Weaver User Interface

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

Abstract: *This article presents the implementation of the user interface used by the Weaver Game Engine. The code provided is intended for creating buttons, text, menus, and other user interface elements. It primarily manages shaders and creates elements that can be moved, rotated, clicked, and made to respond to mouse hovering. Support for adding sound effects is also included. The API is designed to be flexible, allowing users to extend it and customize its behavior by registering new functions.*

1. Introduction

A graphical user interface is how a program communicates with and receives input from the user. In a typical program, we might have menus, buttons, pop-up windows, and other common graphical elements. In games, the user interface is usually conceptually simpler, consisting of a few menus and visual elements that provide information about the game state—for example, a life counter in the corner of the screen.

The truth is that in a computer game, we can assume almost nothing about the appearance of the user interface. A strategy game might include many menus that appear when the player clicks on a unit, while a platform game might only display information about equipped items and a life counter. The appearance and behavior of such elements can vary greatly depending on the game genre.

The only universal characteristic we assume for our interface elements is that they appear on top of the game scene and are not obscured by objects from the game world. Typically, we render the interface after rendering the game world, and elements from the game world cannot interact with the interface. Interfaces are not part of the simulated world—they are merely elements that convey information to the player that would otherwise be difficult to represent.

Our objective here will be creating an API 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);
```

The first argument is a file that will be opened and interpreted to obtain information about the interface texture (for example, it could be an image filename). The second argument is a file containing the OpenGL shader. Both arguments can be NULL. A NULL shader means a default shader will be used. A NULL texture file means the interface will have no texture. The remaining arguments specify the initial position and size of the interface.

We can define shaders more easily by providing an additional library of GLSL functions. This can be done by passing a string containing function definitions in GLSL to the following function:

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

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

We can move interfaces using the function:

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

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

We assume that the position of an interface is given by the coordinates of its center. The x and y axes represent the horizontal and vertical position, respectively. The z axis determines which interface appears in front when multiple interfaces occupy the same space.

We can also rotate it with the function:

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

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

The function above takes as an argument the amount, in radians, by which the interface will be rotated relative to its default orientation.

We can also resize the interface with the function below:

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

```c
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:

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

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

The time parameter in the function above is used when the interface needs to know how many microseconds have passed since the last function call. This is useful for correctly rendering interfaces with animated textures.

However, rendering all previously created interfaces may not always be what the user wants. Sometimes, a developer may want to render only the interfaces created after a certain point in time. For this purpose, the function below creates a marker in the interface history. From that point on, when we request to render all interfaces, only those created after the marker will be rendered.

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

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

But what if we want to render some of the interfaces created before the last history marker? In this case, we can create a new interface that is, in fact, just a reference to a previously created one.

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

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

To interact with all interfaces created up to the last marker, we use the function below. It executes the programmed actions, if any, for each interface when the user hovers over it, moves the mouse away, or clicks on it.

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

```c
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 interfaces that are no longer needed? For this purpose, we use the following function, which deletes all interfaces created after the last history marker. This function also removes the last marker, restoring the state to what it was just before the marker was created. Interfaces created before that marker will be rendered again.

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

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

All of this requires managing our interface history, shaders, and markers, which in turn involves memory allocation and deallocation. There are two types of allocations we use: for permanent data that will remain in memory for a long time, and for temporary data that will be deallocated shortly afterward. We will provide functions to handle allocation and deallocation in both cases.

Section: Local Variables (interface.c):

```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 use the allocation and deallocation functions from the standard library. However, users can provide custom functions to replace them. In addition to these four functions, we also support custom functions that are called immediately before and after loading a new interface. These functions are initially set to NULL, but users can later assign them to other functions:

Section: Local Variables (interface.c):

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

All custom functions are chosen during the API initialization:

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

```c
#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 initialize the deallocation functions with `NULL`. This means that allocated memory will not be deallocated. This can be useful in scenarios where the memory manager uses its own garbage collector and does not want interference.

The function above accepts a variable number of arguments. First, it receives pointers to variables where the window width and height will be stored. These values will be stored at the provided locations:

Section: Local Variables (interface.c):

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

The window size arguments are followed by the custom functions described earlier. After the six initial functions comes a NULL-terminated list of arguments, each consisting of a string followed by a function that creates a new interface given a file. The string represents a filename extension (for example, “gif”, “jpg”, or others). The function corresponding to the file extension has the following type:

Section: Local Macros (interface.c):

```c
typedef void pointer_to_interface_function(void *(*)(size_t), void (*)(void *),
                                          void *(*)(size_t), void (*)(void *),
                                          void (*)(void), void (*)(void),
                                          char *, void *);
```

The function receives as arguments the allocation and deallocation functions, a filename, and a void pointer to the interface that should be populated with information from the file. Each of these functions is expected to correctly interpret files with the previously specified extensions and fill the given interface with textures extracted from the file. In this way, we delegate to the user the responsibility of providing functions capable of creating interfaces from different file formats. The last argument is a void pointer because these functions could be used both for visual user interfaces and for sound interfaces. Each type of interface uses different structs.

Since we have an initialization function, we also need a finalization function:

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

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

Up to this point, we have presented functions and their behavior for visual interfaces. However, you can also create audio interfaces using the function below:

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

