, we will not remove the marking but we will still restore the status to how it was when the first marking was created.

The function is implemented as below:

Section: API Functions Definition (interface.c) (continuation):

```
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;
  // Removing interfaces after the last marking:
  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;
```

```
  }
  // Removing last marking, except if it is also the first marking:
  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;
  MUTEX_SIGNAL(&linked_list_mutex);
}
```

The function first removes all structures created after the last marking. Only after this the last marking is removed, except if it is also the first marking. The pointer to the last structure in the linked list is updated in the beginning of the function, checking the pointer to the element before presented and stored in the last marking that was removed.

7. Rendering Interfaces

To render interfaces, we should take into account the correct order in which each interface is rendered. One of the simplest methods to do it is drawing the interfaces in any order and use th z coordinate nd a z-buffer to check if each interface should be drawn using a depth test. However, when dealing with transparent objects, not always this produces the correct result. If we first draw a tranparent object and after draw an opaque object behind it, parts of the opaque object that should be visible because of the transparency would not be visible.

But for our interfaces, as we are always dealing with bidimentional objecs that always are parallel to the axis z, we instead will store an ordered list of interfaces that determines the correct drawing order for our interfaces. The list will be ordered by the z coordinate and will make all interfaces be drawn before any interface in front of it. Basically, this is called the "painter algorithm". As a painter always begin the drawing by more distant objects, we also will begin drawing more distant interfaces.

If two interfaces havethe same value in their z coordinate, we can draw them in any order.

The pointer that stores our ordered list address is this:

Section: Local Variables (interface.c) (continuation):

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

During the API initialization we should initialize the above variables:

Section: Initializing Interface API (continuation):

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

After finalizing our API, if this interface list was allocated, we need to disallocate it, set this pointer to null again and finalize the mutex:

Section: Finalizing Interface API (continuation):

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

```
z_list_size = 0;
```

As we did in the finalization, when we restore our interface history, we change the list of interfaces that we are viewing, restoring previous interfaces. In this case we need to reset our list making it empty again.

Section: Restoring History:

```
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);
```

We must initialize this list in the rendering function `_Wrender_interface`. Inside this function, we should check if the size of this list is equal the number of active interfaces in the current moment (the number of active interfaces can be checked in the last history marking). If we have a different value, this means that new interfaces were created and we should regenerate our ordered list. The code that perform this can be seen below:

Section: Generating Ordered List of interfaces:

```
if(z_list_size != last_marking -> number_of_interfaces){
  void *p;
  unsigned 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 if(((struct user_interface *) p) -> type == TYPE_LINK)
      z_list[i] = ((struct link *) p) -> linked_interface;
    else if(((struct user_interface *) p) -> type == TYPE_SHADER)
      i --; // Not an 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);
```

```
}
```

Notice that the scenario in which the ordered list should be rebuild with the code above in uncommon. We run the code above when we are rendering for the first time in a main loop or after restoring our interface history. The only expected scenario where we run the code above with more frequency is if the user is creating new interfaces in its program main loop. In such case, every rendering after a new interface creation, runs the code above. However, this is a bad practice and so we will assume that the code above rarely will be executed, besides being placed in the rendering function. In the typical case, we already have an ordered list with all the active interfaces, and we need only to adjust the interface position in the list if the interface is moved.

Once we have the ordered list of interfaces, we can just render each interface in the order they appear in the list. For each one we load the correct shader, pass the attributes, uniforms and varyings to the shader program and use the OpenGL function to render the interface vertices.

We render the interface when the function `_Wrender_interface` is invoked. The function gets as parameter the current time in microseconds. But to know what was the elapsed time between two consecutive renderings, we need to store the previous time received by the function. We will store this previous time in the variable below, that is initialized with zero before receiving the first time:

Section: Local Variables (interface.c):

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

But we also need to initialize this variable in the initialization. Otherwise, the variable would store incorrect values if the API is finalized and initialized again:

Section: Initializing Interface API (continuation):

```
previous_time = 0;
```

And below we show the function that iterates over the ordered list of interfaces and render each one, updating also the time variables. The function first updates the time, after load the interface vertices indicating how they are represented, and finally iterates over interfaces.

