kiss_sdl - Universal low level widget toolkit for SDL2

1. Overview

kiss_sdl is a universal low level widget toolkit for SDL2. Low level may not sound good, but this is exactly what some developers are looking for, as it provides freedom to do whatever one needs. And it is simple and easy to use. At the time of writing this, there were only 2357 lines of code in the whole toolkit together with the header file, kiss_general.c, kiss_posix.c, and the examples, and only 1068 lines of code in kiss_widgets.c and kiss_draw.c, where all the functionality of the toolkit is implemented. At the time of writing this, there were knowingly no other widget toolkits that anyone could make to work with SDL2 (SDL2 is not compatible with SDL1), but even if there were other universal ones, then they are likely much more complex and not so easily changeable as this one. When one writes a graphical user interface for one's game or application, using this toolkit, then it may feel like writing one's own widgets directly in SDL. It is so thin layer, there is not much in between, and thus also not much that restricts one from doing whatever one may want to do. With that widget toolkit it thus is that, SDL is high level, and widget toolkit is low level. SDL (Simple DirectMedia Layer) is also a low level directmedia layer, thus with SDL, this widget toolkit provides a low level framework for writing Graphical User Interfaces (GUI), for games and applications.

KISS is an acronym and a software development approach that means "Keep it short and simple". Different from other widget toolkits where one is supposed to choose from the existing widgets the one that fits one the best, in this toolkit, when there is no widget that satisfies all your needs, modify an existing widget, or write your own. Most other toolkits enable to write your own widgets, but it is not simple to do that, and they are not intended for that. The toolkit was made as simple as possible, to solve the problem of an immense complexity of most of the existing toolkits, which makes it impossible for many people to even use them. At that it was made universal, many simpler toolkits don't enable you to do many things that you want, due to their restricted internal design. This toolkit implements an original innovative approach for a widget toolkit, this new approach enabled to make the widget toolkit simpler. The toolkit is written in C, but it can also be used in C++.

This widget toolkit is made as modifiable as possible, so that it will not stand on the way of doing what the user may want to do. The code of this widget toolkit is split into four files. The file kiss_widgets.c contains only the functions of the widgets, and none of the SDL functions or external library functions are directly called in that file. The three other files serve as abstraction layers, and rewriting the functions in some of these files, enables to redefine how the operations that use the external library functions, are implemented. These three files are kiss_draw.c, for general drawing functions, kiss_general.c, for general functions such as an implementation of the variable size array, and kiss_posix.c, for some of the most necessary POSIX functions. In case of an error, the functions of this toolkit return -1 or NULL, depending on the return type, and the event processing funtions return 0.

Ten widgets are implemented as base widgets, these are a window, multiline label, button, select button, vertical scrollbar, horizontal scrollbar, progress bar, entry box, textbox, and combo box. The two examples show how to use these widgets together. The first example implements a working file selection dialog with scrollable text boxes, the second example shows a use of a combo box. Likely most of the user interfaces can be made by using only these widgets, but the toolkit doesn't prevent adding any additional functionality.

There are four ways to create one's own custom widget, different from the base widgets. First is to create a composite widget. The widgets have coordinates relative to the main window, but it is possible and not so difficult, to add the coordinates of a local window to a certain group of widgets. It is possible to create and free such group of widgets in one function, or process its events or draw it in one function. Combo box is an example of a composite widget. Second is a user added functionality. Write your own event function and call the base function from that. Then functionality can be added in case when the base function returns nonzero, but also new reactions to mouse and keyboard events can be added. This has to be done the in same way as it is done in the base widget event functions though, like at least it should be first checked whether the widget is visible. Third, the functionality can also be added in case if the event is a NULL pointer. Like for games it may be important to add time dependent effects to widgets, for example by making them to continuously change shape, rather than being static. A progress bar is an example of this. And fourth is to write your own widget, by copying the structure and functions of a base widget, and modifying them.

Is it possible to do the event processing in a separate function? Yes it is, by using separate event loops for different widgets, or by passing objects using a variable size array, but because of simplicity this has not been done in the examples of this toolkit. This also makes a re-entrant or thread-safe code possible, and also different widgets can then be drawn in different threads. In such widget toolkit there is essentially nothing central, except in that case a table may be necessary, that shows for every thread whether it has received an expose event and whether it has processed it.

Can the images be compiled into the code, instead of being in separate files? Yes they can, images can be compiled into the C code, also Gimp enables to save them directly in that way. Then with a special function they can be retrieved from the data in the code. But again because of simplicity, this has not been done in the examples of this toolkit.

Can i use other colors, textures, etc? Yes. To use certain other fonts or textures, you may have to rewrite the init function. You can also draw anything you like on the widgets in the drawing function, this will be automatically redrawn every time after the window is exposed.

Can i do this or that other thing? You can do whatever you want, different from some other widget toolkits, this toolkit has not been made to restrict your freedom, or dictating to you how you should do things or what you can do. Just learn the basic things about how it works, the toolkit is simple and thus not difficult to learn, but it would be difficult to do what you want otherwise.

kiss_sdl is fully capable of UTF-8 (8-bit Unicode). The rendered text, the keyboard input and also the C source code, all is in UTF-8.

2. Hello World

There should be some Hello World for everything. The code below creates a simple message box, with only a label and a button. The code is below so that you can see it, you may however not be able to copy it, because pdf does not really contain text, and copying text is thus not always accurate.

To compile this, just create a new directory, copy all files from the kiss_sdl distribution to there, create a file hello.c , and write the following code in it. Then like in Linux, copy kiss_makefile to makefile, and there simply replace kiss_example1 everywhere with hello. Then open terminal in that directory, write make, and then ./hello to run it.

```
#include "kiss_sdl.h"

void button_event(kiss_button *button, SDL_Event *e, int *draw,
     int *quit)
{
     if (kiss_button_event(button, e, draw)) *quit = 1;
}

int main(int argc, char **argv)
{
     SDL_Renderer *renderer;
     SDL_Event e;
     kiss_array objects;
     kiss_window window;
     kiss_label label;
     kiss_button button;
     char message[KISS_MAX_LENGTH];
     int draw, quit;

     quit = 0;
     draw = 1;
     kiss_array_new(&objects);
     renderer = kiss_init("Hello kiss_sdl", &objects, 320, 120);
     kiss_window_new(&window, NULL, 0, 0, 0, kiss_screen_width,
          kiss_screen_height);
     strcpy(message, "Hello World!");
     kiss_label_new(&label, &window, message,
          window.rect.w / 2 - strlen(message) *
          kiss_textfont.advance / 2,
          window.rect.h / 2 - (kiss_textfont.fontheight +
          2 * kiss_normal.h) / 2);
     label.textcolor.r = 255;
     kiss_button_new(&button, &window, "OK",
          window.rect.w / 2 - kiss_normal.w / 2, label.rect.y +
          kiss_textfont.fontheight + kiss_normal.h);
     window.visible = 1;
     while (!quit) {
          SDL_Delay(10);
          while (SDL_PollEvent(&e)) {
               if (e.type == SDL_QUIT) quit = 1;
               kiss_window_event(&window, &e, &draw);
               button_event(&button, &e, &draw, &quit);
          }
          if (!draw) continue;
          SDL_RenderClear(renderer);
          kiss_window_draw(&window, renderer);
          kiss_label_draw(&label, renderer);
          kiss_button_draw(&button, renderer);
          SDL_RenderPresent(renderer);
          draw = 0;
     }
     kiss_clean(&objects);
     return 0;
}
```

All that needs to be included, is kiss_sdl.h.