```c
struct sound *_Wnew_sound(char *filename);
```

To play the sound loaded by the previous function, we use the function below:

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

```c
struct sound *_Wplay_sound(struct sound *snd);
```

This concludes the description of our API functions.

The API behavior can also be modified by defining the macro `W_FORCE_LANDSCAPE`. In an environment where the width is greater than the height, setting this macro has no effect. However, if the window height is greater than the width and this macro is defined, the x and y axes will be rotated 90 degrees counter-clockwise. This feature is useful for creating more consistent user interfaces on mobile devices. In such cases, we can always assume that there is more horizontal than vertical space, and, if necessary, the user can rotate their 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 show below the structure of the file `interface.h`:

File: `src/interface.h`:

```c
#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 code above shows the default boilerplate for defining a header in our C API. The first two lines and the last one are macros that ensure the header is not included more than once in a single compilation unit.

Lines 3, 4, 5, and the three lines before the last one make the header compatible with C++ code. These lines tell the compiler that we are using C code and, therefore, it can apply optimizations assuming that no C++-specific features—such as operator overloading—will be used.

Next, we include a header that allows us to use boolean variables. You may also notice some parts highlighted in red. One of them is labeled "Function Declaration (interface.h)", the same title used for most of the code declared earlier. This means that all previously defined code blocks with that title will be inserted at this point in the file. The other red-highlighted parts represent code that we will define in the following sections.

If you want to understand how the `interface.c` file relates to this header, its structure is as follows:

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. No other files will be created.

1.2. Supporting Multiple Threads

Most of the code defined in this document is portable. The only requirement we assume is that an OpenGL context has already been created and is active. However, there is one non-portable component that must be defined differently depending on the system and environment: mutex support.

A mutex is a data structure that controls access by multiple processes to a shared resource. Its implementation varies depending on the operating system and environment. Due to its non-portability, we introduce it here separately from the rest of the code.

On Linux and BSD, a mutex is defined using the `pthread` library and follows its naming conventions. On Windows, a mutex is called a critical section. In WebAssembly, we do not define mutexes for now, as multithreading is currently not supported 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 the headers for the `pthread` library. On Windows, we simply 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
```

Each declared mutex must be initialized. For this, we use 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 a 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 we can perform. The first is to request ownership of the mutex. If another process is already using it, we must wait until the mutex becomes 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, when we are done using the resource protected by the mutex, we signal that other processes 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 describe the four main data structures managed by the API: shaders, user interfaces, links to previously created interfaces, and history markers.

2.1. Shader Data Structure

To render any object using the video card, we need to specify a set of rendering rules. At what position should the object be drawn? What is its color? What texture should be applied? Should the rendering change based on time or the viewing angle? To answer these questions, we use special programs that run on the GPU, called shaders.

In addition to the shader program itself, we also need to store associated variables. Shader programs access a list of variables during execution, and we can update their values before each run. For every modifiable variable, we must store its address within the shader program. Only with this information can we change the variables value at runtime.

A shader data structure has the following layout:

Section: Local Data Structures (interface.c):

```
struct shader {
  int type;
  void *next; // Linked list pointer
  GLuint program; // The shader program
  /* These are the shader variables that users could modify: */
  GLint uniform_foreground_color, uniform_background_color; // Bg/Fg color
  GLint uniform_model_view_matrix; // Model-view matrix
  GLint uniform_interface_size; // Object size in pixels
  GLint uniform_mouse_coordinate; // Mouse coordinate (interface referential)
  GLint uniform_time; // Time counter
  GLint uniform_integer; // Arbitrary integer
  GLint uniform_texture1; // Object texture
};
```

Since we are using OpenGL data types such as `GLuint`, we need to include the appropriate OpenGL headers:

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

```
#if defined(__linux__) || defined(BSD) || defined(__EMSCRIPTEN__)
#include <EGL/egl.h>
#include <GLES3/gl3.h>
#endif
#if defined(_WIN32)
#pragma comment(lib, "Opengl32.lib")
#include <windows.h>
#include <GL/gl.h>
#endif
```

The first two variables in the data structure exist because each shader will be stored in a linked list that may also contain other types of data structures. The type field indicates which kind of structure is stored in the list, and the pointer to the next element forms the linked list by linking each element to the following one.

In a shader data structure, the type field is always set to `TYPE_SHADER`. The possible values, covering all types of supported data structures, are:

Section: Local Macros (interface.c):