For each interface, we load the shader program and pass to it each uniform and varying necessary stored in the interface struct. After this, we check the interface textures. If we have more than one texture, this means that we could have animated textures (for example, we could have loaded an anmated GIF). If we indeed have an animated texture, we need to check the elapsed time between renderings and the time since we are in the current frame animation. If we see that we need to change to the next animation frame, we update it. And finally we pass to the shader the correct texture. Only after iterate over all interfaces, we store the time in our static global variable that remember when the last rendering happened.

The implementation for the procedure described above is:

Section: API Functions Definition (interface.c) (continuation):

```
void _Wrender_interface(unsigned long long time){
            <Section to be inserted: Generating Ordered List of interfaces>
  {
    unsigned i, elapsed_time;
    if(previous_time != 0)
      elapsed_time = (int) (time - previous_time);
    else
      elapsed_time = 0;
    // Carregando os vrtices do VBO:
    glBindBuffer(GL_ARRAY_BUFFER, interface_vbo);
```

```
  // Especificando como os dados esto representados no VBO:
  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float),
                        (void *) 0);
  glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float),
                        (void *) (3 * sizeof(float)));
  glEnableVertexAttribArray(0);
  glEnableVertexAttribArray(1);
  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);
    glUniform2f(z_list[i] -> shader_program -> uniform_mouse_coordinate,
                z_list[i] -> mouse_x, z_list[i] -> mouse_y);
    // O shader recebe contagem de tempo em segundos mdulo 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. Moving, Rotating and Resizing Interfaces

8.1. Moving Interfaces

Moving an interface means updating its variables (x, y, z). Other than this, we also need to update the model-view matrix to use the new values x and y after converting them to OpenGL coordinates. And if we are changing coordinate z, then we may need to change the position of this interface in the ordered list of interfaces that determine the order of drawing. All these changes must be done after requesting the use of the interface mutex to prevent two simultaneous invocations of this function for the same interface.

The API function that moves interfaces is `_Wmove_interface` and we define the function as:

Section: API Functions Definition (interface.c) (continuation):

```
void _Wmove_interface(struct user_interface *i,
                      float new_x, float new_y, float new_z){
  GLfloat x, y;
  GLfloat cos_theta = cos(i -> _rotation);
  GLfloat sin_theta = sin(i -> _rotation);
  MUTEX_WAIT(&(i -> interface_mutex));
  i -> x = i -> _x = new_x;
  i -> y = i -> _y = new_y;
#if defined(W_FORCE_LANDSCAPE)
  if(*window_height > *window_width){
     i -> _x = *window_width - new_y;
     i -> _y = new_x;
  }
#endif
```

<Section to be inserted: Converting Coordinates and Sizes>

```
  i -> _transform_matrix[12] = x +
    (i -> height / (*window_width)) * sin_theta -
    (i -> width / (*window_width)) * cos_theta;
  i -> _transform_matrix[13] = y -
    (i -> width / (*window_height)) * sin_theta -
    (i -> height / (*window_height)) * cos_theta;
  if(new_z != i -> z){ // Atualizando lista ordenada de interfaces
    unsigned 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. Rotating Interfaces

Rotating an interface involves reserving its mutex, updating the rotation variable and also updating its model-view matrix. The code for this can be seen below:

Section: API Functions Definition (interface.c) (continuation):

```
void _Wrotate_interface(struct user_interface *i, float rotation){
  GLfloat x, y;
  GLfloat cos_theta = cos(rotation);
  GLfloat sin_theta = sin(rotation);
  MUTEX_WAIT(&(i -> interface_mutex));
  i -> rotation = i -> _rotation = rotation;
#if defined(W_FORCE_LANDSCAPE)
  if(*window_height > *window_width)
    i -> _rotation += M_PI_2;
#endif
```

<Section to be inserted: Converting Coordinates and Sizes>

```
  i -> _transform_matrix[0] = (2 * i -> width / (*window_width)) *
    cos_theta;
  i -> _transform_matrix[1] = (2 * i -> width / (*window_height)) *
    sin_theta;
  i -> _transform_matrix[4] = -(2 * i -> height / (*window_width)) *
     sin_theta;
  i -> _transform_matrix[5] = (2 * i -> height / (*window_height)) *
    cos_theta;
  i -> _transform_matrix[12] = x +
    (i -> height / (*window_width)) * sin_theta -
    (i -> width / (*window_width)) * cos_theta;
  i -> _transform_matrix[13] = y -
    (i -> width / (*window_height)) * sin_theta -
    (i -> height / (*window_height)) * cos_theta;
  MUTEX_SIGNAL(&(i -> interface_mutex));
}
```

8.3. Resizing Interfaces

Resizing interfaces, like rotating them, involves only reserving the mutex, updating the associated variables and updating the model-view matrix.

