Windowing Interface Weaver

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

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

Index

1. Introduction

A graphical computer program needs a space where it can draw in the screen. In some environments, lke video-games, each program always controll the entire screen automatically, without asking for it to some operating system. But when a computer program runs in a computer with some modern graphical environment, usually it is necessary to ask for the

creation of some region called “window”. There the program have control over the content and can draw freely.

Besides creating the window, it is also important to have control if we are in fullscreen mode or not if we are in an environment that allows this. And if possible, toggle between fullscreen and windowed mode. If we are in windowed mode, we also should be able to change window size.

1.1. Literary Programming and Notation

This article uses the technique of “Literary Programming” to develop our random number generator API. This technique was presented at [Knuth, 1984] and have as objective develop software in a way that a computer program to be compiled is exactly equal a document written for human beings detailing and explaining the code. This document is not independent of the source code, it is the project source code. Automated tools are used to extract the code from this document, sort it in the right order and produce the code that is passed to a compiler.

For example, in this article we will define two different files: `window.c` and `window.h`. Both of them can be inserted statically in any project or compiled as a shared library. The content of `window.h` is:

(P)

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

```
#ifndef WEAVER_WINDOW
#define WEAVER_WINDOW
#ifdef __cplusplus
extern "C" {
#endif
#include <stdbool.h> // Define type 'bool'
#if !defined(_WIN32)
#include <sys/param.h> // Needed on BSD, but do not exist on Windows
#endif
                    <Section to be inserted: OpenGL Header>
                    <Section to be inserted: Macro Definition>
                <Section to be inserted: Window Data Structures>
                    <Section to be inserted: Window Declarations>
#ifdef __cplusplus
}
#endif
#endif
```

The two first lines and the last one are macros that ensure that the function declaration and variables from this file will always be included at most once in a compiling unit. We also put macros to check if we are compiling this as C or C++. If we are in C++, we assure the compiler that all our functions will be in C-style. We never will modify them with operator overload, for example. This makes the code became more compact.

The red parts in the above code shows that some code will be inserted there in the future. In “RNG Declarations”, for example, we will put there function declarations.

Each piece of code have a title, that in the above example is `src/window.h`. The title shows where the code will be placed. In the case above, the code will directly to a file. In future pieces of code, we will have different titles, including titles matching exactly the names present in the red code above. If a piece of code have as title a name matching the red part in some code, then that code is positioned exactly in that red part when compiling the code.

As a second example of code, we also will declare here that when we are in debug mode (when the

macro W_DEBUG_WINDOW is declared) we will need the declaration of input/output functions, as our code will be more verbose:

Section: Headers:

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

Notice that we still did not define where this code section called "Headers" will be positioned. We can do this later.

1.2. API Functions to Be Defined

In this article we will define the following functions:

Section: Window Declarations:

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

This function will create a new window. By default, it will create a fullscreen window. If the macro W_WINDOW_NO_FULLSCREEN is defined, instead it will create a non-fullscreen window.

If the macro W_DEBUG_WINDOW is defined, this function also will print in the screen information about the graphical environment. Its resolution for example. But possibly other information that can be relevant.

The function will initialize the mouse and keyboard structs.

In case of error, the function will return false.

Section: Window Declarations:

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

This function closes the window, freeing any resource allocated during the window creation by the previous function. This function must be invoked only after a window was created. In case of error, it returns false.

Section: Window Declarations (continuation):

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

This function will render in the screen all pending ommands since last rendering. It will return false in case of error. This function probably will be called in the end of each iteration in a main loop.

Section: Window Declarations:

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

This function gets the screen resolution and store in the pointers passed as argument. If we have more than one screen, it will return data about the main window. In case of error, it returns false and stores zero as the resolution.

Section: Window Declarations:

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

This function will get the pixel density of the screen, measured in DPI (dots per inch). If an error occurs, it will return false and the values passed as arguments will be initialized to zero.

Section: Window Declarations:

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

This function gets the current window size in pixels and store the information in the pointers passed as argument. If we have no window or in case of error, it returns false and store 0 in the pointers. Otherwise, it returns true and store the correct value in the pointers.

Section: Window Declarations:

```
void _Wget_window_input(unsigned long long current_time,
                        struct _Wkeyboard *keyboard,
                        struct _Wmouse *mouse);
```

This is the function that shall be called periodically to update the keyboard and mouse state. The first argument is the current time measured in some unit of time. The next arguments are structs that represent keyboard and mouse and that shall be updated.

Section: Window Declarations (continuation):

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

This function cleans the state for keyboard and mouse structs. It should be called each time we interrupt our periodic invocation of `_Wget_window_input`. Calling these functions the structs are reset to the initial state. And we stop to consider as pressed or released any key or button.

Section: Window Declarations (continuation):

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

This checks if we are in fullscreen mode or not. The result is undefined if a window was not created.

Section: Window Declarations (continuation):

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

This function alternates between fullscreen mode and windowed mode. Depending on the environment, it could fail.

Section: Window Declarations (continuation):

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

This function changes the window size if it exists and if we are not i fullscreen mode. If this operation fails because these conditions are not met, the function returns false.

Section: Window Declarations (continuation):

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

This function register a function to be executed always that our window changes its size. The function gets as arguments the old width and height, followed by the new width and height.

2. Creating and Managing the Window

In this Section we will define the functions and core about creating a window and an OpenGL context compatible with Open GL ES 2.0.

2.1. Getting Screen Resolution and Dot Density

Before creating our window, it is important to check what is our window and screen resolution. If we are running in fullscreen mode, this will be our window size. This is accomplished differently accordig with the operating system.

We will also create a static variable that will act as a cache that memorizes the window height in pixels. Storing this value is useful because futurely we will need to transform the mouse position coordinates. Practically all environments inform coordinates in a window using as origin the upper left corner. However, in Weaver API, we prefer to use the more standard convention from mathematics, where the origin usually is the lower left corner. To transform coordinates according with our convention, it is important to memorize the window height, so that we do not need to measure it all the time:

Section: Local Variables (continuation):

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

In addition to resolution, another measurement that may be relevant is the number of dots per inch, which measures the pixel density per area of our screen. The DPI value can be consulted by some applications, but it is not something we need to worry about having stored in the cache, since it is a value that does not change and will rarely need to be consulted.

2.1.1. Getting Screen Resolution and Dot Density on X

The X Window System, also known as X11, is a windowing system present in many Operating System, as Linux, BSD and even in MacOX, where it exists to ensure compatibility with older programs developed

before their current windowing system. We will begin our function implementation with X11 because is the most widely present windowing system.

X11 works in a client-server architecture. When we create a graphical program, we create a client that communicates with X server using sockets. All operations like window creation, resizing windows and more are made when with the client asking for them sending requests to the server, which executes the requests if possible.

We expect to use X11 when we are not compiling programs to Windows or to Web Assembly, as both the Windows Operating System and web browsers do not have a X11 server. If we will use X11, we need the following header:

Section: Headers (continuation):

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

In X11, as we need to communicate with a server, we assume that we have a variable with some data structure with information about this connection. This variable is called "`display`".

Using such connection, that we assume being a static variable in `window.c`, we read the resolution using the function below:

Section: API Functions:

```
#if defined(__linux__) || defined(BSD)
bool _Wget_screen_resolution(int *resolution_x, int *resolution_y){
  bool keep_alive = true; // Should we keep the connection alive?
  // The first called unction should be this that prepares X11 for any
  // multi-threaded invocation
  XInitThreads();
  // If we still do not have an active connection, we create it:
  if(display == NULL){
    display = XOpenDisplay(NULL);
    keep_alive = false;
  }
  // Now we get the default screen number:
  int screen = DefaultScreen(display);
  // And ask for its resolution:
  *resolution_x = DisplayWidth(display, screen);
  *resolution_y = DisplayHeight(display, screen);
  // If we had not an active connection before invoking this function, we
  // should keep things this way:
  if(!keep_alive){
    XCloseDisplay(display);
    display = NULL;
  }
  // If everything is ok, we should return true:
  return true;
}
#endif
```

Of course, we assume the existence of the following static variable:

Section: Local Variables (continuation):

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

```
static Display *display = NULL; //Connection with server and info about screens
#endif
```

Regarding the screen's dot density, measured in DPI, we must first check the screen resolution, and then use the functions `DisplayWidthMM` and `DisplayHeightMM` to obtain the screen size in millimeters. Calculating the number of dots per inch is done by simply converting millimeters to inches and dividing the number of pixels by that value, both horizontally and vertically.

Section: API Functions (continuation):

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

2.1.2. Getting Screen Resolution and Dot Density on Web Assembly

If we are running Web Assembly, then we are in a web browser. Our "window" will be a HTML canvas. And to obtain the screen size, we need to run javascript code. We can do this with the help of Emscripten API:

Section: API Functions:

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

In WebAssembly, getting the pixel density of the screen is somewhat imprecise. Web browsers typically store a floating-point value as an attribute of the object representing the window called `devicePixelRatio`. It measures the ratio of the size of a CSS pixel to a physical pixel. A value of 1 typically represents a device with 96 DPI. The value can be interpreted as a multiplier over the base value of 96 DPI. Emscripten provides the function `emscripten_get_device_pixel_ratio` as a way to get this value more easily:

Section: API Functions (continuation):

```
#if defined(__EMSCRIPTEN__)
bool _Wget_screen_dpi(int *dpi_x, int *dpi_y){
```

```
  *dpi_x = emscripten_get_device_pixel_ratio() * 96.0;
  *dpi_y = *resolution_x;
  return true;
}
#endif
```

2.1.3. Getting Screen Resolution and Dot Density on Windows

Finally, on Windows, we do this calling an API function that gives us these information. In the case of the resolution:

Section: Funes da API (continuation):

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

And for the dot density:

Section: API Functions (continuation):

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

In the above function, we assumed the existence of a variable called `window` which stores the current windows identifier. This identifier will be defined and initialized later.

2.2. Creating a Window

Now we describe how to create a window in all the supported environments: X11 graphical environment, windows and a web browser running Web Assembly.

2.2.1. Creating a Window on X

The steps to create a window on X11 are:

0. We ask the X libraryq to be prepared for simultaneous communications caming from multiple threads. It is not common needing to do this, but it is a good practice to be prepared if this happens. This must be the first thing to do, before using other library functions.

1. We open a connection with the server. If we succeed, we get as response a list of relevant information about the screen.

2. We read the resolution using the function defined on section 2.1.

3. We send a new message to the server asking to create a new window. We will do it in the default screen, The window size will be the greatest value allowed given the screen resolution.

4. We memorize the initial value for our window height and width.

The above steps are implemented using the functions and macros below:

Section: X11: Create Window:

```
#if defined(__linux__) || defined(BSD)
int screen_resolution_x, screen_resolution_y; //Screen resolution
/* Step 0: */
XInitThreads();
/* Step 1: */
display = XOpenDisplay(NULL);
if(display == NULL){
#if defined(W_DEBUG_WINDOW)
  fprintf(stderr, "ERROR: Failed to connect to X11 server.\n");
#endif
  return false; // Couldn't connect
}
/* Step 2: */
_Wget_screen_resolution(&screen_resolution_x, &screen_resolution_y);
/* Step 3: */
window = XCreateSimpleWindow(display, // X11 connection
                             DefaultRootWindow(display),// Parent window
                             0, 0, // Window position
                             screen_resolution_x, // Width
                             screen_resolution_y, // Height
                             0, 0, // Border width and color
                             0); // Default window color
/* Step 4: */
window_size_x =  screen_resolution_x;
window_size_y = screen_resolution_y;
#endif
```

This code assumes that we have the following declared variable to store and remember the created window:

Section: Local Variables (continuation):

```
#if defined(__linux__) || defined(BSD)
static Window window;    // Created window struct
#endif
```

The code defined above creates a window, but it does not mean that the created window is drawn in the screen. Before drawing the window in the screen we can adjust some of its properties.

The first thing that we need to adjust is that by default the window should be created in fullscreen mode. We do this asking to the window manager do not interefere with the window creating decorations or limiting its size. But we do this only if the user did not disable fullscreen setting a macro:

Section: X11: Create Window (continuation):

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

But what if we are running outside the fullscreen mode and the user has defined macros that configure the window size to a different size than occupy the whole screen? In this case, we need to resize the window before drawing it for the first time. The macros that control the window size when we are not in fullscreen mode are `W_WINDOW_RESOLUTION_X` and `W_WINDOW_RESOLUTION_Y`. If they are non-positive, then this means that we should set their values as the greatest possible given the screen resolution. Otherwise, their values represent the size in pixels for the window. But this is true only outside fullscreen mode:

Section: X11: Create Window (continuation):

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

We also will warn the window manager to not allow window resizing. The window should be resized only using resizing functions that we will soon define:

Section: X11: Create Window (continuation):

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

The above resource needs the following header:

Section: Headers (continuation):

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

```
#endif
```

Another relevant information to adjust is what kind of events is relevant to pass to the program when something happen to the window. For example, we do not consider relevant if the user interacts with another window making our window lose focus. But other events, like the information that the user pressed a key is relevant and our program should be informed.

The list of events that we consider relevant is: the window is created or destroyed, the user presses or releases a mouse button, presses or releases a key in keyboard and if the user moves the mouse pointer. If we do not ask the server to send this information, our program would not be able to know if the user interacts with it pressing keys.

Section: X11: Create Window (continuation):

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

Another important thing is choose the name for our window. Usually this name is presented by the window manager. We let the user choose the name setting the macro `W_WINDOW_NAME`:

Section: X11: Create Window (continuation):

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

If this macro is not defined, we use an empty string:

Section: Macro Definition:

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

Now we configure OpenGL ES. As this is sufficiently laborious, we write the necessary steps in the next session:

Section: X11: Create Window (continuation):

```
#if defined(__linux__) || defined(BSD)
                <Section to be inserted: X11: Configure OpenGL ES>
#endif
```

After adjusting all the configurations, we can draw the created window. For this, we send a request to the server X and wait in a loop until we are notified that the window was created (we asked to be notified of this event when we passed flag `StructureNotifyMask` to function `XSelectInput` previously). The code for this is:

Section: X11: Create Window (continuation):

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

2.2.2. Creating a Window on Web Assembly

One of the most different environments where our API can be executed are web browsers after the code is compiled to Web Assembly. In this case, there are no true window, the space where we can draw in the screen is a HTML canvas. We do not need to worry about the possibility of the user resize the window, for example. But we still need to pay attention to the size of the screen and should allow the resizing of our canvas.

