Development notes of the Guile-SFML bindings

Eduardo Acuña Yeomans

April 2014

About

This is a development documentation, transcribed from the notes I write for the planning and programming related to this proyect. The reader may encounter ideas, struggles, and references related to the development of the bindings.

This notes can be useful for me in the future and to other people that want to use a C library from GNU Guile. However my first language isn't english so any suggestions or corrections can be mailed to `eduardo.acye@gmail.com`.

This document is being written as the bindings are coded, it is supposed to be read in order (it is not a reference manual for the bindings).

Development plan

The goal of this proyect is constructing a clean and simple library for graphics and audio interactive programming.

I have C++ background but currently getting more involved with the Lisp family of languages, in school I had to write a simple game in C++ and chose the SFML library to handle the graphics, input and audio. I liked it so much that i wanted to use the library from a Scheme environment. The first thing i tried was to write the bindings using the static FFI using the Guile manual as a starting point, but it was a lot of not fun C and C++ programming (I have suspended the development of this bindings but the things i got to work live in my `https://github.com/eduardoacye/Guile-SFML-1` repository).

Now the intention is to write the bindings using the Dynamic FFI. The process for getting the proyect done is divided into three steps:

1. Mapping the CSFML structures and functions directly to Guile with the Dynamic FFI.
2. Wrapping the previous bindings to create modules that use GOOPS and Scheme idioms to work with the SFML implementation.
3. Expanding the previous modules with useful functions, macros and classes that are not part of SFML.

There isn't much documentation in the official Guile Reference Manual about writting bindings with the Dynamic Foreign Function Interface, however there are a couple of proyects of bindings for Guile using this method that I think are very helpful, in particular Andy Wingo's guile-figl repository and David Thompson guile-allegro5 and guile-2d repositories.

I recomend you check those out:

- guile-figl at `https://gitorious.org/guile-figl`

- guile-allegro5 at `https://github.com/davexunit/guile-allegro5`
- guile-2d at `https://github.com/davexunit/guile-2d`

1. Raw bindings from CSFML

This is the first step of the development process. And it consist of getting all the SFML structures and functions working from Guile. I'm using the C bindings as a starting point. To start working i had to compile Guile (currently using the 2.0.9 version), SFML 2.1 and CSFML 2.1.

The structure of the bindings is very similar to the directory structure of the CSFML source code. There is one module for each file in CSFML, however there might be some files missing that provide C++ and C functionalities that Guile already have (such as multithreading).

guile-sfml directory tree

- **sfml2**
  - **audio**
    - `todo`
  - **graphics**
    - blend-mode-unsafe.scm
    - circle-shape-unsafe.scm
    - color-unsafe.scm
    - common.scm
    - convex-shape-unsafe.scm
    - font-unsafe.scm
    - glyph-unsafe.scm
    - image-unsafe.scm
    - primitive-type-unsafe.scm
    - rect-unsafe.scm
    - rectangle-shape-unsafe.scm
    - render-states-unsafe.scm
    - render-texture-unsafe.scm
    - render-window-unsafe.scm
    - shader-unsafe.scm
    - shape-unsafe.scm
    - sprite-unsafe.scm
    - text-unsafe.scm
    - texture-unsafe.scm
    - transform-unsafe.scm
    - transformable-unsafe.scm
    - types-unsafe.scm
    - vertex-array-unsafe.scm
    - vertex-unsafe.scm
    - view-unsafe.scm
  - graphics-unsafe.scm

- **network**
  - `todo`
- **system**
  - clock-unsafe.scm
  - common.scm
  - input-stream-unsafe.scm
  - sleep-unsafe.scm
  - time-unsafe.scm
  - vector2-unsafe.scm
  - vector3-unsafe.scm
- system-unsafe.scm
- utils.scm
- **window**
  - common.scm
  - context-unsafe.scm
  - event-unsafe.scm
  - joystick-unsafe.scm
  - keyboard-unsafe.scm
  - mouse-unsafe.scm
  - video-mode-unsafe.scm
  - window-handle-unsafe.scm
  - window-unsafe.scm
- window-unsafe.scm

Common code

The code that has to be aviable from every sub-library, like auxiliary procedures and useful macros are in the `sfml2/utils.scm` file. The code that has to be aviable from every module in a specific sub-library, like the macro for defining a foreign function of that sub-library is in the `sfml2/sub-lib/common.scm` file, where `sub-lib` can be: `audio`, `graphics`, `network`, `system` or `window`.