First, the base event function for button is overwritten by the user, with the function button_event(). The reason for overwriting that function, is to make the program to quit, when pressing the OK button. For overwriting the event function, the base event function is called in the if condition with the same arguments as the overwritten function, except quit, and the if statement body is *quit = 1 , which ends the main loop. Notice the pointers, a variable, when the function may change it, is passed as a pointer. draw and quit are local variables defined in main(), which means that they are passed to the event functions with a reference operator, as &draw and &quit.

The main() function header has to be int main(int argc, char **argv) , i don't even remember why it cannot be int main(void) , but this is how it works.

Next all the necessary variables are defined, there is not much to explain, this is what needs to be defined. As you see, we also define three widgets, window, label and button.

We initialize quit and draw, by assigning 0 to quit and 1 to draw. Assigning 1 to draw makes sure that everything is drawn at the beginning. Next we create the array object, which is necessary to store references to allocated objects, but we don't have to do anything else with it, than just to pass it to kiss_init() and kiss_clean().

kiss_init() creates the SDL main window, and returns the renderer. There is not much about it, the window's width and height have to be passed to it as its last arguments. Next we create the window widget, this would be like a dialog window on which are all our widgets. We pass the window widget structure to it, then NULL, because it is the bottom window, and is not on any other window. The next argument is decorate, it is nonzero in the kiss_sdl examples, but this time we pass 0 to it, to show that the widgets borders can not be decorated. The last 4 arguments are x, y, width and height of the widget. Because we cover all the surface of the SDL window with it, these are 0, 0, and kiss_screen_width and kiss_screen_height, which are the width and height of the SDL window.

Next we copy the text of the message to the char array named message, and create a label with window as its dialog window, and the message.

The low level widget toolkit differs from the advanced widget toolkits such as GTK, in that all the coordinates, widths and heights of the widgets, have to be calculated manually. Also all the coordinates are SDL window coordinates, so when the coordinates have to be relative to the dialog window, the dialog window coordinates have to be added to the widget's coordinates, which is not difficult to do though.

Our dialog window's width is window.rect.w , the width of our label is strlen(message) * kiss_textfont.advance . The member advance in the font structure, is the horizontal distance from the beginning of a character, to the beginning of the next character, in pixels. All coordinates, widths and heights are in pixels. Now to center the widget in the dialog window, its x coordinate has to be window x + window width / 2 - widget width / 2, because it has to be half the widget width less than half the window width. In our

case the window x is 0, so the x coordinate of the label is window.rect.w / 2 - strlen(message) * kiss_textfont.advance / 2.

We want that the label with the button below it, were also verically at the center of the window. We would have one button height between the label and the button, this is a good distance between them. In that case, the total height of all our widgets is kiss_textfont.fontheight + 2 * kiss_normal.h . fontheight is the height of the text font, this should not be confused with the lineheight, which is the vertical distance from the beginning of a line, to the beginning of the next line, which is not the same as fontheight. Thus the y coordinate of the label is window.rect.h / 2 – (kiss_textfont.fontheight + 2 * kiss_normal.h) / 2 , and everything is perfectly centered. kiss_normal is the image structure of the normal state of the button, for every loaded image there is such structure, and all these structures have members h and w, which are the width and height of the image

We make it a bit nicer, and make the message red. label.textcolor.r is the value of the red component of the label's text color. As the color by default is black, then assigning 255 to that, makes the text red. This is how the features of a widget can be changed later after creating it, by changing the values of its structure.

Next we create a button, with the dialog window window, and the text "OK" on it. The x coordinate of the button is calculated in the same way as the x coordinate of the label, as it has to be horizontally centered, thus it is window.rect.w / 2 – kiss_normal.w / 2 . We calculate the y coordinate of the button relative to the y coordinate of the label, thus it is label.rect.y + kiss_textfont.fontheight + kiss_normal.h .

Next we do that magical thing, window.visible = 1 . This makes visible that window, and all the widgets on it.

Next there is the typical main loop, while quit is zero. It starts with SDL_Delay(10) , which gives 10 milliseconds time in every iteration, for the operating system and other programs. Next is the event loop.

Every iteration of the event loop gives a different event e, if there are any events. First we check whether the event is SDL_QUIT, which means that the user closed the window, by clicking on the x at the upper left corner of the window, using a keyboard shortcut or by other ways of closing it. In that case we assign nonzero to quit, to end the main loop, and quit the program.

Next we call the event functions of the window and the button. We should formally call the window event function, just to process the expose event, to find when the window and everything on it has to be redrawn. Though it works without it, as calling the button's event function does the same. Notice that we called the base window event function, but the overwritten button event function, which also has the argument quit.

This is how it is done in a low level widget toolkit, things have to be done manually, like the event functions have to be added to the event loop manually. Which is not difficult though, and at that the widget toolkit is much simpler, and there is much less to learn. But no glorious one function to do all the main loop, and event loop, nothing like that.

And when there is that glorious function, then any thinking person certainly asks, how can one do other things while the widgets are active, or how to add some event processing. And this would not be that simple any more. Abstraction is for hiding details for making certain tasks simpler, but abstraction that always hides everything, is not good.