Here we can create and manipulate the drawing area combining two things: the SDL library, presented by Emscripten as the graphical API and Javascript code to control the canvas where the SDL code is rendered.

Now we include the Emscripten and SDL headers, with additional headers needed by some macros that we will use:

Section: OpenGL Header:

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

First we get the screen resolution with the function defined in Subsubsection 2.1.2:

Section: Web Assembly: Create Window:

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

The next step is initialize the video subsystem in SDL library. This is done with the following initialization:

Section: Web Assembly: Create Window (continuation):

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

Now we will create the window, which in practice means adjust the canvas size where we will draw and initialize it with SDL code. The HTML canvas should have the window inner size, except if macro `W_WINDOW_NO_FULLSCREEN` is defined and both `W_WINDOW_RESOLUTION_X` and `W_WINDOW_RESOLUTION_Y` have positive values. We also ensure that the canvas is visible using Javascript, as it could be hidden (we do it when we execute the function to close the window).

Section: Web Assembly: Create Window (continuation):

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

```
  });
#if defined(W_WINDOW_NO_FULLSCREEN)
  fullscreen_mode = false;
#if defined(W_WINDOW_SIZE_X) && W_Window_Size_X > 0
  window_size_x = W_Window_Size_X;
#endif
#if defined(W_WINDOW_SIZE_Y) && W_Window_Size_Y > 0
  window_size_y = W_Window_Size_Y;
#endif
#endif
  // Adjust OpenGL version
  SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION,
                      W_WINDOW_OPENGL_MAJOR_VERSION);
  SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION,
                      W_WINDOW_OPENGL_MINOR_VERSION);
  // Ebsure that the canvas will be visible
  EM_ASM(
    var el = document.getElementById("canvas");
    el.style.display = "initial";
  );
  window = SDL_SetVideoMode(window_size_x / pixel_ratio,
                            window_size_y / pixel_ratio, 0, SDL_OPENGL);
  if(fullscreen_mode){
    EM_ASM(
      var el = document.getElementById("canvas");
      el.style.position = "absolute";
      el.style.top = "0px";
      el.style.left = "0px";
      el.style.width = window.innerWidth + "px";
      el.style.height = window.innerHeight + "px";
      el.width = (window.innerWidth * window.devicePixelRatio);
      el.height = (window.innerHeight * window.devicePixelRatio);
    );
  }
  if(window == NULL)
    return false;
  glViewport(0, 0, window_size_x, window_size_y);
}
#endif
```

In the aove code, after creating our "window" (which is a canvas), we also set the viewport size. Usually it is not necessary to set its value when we will render in the entire screen. However, when running in a mobile device where the logical pixel size is different than the real device pixel, the default viewport could have wrong values. Because of this, it is important to adjust the viewport explicitly here.

Another necessary thing for the code above is declaring a static variable to memorize if we are in fullscreen mode or not:

Section: Local Variables (continuation):

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

This variable is necessary in Web Assembly because here we, and not some window manager, are

responsible to make some adjusts in our "window" to make it fullscreen. And we also make things to act as a fullscreen window, even if the web broser do not let us became fullscreen: we make our canvas occupy the screen are in the top of the page, so that the user still can enter manually in fullscreen if we fail. Because of this, we should remember if we are in fullscreen mode or not.

The window in this case is considered a SDL surface:

Section: Local Variables (continuation):

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

2.2.3. Creating a Window on (Microsoft) Windows

As in the previous code, we begin asking for the screen resolution:

Section: Windows: Create Window:

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

Next we need to define a class for our window. First we need to give it an arbitrary name in a string format. We will call it "WeaverWindow":

Section: Local Variables (continuation):

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

Now our class need a function to handle signals and messages sent to the window. Such messages are sent when the window is created, destroyed, resized, exposed, for example. We always can redirect the messages to function `DefWindowProc` to deal with them using default actions, but for some messages it is best to define ourselves the correct action to be taken. The function format is:

Section: API Functions (continuation):

```
#if defined(_WIN32)
LRESULT CALLBACK WindowProc(HWND window, UINT msg, WPARAM param1, LPARAM param2){
  switch(msg){
            <Section to be inserted: Windows: Deal with Window Messages>
    default:
      return DefWindowProc(window, msg, param1, param2);
  }
}
#endif
```

But when will we deal with the received message instead of just passing it to `DefWindowProc`? One of the cases is when the window get a message warning that it is being closed:

Section: Windows: Deal with Window Messages:

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

Now we need to create a class for our window. The class should have an unique non-conflicting name. We also pass an identifier with the instance of our program (that we get with `GetModuleHandle`) and the function to deal with signals and messages for the window (in our case, the default `DefWindowProc`).

Section: Windows: Create Window (continuation):

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

For convenince, we used the function `memset` to initialize the struct of our window class, as the majority of its elements can be set to zero to keep the default value. Because of this, we insert the following header:

Section: Cabealhos (continuation):

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

The code above assumes that we have declared a boolean variable that stores if we already created a class:

Section: Local Variables (continuation):

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

After defining the window class, we can create the window with the code below:

Section: Windows: Create Window (continuation):

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

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

We will store the handle to the created window in this variable:

Section: Local Variables (continuation):

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

Before showing the window on the screen we configure OpenGL on it:

Section: Windows: Criar Janela (continuation):

```
#if defined(_WIN32)
                <Section to be inserted: Windows: Configure OpenGL>
#endif
```

Now we ask the system to show the window to the user and then wait in a loop until we get a message saying that the window was created:

Section: Windows: Create Window (continuation):

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

2.2.4. Defining the Window Creation Function for All Environments

In the previous subsubsections, we defined code for window creation in different environments, and all

code was placed inside conditional macros that uses each code only in the right environment. Now we can unite all these previous code blocks in a single function:

Section: API Functions:

```
bool _Wcreate_window(struct _Wkeyboard *keyboard, struct _Wmouse *mouse){
  if(already_have_window == true)
    return false;
                    <Section to be inserted: X11: Create Window>
                <Section to be inserted: Web Assembly: Create Window>
                  <Section to be inserted: Windows: Create Window>
                    <Section to be inserted: Keyboard Initialization>
  _Wflush_window_input(keyboard, mouse);
  already_have_window = true;
  return true;
}
```

The variable that stores if the window already was created will be declared here:

Section: Local Variables (continuation):

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

2.3. Configuring OpenGL

Our API aim to support at least OpenGL S 2.0. In most environments this is not a problem. But on Windows, for example, we have no guarantees that we have OpenGL ES support. In this case, we should use OpenGL 4, but we should support only the functions and macros that exists on OpenGL ES 2.0.

2.3.1. Configuring OpenGL ES in X11

In X11, the interface used to program in OpenGL ES is called EGL and its functions and macros are declared in the following header:

Section: OpenGL Header (continuation):

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

Now we need to create a structure needed by OpenGL ES to store information about the graphical interface: a OpenGL display. Such display can be obtained from the X display that we already created:

Section: X11: Configure OpenGL ES:

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

This variable is declared here:

Section: Local Variables (continuation):

```
#if defined(__linux__) || defined(BSD)
static EGLDisplay *egl_display;
#endif
```

Now we need to obtain a configuration to the creation of a OpenGL context. We specify some requirements and get a configuration using the code below:

Section: X11: Configure OpenGL ES (continuation):

```
{
  bool ret;
  int number_of_configs_returned;
  int requested_attributes[] = {
    // We should support drawing in the screen and textures
    EGL_SURFACE_TYPE,  EGL_WINDOW_BIT | EGL_PBUFFER_BIT,
    // And at least 1 bit to the red color:
    EGL_RED_SIZE, 1,
    // At least 1 bit to the green color:
    EGL_GREEN_SIZE, 1,
    // At least 1 bit to the blue color:
    EGL_BLUE_SIZE, 1,
    // At least 1 bit to the alpha channel:
    EGL_ALPHA_SIZE, 1,
    // At least 1 bit to the depth channel:
    EGL_DEPTH_SIZE, 1,
    EGL_NONE
  };
  ret = eglChooseConfig(egl_display, requested_attributes,
                        &egl_config, 1, &number_of_configs_returned);
  if(ret == EGL_FALSE){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: Could not create valid EGL config.\n");
#endif
    return false;
  }
}
```

The struct that stores which configuration we will use is declared here:

Section: Local Variables (continuation):

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

Like how EGL needs its own display struct, it also needs its own window struct to store information about the window where it will draw. We can initialize such structure from the window struct used by the X library:

Section: X11: Configure OpenGL ES (continuation):

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

```
  return false;
}
```

The EGL window is decared here:

Section: Local Variables (continuation):

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

Now we can create the OpenGL context. We choose the major and minor OpenGL version using macros `W_WINDOW_OPENGL_MAJOR_VERSION` and `W_WINDOW_OPENGL_MINOR_VERSION`. With this information, we create the context:

Section: X11: Configure OpenGL ES (continuation):

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

The variable that stores the handle to the OpenGL context is declared here:

Section: Local Variables (continuation):

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

2.3.2. Configuring OpenGL on Web Assembly

No additional configuration is necessary. We already support OpenGL because when we created a "window" on Subsubsection 2.2.2, we passed a flag that activated OpenGL.

It should be noted that the OpenGL version in this case is WebGL, but this version is compatible with OpenGL ES.

2.3.3. Configuring OpenGL on Windows

To use OpenGL on Windows, first we warn the compiler about the necessary libraries, so that we do not need to pass them explicitly during the linking, and then we include the default library and the OpenGL library:

Section: OpenGL Header (continuation):

```
#if defined(_WIN32)
#pragma comment(lib, "Opengl32.lib")
#pragma comment(lib, "Shell32.lib")
#pragma comment(lib, "User32.lib")
```

```
#pragma comment(lib, "Gdi32.lib")
#include <windows.h>
#include <GL/gl.h>
#endif
```

We also need an struct with information about the device where we will draw. In our case, a window:

Section: Windows: Configure OpenGL:

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

This struct is declared here with the context that we will soon initialize:

Section: Local Variables (continuation):

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

Besides this, configuring OpenGL on Windows is a little messy. To start, the default function that creates an OpenGL context, `wglCreateContext`, is too primitive, without guaranteed support for more recent OpenGL functions and without support for necessary parameters to create a modern OpenGL context. But we can use an alternative function to create a modern OpenGL context: `wglCreateContextAttribsARB`. Using this function we can do some basic stuff, like choosing which OpenGL version we want and we can ask for support to modern OpenGL functions and that support more parameters.

The problem is that the function `wglCreateContextAttribsARB` is not a part of the API, but it is an extension. We must load this function using an auxiliary function before using it to create a context. But to use the function that loads it, we need an existing OpenGL context active.

This can be solved first creating a basic OpenGL context using the default API, loading the function that creates a proper context, creating a modern context, associating it to a window and loading all the necessary function that is not part of the default API. But it is not possible to associate more than one OpenGL context to a single window. So to load our function we need to create a dummy window and dummy context before:

Section: Windows: Configure OpenGL:

```
{
        <Section to be inserted: Windows: Create Dummy Window>
        <Section to be inserted: Windows: Create Dummy Context>
    <Section to be inserted: Windows: Load Initial OpenGL Functions>
  <Section to be inserted: Windows: Remove Dummy Context and Window>
}
```

First let's create our dummy window. As this will not be our rendering window, we do not need to create it using all the specifications from the real window:

Section: Windows: Create Dummy Window:

```
HWND dummy_window;
{
  WNDCLASS dummy_window_class;
  memset(&dummy_window_class, 0, sizeof(WNDCLASS));
  dummy_window_class.lpfnWndProc = WindowProc;
  dummy_window_class.hInstance = GetModuleHandle(NULL);
  dummy_window_class.lpszClassName = "DummyWindow";
  // This is expected to fail if the class already is registered. It happens
  // when we create more than one window with this function. So we just ignore
  // errors returned by this function:
  RegisterClass(&dummy_window_class);
  SetLastError(0);
```

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

Now to create a dummy context first we ned to choose an existing pixel format similar to what we specify and then we set the pixel format in our dummy window:

Section: Windows: Create Dummy Context:

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

And after configuring the pixel format, we can get the dummy OpenGL context:

Section: Windows: Create Dummy COntext (continuation):

```
  // ...
  dummy_context = wglCreateContext(dummy_device_context);
```

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

Now we need to load the functions that we need. Loading existing functions that are not normally accessible because they are considered extensions, can be done using the function `wglGetProcAddress` as in the loading function below that returns a pointer to the desired function:

Section: Local Functions:

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

Notice that as revealed by the code above, the function `wglGetProcAddress` can return up to 5 different values in case of error. Despite the fact that according with the documentation, only `NULL` is the correct way to signal an error.

The two needed functions is one that chooses a pixel format (like the pixel format chosen above, but with more options and resources) and another that creates an OpenGL context (like the dummy context above, but also with more options and resources).

