FreeWRL library interface

Michel Briand

2011-02-12

Contents

1 FreeWRL library specification

1.1 Current situation

1.1.1 Excerpt from libFreeWRL.h

I stripped comments and platform specific code from `libFreeWRL.h`:

```
const char *libFreeWRL_get_version();
typedef struct freewrl_params {
        ...
} freewrl_params_t;
extern freewrl_params_t fw_params;
bool initFreeWRL(freewrl_params_t *params);
void startFreeWRL(const char *url);
void closeFreeWRL();
void terminateFreeWRL();
int ConsoleMessage(const char *fmt, ...);
```

```
void create_EAI();
void create_MIDIEAI();
void doQuit();
bool Anchor_ReplaceWorld();
#define VIEWER_NONE 0
#define VIEWER_EXAMINE 1
#define VIEWER_WALK 2
#define VIEWER_EXFLY 3
#define VIEWER_FLY 4
#define VIEWER_YAWPITCHZOOM 5
void set_viewer_type(const int type);
void setGeometry_from_cmdline(const char *gstring);
void setSnapFile(const char* file);
void setSnapTmp(const char* file);
void setEaiVerbose();
void setScreenDist(const char *optArg);
void setStereoParameter(const char *optArg);
void setShutter(void);
void setEyeDist(const char *optArg);
void setAnaglyphParameter(const char *optArg);
void setSideBySide(void);
void setStereoBufferStyle(int);
void initStereoDefaults(void);
void setLineWidth(float lwidth);
void setSnapGif();
void setPrintShot();
#define RUNNINGASPLUGIN (isBrowserPlugin)
extern char *BrowserFullPath;
extern int _fw_pipe, _fw_FD;
extern int _fw_browser_plugin;
extern int isBrowserPlugin;
extern uintptr_t _fw_instance;
extern char *keypress_string;
void askForRefreshOK();
int checkRefresh();
void resetRefresh();
```

1.1.2 What's used in the Linux front-end

I gathered the functions really used in the front-end code:

```
$ nm .libs/freewrl | sed -n '/ U /p' | sed -e 's/[ \t]*U //' | grep -v GLIBC
ConsoleMessage
doQuit
initFreeWRL
initStereoDefaults
libFreeWRL_get_version
setAnaglyphParameter
setEaiVerbose
setEyeDist
setGeometry_from_cmdline
setLineWidth
setScreenDist
```

```
setShutter
setSideBySide
setSnapFile
setSnapGif
setSnapTmp
setStereoParameter
startFreeWRL
```

Doing the same for data structures and global variables:

```
$ nm .libs/freewrl | sed -n '/ B /p' | sed -e 's/.*[ \t]*B //' | grep -v GLIBC
_fw_browser_plugin
_fw_instance
_fw_pipe
isBrowserPlugin
keypress_string
params
```

Looking (quickly) into the code, I've found:

in `src/bin/main.c`

```
    freewrl_params_t *params = NULL;
```

1.2 What to do to improve the situation

1.2.1 Naming convention

We have to decide about an enforced naming convention.

Not to say that

```
  gnu_or_open_source_convention
```

is better than

```
  MicrosoftOrSoCalledPolishNotation
```

That's not my intent.

But to make all the FreeWRL code homogeneous.

Personnaly I feel the first more readeable. But feel free to comment / propose.

1.2.2 Namespace

All symbols that are visible outside must be declare in `libFreeWRL.h`.

All platform specific cases should be handle through glue code.

- Data types
  All data types that can be used from outside (thus declared in `libFreeWRL.h`) must be consistently named:

```
typedef struct { .... } fwl_params_t;
extern int fwl_is_browser_plugin;
extern char *fwl_keypress_string;
```

instead of:

```
typedef struct freewrl_params { .... } freewrl_params_t;
extern int isBrowserPlugin;
extern char *keypress_string;
```

- Functions
  All `libFreeWRL` functions that can be called from outside (thus declared in `libFreeWRL.h`) must be consistently named:

  ```
  void fwl_init();
  void fwl_quit();
  ```

  instead of:

  ```
  bool initFreeWRL(freewrl_params_t *params);
  void doQuit();
  ```

- MACROs
  All macros that can be used from outside (thus declared in `libFreeWRL.h`) must be consistently named:

  ```
  #define FWL_VIEWER_WALK 2
  #define FWL_IS_BROWSER_PLUGIN (fwl_is_browser_plugin)
  ```

  instead of:

  ```
  #define VIEWER_WALK 2
  #define RUNNINGASPLUGIN (isBrowserPlugin)
  ```

1.2.3 Graphic system initialization

- Current situation
  Correct me if I'm wrong.

  On `Linux` the front-end calls the library which initialize itself the `X11` display and the `GLX` context. Thus the front-end does not have the control here.

  On `Mac` the front-end initialize the `Aqua` display and the `AGL` context.