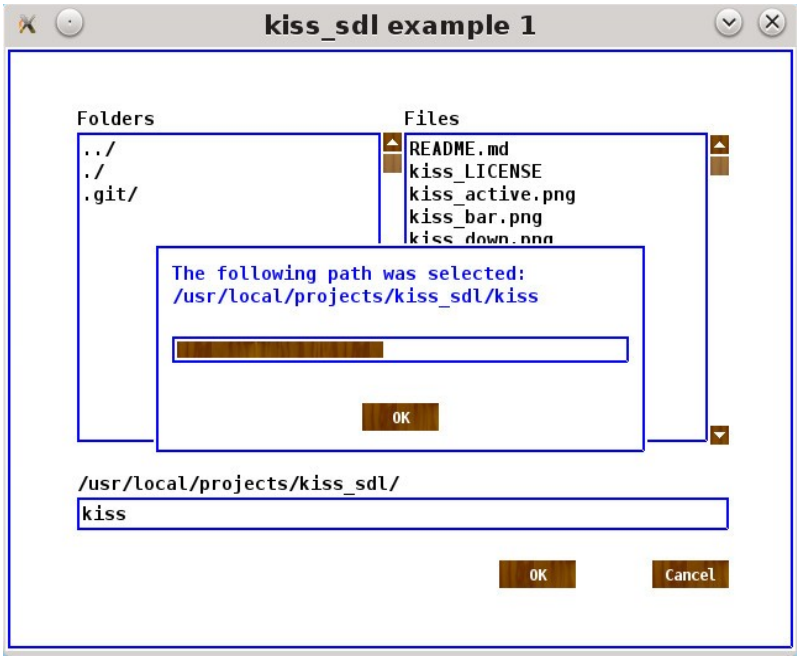
Next in the main loop is the drawing, which we do only when draw is nonzero. Draw is assigned zero again in the end of the drawing, so that something somewhere has to assign nonzero to it again, to draw again. The drawing is a complete redraw of the SDL window, drawing a single frame, so to say. It starts with clearing the renderer, and ends with presenting the renderer, the base SDL functions. The base draw functions of all the three widgets are called in between these, there is not more to it.

After the end of the main loop, kiss_clean() is called, with the reference to the array objects passed to it, this frees all the allocated objects, and quits SDL and all its subsystems. Finally, return 0 has to be there by standard.

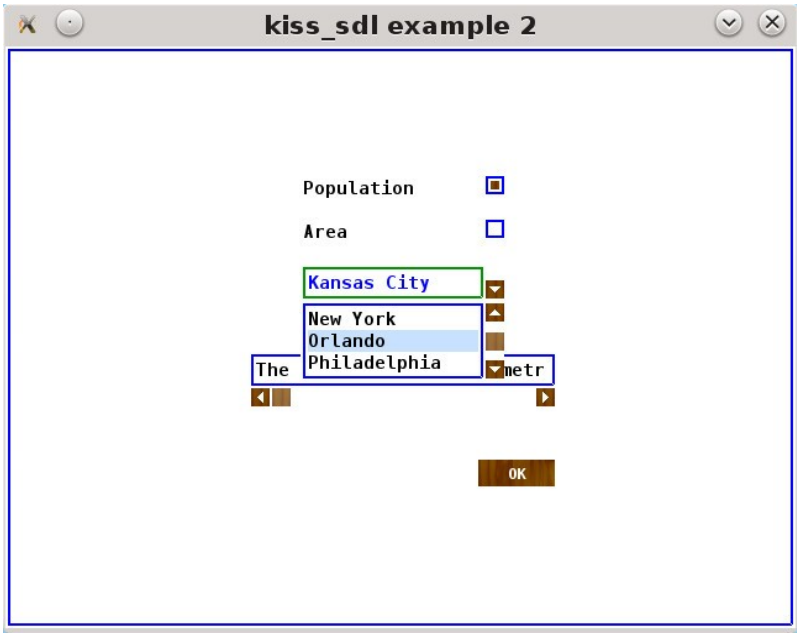
The widgets don't do many automagical things in this simple dialog, only the button becomes lighter when the mouse is on it, and darker when clicked. I simply took a pattern of wood, and cut a proper size piece out of it, for a button. Then i made 3 images out of it, first is with the normal brightness, second is with the brightness increased, and third is with the brightness decreased. But these can be any images, like one may make them 3D, having 3D buttons.

Hope that this gave everyone an idea of how a low level widget toolkit differs from advanced widget toolkits, such as GTK.

3. Example 1



4. Example 2



Only kiss_sdl.h needs to be included. First there is an arrray of structures, initialized with data about the bigger cities in USA.

Next are the user written event functions. In the first two functions, the other of the select buttons is toggled, when a select button is toggled by clicking on it. This gives an effect like radio buttons. One can see that the base function is called in the if condition, and the reaction is when that base function returns nonzero.

Next there is a user written event function for combo box, it provides a reaction when the combo box closes, the chosen text is then in the combobox->entry.text. This is searched in the combo box array (combobox->textbox.array), and its index is calculated. Then depending on the state of the select1 widget (the member of its structure select1->selected) one of the two texts is written to a string, with the data from the cities array, at the previously found index. Next the string is copied to the entry widget (entry->text), limiting it to the width of the entry box by '\0'. Also the scrollbar is set to 0, so that it is at the beginning and doesn't move. And the variable draw is assigned nonzero as always when anything is changed that needs to be drawn.

Next there is a user written scrollbar event function, which is a reaction when the scrollbar's slider moves. First the index of the first character in the string is calculated, by subtracting from the length of the string the number of characters that are visibe, and this number is the maximum number of characters in the entry box, multiplied by the scrollbar's fraction (hscrollbar->fraction). Always 0.5 has to be added when rounding a float expression by casting to int. Then the text starting from that index is copied to the entry box, so that its length is limited by the width.

The user written button function only assigns nonzero to the variable quit, to quit the application.

Everything necessary is defined at the beginning of the main() function, including structures for all widgets. After other initializations, what is important is initializing the variable size array a. The structure of the array a is defined in the stack, the same as the structures of the widgets. A new array shall be created with that structure, and then the names of the cities from the cities array are appended to that array. Later this array is passed as an argument to the function that creates a combo box.

Next all the widgets are created. The widget structures are passed to the new functions as first arguments. The window is the dialog window for all the widgets there, thus its second argument is NULL, the second argument for all other widgets is a reference to that window. The third argument in several widgets is whether to decorate the widgets, which in this example is every time. The coordinates, widths and heights of the widgets are calculated, this may look complicated at first, but it's straightforward.

It is important to assign zero to the hscrollbar.step too, because at first the entry box is empty, and thus the scrollbar should not move.

Next, assigning nonzero to the window.visible does that magic trick that makes all the widgets visible at once, because this window is set as a dialog window for all our widgets.

Next there is the main loop while !quit as usual for such applications, the additional code there can be the implementation of a game or application. It is good to have a delay, it is better to be 5 ms or more, because 5 ms used to be the time period of the operating system's scheduler, this gives some more time for the operating system and other programs.

Next there is the event loop, which goes through all events that currently happened. In that loop there are the event functions of all the widgets, user written, or the base functions for the widgets which the user did not overwrite. In our case the order of these functions is not really important. The event is passed to the functions as an argument.