```
#define TYPE_INTERFACE 1 // A visual interface that could be rendered
#define TYPE_LINK      2 // Link to a previous interface
#define TYPE_MARKING   3 // Marker in the interface history
#define TYPE_SHADER    4 // Shader to render interfaces
#define TYPE_SOUND     5 // A sound effect for interfaces
```

The program variable in the shader data structure stores the compiled shader. All the variables that follow it indicate the locations of configurable variables within the shader program. Not all of these variables will necessarily be present in every shader, but they represent the set of variables supported by our API. They include variables for vertex data, size, position, color, textures, and other related attributes.

2.2. User Interface Data Structure

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

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(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;
  /* Additional data for vector textures or procedural animation */
  void *_internal_data;
  void (*_free_internal_data)(void *);
  void (*_reload_texture)(struct user_interface *);
};
```

Now we will describe the purpose of each part of this structure. The type and next pointer were already described and are the same as those presented in the shader data structure.

After the type and next fields, we store the position in pixels (for axes x and y) and the z-index (axis z) of the interface. Then we store the rotation (in radians) and the width and height (in pixels). These values can be read but should not be changed directly. To modify them, you must use the designated functions defined in later sections. This is necessary because properly updating position, rotation, and size requires modifying the transformation matrix, which is how OpenGL and our shaders internally represent this information. The z-index must also be updated in a separate list that stores the correct rendering order for each element.

Note that we actually have two sets of variables to store position (x, y) and rotation. This is because when the `W_FORCE_LANDSCAPE` macro is defined, and the window height is greater than its width, the x and y axes are rotated, and the interface is rotated 90 degrees as well. In this case, the variables `_x`, `_y`, and `_rotation` store the logical (unrotated) values, while `x`, `y`, and `rotation` store the rotated ones. When the window is wider than it is tall or the macro is not defined, both sets of variables will store the same values.

The `mouse_x` and `mouse_y` attributes store the mouse coordinates relative to the bottom-left corner of the interface, rather than the screen. If the interface is rotated, the coordinate system is rotated accordingly.

The foreground and background colors are in RGBA format, with each component ranging from 0 to 1. These values are passed to the shader, although the shader may or may not use them. Similarly, the integer attribute is passed to the shader, and it is up to the shader to decide whether to make use of it.

The visible attribute determines whether the interface should be rendered. If set to false, it will not appear on the screen.

Next, we have a pointer to the shader data structure associated with this interface, which defines the rendering behavior.

The interaction function pointers are usually initialized to `NULL`. If not, they are called when the mouse enters or leaves the interface, or when a mouse button is pressed or released over it. Additionally, we have a boolean flag that indicates whether the mouse is currently over this interface.

The attributes following the interaction functions are initialized to 0, false, or NULL. However, the loading function (responsible for interpreting the interfaces source file) may change and update these values. This loading function is passed to the APIs initialization function and interprets files with specific extensions to generate textures and other interface-related information.

Next, we have a pointer to the OpenGL texture used by the interface. It is possible to have more than one texture—for example, if the source file contains multiple images or is an animated GIF. In such cases, each animation frame is stored as a separate texture. After successfully loading a texture, the `texture_loaded` flag should be set to true.

If multiple textures are loaded, the animate flag determines whether the interface should animate. This value can be changed to pause or resume the animation.

The following two attributes store the total number of frames in the animation and the index of the current frame. Frame indices start at 0.

The `frame_durations` pointer should point to an array with the duration of each frame. If the number of frames is 0 or 1, this pointer can be NULL.

The `_t` variable keeps track of the elapsed time. It should be set to zero after the interface has been fully loaded and its texture is ready. It is then incremented periodically to determine when to switch to the next animation frame.

The `max_repetition` attribute defines how many times an animated interface should repeat. If the value is zero, the animation loops indefinitely. If it is a positive number, the animation will stop after reaching the specified repetition count and will remain on the last frame.

The `_internal_data`, `_free_internal_data`, and `_reload_texture` fields are optionally initialized by the loading function. `_internal_data` is a pointer to any arbitrary data structure. `_free_internal_data` is a function used to deallocate this structure when the interface is destroyed. `_reload_texture` is a pointer to a function called whenever the interface changes size, or on every animation frame if the interface is animated but contains a single texture. This makes it possible to use vector-based textures that are recalculated on resize, as well as procedural animations in which each frame is generated dynamically rather than preloaded.

2.3. Markers in History

As mentioned in the Introduction, the logic that determines which interfaces are accessible at any given moment relies on markers placed in the interface history. All interfaces created after the most recent marker are accessible, while those created before it are not: they will not be rendered, and the user will not be able to interact with them. Multiple markers can be created, and the most recent existing marker can be deleted, which also removes the interfaces created after it.

To make this possible, each marker must store a pointer to the previous marker and to the previous structure in the linked list. This allows us to replace a deleted marker with the one it references. Each marker also stores the number of interfaces created after it—this corresponds to the number of interfaces that will remain active while the marker exists and is the most recent one.