1.1. SFML/System

The SFML/System library seems to be very simple, the bindings exist for a subset of the SFML code. I opted for excluding the multithreading part. I did this because Guile can provide this features and in the SFML tutorials, the author writes:

> In its newest version (2011), the C++ standard library provides a set of classes for threading. At the time SFML was written, the C++11 standard was not written and there was no standard way of creating threads. When SFML 2.0 was released, there were still a lot of compilers that didn't support this new standard.
>
> If you work with compilers that support the new standard and its `<thread>` header, forget about SFML classes and use it – it will be much better. But if you work with a pre-2011 compiler, or plan to distribute your code and want it to be fully portable, then SFML threading classes are a good solution.

I hope that this desition doesn't break the whole proyect, if I notice that they are essential for the bindings to work, this parts will be written.

Understanding the dynamic foreign function interface

There is very little documentation in the Guile Reference Manual regarding the Dynamic Foreign Function Interface. In the guile-figl, guile-allegro5 and guile-2d code, there are some cool macros to facilitate the implementation of C features (like `define-enumeration` and `define-foreign`).

I don't have the ability to design and implement good macros, however i'll try to explain in this section of the document how i understand this FFI works.

Note: I have not yet worked with the *bytevectors* described in the Guile Manual.

There are two important things that make up a C library: The structures that represent the data and the functions that manipulate the data. The library i'm wrapping is called CSFML that is itself a binding in C for the SFML library (that is written in C++).

The way CSFML is written has two types of structures: the simple compound data in SFML is a C struct (usualy the ones that are composed from standard C types like the 3-dimentional vector that just has 3 float numbers in the struct); the more complicated classes are handled as opaque data types, this are structs without data that are used with pointers to the C++ class that has the data, the functions that are aviable from C are mixed with C++ to perform operations with the opaque pointer, so the C part of the program doesn't know what the C++ part does, it just glue code.

The Guile module we have for working with the FFI is `(system foreign)`. To have access to the functions from the library we have to use the procedure `dynamic-link` to get a linked Scheme object (we'll use this when binding functions).

I'll call the first type of structures *simple data* and the second type of structures *opaque data*. The simple data is going to be wrapped in Guile with `make-c-struct` (read 6.20.5.4 Foreign Structs from the Guile Manual) and the opaque data is going to be wrapped in Guile with `define-wrapped-pointer-type` (read 6.20.5.3 Void Pointers and Byte Access from the Guile Manual).

The C functions will be bind using a macro that calls the `pointer->procedure` with the corresponding library (read 6.20.6 Dynamic FFI from the Guile Manual).

There are enums and unions too, but i have not decided how i'll handle those constructs.

In the following two sections I illustrate an example for wrapping a C structure named cucus that has some functions that manipulate the structure, then i'll proceed to handle pointers to structures and pointers to pointers to structures.

```
struct cucus
{
    int a;
    float b;
    char c;
};

struct cucus y = {666, -.01, 'f'};
struct cucus *ptr_y = &y;

void take_cucus(struct cucus x)
{
    printf("a  ==  %d\n", x.a);
    printf("b  ==  %f\n", x.b);
    printf("c  ==  %c\n", x.c);
}
void take_cucus_ptr(struct cucus * x)
```

```
{
    if(x == 0){
        printf("the pointer is NULL\n");
    }else{
        printf("a  ==  %d\n", (*x).a);
        printf("b  ==  %f\n", (*x).b);
        printf("c  ==  %c\n", (*x).c);
    }
}
void take_cucus_ptr_ptr(struct cucus ** x)
{
    if(x == 0){
        printf("the pointer is NULL\n");
    }else if(*x == 0){
        printf("dereferenced pointer is NULL\n");
    }else{
        printf("a  ==  %d\n", (*(*x)).a);
        printf("b  ==  %f\n", (*(*x)).b);
        printf("c  ==  %c\n", (*(*x)).c);
    }
}
struct cucus give_cucus(void)
{
    struct cucus x = {0, 3.1416, 'z'};
    return x;
}
struct cucus * give_cucus_ptr(void)
{
    return &y;
}
struct cucus ** give_cucus_ptr_ptr(void)
{
    return &(ptr_y);
}
```