Next the combo box and scrollbar event functions are called again outside the event loop, with the event argument NULL. It is explained why it is done below, but in brief, this is necessary for all widgets that have a certain continuous movement.

Next there is the drawing part, which is done only when the variable draw is nonzero, and at the end of it, zero is assigned again to that variable. It begins with clearing the renderer, and ends with presenting everything drawn on the screen. Next there are draw functions of the widgets, which can be overwritten by the user as well, the same way as the event functions. The base draw functions return nonzero when anything needs to be drawn. But in our case no additional drawing is done. The only thing important in our case what concerns the order of the drawing functions, is that the combo box has to be after the scrollbar and the entry box, as it partly covers them when it's open.

Finally all the objects allocated are freed by the kiss_clean(). The array named objects is passed to that function. It is the same array that were passed to kiss_init(), where all the allocated objects were added to that array, and as you noticed, we also added the array

for the combo box, to that array. Adding allocated objects to the array in that way, enables to free them all at once, with one function.

5. Install

To maintain simplicity, no library has been made, but either a static or a dynamic library can be made, and the path to the resources (images and fonts) can be defined by adding a macro definition like -DRESDIR=\"kiss/\" to the compiler's command line. The simplest way to use it, is to copy all files to your project's directory and compile them with your code.

The common way in Windows is to copy all the dll-s from the proper bin directories (or library directories when using Visual Studio) of SDL, SDL_image and SDL_ttf, to your project's directory, this is how SDL programs are usually distributed. Or copy them to the Windows system directory, but then they may have to be replaced when using different versions of SDL2.

The kiss_makefile compiles the examples that show all the widgets in the toolkit. The kiss_makefile was made for Linux, to use it in Windows, edit it, comment the lines LDFLAGS, CFLAGS, EXE1 and EXE2, and uncomment the corresponding lines for 32 bit Windows, 64 bit Windows or Macintosh. The Xcode command line tools also have to be installed standalone in Macintosh. Then change the paths to the directories under which you installed the SDL2 development libraries. By default, it is assumed that the libraries are installed under C:\. No change may be necessary, or the change may be simple, like only changing the version of the library from 2.0.4 to 2.0.6.

In Windows, this toolkit was tested with MinGW-w64, installed from the file i686-5.3.0-release-posix-dwarf-rt_v4-rev0.7z (extracted to C:\ with C:\mingw32\bin added to the PATH), which is a 32 bit version of MinGW-w64 . Other versions of MinGW should also work, but they are not tested. A 32 bit compiler also works with 64 bit Windows, but a 64 bit compiler cannot be used with 32 bit Windows.

It was also tested in the Visual Studio Community 2015. The project files for the Visual Studio are provided, but a similar changes to the library paths should be made if necessary, as described above, in the project -> properties -> configuration properties -> vc++ directories. The paths of the 64 bit SDL2 libraries for x64 are entered, but it was not tested with x64.

For compiling in Visual Studio, run kiss_example1.vcxproj or kiss_example2.vcxproj, make sure that on the toolbar x86 is selected (instead of x64), it was tested with x86, then build. Use one of the following commands to compile either in Linux or Macintosh, or in Windows using MinGW.

```
make -f kiss_makefile

mingw32-make -f kiss_makefile
```

The kiss_sdl project page is https://github.com/actsl/kiss_sdl .

6. Version

1.0.6

7. License

zlib license

8. Macros and enumerations

```
#define KISS_MAX_LENGTH 200
```

The maximum length of a line of text. This is the length of the buffer that includes the terminating '\0' character.

```
#define KISS_MIN_LENGTH 10
```

Only used internally for implementation of the variable size array, the initial size of the array, no need to use in any code.

```
#define KISS_MAX_LABEL 500
```

The maximum length of a label. This is the length of the buffer that includes the terminating '\0' character.

```
enum {OTHER_TYPE, WINDOW_TYPE, RENDERER_TYPE, TEXTURE_TYPE,
     SURFACE_TYPE, FONT_TYPE, STRING_TYPE, ARRAY_TYPE};
```

Used for the ID when storing a reference to an object to the variable size array, to determine the type of the object.

9. Structures

This toolkit is not object oriented, each widget has its own structure, it is not more complicated. There are no callbacks or signals and slots.

```
typedef struct stat kiss_stat;
```

The POSIX structure stat for the file information, the implementation in the Visual Studio is not full POSIX.

```
typedef struct dirent kiss_dirent;
```

The POSIX structure dirent for the directory entry, the implementation in the Visual Studio is not full POSIX.

```
typedef DIR kiss_dir;
```

The POSIX structure DIR for the directory, the implementation in the Visual Studio is not full POSIX.

```
typedef struct kiss_array {
     void **data;
     int *id;
     int length;
     int size;
     int ref;
} kiss_array;
```

The structure for a dynamic array, also called dynamically sized array, which is an array similar to the variable size array in glib, or a vector container in C++. It is an array with unlimited size. data is an array of void pointers and id is an array of integers, both are in the continuous memory space and thus can be indexed as ordinary C arrays. length is the number of elements in these C arrays that have been appended. The initial length is 0, only appending the elements increases the length. ref is the reference count, it should be increased when more references to the array are made, and is decreased by kiss_array_free(). When it reaches zero, the C arrays data and id shall be freed.

```
typedef struct kiss_image {
     SDL_Texture *image;
     int w;
     int h;
} kiss_image;
```

The structure for a loaded image. image is the image loaded in kiss_init(). w and h are the width and height of the loaded images, assigned in kiss_init() by querying the textures.

```
typedef struct kiss_font {
     TTF_Font *font;
     int fontheight;
     int spacing;
     int lineheight;
     int advance;
     int ascent;
} kiss_font;
```

The structure for a loaded font. font is the font loaded in kiss_init(). Next are the attributes of the font, assigned in kiss_init() by querying the font. fontheight is the maximal height in pixels of the characters written with that font. spacing is the vertical space between the lines in pixels, and lineheight is the vertical distance between the beginning of a line and the beginning of the next line. advance is the horizontal distance in pixels between the beginning of a character and the beginning of the next character.

ascent is the number of pixels from the upper edge of the text, to the text's baseline. ascent is used to align widgets to the baseline, which is done in the following way. By adding the y coordinate of the upper edge of the text and the ascent, to the y coordinate of the widget, the widget's upper edge is aligned to the baseline. By then subtracting the height of the widget from the y coordinate of the widget, the widget's lower edge is made to be aligned to the baseline of the text.

```
typedef struct kiss_window {
     SDL_Rect rect;
     int visible;
     int focus;
     int decorate;
     SDL_Color bg;
     struct kiss_window *wdw;
} kiss_window;
```