  On `Windows` the situation is somewhat similar to `Linux` (see `fwWindow32.c`).

  Let's call the graphical system initialization and variables the *context*.

- A more flexible method
  A method to make all platforms use the same codebase for the *context*.

  Beside the `fwl_params_t` structure that has to be extensively used we ought to create another data type for *context* initialization. Let's call it `fwl_context_t` for the sake of simplicity.

  This data type shall be *anonymous*, that is *generic*. Then, each platform will have a *specific* implementation of this type. I.e. a `fwl_context_x11_t`, `fwl_context_aqua_t`, `fwl_context_win_t`, ...

  This type will have two use cases.

The first is when the front-end initializes itself the *context*. I.e. when it initializes the *context* in a way that fits its specific needs. The front-end do that, then fills in the FreeWRL context with actual values for the library to access them when this is needed.

The second use case is when the front-end delegate the graphical context initialization to the library. Through this data structure, the front-end can access the context variables if it needs.

- Example
  Linux `GLX` context and `X11` window identifier in the specific `fwl_context_x11_t`:

  ```
  typedef struct {

          ...
          GLXContext ctx;
          Window win;
          ...

  } fwl_context_x11_t;
  ```

  Windows `WGL` context and `Win32` window identifier in the specfic `fwl_context_win_t`:

  ```
  typedef struct {

          ...
          HGLRC ctx;
          HWND win;
          ...

  } fwl_context_x11_t;
  ```

1.2.4 Events

- Current situation
  All events, coming through a specific graphical system, must be translated into a common, generic event. Because the current situation is a mess. Each platform has its specificities about event. This prevent us to improve the user experience. This will be a concern for iPhone and Android…

  Example of the current situation with the event generated when the window geometry changes. You can see a lot of differences…

  Linux

  ```
  switch(event.type) {
          case ConfigureNotify:
                  setScreenDim (event.xconfigure.width,event.xconfigure.height);
                  break;
  ```

  Windows

  ```
  switch( msg ) {
          case WM_SIZE:
                  GetClientRect(hWnd, &rect);
                  screenWidth = rect.right; /*used in mainloop render_pre setup_project
                  screenHeight = rect.bottom;
                  resize_GL(rect.right, rect.bottom);
                  setScreenDim(rect.right,rect.bottom);
                  break;
  ```

Mac

Where is this ? In the front-end obviously. Then the front-end pass events to the library through the function `handle_aqua()`.

- How to improve
  Create a unique definition for events of all sorts. Let's call it `fwl_event_t` for the sake of simplicity.

  Create a function to translate an `<whatever platform>` event into a `fwl_event_t`.

  Create a function to eat an event. I.e. to act on the event.

  Create a system of callbacks.

  Then, two cases are possible:

  - the library runs the event loop (it has created the *context*). Default callbacks are programmed to be called for each interresting events. The list of default callbacks defines a *viewer behavior* or an *actor*. I.e. *walk* and *fly* are two different *actors*.
  - the front-end is the master (it initialized the *context* and controls the event loop). It programs callbacks for all the events it wants to eat itself. Then pass the event to the library. Which translates the event, and then to act upon it. The callbacks are called by the library *actor* active at this time.

1.2.5 OpenGL binding

A *great job* was accomplished when a lot of OpenGL calls where replaced by macros. That way a compile time selection of the rendering capabilities is possible. If this is possible, the job has to be finished: i.e. for all OpenGL calls.

Basically we can see OpenGL-ES as a subset of OpenGL. But for FreeWRL to work correctly OpenGL-ES requires the use of `VBO`.

1.2.6 An unique function to get/set libFreeWRL variables

- Purpose

  - all variables visible from outside the library that need an accessor (to trap changes, update other variables, ...)
  - all variables exposed to scripting language or external processes (SAI/EAI) that need an accessor (check type, check value, ...)

- Variable list

  - window (type is platform specific)
  - OpenGL window (type is platform specific)
  - GLX/AGL/WGL context (type is platform specific)
  - width
  - height
  - fullscreen (switch)
  - eai (switch)
  - verbose (switch)
  - collision (switch)
  - starting url (world base)

  - viewer mode
  - snap (switch)
  - snap directory
  - snap file pattern
  - EAI verbose (switch)
  - screen dist
  - stereo (switch)
  - shutter glass (switch)
  - eye dist
  - anaglyph
  - side by side (switch)
  - stereo buffer mode (switch)
  - line width
  - snap gif (not used?)
  - print shot (can't understand this?)
  - browser full path (need rework)
  - pipe and all plugin related variables (I rewriting this)
  - keypress$_{\mathrm{string}}$ (need rework)
  - ask for refresh, refresh switch or status (need rework)

- Implementation

  - a new include file could be created with the list of variables (it will be included in libFreeWRL.h when this step is finished)
  - a new source file could be created with the unique accessor function