Section: Local Data Structures (interface.c):

```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;
};
```

Note that this is declared as a local data structure in the `interface.c` file. We did this because the structure is intended for internal use only within our API. Users do not need to manipulate markers directly. Instead, they interact with this data structure indirectly through the functions `_Wmark_history_interface` and `_Wrestore_history_interface`, as described in the Introduction.

2.4. Link to Other Interfaces

If an interface is inaccessible because it was created before the last history marker, it is possible to create a link to it using the `_Wlink_interface` function. Linking to the older interface makes it act as if it were a newly created one, thereby making it accessible again. Creating a link means creating and storing the following struct in the linked list:

Section: Local Data Structures (interface.c):

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

Besides the information required for the linked list, such as the type and the pointer to the next element, the only additional data we need to store in a link is a pointer to the previously linked interface.

3. Initializing and Finalizing the API

The purpose of initialization in this API is to define all the custom functions that will be used for memory allocation, deallocation, and interface creation management. Finalization deallocates the data used to store information about some of these functions. It also resets all custom functions to their default values.

3.1. Initialization

Here, our goal is to set the six custom functions discussed in the Introduction, as well as to receive a variable-length list of functions that will be used to interpret the contents of files with specific extensions.

To store the functions from this variable-length list, we use the following structure:

Section: Local Data Structures (interface.c):

```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, void *target);
};
static unsigned number_of_file_functions_in_the_list = 0;
static struct file_function *list_of_file_functions = NULL;
```

The struct is essentially a pair consisting of a function that extracts textures from a given file (and receives all the custom functions to be used), and a string representing a file extension. After defining this structure, we create a pointer to a list of such structures, which initially represents an empty list.

Later, once this list is no longer empty—that is, after initialization—we will be able to iterate over its elements and retrieve the appropriate loading function using the following 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, void *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 defines a function that returns a function pointer—with many parameters. However, the function itself takes only a single argument: `ext`, a file extension to be searched for in the list.

Since we use a function to compare strings, we must include the standard header for string operations:

Section: Local Headers (interface.c):

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

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

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) /* Part 1: Get 6 functions + window size*/
    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; /* Part 2: How many more functions there are? */
    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); //Part 3:Allocate others
    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 function above is quite verbose because C tends to be verbose when dealing with functions that accept a variable number of arguments, especially when those arguments are pointers to functions with many parameters. Despite the verbosity, the initialization logic is straightforward.

At the end, in red, we marked a placeholder where we will insert additional initialization code, which will be defined in the following sections.

The use of certain features—such as accessing arguments via `va_list`—requires the inclusion of 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, during initialization, we allocate memory to store a list of functions that interpret files, then during finalization we begin by deallocating that memory. Similarly, if initialization involves setting the six basic custom functions, finalization resets those functions to their 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;
}
```

Note that we have reserved additional space, marked in red, for code that may be needed later in the finalization process as the API grows in complexity with the addition of new functions in the following sections.

This finalization function undoes everything performed during initialization. It is safe to initialize and finalize the API repeatedly without issues.

4. Shaders

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

First, all shader source files must specify the shading language version on the first line. We use the GLSL language, which has different versions. We will not require the user to include this information in every shader source, as we will insert it automatically.

To choose the correct GLSL version to use in all shaders, we check the value of the macro `W_GLSL_VERSION`. This macro will be written as the first line of each shader source code. The macro can be defined by the user via appropriate compilation flags or by including the macro definition in the code. If it is not defined, we will use the default string “`#version 100\n`”. This standard value corresponds to the OpenGL ES shading language version 1.00. We will define the macro at the beginning of the file that contains our functions if it does not already exist:

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 only need to check which macro is defined, allowing us to include code conditionally based on that.

Next, we need to specify the default precision for each variable type if the user does not specify one in the variable declaration. This is done using the keyword precision followed by a precision qualifier (`lowp`, `mediump`, `highp`) and a variable type. Here, we will set the highest possible precision as the default. For less critical variables, the user can always lower the precision 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 we must define is how to support libraries in our GLSL code. These libraries can simplify shader definitions. The user can define additional GLSL functions that will be available in all interface shaders by storing their source code in the following variable:

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

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

The variable will be modified by the function introduced in the Introduction to extend GLSL with new functions. This function is extremely simple: it just assigns a string containing the functions source code to the variable mentioned above:

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

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

However, we should not forget that when finalizing our API, we need to undo this assignment. Otherwise, if a user finalizes and then initializes the API again, the second initialization would have the same function definitions assigned as in the first initialization:

Section: Finalizing Interface API:

```
shader_library = "";
```

The list of default variables and attributes passed to the shader must also be included in the source code. We will define them later, but for now, we show that they will be stored 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 this information and a string containing its source code. First, we include the standard input/output library to print error messages on the screen 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 into a complete shader program. Compiling a shader involves creating both types of shaders (vertex and fragment) in OpenGL, compiling them, and linking them into 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 did not show above is how we verify whether the shader was compiled and linked successfully. To check for shader compilation errors, we use the function below. It checks if a compilation error occurred, and if so, it reads the logs to determine what went wrong and prints the message to the screen. Note that here we use the temporary allocation and deallocation functions to manage memory space for the error message. The function also returns whether an error was found.

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 whether an error occurred during the linking stage is similar. However, after linking, we can go even further: we can try to validate the shader by simulating its usage to detect additional errors that could not be caught during syntactic analysis. However, since this is an expensive operation, we will perform it only if the macro `W_DEBUG_INTERFACE` is defined. In that case, we assume 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;
}
```

This concludes the explanation of how we compile a new shader and print messages when errors are found in the source code.

Now we can define the default shader source code that will be used if the user does not provide a custom shader. The goal of this shader is to display 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 consist of string literals stored in the constant variable defined above. In the case of the vertex shader, it will multiply each vertex by the model-view matrix, which contains information about the interfaces size, position, and rotation. It will also assign a texture coordinate to each vertex.

To understand the code, keep in mind that all interfaces will share the same vertices: $(0,0,0), (1,0,0), (1,1,0), (0,1,0)$. Each vertex is associated with a texture coordinate, as 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. When followed in the declared order, they form a square in counter-clockwise order. This tells OpenGL that we are viewing the front of the interface, not its back. We do this to account for the possibility that the user configures OpenGL to not render the back faces of geometric figuresan optimization that is very common and often necessary. By specifying the correct order, we ensure that the interface remains visible even when this option is enabled.

The rotated texture coordinates are used if the macro `W_FORCE_LANDSCAPE` is defined and the window height is greater than its width. In this case, we swap the interfaces width and height and also rotate the texture. This effectively rotates the interface.

These vertices will be loaded into the video card, and afterward we can render them by drawing the corresponding VBO (Vertex Buffer Object). Loading the vertices into the VBO is done during initialization:

Section: Initializing Interface API (continuation):

```
glGenBuffers(1, &interface_vbo);
glBindBuffer(GL_ARRAY_BUFFER, interface_vbo);
// Sending vertices to the video card:
glBufferData(GL_ARRAY_BUFFER, sizeof(interface_vertices), interface_vertices,
             GL_STATIC_DRAW);
```

During finalization, we need to destroy the VBO:

Section: Finalizing Interface API (continuation):

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

By default, all interfaces start as a rectangle centered on the screen, occupying the entire screen area, since OpenGL defines 1 as the width and height of the rendering space. To transform this fixed square into different sizes, positions, and rotations, we use 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 at 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 the attributes, uniforms, and varyings used in the shaders. Attributes are read-only data specified for each vertex. The attributes we will define include the vertex position in the format (x, y, z) and the texture coordinate in the format (x, y). The x and y coordinates will range from 0 to 1, with $z = 0$, as previously described. 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 are also passed to vertices but do not change between them. Therefore, the fragment shader does not need to interpolate their values. We will store the following 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 set and modified in the vertex shader, and their interpolated values will be passed to the fragment shader. Here, we use them for the texture coordinates.

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

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

Note that we did not use all the variables defined in the default shader. Unused variables will be discarded during compilation as an optimization. However, we declared them anyway to list the supported variables that can be used by custom shaders defined by the user.

By the way, speaking of custom shaders, they will be provided by the user through the function `_Wnew_interface` and specified as a path to a file containing the source code. For this case, we will define a function that creates a new shader program from a filename given as an 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;
  // Go to end of file to get its size and return to beginning:
  fseek(fp, 0, SEEK_END);
  source_size = ftell(fp);
  // Allocates and reads 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 we will address in this section is what should be rendered as the default texture when the user supplies no texture for a given interface. In these cases, we will create a texture composed of a single white pixel:

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

```
static GLuint default_texture;
```