The structure for a window widget. This widget is only a rectangle rect, drawn with a color bg, and decorated with the function kiss_decorate() when the member decorate is nonzero. This is a simple widget and its structure contains members that appear in most of the other widget structures. visible and focus are to separately set the visibility and mouse focus for the widget. wdw is a pointer to another window widget that determines the visibility and focus of the widget. The window wdw is a dialog window, on which other widgets are drawn.

```
typedef struct kiss_label {
     SDL_Rect rect;
     int visible;
     char text[KISS_MAX_LABEL];
     SDL_Color textcolor;
     kiss_font font;
     kiss_window *wdw;
} kiss_label;
```

The structure for a label widget. textcolor is the color of the text. font is the font of the text, so different labels can be drawn with different fonts, if additional fonts are loaded. The member named text contains the string to be drawn. This string may contain '\n' characters for line feed, thus the label can be multiline. This structure does not contain a member for focus, as label does not react to mouse events.

```
typedef struct kiss_button {
     SDL_Rect rect;
     int visible;
     int focus;
     int textx;
     int texty;
     char text[KISS_MAX_LENGTH];
     int active;
     int prelight;
     SDL_Color textcolor;
     kiss_window *wdw;
} kiss_button;
```

The structure for a button widget. The members textx and texty are SDL window

coordinates of the upper left corner of the text texture, these are calculated in the kiss_button_new() and there is no need to use them in any code. text is the text drawn on the button. prelight is nonzero when the mouse pointer is on the button. active is nonzero when the button is pushed (held down). The combination of these provides three states of the button, normal (active zero and prelight zero), prelight (prelight nonzero) and active (active nonzero). A separate image corresponds to each of these states.

```
typedef struct kiss_selectbutton {
     SDL_Rect rect;
     int visible;
     int focus;
     int selected;
     kiss_window *wdw;
} kiss_selectbutton;
```

The structure for a select button widget. The select button is a simple widget similar to button, that can be in two states. When selected is nonzero, then the select button is selected and drawn by default as a small filled rectangle, and when it is zero, then the select button is not selected and is drawn by default as a small empty rectangle.

```
typedef struct kiss_vscrollbar {
     int visible;
     int focus;
     SDL_Rect uprect;
     SDL_Rect downrect;
     SDL_Rect sliderrect;
     int maxpos;
     double fraction;
     double step;
     unsigned int lasttick;
     int downclicked;
     int upclicked;
     int sliderclicked;
     kiss_window *wdw;
} kiss_vscrollbar;
```

The structure for a vertical scrollbar widget. Different from other widgets, vertical scrollbar has three rectangles, uprect, downrect and sliderrect, correspondingly for the up arrow, down arraw and slider. These rectangles are used in the same way as rect is used for other widgets, both for drawing the images and to determine whether the mouse events happened inside these rectangles. The member maxpos is the maximum number of pixels by which the slider can go down. It is calculated in the kiss_vscrollbar_new() and there is no need to use it in any code. The fraction is a floating point number in the range 0. to 1. and it is the fraction of the maximal position of the slider, by which the slider is currently down from its upper position. The step is the step by which the slider moves by one mouse click on the down arrow, it is a floating point number in the range 0. to 1. If the step is close to 0., the scrollbar does not react to mouse events.

The member lasttick in this and other widgets, is a number of time ticks when the number of ticks were last read, this is handled by the kiss_vscrollbar_event() internally.

The members downclicked, upclicked and sliderclicked indicate whether the mouse is currently clicked on the up arrow, down arrow or on the slider. In case of the slider the mouse pointer may move away from the slider, sliderclicked remains nonzero until the left mouse button is released. These members are handled by the kiss_vscrollbar_event() internally, and there is no need to use them in any code.

```
typedef struct kiss_hscrollbar {
     int visible;
     int focus;
     SDL_Rect leftrect;
     SDL_Rect rightrect;
     SDL_Rect sliderrect;
     int maxpos;
     double fraction;
     double step;
     unsigned int lasttick;
     int leftclicked;
     int rightclicked;
     int sliderclicked;
     kiss_window *wdw;
} kiss_hscrollbar;
```

The structure for a horizontal scrollbar widget. This widget is similar to the vertical scrollbar widget, with the only difference that the slider moves horizontally, and instead of up and down arrows, there are left and right arrows.

```
typedef struct kiss_progressbar {
     SDL_Rect rect;
     int visible;
     SDL_Rect barrect;
     int width;
     double fraction;
     double step;
     SDL_Color bg;
     unsigned int lasttick;
     int run;
     kiss_window *wdw;
} kiss_progressbar;
```

The structure for a progress bar widget. barrect is the current rectangle for the progress bar, this changes when the progress bar increases or decreases. width is the maximum width of the progress bar, this is calculated by kiss_progressbar_new() and is used internally. fraction is the current fraction of the width of the progress bar, of the value of width. step is a fraction of width, by which the progress bar moves during one kiss_progress_interval. run indicates whether the progress bar currently moves, this enables to make the progress bar to continuously move with the speed determined by step.

```
typedef struct kiss_entry {
     SDL_Rect rect;
     int visible;
     int focus;
     int decorate;
     int textx;
     int texty;
     char text[KISS_MAX_LENGTH];
     int active;
     int textwidth;
     int selection[4];
     int cursor[2];
     SDL_Color normalcolor;
     SDL_Color activecolor;
     SDL_Color bg;
     kiss_window *wdw;
} kiss_entry;
```

The structure for an entry box widget. text is the text in the entry box. textwidth is the width of the text area in pixels, this is calculated in kiss_entry_new(). The maximum number of characters in the entry box is calculated by textwidth and the width of the characters in the font, this enables to also use proportional fonts, by rewriting the function kiss_maxlength() in kiss_draw.c. active indicates whether the entry box is open. When the entry box is open, the color of the text changes from normalcolor to activecolor, also the decoration color changes by default from blue to green, if the entry box is decorated. When active, the entry box is open to editing, until closed by pressing Enter. When the member wdw is not NULL, the entry box also takes mouse focus when active. The cursor and selection are members for implementing additional editing and text processing, these are not currently used.

```
typedef struct kiss_textbox {
     SDL_Rect rect;
     int visible;
     int focus;
     int decorate;
     kiss_array *array;
     SDL_Rect textrect;
     int firstline;
     int maxlines;
     int textwidth;
     int highlightline;
     int selectedline;
     int selection[4];
     int cursor[2];
     SDL_Color textcolor;
     SDL_Color hlcolor;
     SDL_Color bg;
     kiss_window *wdw;
} kiss_textbox;
```

