NAME

Lua-AKFAvatar reference − AKFAvatar specific functions for Lua

DESCRIPTION

This is a reference for the Lua-bindings to AKFAvatar. It does not describe the language Lua. The reference manual for the language Lua can be found on the Internet on **http://www.lua.org/manual/5.2/**.

Don't worry, you don't need to understand or even learn everything. You can do lots of things with very few commands. Just pick the parts you are interested in and experiment with that. There are many parts of Lua or AKFAvatar which you will most probably never need.

File formats

For images the formats **XPM**, **XBM** and uncompressed **BMP** are always supported. More formats may be supported if **SDL_image** and further libraries are installed.

For audio the formats **AU** and **WAV** are supported. Files may be encoded as PCM, µ-law or A-law. There is also a Lua module named **akfavatar-vorbis**, which adds support for Ogg Vorbis audio files.

Basic functionality

local avt = require "lua-akfavatar"

Before you can use the AKFAvatar binding, you have to load it with this function. This gives you the table *avt*, with which you can access the following functions.

avt.encoding(*encoding***)**

Set the used text encoding. If this function is not used, only "ASCII" is supported.

Supported encodings are:
ASCII, UTF-8,
ISO-8859-1, -2, -3, -4, -5, -7, -8, -9, -10, -11, -13, -14, -15, -16,
KOI8-R, KOI8-U,
IBM437 (PC-8), IBM850 (DOS-Latin-1),
WINDOWS-1250, WINDOWS-1251, WINDOWS-1252
SYSTEM

avt.get_encoding()

Gets the current text encoding.

avt.title(*title [,shortname]***)**
avt.set_title(*title [,shortname]***)**

Sets the title and the shortname.

avt.start(*[mode]***)**

This function opens the graphical window or switches into the graphical mode. If it was already started it resets almost everything.

mode Is one of "auto", "window", "fullscreen", or "fullscreen no switch". The default is "auto" which uses a sensible default, or leaves the mode unchanged, if it already was started.

If you want to set the encoding, the title and the background color, you should do it before you call **avt.start()**. All other functions should be used after it.

Example:

```
local avt = require "lua-akfavatar"
avt.encoding("ISO-8859-1")
avt.title("my program")
avt.set_background_color("sky blue")
avt.start()
avt.start_audio()
avt.avatar_image("default")
avt.tell("hello world")
avt.wait_button()
```

Many of the following functions implicitly call **avt.start()** when it is not started yet.

avt.started()
Returns *true* if AKFAvatar it is started, or else it returns *false*.

avt.avatar_image(*data***)**
Loads the avatar image from the *data*. The *data* can be "default" or "none" or a string with image-data or a table with strings from XPM data. On error it returns *nil* and an error message.

avt.avatar_image_file(*filename***)**
Loads the avatar image from a file. The strings "default" or "none" are also accepted. On error it returns *nil* and an error message.

avt.set_avatar_name(*[name]***)**
Set the name of the avatar. This must be used after **avt.change_avatar_image()**.

avt.set_avatar_mode(*mode***)**
Set the avatar mode. The *mode* can be one of "say", "think", "header" or "footer". The avatar mode is only used, when there is an avatar image.

avt.say(...**)**
avt.write(...**)**
Prints text in the balloon.

You can use strings or numbers. It works similar to **io.write()**, but it writes into the balloon instead of the standard output.

Example:
```
avt.say("Three apples cost ", 3 * appleprice, " Euros.\n").
```

avt.print(...**)**
Prints text in the balloon. It works similar to **print**, but it writes into the balloon instead of the standard output.

The difference to **avt.say()** is, that the values are separated by tabulators in the output and each value is automatically converted with **tostring()** and at the end a new line is started.

You can simply replace the print command with this one with **print = avt.print**. (The script ‘interactive_lua.lua’ does this.)

The function **avt.say()** is better suited for programs, while this function is better suited for interactive use or debugging.

avt.tell(...**)**
Prints text in the balloon, but before that the size of the balloon is adjusted so that the text exactly fits.

Example:
```
avt.tell("Three apples cost ", 3 * appleprice, " Euros.").
```

Attention: This function still has problems with tabs ("\t").

avt.newline()
Start a new line. It is the same as "\n" in **avt.say()**, but faster.

avt.set_text_delay(*[delay]***)**
Activate the slow-print mode. With no value for *delay* it uses a default delay. To deactivate the slow-print mode, use the value 0 for *delay*.

avt.clear()
Clears the textfield or viewport. If there is no balloon yet, it is drawn.

avt.flip_page()
Waits a while and then clears the textfield; same as "\f" in **avt.say()**. See also **avt.set_flip_delay(delay)**.

avt.move_in()
Moves the avatar in.

avt.move_out()
Moves the avatar out.

avt.pager(*text [,startline]***)**
Show a longer text with a text-viewer application.