We will create this texture during initialization. The following code generates the texture in OpenGL, binds it as the current 2D texture, and specifies 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);
}
```

During 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 are represented by four vertices forming a square with fixed edge length of 1. What determines each interface's size, position, and rotation is the model-view matrix applied to it.

To understand this matrix, recall that in our default vertex shader (shown earlier), each vertex is converted to a four-dimensional point instead of a two- or three-dimensional one. The line `vec4(vertex_position, 1.0)` takes the 3D coordinates and adds a fourth component with a value of 1.

This is necessary because only in four dimensions can translation, rotation, and scaling all be represented as matrix multiplications. In other words, only in 4D are these operations linear. Since GPUs are highly optimized for matrix multiplications, representing these transformations any other way would harm performance.

Before presenting the final form of the model-view matrix, we will first look at its individual components. Suppose we have a four-dimensional vector $(x0, y0, z0, 1)$ and want to apply only a translation to it. In this case, we can multiply it by the translation matrix shown below:

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

To change the size of an interface—both its width and heightassuming the interface is centered at the origin $(0, 0, 0, 1)$, we can multiply each vertex vector by 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 interface by θ radians, we can use the following matrix, which produces the correct result based on the trigonometric formulas 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 multiply all the matrices corresponding to the desired scale, rotation, and translation. However, the order in which these matrices are multiplied affects the final result. The correct sequence of operations is:

1. First, we center the interface square at the OpenGL origin. (Matrix A).
2. Then, we scale the interface based on its pixel dimensions. (Matrix B)
3. We rotate the interface by θ radians. (Matrix C)
4. We scale the interface again, this time adjusting it to the screen proportions using OpenGL coordinates. (Matrix D)

5. Finally, we translate the interface to its correct position using OpenGL coordinates. (Matrix E)
To perform these operations in the correct order, each vector v should be multiplied as follows:

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

If we use a different order, the transformations may not behave as expected. For example, instead of rotating the interface around its center, we might end up rotating it around its lower-left corner or some other unintended point.

To avoid always needing to multiply all four matrices at runtime, we will compute the final matrix format resulting from their multiplication. This final matrix will be our model-view matrix. For instance, assuming that wwww is the window width and hwhw is the window height, by multiplying matrices E and D, we obtain:

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

Note that we are using $2/w_w$ and $2/h_w$ because we assume the OpenGL coordinate system has a total width and height of 2. Therefore, an object with a width equal to w_w must be scaled to width 2 in OpenGL units to occupy the entire screen. The same reasoning applies to the height.

If we now multiply this result by matrix C, we obtain:

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

Now, if we multiply the result by matrix B, considering that h_p is the objects height in pixels and w_p is its 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 centers the interface at $(0, 0, 0, 1)$. Recall that the interface is originally centered at position $(1/2, 1/2, 0, 1)$. Therefore, matrix A is simply a translation matrix with constant values that shift each vertex 1/2 unit to the left and 1/2 unit 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 shown above is the final form of the model-view matrix for each interface.

However, the (x, y) values above are in OpenGL coordinates. In our API, coordinates are measured differently: the origin is at the lower-left corner of the screen, not at the center as in OpenGL. Additionally, positions are measured in pixels rather than using the fixed size of 2 for the screen as OpenGL does. Therefore, a function that fills the model-view matrix must first convert our coordinates and sizes to OpenGL format.

Assuming we have a pointer `i` to a given interface, this conversion 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;
```

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

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

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

6. Managing Data Structures

As explained and defined before, the data structures managed by our API include: a struct for interfaces, another for linking interfaces, a third for creating history markers, and finally, a linked list to store these structures.

Our linked list will be accessed using two different pointers. The first points to the most recently created and inserted structure in the list. The second points to the most recent history marker inserted in the list. We will also define a mutex to ensure that only one thread can update the linked list at any given time:

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 will control access to the linked list needs to be created during initialization:

Section: Initializing Interface API (continuation):

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

We will include the code to finalize this mutex later, when we define the code that finalizes 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. However, we did not integrate those functions and data structures with the shader data structure. We will do that here in this section.

Our API does not have an exported function to create new shaders. Instead, it creates new shaders when the user creates a new interface and associates it with a custom shader by passing a filename containing the shader source code. That is, when the user calls the API function `_Wnew_interface` with the second parameter specifying a filename with shader source code different from `NULL`.

We also need to create a structure for our default shader, whose source code is in Section 4. The address of the default shader structure 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 receives as an argument a string with the filename that contains the shader source code. The function returns a pointer to the new shader structure and also inserts this data structure into the linked listexcept when `NULL` is passed as the filename. In that case, a new shader structure is created using the default shader source code defined in Section 4, and it is not inserted into 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. After compiling it, the code obtains the addresses of the compiled shader variables that can be modified before executing the shader. These addresses are stored in the shader data structure.

Now, we can initialize the default shader during Initialization:

Section: Initializing Interface API (continuation):

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

To destroy a shader, we need to notify that the compiled shader program will no longer be executed and should be deleted. Then, we deallocate the memory allocated 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 described in the Introduction, all new interfaces will be created using the function `_Wnew_interface`. This function will allocate the new interface, execute the appropriate functions to initialize it, and include it in the linked list. The function is defined as follows:

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, void *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 = 1;
    new_interface -> current_frame = 0;
    new_interface -> frame_duration = NULL;
    new_interface -> _t = 0;
    new_interface -> max_repetition = 0;
    MUTEX_INIT(&(new_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;
    new_interface -> _internal_data = NULL;
    new_interface -> _free_internal_data = NULL;
    new_interface -> _reload_texture = 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 simply allocate the interface using the function set as our permanent allocator, initialize the interfaces variables, insert it into the linked list, and run the appropriate function to load the texture, depending on the filename extension.