The structure for a text box widget. array is a kiss_array that contains text, one line of text in every element of it. This array has to be created by the user, and added to the text box as an argument of kiss_textbox_new(). The user is also responsible for freeing the array. textrect is the actual rectangle for the text, this is calculated in

kiss_textbox_new() and used internally. firstline is the index of the line of the text in the array, that is the first in the text box. maxlines is the maximum number of lines of text in the text box, this is calculated in the kiss_textbox_new(). Highlightline is the number of the line in the text box on which the mouse cursor currently is, starting from 0, it is -1 when the mouse cursor is not on any line. hlcolor is the color of the background of that line, it is set to light blue in the kiss_textbox_new(), but can be changed to another color by the user. selectedline is the number of the line in the text box that is selected by mouse click. Selecting a line causes the event function to return nonzero, and selectedline can be read when reacting to that, to determine what line was selected.

```
typedef struct kiss_combobox {
     int visible;
     char text[KISS_MAX_LENGTH];
     kiss_entry entry;
     kiss_window window;
     kiss_vscrollbar vscrollbar;
     kiss_textbox textbox;
     kiss_window *wdw;
} kiss_combobox;
```

The structure for a combo box widget. Combo box is a composite widget, and thus it contains entry, window (below the text box and scrollbar), vscrollbar and text box, that are all widget structures of the corresponding widgets. text is the initial text in the entry box, and it also contains the final text when the combo box closes either by selecting a line in the text box, or finishing editing the entry box by pressing Enter. The entry box has the window the combo box is on, as its dialog window, and text box and vscrollbar have the combo box window as their dialog window. Thus when the combo box is open, then the ebtry box takes focus from the window below the combo box, but not from the combo box window, that has no dialog window. This also enables the scrollbar to take focus the same time when the entry box has focus, because its focus is taken on a different dialog window.

10. Global variables

The global variables are not declared as constant, to make the code more easily modifiable, but they should be used as constants, by changing them only during the initialization.

```
extern SDL_Color kiss_white, kiss_black, kiss_green, kiss_blue,
     kiss_lightblue;
```

Colors, assigned in kiss_draw.c.

```
extern kiss_font kiss_textfont, kiss_buttonfont;
```

Structures for loaded fonts, assigned in kiss_init(). kiss_textfont is by default used in labels, text boxes and entry boxes, and kiss_buttonfont is only used for the button's texts.

```
extern kiss_image kiss_normal, kiss_prelight, kiss_active,
     kiss_bar, kiss_up, kiss_down, kiss_left, kiss_right,
     kiss_vslider, kiss_hslider, kiss_selected, kiss_unselected;
```

Structures for loaded images, assigned in kiss_init(). As for every loaded image there is a structure, one only has to remember the names of the images, to get the width or height of any of them. normal, prelight and active are images for the three states of a button. bar is the image from the progress bar's bar. up, down, left and right are images for the arrows of the scrollbars. vslider and hslider are images for the sliders of vertical and horizontal scrollbars. selected and unselected are images for the states of a select button.

```
extern double kiss_spacing;
```

Fraction of the line height for space between the text lines, a floating point value in the range 0. to 1., assigned in kiss_draw.c.

```
extern int kiss_textfont_size, kiss_buttonfont_size;
```

By default, both fonts are loaded from the same ttf font file, and the size of the font is different for text and buttons, these sizes are assigned in kiss_draw.c.

```
extern int kiss_click_interval, kiss_progress_interval;
```

The intervals of movement in scroll bars and progress bar, in milliseconds, assigned in kiss_draw.c.

```
extern int kiss_slider_padding;
```

The minimal space between the slider and the arrows in pixels, assigned in kiss_draw.c.

```
extern int kiss_border, kiss_edge;
```

Border is a free space around the edge of the widget in pixels, and edge is the distance from the edge of the widget to the beginning of the decoration, or where the decoration becomes dark enough. Assigned in kiss_draw.c.

```
extern int kiss_screen_width, kiss_screen_height;
```

The width and height of the SDL window, assigned in kiss_init().

11. POSIX functions

These are defined in kiss_posix.c.

```
char *kiss_getcwd(char *buf, int size);
```

An overwritten POSIX function to get the path of the current working directory.

```
int kiss_chdir(char *path);
```

An overwritten POSIX function to change directory.

```
int kiss_getstat(char *pathname, kiss_stat *buf);
```

An overwritten POSIX function to get information about a file or directory entry, and write it to the structure kiss_stat pointed to by buf.

```
kiss_dir *kiss_opendir(char *name);
```

An overwritten POSIX function to get information about a directory.

```
kiss_dirent *kiss_readdir(kiss_dir *dirp);
```

An overwritten POSIX function to get information about a directory entry.

```
int kiss_closedir(kiss_dir *dirp);
```

An overwritten POSIX function to close the directory opened to get information, and free all information returned about the directory and its entry.

```
int kiss_isdir(kiss_stat s);
```

An overwritten POSIX function that returns nonzero when the directory entry is a directory.

```
int kiss_isreg(kiss_stat s);
```

An overwritten POSIX function that returns nonzero when the directory entry is a regular file.

12. General functions

These are defined in kiss_general.c, these functions are not specific to the widget toolkit or any graphical processing.

The general variable size array implemented in this toolkit, enables to add the widget structures to it, and pass them together to an external function. The variable size array also has an ID number for every element, this enables to use unique numbers for every added widget, and by these numbers it is possible to identify these widgets in another function.

Whenever a size limit of a string is used or calculated in the functions of this toolkit, it is always one more than the maximum number of characters. This is a rule, because it is consistent with the minimum necessary buffer size, which is one more than the number of characters because of the terminating '\0' character.

```
int kiss_makerect(SDL_Rect *rect, int x, int y, int h, int w);
```

A function to make an SDL rectangle from the arguments.

```
int kiss_pointinrect(int x, int y, SDL_Rect *rect);
```

A function that returns nonzero when a point with the coordinates x and y, is in the SDL rectangle.

```
int kiss_utf8next(char *str, int index);
```

A function that returns the length of the UTF-8 character at the index (1 to 4, 0 if at the end of string). This and the next function only work correctly when the index is at the beginning of a UTF-8 character or at the terminating '\0' character of the string.

```
int kiss_utf8prev(char *str, int index);
```

A function that returns the length of the UTF-8 character that precedes the character at the index (1 to 4, 0 if at the beginning of string). The previous function and this function can be used to iterate forward and backward in a string, by one UTF-8 character. If the string contains only ASCII characters, then the iteration is by 1.

```
int kiss_utf8fix(char *str);
```

Fixes the string by removing any broken UTF-8 character at the end.

```
char *kiss_string_copy(char *dest, size_t size, char *str1,
     char *str2);
```