To load new functions, first we declare pointers to store the memory position where are our new loaded functions. In the case of the two functions that we want now, their ponters are declared on `window.h` with the code below:

Section: Window Declarations (continuation):

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

We also place the real declaration on file `window.c`:

Section: Global Variables:

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

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

Once we have the pointers, we can initialize them loading and storing in them the functions in which we are interested:

Section: Windows: Load Initial OpenGL Functions:

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

And finally, after loading these two functions, we do not need anymore the dummy window and context:

Section: Windows: Remove Dummy Context and Window:

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

Now we are ready to choose our pixel format (the OpenGL configuration) in the modern way with our new function:

Section: Windows: Configure OpenGL (continuation):

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

```
#endif
    return false;
  }
  DescribePixelFormat(device_context, pixel_format_index,
                      sizeof(pixel_format_descriptor), &pixel_format_descriptor);
  if(!SetPixelFormat(device_context, pixel_format_index,
                     &pixel_format_descriptor)){
#if defined(W_DEBUG_WINDOW)
    fprintf(stderr, "ERROR: 'SetPixelFormat' failed.\n");
#endif
    return false;
  }
}
```

Now we will specify that we want a context with OpenGL version specified by two macros. And then use the funtion that creates a modern OpenGL context:

Section: Windows: Configure OpenGL (continuation):

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

While creating a real OpenGL context, we used some macros that are not defined. As we use them locally, we declare them in the header of file `window.c`:

Section: Headers (continuation):

```
#define WGL_TYPE_RGBA_ARB                   0x202B
#define WGL_PIXEL_TYPE_ARB                  0x2013
#define WGL_COLOR_BITS_ARB                  0x2014
#define WGL_DEPTH_BITS_ARB                  0x2022
#define WGL_STENCIL_BITS_ARB                0x2023
#define WGL_ACCELERATION_ARB                0x2003
#define WGL_DOUBLE_BUFFER_ARB               0x2011
#define WGL_CONTEXT_FLAGS_ARB               0x2094
#define WGL_DRAW_TO_WINDOW_ARB              0x2001
#define WGL_SUPPORT_OPENGL_ARB              0x2010
```

```
#define WGL_FULL_ACCELERATION_ARB               0x2027
#define WGL_CONTEXT_MAJOR_VERSION_ARB           0x2091
#define WGL_CONTEXT_MINOR_VERSION_ARB           0x2092
#define WGL_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB 0x0002
```

But we want to support in our API the functions compatible with OpenGL ES 2.0. Lots of such functions also are not part of the API presented by WGL n Windows. What we need to do is load such functions as extensions. For example, we start declaring the pointers for functions related with shader creation and management:

Section: Window Declarations (continuation):

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

Next we position he pointers as global variables in our file `window.c`:

Section: Global Variables:

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

Next we load for each of these pointers the corresponding function using the function that we defined a little above, before all the pointer declarations:

Section: Windows: Configure OpenGL (continuation):

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

When we create a shader with `glCreateShader`, we need to choose what kind of shader should be

created passing one of the following macros as argument:

Section: Macro Definition (continuation):

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

The type `GLchar` also must be defined on Windows:

Section: Macro Definition (continuation):

```
#if defined(_WIN32)
typedef char GLchar;
#endif
```

After compiling a shader, usually the user wants to check if the compilation was successful. This is done using functions that make queries about shaders, `glGetShaderiv` in particular. We define all functions related with queries about shaders here:

Section: Window Declarations (continuation):

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

The pointers are positioned here:

Section: Global Variables (continuation):

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

Each function is loaded to its respective pointer with:

Section: Windows: Configure OpenGL (continuation):

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

When the function `glGetShaderiv` is used, we can select which information about the shader we want to know passing one of the following macros:

Section: Macro Definition (continuation):

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

When the function `glGetShaderPrecisionFormat` is used to check for the precision of a given type, the chosen type is defined passing one of these macros:

Section: Macro Definition (continuation):

```
#if defined(_WIN32)
#define GL_LOW_FLOAT    0x8DF0
```

```
#define GL_MEDIUM_FLOAT 0x8DF1
#define GL_HIGH_FLOAT   0x8DF2
#define GL_LOW_INT      0x8DF3
#define GL_MEDIUM_INT   0x8DF4
#define GL_HIGH_INT     0x8DF5
#endif
```

When the function `glGetVertexAttribfv` or `glGetVertexAttribiv` are used to obtain information about vertices, the wanted information is informed passing as argument one of the following macros:

Section: Macro Definition (continuation):

```
#if defined(_WIN32)
#define GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING 0x889F
#define GL_VERTEX_ATTRIB_ARRAY_ENABLED        0x8622
#define GL_VERTEX_ATTRIB_ARRAY_SIZE           0x8623
#define GL_VERTEX_ATTRIB_ARRAY_STRIDE         0x8624
#define GL_VERTEX_ATTRIB_ARRAY_TYPE           0x8625
#define GL_VERTEX_ATTRIB_ARRAY_NORMALIZED     0x886A
#define GL_CURRENT_VERTEX_ATTRIB              0x8626
#endif
```

When using function `glGetVertexzAttribPointerv`, we must pass as one of the arguments the following macro:

Section: Macro Definition (continuation):

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

Let's also define this macro that is useful to query about which GLSL implementation we have:

Section: Macro Definition (continuation):

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

Once the user created and compiled shaders, usually she would need to create a program and link the shaders to the program. To enable this, we declare below the pointer for all functions about program management:

Section: Window Declarations (continuation):

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

These declared ponters are effectively positioned here:

Section: Global Variables (continuation):

```
#if defined(_WIN32)
GLuint (__stdcall *glCreateProgram)(void);
void (__stdcall *glAttachShader)(GLuint, GLuint);
```

```
void (__stdcall *glDetachShader)(GLuint, GLuint);
void (__stdcall *glLinkProgram)(GLuint);
void (__stdcall *glUseProgram)(GLuint);
void (__stdcall *glDeleteProgram)(GLuint);
#endif
```

And we initialize them with the correct functions:

Section: Windows: Configure OpenGL (continuation):

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

After finishing the program creation, usually the user wants to know if it was successful. For this, we need to prepare the pointers for functions that deal with program queries:

Section: Window Declarations (continuation):

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

We place the real pointers here:

Section: Global Variables (continuation):

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

And we initialize them with the real functions:

Section: Windows: Configure OpenGL (continuation):

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

```
if(glGetProgramInfoLog == NULL) return false;
glValidadeProgram = (void (__stdcall *)(GLuint))
                     load_function("glValidateProgram");
if(glValidadeProgram == NULL) return false;
```

When we use `glGetProgramiv` to obtain information about a program, we pass one of the following macros as argument to select which information we want to know:

Section: Macro Definition (continuation):

```
#if defined(_WIN32)
#define GL_DELETE_STATUS               0x8B80
#define GL_LINK_STATUS                 0x8B82
#define GL_VALIDATE_STATUS             0x8B83
#define GL_INFO_LOG_LENGTH             0x8B84
#define GL_ATTACHED_SHADERS            0x8B85
#define GL_ACTIVE_ATTRIBUTES           0x8B89
#define GL_ACTIVE_ATTRIBUTE_MAX_LENGTH 0x8B8A
#define GL_ACTIVE_UNIFORMS             0x8B86
#define GL_ACTIVE_UNIFORM_MAX_LENGTH   0x8B87
#endif
```

To obtain and set vertices attributes in a shader, we can use functions that will be associated with the following pointers:

Section: Window Declarations (continuation):

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

The pointers are positioned here:

Section: Global Variables (continuation):

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

And they are initialized here:

Section: Windows: Configure OpenGL (continuation):

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

The type of each vertex attribute returned by `glGetActiveAttrib` can be:

Section: Macro Definition (continuation):

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

And finally, the last functions related with shaders are the ones responsible to deal with uniform variables:

Section: Window Declarations (continuation):

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

These 21 pointers are positioned here:

Section: Global Variables (continuation):

```
#if defined(_WIN32)
GLint (__stdcall *glGetUniformLocation)(GLuint, const GLchar *);
void (__stdcall *glGetActiveUniform)(GLuint, GLuint, GLsizei, GLsizei *, GLint *,
                                     GLenum *, GLchar *);
void (__stdcall *glUniform1f)(GLint, GLfloat);
```

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

And now we need to initialize all these 21 pointers:

Section: Windows: Configure OpenGL (continuation):

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

```
glUniform1fv = (void (__stdcall *)(GLint, GLsizei, const GLfloat *))
                load_function("glUniform1fv");
if(glUniform1fv == NULL) return false;
glUniform2fv = (void (__stdcall *)(GLint, GLsizei, const GLfloat *))
                load_function("glUniform2fv");
if(glUniform2fv == NULL) return false;
glUniform3fv = (void (__stdcall *)(GLint, GLsizei, const GLfloat *))
                load_function("glUniform3fv");
if(glUniform3fv == NULL) return false;
glUniform4fv = (void (__stdcall *)(GLint, GLsizei, const GLfloat *))
                load_function("glUniform4fv");
if(glUniform4fv == NULL) return false;
glUniform1iv = (void (__stdcall *)(GLint, GLsizei, const GLint *))
                load_function("glUniform1iv");
if(glUniform1iv == NULL) return false;
glUniform2iv = (void (__stdcall *)(GLint, GLsizei, const GLint *))
                load_function("glUniform2iv");
if(glUniform2iv == NULL) return false;
glUniform3iv = (void (__stdcall *)(GLint, GLsizei, const GLint *))
                load_function("glUniform3iv");
if(glUniform3iv == NULL) return false;
glUniform4iv = (void (__stdcall *)(GLint, GLsizei, const GLint *))
                load_function("glUniform4iv");
if(glUniform4iv == NULL) return false;
glUniformMatrix2fv = (void (__stdcall *)(GLint, GLsizei, GLboolean,
                                         const GLfloat *))
                     load_function("glUniformMatrix2fv");
if(glUniformMatrix2fv == NULL) return false;
glUniformMatrix3fv = (void (__stdcall *)(GLint, GLsizei, GLboolean,
                                         const GLfloat *))
                     load_function("glUniformMatrix3fv");
if(glUniformMatrix3fv == NULL) return false;
glUniformMatrix4fv = (void (__stdcall *)(GLint, GLsizei, GLboolean,
                                         const GLfloat *))
                     load_function("glUniformMatrix4fv");
if(glUniformMatrix4fv == NULL) return false;
```

The uniform variables can have the same types than vertex attributes and also these new types:

Section: Macro Definition (continuation):

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

The last thing necessary to render simple images is send vertices to the video card. One of the ways to do this is sending a pointer instead of the entire vertex list. This can be done using the following functions related to vertices:

Section: Window Declarations (continuation):

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

These pointers are positioned as global variables:

Section: Global Variables (continuation):

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

And initialize them:

Section: Windows: Configure OpenGL (continuation):

```
glVertexAttrib1f = (void (__stdcall *)(GLuint, GLfloat))
                    load_function("glVertexAttrib1f");
if(glVertexAttrib1f == NULL) return false;
glVertexAttrib2f = (void (__stdcall *)(GLuint, GLfloat, GLfloat))
                      load_function("glVertexAttrib2f");
if(glVertexAttrib2f == NULL) return false;
glVertexAttrib3f = (void (__stdcall *)(GLuint, GLfloat, GLfloat, GLfloat))
                      load_function("glVertexAttrib3f");
if(glVertexAttrib3f == NULL) return false;
glVertexAttrib4f = (void (__stdcall *)(GLuint, GLfloat, GLfloat, GLfloat,
                                       GLfloat))
                    load_function("glVertexAttrib4f");
```

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

And besides the types that are already defined, we also need to define the following type for vertex attributes:

Section: Macro Definition (continuation):

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

Now we will define the pointers for functions related to buffer objects. The pointers are:

Section: Window Declarations (continuation):

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

And the declarations:

Section: Global Variables (continuation):

```
#if defined(_WIN32)
void (__stdcall *glGenBuffers)(GLsizei, GLuint *);
void (__stdcall *glDeleteBuffers)(GLsizei, const GLuint *);
void (__stdcall *glBindBuffer)(GLenum, GLuint);
void (__stdcall *glBufferData)(GLenum, GLsizeiptr, const void *, GLenum);
```

```
void (__stdcall *glBufferSubData)(GLenum, GLintptr, GLsizeiptr, const void *);
void (__stdcall *glIsBuffer)(GLuint);
void (__stdcall *glGetBufferParameteriv)(GLenum, GLenum, GLint *);
#endif
```

And the initialization:

Section: Windows: Configure OpenGL (continuation):

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

When we bind a buffer with `glBindBuffer`, it can have one of the following types:

Section: Macro Definition (continuation):

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

When we pass data to a buffer with `glBufferData`, we can choose between the following usages for the data:

Section: Macro Definition (continuation):

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

When we ask information about a buffer with `glGetBufferParameteriv`, we can ask for the following data:

Section: Macro Definition (continuation):

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

And these new functions require the following new types that should be large enough to store a pointer,

even not being necessarily pointers. The difference between them is that one is signed and the other is unsigned:

Section: Macro Definition (continuation):

```
#if defined(_WIN32)
// This includes a signed size_t definition:
#include <BaseTsd.h>
typedef size_t GLsizeiptr;
typedef SSIZE_T GLintptr;
#endif
```

The two functions about setting the viewport and the clipping are:

Section: Window Declarations (continuation):

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

Their declarations:

Section: Global Variables (continuation):

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

And the initialization:

Section: Windows: Configure OpenGL (continuation):

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

A type `GLclampf` represents a floating point number that should be in the range between 0 and 1. We can represent it by a regular floating point number.