Destroying an interface means checking whether some of its variables are different from `NULL`. If so, it means these variables were allocated by the loading function and should also be deallocated. Examples of such variables include the texture and the list of frame durations for animated textures. The destructor function must first deallocate these variables, and only then deallocate 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 -> mutex));
  if(interface_struct -> _free_internal_data != NULL &&
     interface_struct -> _internal_data != NULL){
    interface_struct -> _free_internal_data(interface_struct -> _internal_data);
  }
  if(permanent_free != NULL)
    permanent_free(interface_struct);
}
```

6.3. Creating and Destroying Link to an Existing Interface

As explained, instead of creating a new interface, we can create a link to an existing one. This causes the linked interface to behave as if it were newly created, at least with respect to functions that interact with it. A link is simply a pointer to another interface in the linked list and is created using the function _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 simply by calling the deallocation function. For this reason, we will not define an auxiliary function to perform this action.

6.4. Creating Markers in the Interface History

To create a new marker in the interface history, the user should call the function _Wmark_history_interface. Once this marker is set, all interfaces created before it become inaccessible until the marker 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);
  }
}
```

Since it is important to have at least one marker in the history to keep track of the number of active interfaces, the first marker is created during initialization and will only be removed during finalization. If the user attempts to remove this initial marker using the API function before finalization, the action will be ignored.

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 will be defined in the next section. It destroys and removes all structures created after the most recent marker, and then it removes and destroys that marker. However, it will never destroy the initial marker. For this reason, in previous code, we manually destroyed the first marker using `permanent_free`.

6.5. Removing Markers and Interfaces

We will remove and deallocate interfaces using the function that restores the history to a previous point. This function will remove all interfaces created after the most recent marker and then remove that marker itself. This is handled by the function `_Wrestore_history_interface`.

After executing this function, the most recent marker will be removed, and we will revert to the marker that existed before it. If there is no previous marker, the current marker will not be removed, but the system will still be restored to the state it had when the initial marker was created.

The function is implemented as follows:

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 marker, 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 most recent marker. Only after this is the last marker itself removed—except when it is also the initial marker. The pointer to the last structure in the linked list is updated at the beginning of the function by retrieving the pointer to the previous element, which is stored in the most recent marker that is being removed.

7. Rendering Interfaces

To render interfaces, we must consider the correct order in which each interface is drawn. One simple method is to draw the interfaces in any order and use the z coordinate and a z-buffer to determine whether each interface should be drawn using a depth test. However, when dealing with transparent objects, this approach does not always produce the correct result. For example, if we first draw a transparent object and then draw an opaque object behind it, parts of the opaque object that should be visible through the transparency may not be visible.

Since our interfaces are always two-dimensional objects parallel to the z axis, we instead maintain an ordered list of interfaces that defines the correct drawing order. This list is ordered by the z coordinate, ensuring that all interfaces are drawn before any interface in front of them. This approach is essentially the "painters algorithm": just as a painter begins painting distant objects first, we begin by drawing the most distant interfaces.

If two interfaces have the same z coordinate value, they can be drawn in any order.

The pointer storing the address of this ordered list is:

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;
```

During the API initialization, we should initialize the variables mentioned above:

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;
```

```c
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 to the number of active interfaces at the current moment (the number of active interfaces can be checked in the last history marker). If the values differ, this means new interfaces were created, and we should regenerate our ordered list. The code that performs this can be seen below:

Section: Generating Ordered List of interfaces:

```c
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;
  }
  // Ordering list (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 rebuilt using the code above is uncommon. We run that code when rendering for the first time in a main loop or after restoring our interface history. The only expected scenario in which the code runs more frequently is when the user is creating new interfaces in their programs main loop. In such cases, every rendering after a new interface creation will execute the code above. However, this is considered bad practice, so we assume that the code above will rarely be executed, despite being placed inside the rendering function. Typically, we already have an ordered list with all active interfaces, and we only need to adjust the interfaces position in the list if the interface is moved.

Once we have the ordered list of interfaces, we can simply 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.

The interface is rendered when the function _Wrender_interface is invoked. This function receives the current time in microseconds as a parameter. But to know 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, which 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 during the initialization. Otherwise, the variable would hold incorrect values if the API is finalized and initialized again:

Section: Initializing Interface API (continuation):

```
previous_time = 0;
```

Below we show the function that iterates over the ordered list of interfaces and renders each one, also updating the time variables. The function first updates the time, then loads the interface vertices, indicating how they are represented, and finally iterates over the interfaces.

For each interface, it loads the shader program and passes to it each uniform and varying variable necessary, which are stored in the interface struct. After this, it checks the interface textures. If there is more than one texture, this means we could have animated textures (for example, an animated GIF).