If the *startline* is given and it is greater than 1, then it starts in that line. But you still can scroll back from there.

avt.wait(*[seconds]***)**
Waits a given amount of seconds (may be a fraction).

If no value is given, it waits "a while".

avt.ticks()
Returns a value, which is increased each millisecond. This can be used for timing.

avt.show_avatar()
Shows only the avatar without any balloon.

avt.bell()
Makes a sound or flashes the display if audio is not started.

avt.flash()
Flashes the display.

avt.show_image(*data***)**
Shows an image. The *data* can be a string with image-data or a table with strings from XPM data. It returns *true* on success, or *false* on error. If it succeeds you should call **avt.wait()** or **avt.wait_button()** or **avt.get_key()**.

avt.show_image_file(*filename***)**
Load an image and show it. It returns *true* on success, or *false* on error. You can search for the file with *avt.search()*. If it succeeds you should call **avt.wait()** or **avt.wait_button()** or **avt.get_key()**.

avt.subprogram(*function, [arg1, ...]***)**
Call the function as a subprogram.

On a quit-request (pressing the <ESC>-key or the close button of the window) it just returns to the main program.

On success it returns the results of the function, on a quit-request it returns nothing. Errors are treated normally.

To call a subprogram from a separate file, use **dofile**:
```
avt.subprogram(dofile, "subprogram.lua")
```

avt.optional(*modname***)**
Loads a module like **require**, but the module is not required, but optional. That means, it is not an error, when it cannot be loaded.

Lua-AKFAvatar need not be initialized to use this function.

Unicode

avt.say_unicode(...**)**
Prints text in the balloon with special support for Unicode characters.

This command is very similar to **avt.say()**. Strings are handled in the same way, depending on the selected encoding, but numbers are interpreted as Unicode codepoints, independent from the selected encoding.

This can be used to insert characters, which are not available in the current string encoding. Sometimes it is also just easier to use.

Example:

```
avt.say_unicode("Three apples cost ",
                tostring(3 * appleprice), 0x20AC,
                ".\n").
```

As you can see, you have to use **tostring()** when nummerical expressions are to be shown as numbers. The number *0x20AC* however represents the Euro currency sign. (The '0x' introduces hexadecimal numbers.)

By the way, a bunch of unicode numbers are handled much more efficiently than strings.

avt.printable(*codepoint***)**
Checks whether the given *codepoint* represents a printable Unicode character. It returns *true* when this is the case or *false* if the font doesn't have a printable glyph for it (it may be a supported control character). This function is independent from the chosen encoding. It can be used together with **avt.say_unicode(...)**.

avt.combining(*codepoint***)**
Checks whether the given *codepoint* represents a combining Unicode character and is treated as such. It returns *true* when this is the case or *false* if not. It is only really applicable for supported characters.

avt.detect_utf8(*string [, maxlength]***)**
Detect if *string* is valid UTF-8 (or ASCII).

Checks up to *maxlength* bytes, but it finishes to check an incomplete sequence.

avt.toutf8(*codepoint [, ...]***)**
Takes one or more Unicode *codepoints* and returns an UTF-8 encoded string for that.

avt.utf8codepoints(*str***)**
This can be used to get all Unicode codepoints from an UTF-8 encoded string as numbers.

Example:

```
for c in avt.utf8codepoints(s) do
  avt.say_unicode(c)
  avt.say(string.format(" = U+%04X", c))
  avt.newline()
end
```

Sizes and positions

avt.set_balloon_size(*[height] [, width]***)**
Sets the size of the balloon. No values or values of 0 set the maximum size.

avt.set_balloon_width(*[width]***)**
Sets the width of the balloon. No value or 0 sets the maximum.

avt.set_balloon_height(*[height]***)**
Sets the height of the balloon. No value or 0 sets the maximum.

avt.get_max_x()
Get the maximum x position of the cursor in the balloon (ie. the width).

avt.get_max_y()
Get the maximum y position of the cursor in the balloon (ie. the height).

avt.where_x()
Get the x position of the cursor in the balloon.

avt.where_y()
Get the y position of the cursor in the balloon.