Section: Macro Definition (continuation):

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

Now wee will prepare the functions about texturing:

Section: Window Declarations (continuation):

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

And we declare these functions in `window.c`:

Section: Global Variables (continuation):

```
#if defined(_WIN32)
void (__stdcall *glActiveTexture)(GLenum); // TEXTURE0..TEXTURE1...
```

```
void (__stdcall *glCompressedTexImage2D)(GLenum, int, GLenum, GLsizei,
                                         GLsizei, int, GLsizei, void *);
void (__stdcall *glCompressedTexSubImage2D)(GLenum, int, int, int, GLsizei,
                                            GLsizei, GLenum, GLsizei,
                                            void *);
void (__stdcall *glGenerateMipmap)(GLenum);
#endif
```

Here we initialize these 17 functions making them usable:

Section: Windows: Configure OpenGL (continuation):

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

The above functions require the following macros to be used in the enumerations:

Section: Macro Definition (continuation):

```
#if defined(_WIN32)
#define GL_RGB                          0x1907
#define GL_RGBA                         0x1908
#define GL_ALPHA                        0x1906
#define GL_TEXTURE0                     0x84C0
#define GL_TEXTURE1                     0x84C1
#define GL_TEXTURE2                     0x84C2
#define GL_TEXTURE3                     0x84C3
#define GL_TEXTURE4                     0x84C4
#define GL_TEXTURE5                     0x84C5
#define GL_TEXTURE6                     0x84C6
#define GL_TEXTURE7                     0x84C7
#define GL_TEXTURE8                     0x84C8
#define GL_TEXTURE9                     0x84C9
#define GL_TEXTURE10                    0x84CA
#define GL_TEXTURE11                    0x84CB
#define GL_TEXTURE12                    0x84CC
#define GL_TEXTURE13                    0x84CD
#define GL_TEXTURE14                    0x84CE
#define GL_TEXTURE15                    0x84CF
#define GL_TEXTURE16                    0x84D0
#define GL_TEXTURE17                    0x84D1
#define GL_TEXTURE18                    0x84D2
#define GL_TEXTURE19                    0x84D3
#define GL_TEXTURE20                    0x84D4
#define GL_TEXTURE21                    0x84D5
```

```
#define GL_TEXTURE22                        0x84D6
#define GL_TEXTURE23                        0x84D7
#define GL_TEXTURE24                        0x84D8
#define GL_TEXTURE25                        0x84D9
#define GL_TEXTURE26                        0x84DA
#define GL_TEXTURE27                        0x84DB
#define GL_TEXTURE28                        0x84DC
#define GL_TEXTURE29                        0x84DD
#define GL_TEXTURE30                        0x84DE
x#define GL_TEXTURE31                        0x84DF
#define GL_LUMINANCE                        0x1909
#define GL_TEXTURE_2D                       0x0DE1
#define GL_UNSIGNED_BYTE                    0x1401
#define GL_TEXTURE_WRAP_S                   0x2802
#define GL_TEXTURE_WRAP_T                   0x2803
#define GL_LUMINANCE_ALPHA                  0x190A
#define GL_TEXTURE_MAG_FILTER               0x2800
#define GL_TEXTURE_MIN_FILTER               0x2801
#define GL_UNSIGNED_SHORT_5_6_5             0x8363
#define GL_UNSIGNED_SHORT_4_4_4_4           0x8033
#define GL_UNSIGNED_SHORT_5_5_5_1           0x8034
#define GL_MAX_TEXTURE_IMAGE_UNITS          0x8872
#define GL_TEXTURE_CUBE_MAP_POSITIVE_X      0x8515
#define GL_TEXTURE_CUBE_MAP_POSITIVE_Y      0x8517
#define GL_TEXTURE_CUBE_MAP_POSITIVE_Z      0x8519
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_X      0x8516
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_Y      0x8518
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_Z      0x851A
#define GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS   0x8B4C
#define GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS 0x8B4D
#endif
```

The per fragment OpenGL ES functions are a series of tests and operations that modify the color of each pixel before it is drawn in a surface. Tipically if one of these tests fail, the pixel is discarded. The functions related with these operations are:

Section: Window Declarations (continuation):

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

And these functions are declared on `window.c`:

Section: Global Variables (continuation):

```
#if defined(_WIN32)
void (__stdcall *glSampleCoverage)(GLclampf, bool);
void (__stdcall *glStencilFuncSeparate)(GLenum, GLenum, int, unsigned int);
```

```
void (__stdcall *glStencilOpSeparate)(GLenum, GLenum, GLenum, GLenum);
void (__stdcall *glBlendEquation)(GLenum);
void (__stdcall *glBlendEquationSeparate)(GLenum, GLenum);
void (__stdcall *glBlendFuncSeparate)(GLenum, GLenum);
void (__stdcall *glBlendColor)(GLclampf, GLclampf, GLclampf, GLclampf);
#endif
```

And their initialization:

Section: Windows: Configure OpenGL (continuation):

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

The use of the above functions require the following macros to be used as enumerations:

Section: Macro Definition (continuation):

```
#if defined(_WIN32)
#define GL_ONE                      1
#define GL_ZERO                     0
#define GL_LESS                     0x0201
#define GL_INCR                     0x1E02
#define GL_DECR                     0x1E03
#define GL_KEEP                     0x1E00
#define GL_BACK                     0x0405
#define GL_FRONT                    0x0404
#define GL_EQUAL                    0x0202
#define GL_NEVER                    0x0200
#define GL_ALWAYS                   0x0207
#define GL_LEQUAL                   0x0203
#define GL_GEQUAL                   0x0206
#define GL_INVERT                   0x150A
#define GL_REPLACE                  0x1E01
#define GL_GREATER                  0x0204
#define GL_NOTEQUAL                 0x0205
#define GL_FUNC_ADD                 0x8006
```

```
#define GL_INCR_WRAP                0x8507
#define GL_DECR_WRAP                0x8508
#define GL_SRC_ALPHA                0x0302
#define GL_DST_ALPHA                0x0304
#define GL_SRC_COLOR                0x0300
#define GL_DST_COLOR                0x0306
#define GL_FUNC_SUBTRACT            0x800A
#define GL_FRONT_AND_BACK           0x0408
#define GL_CONSTANT_COLOR           0x8001
#define GL_CONSTANT_ALPHA           0x8003
#define GL_SRC_ALPHA_SATURATE¥    0x0308
#define GL_ONE_MINUS_SRC_COLOR      0x0301
#define GL_ONE_MINUS_DST_COLOR      0x0307
#define GL_ONE_MINUS_SRC_ALPHA      0x0303
#define GL_ONE_MINUS_DST_ALPHA      0x0305
#define GL_FUNC_REVERSE_SUBTRACT    0x800B
#define GL_ONE_MINUS_CONSTANT_COLOR 0x8002
#define GL_ONE_MINUS_CONSTANT_ALPHA 0x8004
#endif
```

About the functions that affect all the framebuffer, we have 2 functions to prepare:

Section: Window Declarations (continuation):

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

And their declarations in `window.c`:

Section: Global Variables (continuation):

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

The initialization:

Section: Windows: Configure OpenGL (continuation):

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

The last set of functions that we must prepare are the ones that manage framebuffer objects:

Section: Window Declarations (continuation):

```
#if defined(_WIN32)
extern void (__stdcall *glBindFramebuffer)(GLenum, unsigned int);
extern void (__stdcall *glDeleteFramebuffers)(GLsizei, unsigned int *);
extern void (__stdcall *glGenFramebuffers)(GLsizei, unsigned int *);
extern void (__stdcall *glBindRenderbuffer)(GLenum, unsigned int);
extern void (__stdcall *glDeleteRenderbuffers)(GLsizei, const unsigned int *);
extern void (__stdcall *glGenRenderbuffers)(GLsizei, unsigned int *);
extern void (__stdcall *glRenderbufferStorage)(GLenum, GLenum, GLsizei, GLsizei);
```

```
extern void (__stdcall *glFramebufferRenderbuffer)(GLenum, GLenum, GLenum,
                                                   unsigned int);
extern void (__stdcall *glFramebufferTexture2D)(GLenum, GLenum, GLenum,
                                                unsigned int, int);
extern void (__stdcall *glCheckFramebufferStatus)(GLenum);
extern boolean (__stdcall *glIsFrabuffer)(unsigned int);
extern void (__stdcall *glGetFramebufferAttachmentParameteriv)(GLenum, GLenum,
                                                               GLenum, int *);
extern boolean (__stdcall *glIsRenderbuffer)(unsigned int);
extern void (__stdcall *glGetRenderbufferParameteriv)(GLenum, GLenum, int *);
#endif
```

We declare these 14 functions also in `window.c`:

Section: Global Variables (continuation):

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

And the initialization:

Section: Windows: Configure OpenGL (continuation):

```
glBindFramebuffer = (void (__stdcall *)(GLenum, unsigned int))
                        load_function("glBindFramebuffer");
if(glBindFramebuffer == NULL) return false;
glDeleteFramebuffers = (void (__stdcall *)(GLsizei, unsigned int *))
                           load_function("glDeleteFramebuffers");
if(glDeleteFramebuffers == NULL) return false;
glGenFramebuffers = (void (__stdcall *)(GLsizei, unsigned int *))
                        load_function("glGenFramebuffers");
if(glGenFramebuffers == NULL) return false;
glBindRenderbuffer = (void (__stdcall *)(GLenum, unsigned int))
                         load_function("glBindRenderbuffer");
if(glBindRenderbuffer == NULL) return false;
glDeleteRenderbuffers = (void (__stdcall *)(GLsizei, const unsigned int *))
                           load_function("glDeleteRenderbuffers");
if(glDeleteRenderbuffers == NULL) return false;
```

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

The necessary macros to use these functions are:

Section: Macro Definition (continuation):

```
#if defined(_WIN32)
#define GL_RGBA4                                        0x8056
#define GL_RGB565                                       0x8D62
#define GL_RGB5_A1                                      0x8057
#define GL_FRAMEBUFFER                                  0x8D40
#define GL_RENDERBUFFER                                 0x8D41
#define GL_STENCIL_INDEX8                               0x8D48
#define GL_DEPTH_ATTACHMENT                             0x8D00
#define GL_DEPTH_COMPONENT16                            0x81A5
#define GL_COLOR_ATTACHMENT0                            0x8CE0
#define GL_STENCIL_ATTACHMENT                           0x8D20
#define GL_RENDERBUFFER_WIDTH                           0x8D42
#define GL_RENDERBUFFER_HEIGHT                          0x8D43
#define GL_FRAMEBUFFER_COMPLETE                         0x8CD5
#define GL_RENDERBUFFER_RED_SIZE                        0x8D50
#define GL_RENDERBUFFER_BLUE_SIZE                       0x8D52
#define GL_RENDERBUFFER_GREEN_SIZE                      0x8D51
#define GL_RENDERBUFFER_ALPHA_SIZE                      0x8D53
```

```
#define GL_RENDERBUFFER_DEPTH_SIZE                      0x8D54
#define GL_RENDERBUFFER_STENCIL_SIZE                    0x8D55
#define GL_RENDERBUFFER_INTERNAL_FORMAT                 0x8D44
#define GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE           0x8CD0
#define GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME           0x8CD1
#define GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL         0x8CD2
#define GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE 0x8CD3
#endif
```

2.3.4. Choosing OpenGL Version

In the previously defined code, in all environments. we choosed the OpenGL version according with the following two macros:

1) `W_WINDOW_OPENGL_MAJOR_VERSION`

2) `W_WINDOW_OPENGL_MINOR_VERSION`

We will now define what these macros values will be if the user did not personalized them to some value.

When we are running in X11, we can use EGL to create an OpenGL ES context. In this case, we prefer version 3.0, as is the newer version that also supports all OpenGL ES 2.0 resources.

If we are running in a browser using Web Assembly, we want to use WebGL 2, as this version is equivalent to OpenGL Es 3.0.

If we are running on Windows, not all drivers and hardware support OpenGL ES API. Because of this, we choose to use OpenGL 4.0, as this is a more common version that support the modern OpenGL functions used by OpenGL ES.

Our macro that sets the OpenGL version, because of this, is:

Section: Headers (continuation):

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

2.4. Closing a Window

Closing a window makes the window disappear and also finalizes any data initialized during window creation. This includes the OpenGL context.

2.4.1. Closing a Window on X

Closing a window in X11 means calling the function that asks the server to close the window and also closing the connection with the server. This is done calling respectively `XDestroyWindow` and `XCloseDisplay`. We also check if we really have an existing window before trying to close and destroy it.

Section: API Functions:

```
#if defined(__linux__) || defined(BSD)
bool _Wdestroy_window(void){
  if(already_have_window == false)
    return false;
  eglMakeCurrent(egl_display, EGL_NO_SURFACE, EGL_NO_SURFACE,
                 EGL_NO_CONTEXT );
```

```
  eglDestroySurface(egl_display, egl_window);
  eglDestroyContext(egl_display, egl_context);
  eglTerminate(egl_display);
  XDestroyWindow(display, window);
  XCloseDisplay(display);
  display = NULL;
  already_have_window = false;
  return true;
}
#endif
```

2.4.2. Closing a Window in Web Assembly

Closing a window when running in a web browser means finalize the SDL structures and hide the canvas where we were drawing. We do this with the following function:

Section: API Functions (continuation):

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

2.4.3. Closing a Window on Windows

Closing a window in Windows means calling a function to destroy the window:

Section: API Functions (continuation):

```
#if defined(_WIN32)
bool _Wdestroy_window(void){
  if(already_have_window == false)
    return false;
  wglMakeCurrent(NULL, NULL);
  wglDeleteContext(wgl_context);
  DestroyWindow(window);
  already_have_window = false;
  return true;
}
#endif
```

2.5. Rendering the Window

In a graphical program, usually we have some main loop and in this loop, we call some function that draws in our window. This means asking to update the window after making OpenGL calls to draw. How we do this depends on our environment and API.

2.5.1. Rendering Window on X

In X, when we asked EGL to create a window, by default it creates a double buffered window. Rendering in the screen means in this case to swap the window buffers. The back buffer where we were drawing became the front buffer, what is shown in the screen, and vice-versa. This is done by function `eglSwapBuffers`.