Binding the structures

There is one list and two functions that are defined for each simple data structure:

- List of C types
- Procedure structure to pointer
- Procedure pointer to structure

This simple data structures are represented at the scheme level as lists, but in the C side are normal structs. Assuming we have made a dynamic link with the shared library that has the struct and functions definition above we define in scheme:

```
;;; List of C types
(define cucus-type (list int float int8))

;;; Make a C struct from a list of Scheme values
```

```
(define (cucus->pointer lst-struct)
    (make-c-struct cucus-type lst-struct))

;;; Return a list of Scheme values from a C struct
(define (pointer->cucus c-struct)
    (parse-c-struct c-struct cucus-type))
```

It is important to note that the `(system foreign)` module doesn't export a `char` type, however the size of the char is 1 byte and we represent it as an integer (the `int8` from the code above is the `char`).

We can handle the cucus C struct as opaque data with the `define-wrapped-pointer-type` procedure, a simple definition for this is:

```
;;; Scheme record for representing an opaque C cucus struct
(define-wrapped-pointer-type cucus
    cucus?                            ; predicate
    wrap-cucus                        ; pointer to record
    unwrap-cucus                      ; record to pointer
    (lambda (c port)                  ; printer
        (display "<cucus " port)
        (display (cucus-parse (unwrap-cucus c)) port)
        (display ">" port)))
```

Binding the functions

The functions are defined with `pointer->procedure`, and each function in the library must have an equivalent procedure in Guile. It is necessary to know the name of the C function (we use a Scheme string to represent it) and have aviable the library link as a Scheme object (In this example we use `(define libcucus (dynamic-link ".../libcucus"))` so the name of this object is `libcucus`).

```
;;; Binding for the take_cucus C function in libcucus
(define take-cucus
    (pointer->procedure void
                        (dynamic-func "take_cucus" libcucus)
                        (list cucus-type)))

;;; Binding for the take_cucus_ptr C function in libcucus
(define take-cucus-ptr
    (pointer->procedure void
                        (dynamic-func "take_cucus_ptr" libcucus)
                        (list '*)))

;;; Binding for the take_cucus_ptr_ptr C function in libcucus
(define take-cucus-ptr
    (pointer->procedure void
                        (dynamic-func "take_cucus_ptr" libcucus)
                        (list '*)))

;;; Binding for the give_cucus C function in libcucus
(define give-cucus
    (pointer->procedure cucus-type
                        (dynamic-func "give_cucus" libcucus)
                        (list)))

;;; Binding for the give_cucus_ptr C function in libcucus
(define give-cucus-ptr
    (pointer->procedure '*
                        (dynamic-func "give_cucus_ptr" libcucus)
                        (list)))

;;; Binding for the give_cucus_ptr_ptr C function in libcucus
(define give-cucus-ptr-ptr
    (pointer->procedure '*
                        (dynamic-func "give_cucus_ptr_ptr" libcucus)
                        (list)))
```

With these definitions we can manipulate the cucus C struct and call the functions that manipulate the structure.

I use a macro for binding the C functions, I took it from David Thompson (`guile-2d/2d/wrappers/ftgl.scm` from the github repository) with a little modification to take a symbol for the C function name instead of a string (just to make it look more like the function declaration):

```
(define-syntax define-ff
    (syntax-rules ()
        ((define-ff name ret c-name args)
         (define name (pointer->procedure ret
                                          (dynamic-func (symbol->string 'c-name) lib)
                                          args)))))
```

Unsafe version of the System sub-library

The first thing I did was determine all the structures that needed to be binded, then classify them by simple data or opaque data. The next step is knowing what C functions are involved and then listing the order they had to be implemented.