avt.home_position()
Returns *true* if the cursor is in the home position or *false* otherwise. (This also works for right-to-left writing.)

avt.move_x(*x***)**
Moves the cursor to the given *x* position.

avt.move_y(*y***)**
Moves the cursor to the given *y* position.

avt.move_xy(*x, y***)**
Moves the cursor to the given *x* and *y* position.

avt.save_position()
Save the current cursor position.

avt.restore_position()
Restore the last saved cursor position.

avt.next_tab()
Moves the cursor to the next tabulator position.

avt.last_tab()
Moves the cursor to the previous tabulator position.

avt.reset_tab_stops()
Reset tab stops to every eigth column.

avt.clear_tab_stops()
Clears all tab stops.

avt.set_tab(*x, true*| *false***)**
Set or clear tab in position *x*.

avt.delete_lines(*line, number***)**
Deletes given *number* of lines, starting from *line*; the rest is scrolled up.

avt.insert_lines(*line, number***)**
Inserts given *number* of lines, starting at *line*; the rest is scrolled down.

avt.insert_spaces(*number***)**
Insert *number* spaces at the current cursor position. The rest of the line is moved.

avt.delete_characters(*number***)**
Delete *number* characters at the current cursor position. The rest of the line is moved.

avt.erase_characters(*number***)**
Erase *number* of characters. The characters are overwritten with spaces.

avt.backspace()
Go back one character. Does nothing if the cursor is at the beginning of the line.

Text style

avt.markup(*true*| *false***)**
Set the markup mode. In the markup mode the character "_" toggles the underlined mode on or off and the character "*" toggles the bold mode on or off. Both characters are never displayed in markup mode!

You can always use the overstrike technique, which doesn’t reserve any characters, but is harder to use.

avt.underlined(*true*| *false***)**
Set the underlined mode.

avt.get_underlined()
Returns *true* if the underlined mode is active or *false* otherwise.

avt.bold(*true*| *false***)**
Set the bold mode.

avt.get_bold()
Returns *true* if the bold mode is active or *false* otherwise.

avt.inverse(*true*| *false***)**
Set the inverse mode.

avt.get_inverse()
Returns *true* if the inverse mode is active or *false* otherwise.

avt.normal_text()
Resets the text to normal settings.

Colors

avt.set_background_color(*color***)**
Sets the background color of the window.

Colors can either be given as English names or as RGB value with 6 hexadicimal digits.

Examples

```
avt.set_background_color("sky blue")
avt.set_background_color(0x8B4513)
avt.set_background_color("#8B4513") --> deprecated
avt.set_background_color("#555") --> deprecated
```

avt.set_balloon_color(*color***)**
Sets the color of the balloon.

avt.set_text_color(*color***)**
Sets the text color.

avt.set_text_background_color(*color***)**
Sets the text background color.

avt.set_text_background_balloncolor()
Sets the text background color to the color of the balloon.

avt.set_bitmap_color(*color***)**
Sets the foregroud color for bitmaps. The background is always transparent.

avt.get_color(*color_number***)**
Get a color for a given integer value.

AKFAvatar has an internal palette with color names to use. With this function you can scan through that list. It returns the name and RGB value as strings, or it returns nothing on error.

avt.colors()
Iterator for internal color names.

AKFAvatar has an internal palette with color names to use. With this function you can scan through that list with a generic **for** loop.

```
require "lua-akfavatar"
for nr, name, rgb in avt.colors() do
  avt.normal_text()
  avt.newline()
  avt.say(string.format("%3d) %5s, %-25s", nr, rgb, name))
```

```
  avt.set_text_background_color(name) -- either name or rgb
  avt.clear_eol()
  avt.wait(0.7)
end
avt.wait_button()
```

If you don't need the *rgb* value, you can leave that variable away.

Interaction

avt.wait_button()

Waits until a button is pressed.

avt.decide()

Ask the user to make a positive or negative decision. Returns *true* or *false*.

avt.ask(*[question]***)**

Shows the *question*, if given, and waits for the user to enter something. Returns the result as string.

The following example shows how to force the input of a number:

```
require "lua-akfavatar"
avt.save_position()
repeat
  avt.restore_position()
  number = tonumber(avt.ask("Enter a number: "))
until number
avt.say("The number is ", number)
avt.wait_button()
```

avt.file_selection(*[filter]***)**

Start a file-chooser in the balloon. It starts in the current working directory. When a directory is chosen it changes the working directory to that one. At the end it returns the selected filename (which is in the then current working directory) or *nil* on error.