The code to resize interfaces is:

Section: API Functions Definition (interface.c) (continuation):

```
void _Wresize_interface(struct user_interface *i,
                        float new_width, float new_height){
  GLfloat x, y;
  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;
              <Section to be inserted: Converting Coordinates and Sizes>
  i -> _transform_matrix[0] = (2 * i -> width / (*window_width)) *
    cos_theta;
  i -> _transform_matrix[1] = (2 * i -> width / (*window_height)) *
    sin_theta;
  i -> _transform_matrix[4] = -(2 * i -> height / (*window_width)) *
     sin_theta;
  i -> _transform_matrix[5] = (2 * i -> height / (*window_height)) *
    cos_theta;
  i -> _transform_matrix[12] = x +
    (i -> height / (*window_width)) * sin_theta -
    (i -> width / (*window_width)) * cos_theta;
  i -> _transform_matrix[13] = y -
    (i -> width / (*window_height)) * sin_theta -
    (i -> height / (*window_height)) * cos_theta;
  MUTEX_SIGNAL(&(i -> interface_mutex));
}
```

8. Interacting with Interfaces

Given the active interfaces, we can interact with them passing the mouse cursor over them or clicking with some mouse button on them. To manage these interactions we use the function `_Winteract_interface` that gives us information about what the mouse is doing and automatically executes functions associated with interactions with interfaces.

To make this possible we need to memorize the mouse button states. Because we need to know not only the current mouse button status, but also the previous status. To memorize this, we will use the following variables, one for each mouse button:

Section: Local Variables (interface.c):

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

We initialize these variables with false during the API initialization:

Section: Initializing Interface API (continuation):

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

Knowing the previous mouse state and the current state, we are able to notice not only that the user is clicking in something, but also if the clicking is beginning in this frame or it is just continuing. We know not only if the user is not clicking, but also know if the button was released at some moment between the last frame and the current frame. Only with these information we can execute correctly the functions that interact with each interface.

In a given frame we can interact only with a single interface, as we have only a single mouse cursor. But if more than one interface occupy the same position, how could we know

which one is the correct interface? Well, fortunately we a list of active interfaces where they are ordered according with the order in which they are draw. When more than one interface occupy the same point where the mouse cursor is, the correct interface is the one that is in the front of others.

So we iterate over each interface in the reverse order in which they are drawn and check the first interface under the mouse cursor. This is the current interface. If there is another interface marked as current one, we remove the mark and consider this the previous interacted interface. If necessary, and if they exist, we execute functions associated with the previous interface (the mouse cursor abandoned this interface) and with the current one (if the mouse hovered over it, or clicked on it).

The code to perform this is:

Section: API Functions Definition (interface.c) (continuation):

```
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;
          <Section to be inserted: Converting Mouse Coordinates to x and y>
    z_list[i] -> mouse_x = x - z_list[i] -> x + (z_list[i] -> width / 2);
    z_list[i] -> mouse_y = y - z_list[i] -> y + (z_list[i] -> height / 2);
    if(current == NULL &&
       z_list[i] -> mouse_x  > 0 && z_list[i] -> mouse_x < z_list[i] -> width &&
       z_list[i] -> mouse_y  > 0 && z_list[i] -> mouse_y < z_list[i] -> height)
      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;
}
```

This is a very direct code, but we ommited the part where we transform the mouse coordinates in coordinates (x, y) to compare with each interface. If the interface is not rotated, no transformation is necessary and we could use directly the mouse coordinates. But if the interface is rotated, the easiest way to check if the mouse is over it is ignoring its rotation and rotate the mouse in the opposite angle, with axis on the center of the interface. After this transformation, we check if the coordinate is over the interface in the exactly same manner.

Our code to generate (x, y) just checks if the interface is rotated. If not, no transformation is necessary. If yes, we just make some trigonometric calculations to determine the transformed coordinate:

Section: Converting Mouse Coordinates to x and 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;
}
```

References

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