- file: vector2-unsafe.scm (simple data)
- file: vector3-unsafe.scm (simple data)
- file: time-unsafe.scm (simple data)
  - file: clock-unsafe.scm (opaque data)
  - file: sleep-unsafe.scm (no data)

There are another two files not mentioned in this order: `common.scm` (in the `sfml2/system/` folder) and `system-unsafe.scm` (in the `sfml2/` folder). The first of this files contains the macro `define-ff` with the corresponding link to the `libcsfml-system` library and the other file is just going to export all of the symbols exported from every module in this sub-library.

Note: The word *unsafe* is present in the names of the files because the procedures and data defined handle C pointers and programming with this modules is considered dangerous. In the second step of the project a wrapper for this modules will be written so that everything is in scheme land.

Naming conventions: I use only lower-case; prefixed `"sf-"` to the type name in the list of C types in the structure definition, `pointer->type`, `type->pointer` procedures, opaque data name (like `<sf-type>`), type predicate, `wrap` and `unwrap` procedures; the function bindings of the CSFML library are prefixed with `"%sf-"` for identifying as unsafe procedures.

1.2. SFML/Window

The window sub-library has more structures and functions than the system sub-library but the process of implementing the bindings is the same. There are two struggles I encounter reviewing the code: First, the CSFML library uses unions to represent the event data (a union of structs) so there is going to be an implementation of unions in Guile or a workaround this issue; the second struggle is that a file (`WindowHandle.h`) contains platform specific things (so i have to program this part of the bindings differently for UNIX, Windows and Mac), my intentions are getting the bindings to work on Linux, but I'll try to make them work for the other platforms as well.

Unsafe version of the Window sub-library

Here is the list of the Scheme files in order of implementation and the structure types:

- file: context-unsafe.scm (opaque data)
- file: joystick-unsafe.scm (enums)
- file: keyboard-unsafe.scm (enums)
- file: video-mode-unsafe.scm (simple data)
- file: window-handle-unsafe.scm (no data)
- file: event-unsafe.scm (enums, simple data and union)
- file: window-unsafe.scm (enum, simple data and opaque data)
- file: mouse-unsafe.scm (enum)

Enums implementation

The enum binding is implemented with `define`. Some functions of the CSFML library have an enum type as an argument, so each enum is going to be defined with the type `int` and each value of the enum is going to be associated with a numeric value.

For example, in the file `/include/SFML/Window/Mouse.h` of the CSFML repository, the author describes the sfMouseButton enum:

```
typedef enum
{
  sfMouseLeft, ///< The left mouse button
  sfMouseRight, ///< The right mouse button
  sfMouseMiddle, ///< The middle (wheel) mouse button
  sfMouseXButton1, ///< The first extra mouse button
  sfMouseXButton2, ///< The second extra mouse button

  sfMouseButtonCount ///< Keep last -- the total number of mouse buttons
} sfMouseButton;
```

The translation to Scheme I picked is a direct definition of the symbols with the value set manually:

```
;;; enum binding for sfMouseButton
(define sf-mouse-button int)

(define sf-mouse-left         0)
(define sf-mouse-right        1)
```

```
(define sf-mouse-middle       2)
(define sf-mouse-x-button-1   3)
(define sf-mouse-x-button-2   4)
(define sf-mouse-button-count 5)
```

The `sf-mouse-button` definition is not necessary but the foreign function definitions are cleaner.
For example if we consider the C function:

```
CSFML_WINDOW_API sfBool sfMouse_isButtonPressed(sfMouseButton button);
```

Instead of writting:

```
(define-ff %sf-mouse-is-button-pressed
  sf-bool sfMouse_isButtonPressed (list int))
```

We write:

```
(define-ff %sf-mouse-is-button-pressed
  sf-bool sfMouse_isButtonPressed (list sf-mouse-button))
```

This way we know that the function accepts a value from the sfMouseButton enum.