A general function that copies two strings to dest. The maximum number of characters copied is one less than size, and the terminating '/0' character is always written. Both source1 and source2 can be NULL pointers. Returns NULL in case of error.

```
int kiss_string_compare(const void *a, const void *b);
```

A general function to compare two strings, that can be used as an argument of the functions for searching and sorting arrays.

```
char *kiss_backspace(char *str);
```

A general function that deletes the last character in the string.

```
int kiss_array_new(kiss_array *a);
```

Creates a new variable size array. The kiss_array structure for the array has to be created and provided by the user.

```
void *kiss_array_data(kiss_array *a, int index);
```

Returns the data element of the variable size array, at the index.

```
int kiss_array_id(kiss_array *a, int index);
```

Returns the ID element of the variable size array, at the index.

```
int kiss_array_assign(kiss_array *a, int index, int id,
     void *data);
```

Assigns ID and data to an element of the variable size array, at the index.

```
int kiss_array_append(kiss_array *a, int id, void *data);
```

Appends ID and data elements to the end of the variable size array.

```
int kiss_array_appendstring(kiss_array *a, int id, char *text1,
     char *text2);
```

Appends a string made by adding text1 and text2, to the variable size array. The length of the string is limited by KISS_MAX_LENGTH.

```
int kiss_array_insert(kiss_array *a, int index, int id,
     void *data);
```

Inserts ID and data elements before the variable size array element at the index.

```
int kiss_array_remove(kiss_array *a, int index);
```

Removes the variable size array element at the index.

```
int kiss_array_free(kiss_array *a);
```

Frees the variable size array. Does not free the kiss_array structure, freeing that structure is the responsibility of the user.

13. Draw functions

These are defined in kiss_draw.c.

```
unsigned int kiss_getticks(void);
```

Gets the number of milliseconds since the SDL library initialization. Had to be abstracted to kiss_draw.c only because none of the external library functions are directly called in the kiss_widgets.c. By external library functions are meant any external functions other than the standard C library functions.

```
int kiss_maxlength(kiss_font font, int width, char *str1,
     char *str2);
```

Returns the maximum length of string that can be rendered from the strings with the given font, plus one character for the terminating '\0' by the rule, without exceeding the length of width pixels. Can be rewritten for proportional fonts.

```
int kiss_textwidth(kiss_font font, char *str1, char *str2);
```

Returns the width of the text rendered with the given font, in pixels. Works with proportional fonts. Either of the strings can be NULL.

```
int kiss_renderimage(SDL_Renderer *renderer, kiss_image image,
     int x, int y, SDL_Rect *clip);
```

Renders image to the SDL window at the coordinates x and y with the width and height provided by the rectangle clip, or the whole texture when clip is a NULL pointer.

```
int kiss_rendertext(SDL_Renderer *renderer, char *text,
     int x, int y, kiss_font font, SDL_Color color);
```

Renders text to the SDL window at the coordinates x and y with the given font and color.

```
int kiss_fillrect(SDL_Renderer *renderer, SDL_Rect *rect,
     SDL_Color color);
```

Renders a filled rectangle rect to the SDL window with color.

```
int kiss_decorate(SDL_Renderer *renderer, SDL_Rect *rect,
     SDL_Color color, int edge);
```

A function for decorating the edge of a widget, by default renders a rectangle around the edge of the rectangle rect with color. edge is the distance from the edge of the widget to the rendered rectangle, it is usually kiss_edge. All functions for creating the widgets that are decorated in that way, have an argument decorate, the edges of these widgets shall not be decorated when that argument is zero. The function can be rewritten for decorating the edge of the widget differently.

```
SDL_Renderer* kiss_init(char* title, kiss_array *a, int w, int h);
```

Creates a new SDL window with title, width w and height h, by default the window is centered on the screen, and returns renderer. It also initializes kiss_sdl, including loading all images and fonts. The array a has to be provided by the user, the created objects are appended to it, so that they can later be freed in the kiss_clean(). This function can be rewritten for using different font or image libraries.

```
int kiss_clean(kiss_array *a);
```

Frees all objects added to the array a, and the array itself. This function is specific to kiss_sdl, and not general. Does not free the structure a, freeing that if it is created

dynamically, is the responsibility of the user. When rewriting kiss_init(), this function may have to be rewritten also, to free different images, fonts or other objects.

14. Widget functions

These are defined in kiss_widgets.c.

The functions for creating widgets have an argument wdw which, if provided, makes the widget to become visible when the window widget pointed by that argument becomes visible, and not visible when that window becomes not visible. That window is thus like a dialog window on which all the widgets in the dialog are drawn, the examples show how to do that. Some of these functions also have an additional argument named decorate, this determines whether the widget should be decorated by the default function.

Widgets can be made visible and invisible, an invisible widget is also inactive and doesn't perform any functions. In the examples, all widgets are created first, and then only these that should be active and present on the screen, are made visible. It is also possible to create and free widgets dynamically. Making a user interface in that way may sound strange at first, but all the user written functions for widgets have to be there anyway, a lot has to be there for all widgets. For bigger interfaces it is possible to create widgets dynamically. But making it only by switching the visibility, enables a simpler implementation of a user interface.

In addition to visibility, the widgets also may or may not have focus. Focus determines whether the widget processes mouse events. The focus of the widget is determined by the focus of the window wdw. If this window is provided and has no focus, the focus of a widget is determined by the focus member of the structure of that widget.

The event processing functions are called after each other in the event processing loop. The scrollbar, progressbar and combo box event functions are also called after the event loop with the event argument NULL. The base function returns nonzero when an event happened in that widget which the user may want to additionally process. The event functions have an additional argument, a pointer to an integer that is assigned a nonzero value when the widgets have to be redrawn.