If we indeed have an animated texture, we check the elapsed time between renderings and the time spent in the current animation frame. If it is time to change to the next animation frame, we update it. Finally, we pass the correct texture to the shader.

Only after iterating over all interfaces do we store the current time in our static global variable that remembers when the last rendering occurred.

The implementation of 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;
    // Default framebuffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    // Using VBO vertices:
    glBindBuffer(GL_ARRAY_BUFFER, interface_vbo);
    // Specifying how vertices are stored in 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;
      // Animating texture
      if(z_list[i] -> animate &&
         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] -> max_repetition != 0){
          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;
          if(z_list[i] -> number_of_frames == 1 &&
             z_list[i] -> _reload_texture != NULL){
Ψ    z_list[i] -> _reload_texture(z_list[i]); // Procedural texture
Ψ    // Rendering procedural texture can contaminate the OpenGL state.
Ψ    // So we need to set it again in this case:
Ψ    glBindFramebuffer(GL_FRAMEBUFFER, 0);
Ψ    glBindBuffer(GL_ARRAY_BUFFER, interface_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);
Ψ    // It can be relatively time-consuming to reload a texture, and
Ψ    // it also often requires memory allocations. Because of this, we
Ψ    // will limit the number of invocations of this function to 1 per
Ψ    // frame:
Ψ    if(z_list[i] -> _t > z_list[i] -> frame_duration[0])
Ψ      z_list[i] -> _t = z_list[i] -> frame_duration[0];
Ψ  }
          else if(z_list[i] -> current_frame == 0 && z_list[i] -> max_repetition > 0)
Ψ    z_list[i] -> max_repetition --;
Ψ  if(z_list[i] -> frame_duration[z_list[i] -> current_frame] == 0)
Ψ    break;
        }
      }
      // Choosing correct shader:
      glUseProgram(z_list[i] -> shader_program -> program);
      // Passing uniforms:
      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);
      // 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 (x, y, z) variables. In addition to this, we also need to update the model-view matrix to reflect the new x and y values, after converting them to OpenGL coordinates. And if we are changing the z coordinate, then we may need to update the position of this interface in the ordered list of interfaces that determines the drawing order.

All these changes must be performed after acquiring the interface mutex, to prevent two simultaneous invocations of this function on the same interface.

The API function that moves interfaces is `_Wmove_interface`, and we define it 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 -> 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 -> mutex));
}
```

8.2. Rotating Interfaces

Rotating an interface involves acquiring its mutex, updating the rotation variable, and also updating its model-view matrix. The corresponding code is shown 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 -> 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 -> mutex));
}
```

8.3. Resizing Interfaces

Resizing interfaces, like rotating them, involves simply acquiring 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 -> mutex));
  i -> width = new_width;
  i -> height = new_height;
  if(i -> _reload_texture != NULL)
    i -> _reload_texture(i); // Recreate texture if we have a vector texture
            <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 -> mutex));
}
```

8. Interacting with Interfaces

Given the active interfaces, we can interact with them by moving the mouse cursor over them or clicking on them with a mouse button. To manage these interactions, we use the function `_Winteract_interface`, which provides information about the mouse's behavior and automatically executes the functions associated with interface interactions.

To make this possible, we need to keep track of the mouse button states. This means storing not only the current state of each mouse button but also its previous state. To do 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 and current mouse states, we can detect not only whether the user is clicking something, but also whether the click has just started in the current frame or is merely continuing. Similarly, we can tell not just whether the user is not clicking, but also whether the button was released at some point between the previous and current frames. This information is essential for correctly executing the functions that handle interactions with each interface.

In a given frame, we can interact with only one interface at a time, as we have only a single mouse cursor. But if more than one interface occupies the same position, how do we determine which one should respond to interaction? Fortunately, we have an ordered list of active interfaces, sorted by their rendering order. When multiple interfaces overlap at the cursor position, the correct one is the front-most interface in this list.

Therefore, we iterate over the interfaces in reverse drawing order and check which is the first one under the mouse cursor. This becomes the currently active interface. If another interface was previously marked as active, we remove its mark and treat it as the previously interacted interface. If needed, and if such interfaces exist, we call the appropriate functions: one to handle the cursor leaving the previous interface, and another to handle the cursor hovering over or clicking on the new current interface.

The code that performs 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 code is straightforward, but we omitted the part where we transform the mouse coordinates into (x, y) coordinates to compare them against each interface. If the interface is not rotated, no transformation is needed—we can use the mouse coordinates directly. However, if the interface is rotated, the easiest way to determine whether the mouse is over it is to ignore the interfaces rotation and instead rotate the mouse position by the opposite angle, using the center of the interface as the axis. After this transformation, we can check whether the mouse is over the interface using the usual method.

Our code to generate (x, y) first checks whether the interface is rotated. If not, no transformation is performed. Otherwise, we apply trigonometric calculations to compute 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.