However, in addition to the command to ensure screen rendering, we must worry about another scenario: there are a myriad of window managers in X, and a very wide variety of environments in which our window will run. We do not have much control over some events, such as the fact that our window may lose keyboard and mouse focus, but continue to occupy the entire screen if it is a fullscreen window. Since this is a problematic scenario, since it can make us unable to interact with the system, we must ensure that if we are in fullscreen and detect that our window has lost focus, we must regain it. But we only do this if our window is actually visible at the time (or X causes a fatal error, we should always check this before changing focus to a window):

Section: API Functions (continuation):

```
#if defined(__linux__) || defined(BSD)
bool _Wrender_window(void){
  if(_Wis_fullscreen()){ // Ensure input focus when in fullscreen
    Window focused_window;
    int focus_status;
    XGetInputFocus(display, &focused_window, &focus_status);
    if(focused_window != window){
      XWindowAttributes attr;
      XGetWindowAttributes(display, window, &attr);
      if(attr.map_state == IsViewable)
        XSetInputFocus(display, window, RevertToParent, CurrentTime);
    }
  }// Swap buffers for rendering:
  return eglSwapBuffers(egl_display, egl_window);
}
#endif
```

2.5.2. Rendering Window on Web Assembly

The function `emscripten_sleep` is what makes the window update in a main loop. However, using it is a bad practice. In Web Assembly, programs should not run in an infinite loop, but a function should be registered to be executed over and over again instead of a more usual infinite loop. When we register a function this way, we do not need to use any additional function to ask to update our window. Therefore, in the following code we use only a flush command to ensure that all OpenGL calls are sent. But we do not need to make anything to update the screen, assuming that the program do not follow a bad practice:

Section: API Functions (continuation):

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

2.5.3. Rendering Window on Windows

In Windows, the WGL function responsible for swapping buffers in the window making our drawings visible is called `wglSwapLayerBuffers`:

Section: API Functions (continuation):

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

2.6. Getting Window Size

Next we will define the function that stores in pointers passed as aguments the size of our current window.

2.6.1. Getting Window Size on X

In X11, the Xlib API gives us the function `XGetGeometry` that gets information about some window or pixmap (image). This function returns a lot of information that we are not interested. But also returns the window width and height that we need:

Section: API Functions (continuation):

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

2.6.2. Getting Window Size on Web Assembly

In a web browser, we do not have a window, but a canvas. By convention, the canvas that we use as a window have the ID “canvas”. If we want to know its size, we need to use Javascript to get the canvas and obtain its width and height.

Section: API Functions (continuation):

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

```
    return document.getElementById("canvas").height;
  });
  window_size_x = *width;
  window_size_y = *height;
  if(*width > 0 && *height > 0)
    return true;
  else{
    *width = 0;
    *height = 0;
    return false;
  }
}
#endif
```

2.6.3. Getting Window Size on Windows

On Windows we use a call to `GetWindowRect` to obtain the window size. Which is stored in a `RECT` structure:

Section: API Functions (continuation):

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

2.7. Checking Fullscreen Mode

Here we will define the function that checks if we are in fullscreen mode or not.

2.7.1. Checking Fullscreen Mode on X

In X11, we create a fullscreen window adjusting a window attribute to make the window override any window manager configuration and occupy the whole screen. To check if we are in fullscreen mode, we just check if such attribute is marked or not:

Section: API Functions:

```
#if defined(__linux__) || defined(BSD)
bool _Wis_fullscreen(void){
  XWindowAttributes attributes;
  if(!already_have_window || display == NULL)
```

```
    return false;
  XGetWindowAttributes(display, window, &attributes);
  return (attributes.override_redirect == true);
}
#endif
```

2.7.2. Checking Fullscreen Mode on Web Assembly

Running in a web browser, we need to use Javascript to check if we are running in fullscreen mode or not. Note that here we not always will be able to set fullscreen mode, as this depends on the browser configuration. The function below will be a way to test if we succeed or not in entering fullscreen mode.

Section: API Functions (continuation):

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

2.7.3. Checking Fullscreen Mode on Windows

Detect a fullscreen mode on Windows means using the function `SHQueryUserNotificationState` to obtain the state of the screen. If we are in a fullscreen window, we will know getting as result `QUNS_BUSY`:

Section: API Functions (continuation):

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

2.8. Changing Between Fullscreen and Windowed Mode

Here we present the functions that toggles between fulllscreen and windowed mode in different environments.

Doing this correctly involves remembering the window size whet it is in windowed mode. If we enter fullscreen and then return to windowed mode, the winow size should be the last remembered size. We will store the last size for our window in windowed mode in the variables below. The initial values for these variables can be controlled by some macros if them exist. If not, we set their values to zero, meaning that we have no previous stored value for the window size:

Section: Local Variables (continuation):

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

```
static unsigned last_window_size_y = 0;
#endif
```

Also when we enter or exit fullscreen, or if we change the size of our window, it could be necessary for the user to recompute the position for elements in the screen. Because of this, we will et the user to define a funtion to be executed when this happens. If no function is passed by the user, then nothing additional is executed. The pointer that will store the user-defined function, if it exists, is:

Section: Local Variables (continuation):

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

To register a new function and store its addresss in the above pointer, we invoke the following function:

Section: API Functions (continuation):

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

2.8.1. Changing Between Fullscreen and Windowed Mode on X

Toggling between windowed and fulscreen mode means changing one attribute in our window, the attriute that determines if the window manager will have control over our window to draw window decoration, or if otherwise we should ignore the window manager and control entirely the window size and position.

As this means alternating the control over the window to the window manager and the X server, after changing this attribute we need to withdraw our window, making it disappear momentaneally, and then map our window again, making it appear with or without window decorators. If we are entering now in fullscreen mode, we must also move and resize the window to make it occupy the whole screen again.

Section: API Functions (continuation):

```
#if defined(__linux__) || defined(BSD)
void _Wtoggle_fullscreen(void){
  int new_size_x, new_size_y, old_size_x, old_size_y;
  bool changing_to_fullscreen = false;
  XSetWindowAttributes attributes;
  if(!already_have_window)
    return;
  _Wget_window_size(&old_size_x, &old_size_y);
  XWithdrawWindow(display, window, 0);
  if(!_Wis_fullscreen()){
    attributes.override_redirect = true;
    changing_to_fullscreen = true;
    XMoveWindow(display, window, 0, 0);
  }
  else
    attributes.override_redirect = false;
  XChangeWindowAttributes(display, window, CWOverrideRedirect,
                          &attributes);
  XMapWindow(display, window);
  { // After request the mapping for the window, wait until it became mapped
    XEvent e;
    do{
      XNextEvent(display, &e);
    } while(e.type != MapNotify);
  }
  if(changing_to_fullscreen){
```

```
    _Wget_screen_resolution(&new_size_x, &new_size_y);
    XMoveResizeWindow(display, window, 0, 0, new_size_x, new_size_y);
                    <Section to be inserted: X11: Wait for Resizing>
  }
  else{
    new_size_x = ((last_window_size_x > 0)?(last_window_size_x):(800));
    new_size_y = ((last_window_size_y > 0)?(last_window_size_y):(600));
    XResizeWindow(display, window, new_size_x, new_size_y);
                    <Section to be inserted: X11: Wait for Resizing>
  }
  _Wget_window_size(&new_size_x, &new_size_y);
  glViewport(0, 0, new_size_x, new_size_y);
  if(resizing_function != NULL)
    resizing_function(old_size_x, old_size_y, new_size_x, new_size_y);
}
#endif
```

In the above function, we sent to the X server a request to change our window size. However, this does not mean that when the function `XResizeWindow` or `XResizeWindow` return, the window is already in the new size. It is important to wait until the size is successfully changed, otherwise we could return incorrect or inconsistent values in other functions. For example, if the window asks what is our window size after toggling the fullscreen. Because of this in the abocve code we wait until the size change is completed. This is more important when we exit

However, we cannot wait forever. We could be running in a window manager that ignores our request to change the window size when we are not in fullscreen mode. Because of this, we should have a maximum waiting time. The code below shows how we implement the waiting code. It waits until our window have the new size, but if this does not happen in 0.1 seconds, it halts the waiting:

Section: X11: Wait for Resizing:

```
{ // Wait at most 0.1 seconds for the resizing
  struct timeval begin_time, now;
  int new_width, new_height;
  gettimeofday(&begin_time, NULL);
  do{
    _Wget_window_size(&new_width, &new_height);
    gettimeofday(&now, NULL);
  } while((new_width != new_size_x || new_height != new_size_y) &&
          (now.tv_sec - begin_time.tv_sec) * 1000000 +
          (now.tv_usec - begin_time.tv_usec) < 100000);
}
```

And the function `gettimeofday` requires the following header:

Section: Headers (continuation):

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

2.8.2. Changing Between Fullscreen and Windowed Mode on Web Assembly

In a web browser, when we try to enter fullscreen mode, two things can happen. The browser can let us enter fullscreen mode or not. In the second case, we still try to simulate a fullscreen making our HTML canvas having the same size than the screen and occupy the top part of the document. We do this to let the user enter manually in fullscreen using the browser options.

This means that we could be in a true fullscreen or in a simulated one. Because of this we store

if we are in fullscreen mode (real or simulated) in the static variable `fullscreen_mode`. The function `_Wis_fullscreen`, on the other hand, only informs if we are in the real fullscreen mode.

Our function that toggles the windowed and fullscreen mode will make the changes according with the static variable. Even if we cannot enter a real fullscreen, we toggle between the windowed mode and the simulated fullscreen mode:

Section: API Functions (continuation):

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

2.8.3. Changing Between Fullscreen and Windowed Mode on Windows

On Windows we controll the fullscreen mode changing between styles for our window. We can read and write new information about a style with the functions `GetWindowLongA` and `SetWindowLongA`:

Section: API Functions (continuation):

```
#if defined(_WIN32)
void _Wtoggle_fullscreen(void){
  int new_size_x, new_size_y, old_size_x, old_size_y;
  DWORD window_style = GetWindowLongPtrA(window, GWL_STYLE);
  if(!already_have_window)
    return;
  _Wget_window_size(&old_size_x, &old_size_y);
  if(window_style & WS_POPUP){ // We are in fullscreen. Enter windowed mode.
    SetWindowLong(window, GWL_STYLE, WS_OVERLAPPED | WS_CAPTION | WS_VISIBLE);
    if(last_window_size_x > 0 && last_window_size_y > 0)
      SetWindowPos(window, HWND_TOP, 0, 0, last_window_size_x, last_window_size_y,
                   SWP_FRAMECHANGED | SWP_SHOWWINDOW);
    else
      SetWindowPos(window, HWND_TOP, 0, 0, last_window_size_x, last_window_size_y,
                   SWP_FRAMECHANGED | SWP_SHOWWINDOW | SWP_NOSIZE);
  } else{ // We are in windowed mode. Enter fullscreen.
    _Wget_screen_resolution(&new_size_x, &new_size_y);
    SetWindowLongPtr(window, GWL_STYLE, WS_POPUP | WS_VISIBLE);
    SetWindowPos(window, HWND_TOP, 0, 0, new_size_x, new_size_y,
                 SWP_SHOWWINDOW | SWP_FRAMECHANGED);
  }
  _Wget_window_size(&new_size_x, &new_size_y);
  glViewport(0, 0, size_x, size_y);
  if(resizing_function != NULL)
    resizing_function(old_size_x, old_size_y, new_size_x, new_size_y);
}
#endif
```

2.9. Resizing the Window

Here we will define how to resize our window in different environments, provided that the window exists and that we are not in fullscreen mode. In case of error, the function returns false. Otherwise, it returns true.

2.9.1. Resizing the Window on X

Resize on X11 means using the function `XResizeWindow`, which we already used in other functions. Besides this, we also need to update the new maximum and minimum size for our window to be equal the new size using `XSetWMNormalHints`. We do this becaus we do not let the user change the window size by means outside this resizing function. Finally, after resizing the window we wait for confirmation that the operation succeed measuring the new window size. We wait at most 0.5 seconds for confirmation. And like when we toggled the fullscreen mode, here we also update the OpenGL viewport dize and run the function passed by the user associated with the window resize, if it exists.

Section: API Functions (continuation):

```
#if defined(__linux__) || defined(BSD)
bool _Wresize_window(int width, int height){
  int old_size_x, old_size_y, new_size_x = width, new_size_y = height;
  if(_Wis_fullscreen() || width <= 0 || height <= 0 || !already_have_window)
```

```
    return false;
  _Wget_window_size(&old_size_x, &old_size_y);
  {
    XSizeHints hints;
    hints.flags = PMinSize | PMaxSize;
    hints.min_width = hints.max_width = width;
    hints.min_height = hints.max_height = height;
    XSetWMNormalHints(display, window, &hints);
  }
  last_window_size_x = width;
  last_window_size_y = height;
  XResizeWindow(display, window, width, height);
```

<Section to be inserted: X11: Wait for Resizing>

```
  glViewport(0, 0, width, height);
  if(resizing_function != NULL)
    resizing_function(old_size_x, old_size_y, width, height);
  return true;
}
#endif
```

2.9.2. Resizing the Window on Web Assembly

On Web Assembly, to change the size of our canvas, if we are not in fullscreen mode (the real one or the simulated one), we just access our canvas by its ID and edit its CSS properties and attributes about its size:

Section: API Functions (continuation):

```
#if defined(__EMSCRIPTEN__)
bool _Wresize_window(int width, int height){
  int old_size_x, old_size_y;
  if(fullscreen_mode || width <= 0 || height <= 0 || !already_have_window)
    return false;
  _Wget_window_size(&old_size_x, &old_size_y);
  last_window_size_x = width;
  last_window_size_y = height;
  EM_ASM({
    var el = document.getElementById("canvas");
    el.style.width = ($0 / window.devicePixelRatio) + "px";
    el.width = $0;
    el.style.height = ($1 / window.devicePixelRatio) + "px";
    el.height = $1;
  }, width, height);
  glViewport(0, 0, width, height);
  if(resizing_function != NULL)
    resizing_function(old_size_x, old_size_y, width, height);
  return true;
}
#endif
```

2.9.3. Resizing the Window on Windows

On Windows, resizing the window means calling `SetWindowPos`. We do it only if we are not on fullscreen mode. And after resizing, if it exists, we execute the user custom function to be run when the window size changes:

Section: API Functions (continuation):

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

3. Managing Input

In this section we will deal with input detection. As when the user presses a key or moves the mouse.

3.1. Defining the Keyboard and Mouse

When we have a window, we can detect inputs made by the user when that window has focus. We can detect events made by the keyboard and mouse.