The drawing functions are called after each other in every cycle, after the event processing loop, when there is a need to redraw the widgets. The user may write one's own drawing functions and call the base functions inside them, to do an additional drawing. The order of the drawing functions in the loop is important, the next functions draw over the drawing done by the previous functions. Especially when combo boxes are used, as their popup text boxes draw over the widgets below the combo box.

```
int kiss_window_new(kiss_window *window, kiss_window *wdw,
     int decorate, int x, int y, int w, int h);
int kiss_window_event(kiss_window *window, SDL_Event *event,
     int *draw);
int kiss_window_draw(kiss_window *window, SDL_Renderer *renderer);
```

Base functions for the window widget. The window widget is a simple widget which is only a filled rectangle with width w and height h, drawn on the SDL window at the coordinates x and y. The wdw argument in the new function, the same as with other widgets, may provide a dialog window on which this widget is drawn, or may be NULL. The draw function makes the widget visible, when its dialog window is visible.

```
int kiss_label_new(kiss_label *label, kiss_window *wdw,
     char *text, int x, int y);
int kiss_label_draw(kiss_label *label, SDL_Renderer *renderer);
```

Base functions for the label widget. The label widget is a simple widget which is only a text drawn on the SDL window at the coordinates x and y. The text may contain '\n' characters, providing a multiline text. The label widget does not react to any events, thus it has no event function. A way to process events is to add a window widget beneath the label widget.

```
int kiss_button_new(kiss_button *button, kiss_window *wdw,
     char *text, int x, int y);
int kiss_button_event(kiss_button *button, SDL_Event *event,
     int *draw);
int kiss_button_draw(kiss_button *button, SDL_Renderer *renderer);
```

Base functions for the button widget, drawn at coordinates x and y on the SDL window, with text written on it. The new function for a button calculates its rectangle, copies the button's text and calculates its position, assigns the window argument to the member of its structure, and initializes other members of its structure to zero. The event function first writes one to the draw argument when the SDL window is exposed. Then it checks whether the button or the dialog window of the button has a mouse focus, and returns from the function if neither is true. Then it checks the mouse buttons and mouse motion, changes the state of the button based on these events, and returns nonzero when the button is released. The draw function makes the button visible, when its dialog window is visible. It then renders the image that corresponds to the current state of the button, and renders text to the button.

```
int kiss_selectbutton_new(kiss_selectbutton *selectbutton,
     kiss_window *wdw, int x, int y);
int kiss_selectbutton_event(kiss_selectbutton *selectbutton,
     SDL_Event *event, int *draw);
int kiss_selectbutton_draw(kiss_selectbutton *selectbutton,
     SDL_Renderer *renderer);
```

Base functions for the select button widget, drawn at coordinates x and y on the SDL window. The select button is a simple widget similar to button, that can be in two states, toggled by clicking on the select button.

```
int kiss_vscrollbar_new(kiss_vscrollbar *vscrollbar,
     kiss_window *wdw, int x, int y, int h);
int kiss_vscrollbar_event(kiss_vscrollbar *vscrollbar,
     SDL_Event *event, int *draw);
int kiss_vscrollbar_draw(kiss_vscrollbar *vscrollbar,
     SDL_Renderer *renderer);
```

Base functions for the vertical scrollbar widget, drawn at coordinates x and y on the SDL window, with height h. The event function returns nonzero when the slider moves, so that the user can do further processing, using the value of the fraction member of the widget's structure. The event function also has to be called in the main loop outside the event processing loop, with the event argument NULL. This is because when clicking on either of the arrows, and holding the left mouse button down, the slider moves after every certain number of time ticks, thus the event function has to be periodically called to read the time ticks.

```
int kiss_hscrollbar_new(kiss_hscrollbar *hscrollbar,
     kiss_window *wdw, int x, int y, int w);
int kiss_hscrollbar_event(kiss_hscrollbar *hscrollbar,
     SDL_Event *event, int *draw);
int kiss_hscrollbar_draw(kiss_hscrollbar *hscrollbar,
     SDL_Renderer *renderer);
```

Base functions for the horizontal scrollbar widget, drawn at coordinates x and y on the SDL window, with width w. Similar to the vertical scrollbar widget, except that the slider moves horizontally.

```
int kiss_progressbar_new(kiss_progressbar *progressbar,
     kiss_window *wdw, int x, int y, int w);
int kiss_progressbar_event(kiss_progressbar *progressbar,
     SDL_Event *event, int *draw);
int kiss_progressbar_draw(kiss_progressbar *progressbar,
     SDL_Renderer *renderer);
```

Base functions for the progress bar widget, drawn at coordinates x and y on the SDL window, with width w. The progress bar widget is made to indicate the time passed while doing a long processing, by increasing the width of the progress bar. The event function also has to be called in the main loop outside the event processing loop, with the event argument NULL. This is because when made to run, the width of the progress bar increases after every certain number of time ticks, thus the event function has to be periodically called to read the time ticks.

```
int kiss_entry_new(kiss_entry *entry, kiss_window *wdw,
     int decorate, char *text, int x, int y, int w);
int kiss_entry_event(kiss_entry *entry, SDL_Event *event,
     int *draw);
int kiss_entry_draw(kiss_entry *entry, SDL_Renderer *renderer);
```

Base functions for the entry box widget, drawn at coordinates x and y on the SDL window, with width w, with the initial text provided as an argument of the new function. By clicking in the entry box when it's not active, it becomes active. When active, the entry box is open to editing, until closed by pressing Enter. The entry box, and also the entry box in the combo box, takes the keyboard focus when active, and also the mouse focus when the dialog window is provided. The editing provided is that of the early versions of Unix, backspace deletes the last character and ctrl-u deletes the entire text. An additional functionality can be added, such as clipboard, but what it may be depends on the particular needs.

```
int kiss_textbox_new(kiss_textbox *textbox, kiss_window *wdw,
     int decorate, kiss_array *a, int x, int y, int w, int h);
int kiss_textbox_event(kiss_textbox *textbox, SDL_Event *event,
     int *draw);
int kiss_textbox_draw(kiss_textbox *textbox,
     SDL_Renderer *renderer);
```

Base functions for the text box widget, drawn at coordinates x and y on the SDL window, with width w and height h. The array a is a kiss_array that contains text, one line of text in every element of it, the array has to be provided by the user. By default the text box acts like a list box, with the event function returning nonzero when clicking on a line of text. But text box is not a list box, it is what it is called, a text box. The text editors such as vim have the text internally stored exactly in the same way as in the text box, as an array of pointers to strings. Thus everything can be done there, but this depends on what one may want to do, and there is a huge number of such things that can be done.

```
int kiss_combobox_new(kiss_combobox *combobox, kiss_window *wdw,
     char *text, kiss_array *a, int x, int y, int w, int h);
int kiss_combobox_event(kiss_combobox *combobox, SDL_Event *event,
     int *draw);
int kiss_combobox_draw(kiss_combobox *combobox,
     SDL_Renderer *renderer);
```

Base functions for the combo box widget, drawn at coordinates x and y on the SDL window. The width w and height h at that, are not the width and height of the whole combo box, neither closed or opened, but the width and height of the text box of the combo box. Combo box is a composite widget, and thus it contains an entry box, a window, vertical scrollbar and text box. The text in the entry box can be edited. Ending the editing by pressing Enter, closes the combo box and makes its event function to return nonzero. Selecting a line in the text box also closes the combo box and makes its event function to return nonzero. The event function also has to be called in the main loop outside the event processing loop, with the event argument NULL. This is because the same applies to the vertical scrollbar of the combo box, as to the scrollbar widgets.