The *filter*, if given, should be a function. It gets a filename as parameter. The file is always in the current working directory. If the filter function returns *false* or *nil* or nothing then the filename is not shown, otherwise it is shown.

Example:

```
textfile = avt.file_selection(
   function(n)
     return string.find(n,"%.te?xt$")
   end)
```

Of course *filter* can also be the name of a previously defined function.

avt.color_selection()

Start a color-chooser in the balloon. It returns two strings: first the English name for the color and second the hexadicimal RGB definition. Both values can be used for selecting colors.

avt.get_key()

Waits for a key to be pressed and returns the unicode codepoint of the character. For some function keys it yields a number from the unicode private use section.

avt.key_pressed()

Checks whether a key was pressed. To get the key use **avt.get_key()**.

avt.clear_keys()

Clears the key buffer

avt.push_key(*codepoint***)**
Simulates a pressed key.

avt.navigate(*buttons***)**
Shows a navigation bar with the given buttons.

For buttons use a string with the following characters:

l:	left
r:	right (play)
d:	down
u:	up
x:	cancel
f:	(fast)forward
b:	(fast)backward
p:	pause
s:	stop
e:	eject
*:	circle (record)
+:	plus (add)
-:	minus (remove)
?:	help
' ':	spacer (no button)

Pressing a key with one of those characters selects it. For the directions you can also use the arrow keys. The <Pause> key returns "p". The <Help> key or <F1> return "?".

It returns the approriete character or a number.

If audio output ends while this function is active, it automatically pushes either 'f' (forward) or 's' (stop). If both are given, 'f' (forward) has precedence.

avt.menu(*items***)**
avt.long_menu(*items***)**
Shows a menu with the *items*. The *items* can be either an array with strings. Then It returns the number of the selected item.

Or *items* can be again arrays starting with a string, followed by one or more results. The results can be of any valid Lua type, including functions.

The menu starts in the line of the current cursor position. So you could put a headline before the menu.

```
avt.clear()
avt.say("Please select your favourite food:\n")
local item = avt.long_menu {
  "Chicken",
  "Chips",
  "Pizza",
  "Spinach"}
```

avt.choice(*start_line, items [, key] [, back] [,forward]***)**
This can be used for menus. It is a more basic function than **avt.menu()**. It returns the number of the selected item.

start_line:
line, where choice begins

items: number of items/lines

key: first key, like "1" or "a", 0 for no keys

back: set to *true*, when the first entry is a back function

forward:
set to *true*, when the last entry is a forward function

Audio Output

avt.start_audio()

Starts the audio subsystem.

On success it returns *true*, on error it returns *nil* and an error message.

avt.load_audio_file(*[filename [,playmode]]***)**
avt.load_base_audio_file(*[filename [,playmode]]***)**

Reads audio data from a file. You can search for the file with **avt.search()**.

Lua modules may add support for more audio formats to **avt.load_audio_file()** (for example the module **akfavatar-vorbis**).

When no *filename* is given, or the *filename* is *nil* or an empty string, it returns a silent audio element, ie. you can call its methods, it just doesn't play anything.

The *playmode* can be one of "load", "play" or "loop".

On error it returns *nil* and an error message. (Note: in version 0.19.0 it also returned a silent audio element then.)

avt.load_audio_stream(*file handle [,size [,playmode]]***)**
avt.load_base_audio_stream(*file handle [,size [,playmode]]***)**

Reads audio data from an open file handle. The file must be searchable.

Lua modules may add support for more audio formats to **avt.load_audio_stream()** (for example the module **akfavatar-vorbis**).

If size is 0 or not given, it assumes the audio is the rest of the file.

The *playmode* can be one of "load", "play" or "loop".

On error it returns *nil* and an error message.

avt.load_audio(*[audio_data [,playmode]]***)**
avt.load_base_audio(*[audio_data [,playmode]]***)**

Reads audio data from a string. Otherwise the same as **avt.load_audio_file()**.

Lua modules may add support for more audio formats to **avt.load_audio()** (for example the module **akfavatar-vorbis**).

When no *audio_data* is given, or the *audio_data* is *nil* or an empty string, it returns a silent audio element, ie. you can call its methods, it just doesn't play anything.

The *playmode* can be one of "load", "play" or "loop".

On error it returns *nil* and an error message. (Note: in version 0.19.0 it also returned a silent audio element then.)

avt.silent()

Returns a silent audio element, ie. you can call its methods, it just doesn't play anything.