For us the keyboard will be the following global struct variable:

Section: Window Data Structures:

```
struct _Wkeyboard{
  long key[W_KEYBOARD_SIZE + 1]; // Key array: a timer for each key
};
```

In the key array above, we will have one position for each keyboard key where we will store a number to act as a timer for each key. The meaning of the number stored in each position depends on the value. The options are:

1) If we store zero, this means that the key is not being pressed.

2) If we store 1, this means that the key started to being pressed now.

3) If the number is positive and greater than 1, this means that the key is being pressed and still was not released. The number represents for how long the key is being pressed in some unit of time.

4) If the number is negative, this means that the key was released now. And the opposite of such number means for how long the key was pressed before being released in some unit of time.

The number of keys depends on the operating system and environment. The macro `W_KEYBOARD_SIZE` is the number of supported keys. We add to the keyboard struct one additional position to map unsupported keys. The last position never will store any value different than zero.

The mouse will be represented by the following struct:

Section: Window Data Structures (continuation):

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

The buttons follow the same convention than the keys in a keyboars. We can detect if a button is pressed, for how long it is being pressed, if the button was released and for how long the button was pressed before being released.

The variables `x` and `y` represent the pointer position in pixels from our window. The variables `dx` and `dy` represent the pointer velocity in pixels per unit of time. The variables `ddx` and `ddy` represent the pointer acceleration in pixels per unit of time squared.

Given these structs, we need to periodically call the following function to update them:

Section: API Functions (continuation):

```
void _Wget_window_input(unsigned long long current_time,
                        struct _Wkeyboard *keyboard,
                        struct _Wmouse *mouse){
  if(already_have_window == false)
    return;
                <Section to be inserted: Before Reading Window Events>
                     <Section to be inserted: Get Window Events>
                <Section to be inserted: After Reading Window Events>
}
```

This function works reading in a loop the list of events received by the window and then updating the mouse and keyboard structs. For example, on X11, the reading of events is done this way:

Section: Get Window Events:

```
#if defined(__linux__) || defined(BSD)
XEvent event;
while(XPending(display)){
  XNextEvent(display, &event);
                    <Section to be inserted: X11: Manage Events>
}
#endif
```

If we are in a Web Assembly environment, we use SDL API to read the events:

Section: Get Window Events (continuation):

```
#if defined(__EMSCRIPTEN__)
SDL_Event event;
while(SDL_PollEvent(&event)){
                <Section to be inserted: Web Assembly: Manage Events>
}
#endif
```

On Windows the events are called “messages”. And we read them in the following way:

Section: Get Window Events (continuation):

```
#if defined(_WIN32)
MSG event;
while(PeekMessage(&event, window, WM_KEYFIRST, WM_KEYLAST, PM_REMOVE)){
              <Section to be inserted: Windows: Manage Keyboard Events>
}
while(PeekMessage(&event, window, WM_MOUSEFIRST, WM_MOUSELAST, PM_REMOVE)){
               <Section to be inserted: Windows: Manage Mouse Events>
}
#endif
```

In the Windows case we can filter the events and then we can treat separatelly the mouse and keyboard events, ignoring all others.

3.2. Reading the Keyboard

Here we define how we monitor the keyboard on different environments to update the keyboard structure.

The events in all environments will warn when a key is pressed or released. But they will not say nothing about for how long they were pressed. We need to store and deduce this information. For this, we will need an array of keys being pressed and released at the present time. In the event loop we just update this array:

Section: Local Variables:

```
static struct{
  unsigned key; // Which key was pressed?
  unsigned long long time;    // When it was pressed?
} pressed_keys[32];
unsigned released_keys[32]; // Which keys were released?
```

As a part of the keyboard initialization, and also to reset its status, we should always call code to mark all positions in the array of pressed and released keys as having the key zero. This empties the lists, as we use the null key as the mark for the end of our list, like the null character is used to mark the end of a string. We also need to set as zero all keys oin the keyboard key array. The code to initialize or reinitialize the keyboard is:

Section: Initializing or Reinitializing Keyboard:

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

}
```

3.2.1. Reading the Keyboard on X

In X11, each key in the keyboard have a different and unique code between8 and 255. This code represents the phisical pressed key, but have no relation with an specific symbol associated to the key. The symbol depends on the key mapping. To obtain the symbol associated with the key, we need to translate the code to a symbol (`keycode` to `keysym`). Anyway, this means that we need to store information about a potential number of almost 256 keys.

Section: Macro Definition (continuation):

```
#if defined(__linux__) || defined(BSD)
#define W_KEYBOARD_SIZE 257
#endif
```

To detect if a key was pressed on X11, we use the following code:

Section: X11: Manage Events:

```
if(event.type == KeyPress){
  unsigned key = event.xkey.keycode;
            <Section to be inserted: Add 'key' to List of Pressed Keys>
}
```

To detect a key release before removing the key from list of pressed keys and adding it to list of removen keys:

Section: X11: Manage Events (continuation):

```
if(event.type == KeyRelease){
  unsigned key = event.xkey.keycode;
```

<Section to be inserted: Remove 'key' from Pressed Keys and Add to Released Keys>
}

While the above code manages correctly the keycodes, the user also needs to know what these keycodes represent. Knowing that the key 0xff0d was pressed means very little. Knowing that the key "Enter" was pressed is much more meaningful. The user should be able to check the content of `keyboard.key[W_ENTER]` without needing to know the code for the key. For this, during the keyboard initialization, we should discover all the symbols supported by our keyboard, giving to them correct names. As this depends on the keyboard mapping, we can discover this only during the program execution. The code to discover the position for all supported keys are:

Section: Keyboard Initialization:

```
#if defined(__linux__) || defined(BSD)
{
  int i;
  // Produce keyboard events only if they really happen:
  XkbSetDetectableAutoRepeat(display, true, NULL);
  for(i = 8; i < 256; i ++){
    unsigned long value = XkbKeycodeToKeysym(display, i, 0, 0);
    switch(value){
    case 0: break;
    case XK_Escape: W_ESC = i; break;
    case XK_BackSpace: W_BACKSPACE = i; break;
    case XK_Tab: W_TAB = i; break;
    case XK_Return: W_ENTER = i; break;
    case XK_Up:   W_UP   = i; break; case XK_Down:  W_DOWN =  i; break;
    case XK_Left: W_LEFT = i; break; case XK_Right: W_RIGHT = i; break;
    case XK_0: W_0 = i; break;     case XK_1: W_1 = i; break;
    case XK_2: W_2 = i; break;     case XK_3: W_3 = i; break;
    case XK_4: W_4 = i; break;     case XK_5: W_5 = i; break;
    case XK_6: W_6 = i; break;     case XK_7: W_7 = i; break;
    case XK_8: W_8 = i; break;     case XK_9: W_9 = i; break;
    case XK_minus: W_MINUS = i; break;    case XK_plus: W_PLUS = i; break;
    case XK_F1: W_F1 = i; break;   case XK_F2: W_F2 = i; break;
    case XK_F3: W_F3 = i; break;   case XK_F4: W_F4 = i; break;
    case XK_F5: W_F5 = i; break;   case XK_F6: W_F6 = i; break;
    case XK_F7: W_F7 = i; break;   case XK_F8: W_F8 = i; break;
    case XK_F9: W_F9 = i; break;   case XK_F10: W_F10 = i; break;
    case XK_F11: W_F11 = i; break; case XK_F12: W_F12 = i; break;
    case XK_Shift_L: W_LEFT_SHIFT = i; break;
    case XK_Shift_R: W_RIGHT_SHIFT = i; break;
    case XK_Control_L: W_LEFT_CTRL = i; break;
    case XK_Control_R: W_RIGHT_CTRL = i; break;
    case XK_Alt_L: W_LEFT_ALT = i; break;
    case XK_Alt_R: W_RIGHT_ALT = i; break;
    case XK_Super_L: W_LEFT_SUPER = i; break;
    case XK_Super_R: W_RIGHT_SUPER = i; break;
    case XK_space: W_SPACE = i; break;
    case XK_A: W_A = i; break;   case XK_b: W_B = i; break;
    case XK_c: W_C = i; break;   case XK_d: W_D = i; break;
    case XK_e: W_E = i; break;   case XK_f: W_F = i; break;
    case XK_g: W_G = i; break;   case XK_h: W_H = i; break;
```

```
    case XK_i: W_I = i; break;   case XK_j: W_J = i; break;
    case XK_k: W_K = i; break;   case XK_l: W_L = i; break;
    case XK_m: W_M = i; break;   case XK_n: W_N = i; break;
    case XK_o: W_O = i; break;   case XK_p: W_P = i; break;
    case XK_q: W_Q = i; break;   case XK_r: W_R = i; break;
    case XK_s: W_S = i; break;   case XK_t: W_T = i; break;
    case XK_u: W_U = i; break;   case XK_v: W_V = i; break;
    case XK_w: W_W = i; break;   case XK_x: W_X = i; break;
    case XK_y: W_Y = i; break;   case XK_z: W_Z = i; break;
    case XK_Insert: W_INSERT = i; break;
    case XK_Home: W_HOME = i; break;
    case XK_Page_Up: W_PAGE_UP = i; break;
    case XK_Delete: W_DELETE = i; break;
    case XK_End: W_END = i; break;
    case XK_Page_Down: W_PAGE_DOWN = i; break;
    default: break;
    }
  }
}
#endif
```

The use of function `XkbKeycodeToKeysym` requires the header:

Section: Headers (continuation):

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

All these variables that represent positions in our array of keys need to be declared:

Section: Window Declarations (continuation):

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

And we should initialize them with `W_KEYBOARD_SIZE`. This value represents a non-assigned key that never will be pressed or released. Only after the keyboard initialization we assign them correct values:

Section: Global Variables (continuation):

```
int W_BACKSPACE = W_KEYBOARD_SIZE, W_TAB = W_KEYBOARD_SIZE,
    W_ENTER = W_KEYBOARD_SIZE, W_UP = W_KEYBOARD_SIZE, W_DOWN = W_KEYBOARD_SIZE,
    W_LEFT = W_KEYBOARD_SIZE, W_RIGHT = W_KEYBOARD_SIZE, W_0 = W_KEYBOARD_SIZE,
    W_1 = W_KEYBOARD_SIZE, W_2 = W_KEYBOARD_SIZE, W_3 = W_KEYBOARD_SIZE,
    W_4 = W_KEYBOARD_SIZE, W_5 = W_KEYBOARD_SIZE, W_6 = W_KEYBOARD_SIZE,
    W_7 = W_KEYBOARD_SIZE, W_8 = W_KEYBOARD_SIZE, W_9 = W_KEYBOARD_SIZE,
    W_MINUS = W_KEYBOARD_SIZE, W_PLUS = W_KEYBOARD_SIZE, W_F1 = W_KEYBOARD_SIZE,
    W_F2 = W_KEYBOARD_SIZE, W_F3 = W_KEYBOARD_SIZE, W_F4 = W_KEYBOARD_SIZE,
    W_F5 = W_KEYBOARD_SIZE, W_F6 = W_KEYBOARD_SIZE, W_F7 = W_KEYBOARD_SIZE,
    W_F8 = W_KEYBOARD_SIZE, W_F9 = W_KEYBOARD_SIZE, W_F10 = W_KEYBOARD_SIZE,
```

```
    W_F11 = W_KEYBOARD_SIZE, W_F12 = W_KEYBOARD_SIZE,
    W_LEFT_SHIFT = W_KEYBOARD_SIZE, W_RIGHT_SHIFT = W_KEYBOARD_SIZE,
    W_LEFT_ALT = W_KEYBOARD_SIZE, W_RIGHT_ALT = W_KEYBOARD_SIZE,
    W_LEFT_CTRL = W_KEYBOARD_SIZE, W_RIGHT_CTRL = W_KEYBOARD_SIZE,
    W_LEFT_SUPER = W_KEYBOARD_SIZE, W_RIGHT_SUPER = W_KEYBOARD_SIZE,
    W_SPACE = W_KEYBOARD_SIZE, W_A = W_KEYBOARD_SIZE, W_B = W_KEYBOARD_SIZE,
    W_C = W_KEYBOARD_SIZE, W_D = W_KEYBOARD_SIZE, W_E = W_KEYBOARD_SIZE,
    W_F = W_KEYBOARD_SIZE, W_G = W_KEYBOARD_SIZE, W_H = W_KEYBOARD_SIZE,
    W_I = W_KEYBOARD_SIZE, W_J = W_KEYBOARD_SIZE, W_K = W_KEYBOARD_SIZE,
    W_L = W_KEYBOARD_SIZE, W_M = W_KEYBOARD_SIZE, W_N = W_KEYBOARD_SIZE,
    W_O = W_KEYBOARD_SIZE, W_P = W_KEYBOARD_SIZE, W_Q = W_KEYBOARD_SIZE,
    W_R = W_KEYBOARD_SIZE, W_S = W_KEYBOARD_SIZE, W_T = W_KEYBOARD_SIZE,
    W_U = W_KEYBOARD_SIZE, W_V = W_KEYBOARD_SIZE, W_W = W_KEYBOARD_SIZE,
    W_X = W_KEYBOARD_SIZE, W_Y = W_KEYBOARD_SIZE, W_Z = W_KEYBOARD_SIZE,
    W_INSERT = W_KEYBOARD_SIZE, W_HOME = W_KEYBOARD_SIZE,
    W_PAGE_UP = W_KEYBOARD_SIZE, W_DELETE = W_KEYBOARD_SIZE,
    W_END = W_KEYBOARD_SIZE, W_PAGE_DOWN = W_KEYBOARD_SIZE,
    W_ESC = W_KEYBOARD_SIZE, W_ANY = 0;
```

3.2.2. Reading the Keyboard on Web Assembly

Emscripten gives us access to keyboard keys using SDL API. The number of supported keys in a keyboard is givn by macro `SDL_NUM_SCANCODES` which, at the time of this writting is equalo 512 different keys. Probably this value will not change in the next future.

Section: Macro Definition (continuation):

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

In X11, we presented in the last subsection that there is a difference between the phisical key and the symbol associated with it. The same difference appear here in SDL. The phisical key in X was called `Keycode` and here in SDL is called `Scancode`. The symbol associated, that can change depending on the key mapping, in X was called `Keysym` and here is called `Scancode`. Yes, the terms are confusing because a scancode means totally different things in both APIs. Using the SDL naming, we detect with the following code if a key was pressed:

Section: Web Assembly: Manage Events:

```
if(event.type == SDL_KEYDOWN){
  unsigned key = event.key.keysym.scancode;
            <Section to be inserted: Add 'key' to List of Pressed Keys>
}
```

To detect that a key is released, we use the following code:

Section: Web Assembly: Manage Events (continuation):

```
if(event.type == SDL_KEYUP){
  unsigned key = event.key.keysym.scancode;
    <Section to be inserted: Remove 'key' from Pressed Keys and Add to Released
Keys>
}
```

The above code is identical to what was present in the X11 version. We just had to update the names to match SDL structs. Now we need the code that initializes correctly the name for our supported keys:

Section: Keyboard Initialization:

```
#if defined(__EMSCRIPTEN__)
{
  int i;
  for(i = 0; i < W_KEYBOARD_SIZE; i ++){
    // TODO: When Emscripten implements 'SDL_GetKeyFromScancode', use it:
    unsigned long value = SDL_SCANCODE_TO_KEYCODE(i);
    switch(value){
    case 0: break;
    case SDLK_ESCAPE: W_ESC = i; break;
    case SDLK_BACKSPACE: W_BACKSPACE = i; break;
    case SDLK_TAB: W_TAB = i; break;
    case SDLK_RETURN: W_ENTER = i; break;
    case SDLK_UP:   W_UP   = i; break; case SDLK_DOWN:  W_DOWN =  i; break;
    case SDLK_LEFT: W_LEFT = i; break; case SDLK_RIGHT: W_RIGHT = i; break;
    case SDLK_0: W_0 = i; break;     case SDLK_1: W_1 = i; break;
    case SDLK_2: W_2 = i; break;     case SDLK_3: W_3 = i; break;
    case SDLK_4: W_4 = i; break;     case SDLK_5: W_5 = i; break;
    case SDLK_6: W_6 = i; break;     case SDLK_7: W_7 = i; break;
    case SDLK_8: W_8 = i; break;     case SDLK_9: W_9 = i; break;
    case SDLK_MINUS: W_MINUS = i; break;    case SDLK_PLUS: W_PLUS = i; break;
    case SDLK_F1: W_F1 = i; break;   case SDLK_F2: W_F2 = i; break;
    case SDLK_F3: W_F3 = i; break;   case SDLK_F4: W_F4 = i; break;
    case SDLK_F5: W_F5 = i; break;   case SDLK_F6: W_F6 = i; break;
    case SDLK_F7: W_F7 = i; break;   case SDLK_F8: W_F8 = i; break;
    case SDLK_F9: W_F9 = i; break;   case SDLK_F10: W_F10 = i; break;
    case SDLK_F11: W_F11 = i; break; case SDLK_F12: W_F12 = i; break;
    case SDLK_LSHIFT: W_LEFT_SHIFT = i; break;
    case SDLK_RSHIFT: W_RIGHT_SHIFT = i; break;
    case SDLK_LCTRL: W_LEFT_CTRL = i; break;
    case SDLK_RCTRL: W_RIGHT_CTRL = i; break;
    case SDLK_LALT: W_LEFT_ALT = i; break;
    case SDLK_RALT: W_RIGHT_ALT = i; break;
    case SDLK_SPACE: W_SPACE = i; break;
    case SDLK_a: W_A = i; break;   case SDLK_b: W_B = i; break;
    case SDLK_c: W_C = i; break;   case SDLK_d: W_D = i; break;
    case SDLK_e: W_E = i; break;   case SDLK_f: W_F = i; break;
    case SDLK_g: W_G = i; break;   case SDLK_h: W_H = i; break;
    case SDLK_i: W_I = i; break;   case SDLK_j: W_J = i; break;
    case SDLK_k: W_K = i; break;   case SDLK_l: W_L = i; break;
    case SDLK_m: W_M = i; break;   case SDLK_n: W_N = i; break;
    case SDLK_o: W_O = i; break;   case SDLK_p: W_P = i; break;
    case SDLK_q: W_Q = i; break;   case SDLK_r: W_R = i; break;
    case SDLK_s: W_S = i; break;   case SDLK_t: W_T = i; break;
    case SDLK_u: W_U = i; break;   case SDLK_v: W_V = i; break;
    case SDLK_w: W_W = i; break;   case SDLK_x: W_X = i; break;
    case SDLK_y: W_Y = i; break;   case SDLK_z: W_Z = i; break;
    case SDLK_INSERT: W_INSERT = i; break;
    case SDLK_HOME: W_HOME = i; break;
    case SDLK_PAGEUP: W_PAGE_UP = i; break;
    case SDLK_DELETE: W_DELETE = i; break;
    case SDLK_END: W_END = i; break;
```

```
    case SDLK_PAGEDOWN: W_PAGE_DOWN = i; break;
    default: break;
    }
  }
}
#endif
```

3.2.3. Reading the Keyboard on Windows

On windows the phisical keys are represented by a `Scancode` and the symbol associated with it is called a "virtual-key code". The number of different scancodes is always 256.

Section: Macro Definition (continuation):

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

Detecting keys being pressed is done using exactly he same code that for X11 and SDL, just with some minor changes and also using different names, as Windows calls differently its functions and structs:

Section: Windows: Manage Keyboard Events:

```
if(event.message == WM_KEYDOWN){
  unsigned key = (event.lParam & 0x00ff0000) >> 16;
            <Section to be inserted: Add 'key' to List of Pressed Keys>
}
```

Note that on Windows, to get the scancode of a given key, we need to use some bit manipulation. Other than this, the code is the same than what was already presented for X11 and SDL. The code to detect a key being released is:

Section: Windows: Manage Keyboard Events (continuation):

```
if(event.message == WM_KEYUP){
  unsigned key = (event.lParam & 0x00ff0000) >> 16;
    <Section to be inserted: Remove 'key' from Pressed Keys and Add to Released
Keys>
}
```

And finally, we initialize the names for each key, making them point to the right position in our key array:

Section: Keyboard Initialization:

```
#if defined(_WIN32)
{
  int i;
  for(i = 0; i < W_KEYBOARD_SIZE; i ++){
    unsigned long value = MapVirtualKey(i, MAPVK_VSC_TO_VK_EX);
    switch(value){
    case 0: break;
    case VK_ESCAPE: W_ESC = i; break;
    case VK_BACK: W_BACKSPACE = i; break;
    case VK_TAB: W_TAB = i; break;
    case VK_RETURN: W_ENTER = i; break;
    case VK_UP:   W_UP   = i; break; case VK_DOWN:  W_DOWN =  i; break;
    case VK_LEFT: W_LEFT = i; break; case VK_RIGHT: W_RIGHT = i; break;
    case '0': W_0 = i; break;     case '1': W_1 = i; break;
    case '2': W_2 = i; break;     case '3': W_3 = i; break;
```

```
    case '4': W_4 = i; break;     case '5': W_5 = i; break;
    case '6': W_6 = i; break;     case '7': W_7 = i; break;
    case '8': W_8 = i; break;     case '9': W_9 = i; break;
    case VK_OEM_MINUS: W_MINUS = i; break; case VK_OEM_PLUS: W_PLUS = i; break;
    case VK_F1: W_F1 = i; break;   case VK_F2: W_F2 = i; break;
    case VK_F3: W_F3 = i; break;   case VK_F4: W_F4 = i; break;
    case VK_F5: W_F5 = i; break;   case VK_F6: W_F6 = i; break;
    case VK_F7: W_F7 = i; break;   case VK_F8: W_F8 = i; break;
    case VK_F9: W_F9 = i; break;   case VK_F10: W_F10 = i; break;
    case VK_F11: W_F11 = i; break; case VK_F12: W_F12 = i; break;
    case VK_LSHIFT: W_LEFT_SHIFT = i; break;
    case VK_RSHIFT: W_RIGHT_SHIFT = i; break;
    case VK_LCONTROL: W_LEFT_CTRL = i; break;
    case VK_RCONTROL: W_RIGHT_CTRL = i; break;
    case VK_MENU: W_LEFT_ALT = i; break;
    case VK_RMENU: W_RIGHT_ALT = i; break;
    case VK_SPACE: W_SPACE = i; break;
    case 'A': W_A = i; break;   case 'B': W_B = i; break;
    case 'C': W_C = i; break;   case 'D': W_D = i; break;
    case 'E': W_E = i; break;   case 'F': W_F = i; break;
    case 'G': W_G = i; break;   case 'H': W_H = i; break;
    case 'I': W_I = i; break;   case 'J': W_J = i; break;
    case 'K': W_K = i; break;   case 'L': W_L = i; break;
    case 'M': W_M = i; break;   case 'N': W_N = i; break;
    case 'O': W_O = i; break;   case 'P': W_P = i; break;
    case 'Q': W_Q = i; break;   case 'R': W_R = i; break;
    case 'S': W_S = i; break;   case 'T': W_T = i; break;
    case 'U': W_U = i; break;   case 'V': W_V = i; break;
    case 'W': W_W = i; break;   case 'X': W_X = i; break;
    case 'Y': W_Y = i; break;   case 'Z': W_Z = i; break;
    case VK_INSERT: W_INSERT = i; break;
    case VK_HOME: W_HOME = i; break;
    case VK_PRIOR: W_PAGE_UP = i; break;
    case VK_DELETE: W_DELETE = i; break;
    case VK_END: W_END = i; break;
    case VK_NEXT: W_PAGE_DOWN = i; break;
    default: break;
    }
  }
}
#endif
```

3.2.4. Additional Code for Keyboard Support

With the code written above we can detect when a new key is pressed and when it is released. Such code is Operating System dependent. But we still did not write the code to manage our list of pressed and released keys and how we use such list to update the keyboard struct.

First of all, lets define how we add a key to the list of pressed keys. Remember that at this point, in all Operating System and environments, we wrote code to store what key was pressed in the variable `key`. And remember that our list of pressed keys represent the end of the list as a position where a null key is stored. Knowing this, adding a key to the list means iterating in the list until finding a null key and replacing it by our new pressed key:

Section: Add 'key' to List of Pressed Keys:

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

Now to remove a key from the list of pressed keys and add it to the list of relesed keys, we first need to iterated over pressed keys until find the correct key. Then we move all other keys to the right to one position in the left. This erases a key from the list of pressed keys, After this, we iterate over the list of released keys until we find an empty position, where we put our new released key:

Section: Remove 'key' from Pressed Keys and Add to Released Keys:

```
{
  int i;
  long stored_time = -1;
  for(i = 0; i < 32; i ++){ // Removing from pressed keys
    if(pressed_keys[i].key == key){
      int j;
      stored_time = pressed_keys[i].time;
      for(j = i; j < 31; j ++){
        pressed_keys[j].key = pressed_keys[j + 1].key;
        pressed_keys[j].time = pressed_keys[j + 1].time;
        if(pressed_keys[j].key == 0)
          break;
      }
      pressed_keys[31].key = 0;
      break;
    }
  }
  for(i = 0; i < 32; i ++){ // Adding to released keys:
    if(released_keys[i] == 0)
      released_keys[i] = key;
  }
  if(i == 32) // Too many key releases, ignore and clean keyboard
    keyboard -> key[key] = 0;
  else{
    // Updating keyboard struct:
    keyboard -> key[key] = - (long) (current_time - stored_time);
    if(keyboard -> key[key] == 0)
      keyboard -> key[key] = -1; // Press/release in the same frame
  }
```

```
}
```

But how should we use the information of our list of pressed and released keys to update our keyboard struct? First, in each frame, after updating our list of pressed keys, we should iterate over the list and for each key, update in the keyboard struct for how much time they are being pressed:

Section: After Reading Window Events:

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

Now the timer for pressed keys will keep updating, not remaining always with the number 1.

When the key is released, we already written code that turned to negative its timer. But we still did not write the code that returns the timer for the key to zero in the frame after the release. Before reading thew input in each frame and before updating the list of pressed keys, we should iterate over the list of released keys and set their timers to zero, as the keys now are not being pressed or released. Doing this we also should remove all keys from the list of released keys:

Section: Before Reading Window Events:

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

3.3. Reading the Mouse

Like for the keyboard keys, we also need to store a list of pressed buttons and released buttons. But the number of buttons are much smaller. Our list will only store at most 4 buttons. It is unrealistic to give meaning for more than 4 button presses in a mouse:

Section: Local Variables (continuation):

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

But unlike a keyboard, a mouse can move and produce movements in the mouse pointer. If we want to know not only the mouse pointer position, but also its velocity and acceleration, we should store the previous velocity. We use these variables for this:

Section: Local Variables (continuation):

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

But when we read or the first time the mouse position, we have no way to compute its velocity, as we do not have a previous position. It makes no sense compute some value for the mouse velocity at this moment. Likewise, in the frame after the initialization, despite knowing a previous position and knowing how to compute the velocity, we do not have a previous velocity, so we cannot compute the acceleration. Only in the second frame after the initialization we can compute both position, velocity and acceleration. Until then, we should let thes unknown values as zero. Because of this, we will use a variable to store in which stage of mouse initialization we are. The value 3 means that it was not initialized and we do not know not even the position. The value 2 means that we initialized in this frame and we know only the current position. The value 1 means that we now can compute velocity. And zero means that the initialization is over and we know all values. We store this in the following variable:

Section: Local Variables (continuation):

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

And we initialize the mouse making the list of released and pressed buttons empty and also we initialize all buttons, the position, velocity and acceleration as zero:

Section: Initializing or Reinitializing Mouse:

```
{
  int i;
  for(i = 0; i < 4; i ++){
    pressed_buttons[i].button = 0;
    released_buttons[i] = 0;
  }
  for(i = 0; i < W_MOUSE_SIZE; i ++)
    mouse -> button[i] = 0;
  mouse -> x = mouse -> y = mouse -> dx = mouse -> dy = mouse -> ddx = mouse -> ddy = 0;
  last_mouse_dx = last_mouse_dy = 0;
  mouse_initialization = 3;
                    <Section to be inserted: Get Mouse Position>
}
```

3.3.1. Reading the Mouse on X

In X11, the number of supported buttons in a mouse are 5, and they are numbered from 1 to 5. We will assign to number 0 the meaning of "any pressed button". For mouses we do not need to worry about different mappings giving different meanings for the buttons. Therefore, we can define the number of buttons in the mouse and their positions in the button array with the following macros:

Section: Macro Definition (continuation):

```
#if defined(__linux__) || defined(BSD)
#define W_MOUSE_SIZE 6
#define W_MOUSE_LEFT   Button1
#define W_MOUSE_MIDDLE Button2
#define W_MOUSE_RIGHT  Button3
#define W_MOUSE_X1     Button4
#define W_MOUSE_X2     Button5
#endif
```

About the button presses, to detect them we should search for an event called `ButtonPress`. And then we discover the button pressed, adding it to our list of pressed buttons:

Section: X11: Manage Events (continuation):

```
if(event.type == ButtonPress){
  unsigned button = event.xbutton.button;
```

```
        <Section to be inserted: Add 'button' to List of Pressed Buttons>
}
```

To detect a button being released, we should check for events of type `ButtonRelease` and read in the event which button was reeased. Then, we remove such button from the list of pressed buttons and add it to the list of released buttons:

Section: X11: Manage Events (continuation):

```
if(event.type == ButtonRelease){
  unsigned button = event.xbutton.button;
      <Section to be inserted: Remove 'button' from Pressed Buttons and Add to
Released Buttons>
}
```

And we also need to detect mouse movements. For this, we check for events about mouse movement:

Section: X11: Manage Events (continuation):

```
if(event.type == MotionNotify){
  int x, y;
  x = event.xmotion.x;
  y = (window_size_y - 1) - event.xmotion.y;
            <Section to be inserted: Update Mouse Position 'x' and 'y'>
}
```

Notice that we had to transform the coordinate y. Beause Xlib treats the top of the window as coordinate zero and we want to consider the bottom of the screen as coordinate zero.

Besides this, we also need to get the initial mouse position even if the user did not move the mouse pointer. For this, we use the code below that calls function `XQueryPointer`. This function returns a lot of information about mouse pointer coordinates, not only relative to the current window, but relative to parent and child windows. We ignore most of these informations, keeping only information about mouse position relative to our current window.

Section: Get Mouse Position:

```
#if defined(__linux__) || defined(BSD)
{
  int x, y;
  Window root_return, child_return;
  int root_x_return, root_y_return;
  unsigned mask_return;
  XQueryPointer(display, window, &root_return, &child_return,
                &root_x_return, &root_y_return, &x, &y, &mask_return);
  // Transforming 'y' coordinate:
  y = (window_size_y - 1) - y;
            <Section to be inserted: Update Mouse Position 'x' and 'y'>
}
#endif
```

3.3.2. Reading the Mouse on Web Assembly

Using the SDL API from Emscripten, we also support on Web Assembly a total of five different mouse buttons, each one numbered between 1 and 5. We can also use the position 0 to mean "any key":

Section: Macro Definition (continuation):

```
#if defined(__EMSCRIPTEN__)
#define W_MOUSE_SIZE 6
#define W_MOUSE_LEFT   SDL_BUTTON_LEFT
#define W_MOUSE_MIDDLE SDL_BUTTON_MIDDLE
```

```
#define W_MOUSE_RIGHT  SDL_BUTTON_RIGHT
#define W_MOUSE_X1     SDL_BUTTON_X1
#define W_MOUSE_X2     SDL_BUTTON_X2
#endif
```

We detect that some mouse button was pressed with the code below:

Section: Web Assembly: Manage Events (continuation):

```
if(event.type == SDL_MOUSEBUTTONDOWN){
  unsigned button = event.button.button;
          <Section to be inserted: Add 'button' to List of Pressed Buttons>
}
```

We detect that a button was released with the code below:

Section: Web Assembly: Manage Events (continuation):

```
if(event.type == SDL_MOUSEBUTTONUP){
  unsigned button = event.button.button;
      <Section to be inserted: Remove 'button' from Pressed Buttons and Add to
Released Buttons>
}
```

And we detect below the mouse movement:

Section: Web Assembly: Tratar Eventos (continuation):

```
if(event.type == SDL_MOUSEMOTION){
  int x, y;
  x = event.motion.x;
  y = (window_size_y - 1) - event.motion.y;
             <Section to be inserted: Update Mouse Position 'x' and 'y'>
}
```

To get the initial mouse position, even if the mouse pinter is not moving, we use the following code:

Section: Get Mouse Position (continuation):

```
#if defined(__EMSCRIPTEN__)
{
  int x, y;
  SDL_GetMouseState(&x, &y);
  // Transforming 'y' coordinate:
  y = (window_size_y - 1) - y;
             <Section to be inserted: Update Mouse Position 'x' and 'y'>
}
#endif
```

3.3.3. Reading the Mouse on Windows

On Windows we do not have default numbers assigned to each of the mouse buttons. Instead, each button generate different kind of events instead of a single "button press" event. But we still need to associate each button with a position in our mouse button array. Therefore, we create our own numbers with the macros below:

Section: Macro Definition (continuation):

```
#if defined(_WIN32)
#define W_MOUSE_SIZE 6
#define W_MOUSE_LEFT   1
#define W_MOUSE_MIDDLE 2
```

```
#define W_MOUSE_RIGHT  3
#define W_MOUSE_X1     4
#define W_MOUSE_X2     5
#endif
```

As Windoews generate a different event for each button press, and also treats differently the buttons X1 and X2, this gives us more work to treat te different button pressings:

Section: Windows: Manage Mouse Events:

```
if(event.message == WM_LBUTTONDOWN){
  unsigned button = W_MOUSE_LEFT;
          <Section to be inserted: Add 'button' to List of Pressed Buttons>
}
else if(event.message == WM_MBUTTONDOWN){
  unsigned button = W_MOUSE_MIDDLE;
          <Section to be inserted: Add 'button' to List of Pressed Buttons>
}
else if(event.message == WM_RBUTTONDOWN){
  unsigned button = W_MOUSE_RIGHT;
          <Section to be inserted: Add 'button' to List of Pressed Buttons>
}
else if(event.message == WM_XBUTTONDOWN){
  unsigned button = W_MOUSE_X2;
  if((event.wParam >> 16) & 0x0001){
    unsigned button = W_MOUSE_X1;
  }
          <Section to be inserted: Add 'button' to List of Pressed Buttons>
}
```

To detect when a mouse button is released, we also need to check for all different events that each button generate, treating buttons X1 and X2 differently:

Section: Windows: Manage Mouse Events (continuation):

```
if(event.message == WM_LBUTTONUP){
  unsigned button = W_MOUSE_LEFT;
      <Section to be inserted: Remove 'button' from Pressed Buttons and Add to
Released Buttons>
}
else if(event.message == WM_MBUTTONUP){
  unsigned button = W_MOUSE_MIDDLE;
      <Section to be inserted: Remove 'button' from Pressed Buttons and Add to
Released Buttons>
}
else if(event.message == WM_RBUTTONUP){
  unsigned button = W_MOUSE_RIGHT;
      <Section to be inserted: Remove 'button' from Pressed Buttons and Add to
Released Buttons>
}
else if(event.message == WM_XBUTTONUP){
  unsigned button = W_MOUSE_X2;
  if((event.wParam >> 16) & 0x0001){
    unsigned button = W_MOUSE_X1;
  }
      <Section to be inserted: Remove 'button' from Pressed Buttons and Add to
```

Released Buttons>
```
}
```

And here is how we detect the event in which the mouse pointer moves:

Section: Windows: Manage Mouse Events (continuation):

```
if(event.message == WM_MOUSEMOVE){
  int x, y;
  x = (event.lParam & 0xffff);
  y = (window_size_y - 1) - (event.lParam >> 16);
              <Section to be inserted: Update Mouse Position 'x' and 'y'>
}
```

Like in all other environments, we change the coordinate to make it relative with thew lower-left corner, not with the upper-left corner as is more usual.

And to get the initial mouse position, even if the mouse pointer is not moving, we use the code:

Section: Get Mouse Position:

```
#if defined(_WIN32)
{
  int x, y;
  POINT point;
  // Get screen coordinate:
  GetCursorPos(&point);
  // Convert to window coordinate:
  ScreenToClient(window, &point);
  // Get coordinate, transforming coordinate 'y':
  x = point.x;
  y = (window_size_y - 1) - point.y;
              <Section to be inserted: Update Mouse Position 'x' and 'y'>
}
#endif
```

3.3.4. Additional Code for Mouse Support

We defined in previous Subsections specific code for each different enironment to detect mouse button presses and mouse movement. Now we will define here the code common for all environments.

First we will show the code to add a button to the list of pressed buttons; Recall that the list is an array with at most 4 elements where we mark the premature end of the list with a null button:

Section: Add 'button' to List of Pressed Buttons:

```
{
  int i;
  for(i = 0; i < 4; i ++){
    if(pressed_buttons[i].button == button) // Already pressed
      break;
    if(pressed_buttons[i].button == 0){ // Is being pressed now
      pressed_buttons[i].button = button;
      pressed_buttons[i].time = current_time;
      // Updating mouse:
      mouse -> button[button] = 1;
      break;
    }
  }
  if(i == 4) continue; // Ignoring: too many button presses
```

```
}
```

And we also can remove a button from the list of pressed buttons. When we do it, we add it to the list of released buttons:

Section: Remove 'button' from Pressed Buttons and Add to Released Buttons:

```
{
  int i;
  long stored_time = -1;
  for(i = 0; i < 4; i ++){ // Removing from pressed buttons
    if(pressed_buttons[i].button == button){
      int j;
      stored_time = pressed_buttons[i].time;
      for(j = i; j < 3; j ++){
        pressed_buttons[j].button = pressed_buttons[j + 1].button;
        pressed_buttons[j].time = pressed_buttons[j + 1].time;
        if(pressed_buttons[j].button == 0)
          break;
      }
      pressed_buttons[3].button = 0;
      break;
    }
  }
  for(i = 0; i < 4; i ++){ // Adding to released buttons:
    if(released_buttons[i] == 0)
      released_buttons[i] = button;
  }
  if(i == 4) // Too many button releases, ignore and clean mouse
    mouse -> button[button] = 0;
  else{
    // Updating mouse struct:
    mouse -> button[button] = - (long) (current_time - stored_time);
    if(mouse -> button[button] == 0)
      mouse -> button[button] = -1; // Press/release in the same frame
  }
}
```

And as we did for the keyboard, after updating the list of pressed buttons, we update in the mouse struct for how long each pressed button is being pressed:

Section: After Reading Window Events (continuation):

```
{
  int i;
  // Update the information if any button is being pressed:
  mouse -> button[W_ANY] = (pressed_buttons[0].button != 0);
  for(i = 0; i < 4; i ++){
    if(pressed_buttons[i].button == 0)
      break;
    if(current_time > pressed_buttons[i].time)
      mouse -> button[pressed_buttons[i].button] =
        (current_time - pressed_buttons[i].time);
  }
}
```

And before we updte the list of pressed buttons, we empty the list of released buttons and clean the

position in the button array marking it as not being pressed anymore:

Section: After Reading Window Events:

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

Now the code that update the mouse position, velocity and acceleration (depending on the state of mouse initialization). We run this code after we read a new x and y coordinate for the mouse when we detect a movement and also when we get these values for the first time during initialization.

Section: Update Mouse Position 'x' and 'y':

<Section to be inserted: Correct Mouse Position 'x' and 'y' if Forcing Landscape Mode>

```
{
  if(mouse_initialization < 3){ // If we know previous position
    mouse -> dx = (x - mouse -> x);
    mouse -> dy = (y - mouse -> y);
  }
  mouse -> x = x;
  mouse -> y = y;
  if(mouse_initialization < 2){ // If we know previous velocity
    mouse -> ddx = (mouse -> dx - last_mouse_dx);
    mouse -> ddy = (mouse -> dy - last_mouse_dy);
  }
  last_mouse_dx = mouse -> dx;
  last_mouse_dy = mouse -> dy;
  if(mouse_initialization > 0)
    mouse_initialization --;
}
```

We also will correct above the mouse coordinate if the macro `W_FORCE_LANDSCAPE` is defined and if the window height is greater than the window width. In this case, we will rotate the axis x and y of our coordinate system. We do the correction using this code:

Section: Correct Mouse Position 'x' and 'y' if Forcing Landscape Mode:

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

3.4. Additional Input Functions

When we read the input for the first time in our window, or when we read the input after something interrupted us for some time, then a lot of user input can be accumulated. In such cases, we will not be able to treat such delayed input in aan appropriate manner and the best is ignore it. For this case, we will use

the following function:

Section: API Functions (continuation):

```
void _Wflush_window_input(struct _Wkeyboard *keyboard,
                          struct _Wmouse *mouse){
  // Read accumulated input:
  _Wget_window_input(~0x0, keyboard, mouse);
  // Empty all input information about the keyboard:
            <Section to be inserted: Initializing or Reinitializing Keyboard>
  // Empty all input information abut the mouse:
              <Section to be inserted: Initializing or Reinitializing Mouse>
}
```

4. Final File Sctructure

The file with all the necessary source code for the functions defined in this article will have the following organization:

(P)

Arquivo: `src/window.c`:

```
#include "window.h"
                              <Section to be inserted: Headers>
                          <Section to be inserted: Local Functions>
                         <Section to be inserted: Global Variables>
                          <Section to be inserted: Local Variables>
                          <Section to be inserted: API Functions>
```