Union implementation

As of this date, the SFML binding only uses a union type in one occasion (in event.h). Currently the `(system foreign)` module doesnt give a way of defining unions, but I wrote a simple (and not efficient) way of using them.

The C union type is a heterogeneous data type, similar to the C `struct`, with the difference that a union instance can just contain one of the values at the same time. To explain myself better, here is an example comparing the two:

```
// size of the types in my computer
typedef struct
{
  char a;        // 1 byte
  int b;         // 4 bytes
  double c;      // 8 bytes
} cucus_struct;

typedef union
{
  char a;        // 1 byte
  int b;         // 4 bytes
  double c;      // 8 bytes
} cucus_union;
```

The cucus_struct type contains 13 bytes of data, while the cucus_union type contains only 8 bytes of data. The union has the size of the biggest element, and the other types are aligned with offset 0 in the structure. Unions are usually used when we know that only one of the values will be used.

The way I use the union type in the bindings is the following:

```
;;; union binding for the sf-event union of structures
(define sf-event (list sf-event-type
                       sf-size-event
                       sf-key-event
                       sf-text-event
                       sf-mouse-move-event
                       sf-mouse-button-event
                       sf-mouse-wheel-event
                       sf-joystick-move-event
                       sf-joystick-button-event
                       sf-joystick-connect-event))

(define (pointer->sf-event c-union type)
  (parse-c-union sf-event c-union type))

(define (sf-event->pointer type lst-union)
  (make-c-union sf-event type lst-union))

;;; Alias for the sfEvent*
(define sf-event-* '*)
```

I wanted to make the union definition similar to the way I define structures with the `make-c-struct` and `parse-c-struct` procedures, so I defined two new procedures: `make-c-union` and `parse-c-union`. The idea is to have a list of the types inside the union (in the example `sf-event`), this list can contain a type like `int` (the type of `sf-event-type`) or a list of types representing a structure, like `(list int uint32)` (the type of `sf-text-event`) (A union of unions can not be implemented currently (I hope to provide a fix for making the union type similar to the C union type)).

Here is the implementation of the two procedures (they are written in the file `sfml2/utils.scm`):

```
(define (make-c-union all-types type vals)
  (define bv (make-bytevector (apply max (map sizeof all-types)) 0))
  (define ptr (make-c-struct (if (pair? type) type (list type))
     (if (pair? vals) vals (list vals))))
  (define bv2 (pointer->bytevector ptr (sizeof type)))
  (unless (member type all-types)
    (error "type not in union" type all-types))
  (bytevector-copy! bv2 0 bv 0 (bytevector-length bv2))
  (bytevector->pointer bv))

(define (parse-c-union all-types union type)
  (unless (member type all-types)
    (error "type not in union" type all-types))
  (parse-c-struct union (if (pair? type) type (list type))))
```

1.3. SFML/Graphics

The implementation of this sub-library was more tedious but staightforward than the Window sub-library (A lot of things to write that were practically the same).

Unsafe version of the Graphics sub-library

The order of implementation was divided in 4 sections:

1. PrimitiveType, Rect, Types, Color, BlendMode
2. Glyph, Image, Texture, Transform, Vertex, View
3. CircleShape, ConvexShape, Font, RectangleShape, RenderStates, Shader, Shape, Sprite, Text, Transformable, VertexArray
4. RenderWindow, RenderTexture

Function pointers binding

The only thing I have not checked nor implemented correctly are the function pointer bindings, all the functions that have as arguments a function pointer I defined with the `'*` type. The problem is that I don't even know how they are used in the SFML library, so this will be revised later.

1.4. SFML/Audio

1.5. SFML/Network

1.6. Testing bindings: SFML 2.1 tutorials

Basic application

Graphics tutorial

Autio tutorial

2. SFML adapted for Guile

2.1. SFML/System

2.2. SFML/Window

2.3. SFML/Graphics

2.4. SFML/Audio

2.5. SFML/Network

2.6. Testing library: Programs constructed with SFML

CodeFrame port

Keyboard piano

3. Automatón