Example:

```
audio = avt.load_audio_file(filename) or avt.silent()
```

In this example you get a silent sound when the file could not be read.

avt.alert()
Returns a pseudo audio element, that calls **avt.bell()** when you play it.

avt.audio_playing(*[audio_data]***)**
Checks if the audio is currently playing. If *audio_data* is given and is not *nil* then it checks, if the specified audio is playing. This can also be checked with *audio***:playing()**.

avt.wait_audio_end()
Waits until the audio output ends.

This also ends an audio-loop, but still plays to the end of the current sound.

avt.stop_audio()
Stops the audio output immediately.

avt.pause_audio(*true*| *false***)**
pause (*true*) or resume (*false*) the audio output

*audio***:play()**
*audio***()** Plays the *audio* data. The *audio* must have been loaded by **avt.load_audio_file()** or **avt.load_audio_string()**.

Only one sound can be played at the same time. When you play another sound the previous one is stopped. Use **avt.wait_audio_end()** to play sounds in a sequence.

The audio can also be played by calling the audio variables like a function.

```
play_audio_file = function (filename)
  local sound = avt.load_audio_file(avt.search(filename))
  if sound then sound:play() end
end
```

*audio***:loop()**
Plays *audio* data in a loop. The *audio* must have been loaded by **avt.load_audio_file()** or **avt.load_audio_string()**.

This is for example useful for short pieces of music.

You can stop the audio loop with **avt.wait_audio_end()** or **avt.stop_audio()**.

*audio***:playing()**
Checks if this *audio* data is currently playing. The *audio* must have been loaded by **avt.load_audio_file()** or **avt.load_audio_string()**.

This is the same as **avt.audio_playing(audio)**.

*audio***:free()**
Frees the *audio* data. If this *audio* is currently playing, it is stopped.

Audio data is also freed by the garbage collector.

avt.set_audio_end_key (*key***)**
Define a key that should automatically be pressed when audio ends. The *key* should be given as a number for the Unicode codepoint. Use *0* to deactivate it.

The function returns the value that was previously set.

avt.quit_audio()
Quit the audio subsystem.

This is not needed in normal programs. Only use this, if you are sure you need it.

File-System

avt.dirsep
This variable contains the systems directory separator; either "/" or "\".

avt.datapath

This variable contains the default searchpath for the function **avt.search()** (see below). Directories are separated by semicolons. There are no patterns like in the paths for Lua modules, just directories. This variable is initialized by the environment variable *AVT-DATAPATH* or it gets a system-specific default.

avt.search(*filename [,path]***)**

Searches for a file with the given *filename* in the given *path*. If no *path* is given, it uses the variable *avt.datapath* as default.

On success it returns the full path to the file, if the file is not found, it returns *nil* and an error message.

avt.get_directory()
avt.getcwd()

Returns the current working directory. On error it returns *nil* and an error message.

avt.set_directory(*directory***)**
avt.chdir(*directory***)**

Sets the working directory to *directory*. If *directory* is *nil*, nothing or an empty string, it does nothing.

Returns *true* on success or on error *nil* and an error message.

Example:

```
avt.set_directory(os.getenv("HOME") or os.getenv("USERPROFILE"))
```

avt.directory_entries(*[directory]***)**

Get a list of directory entries of the given *directory* or the current directory if none is given.

On success it returns a table (an array) and the number of entries. On error it returns *nil* and an error message.

The list contains normal files, including hidden files, subdirectories and any other type of entry. It does not contain "." or "..".

Note: the names are in a system specific encoding. To display the names you either have to change the encoding of the display with **avt.encoding("SYSTEM")** or convert the names like this: **avt.say(avt.recode(name, "SYSTEM"))**.

avt.entry_type(*entry***)**

Get the type of a directory entry and its size.

On success it returns the type of the directory entry as string and the size as number. The type can be one of "file", "directory", "character device", "block device", "fifo", "socket" or "unknown".

On error it returns *nil* and an error message.

Symbolic links are followed. That means, you get the type of the resulting entry. Broken links are treated like not existing entries.

Miscellaneous

avt.language

This variable contains a language code for messages. It should be a two-letter code conforming to ISO 639-1. The variable is not set, when the language could not be detected.

avt.translate(*text***)**

Translate the *text*, if possible.

See section **Translations** for how to provide translations.

avt.recode(*string, fromcode [,tocode]***)**

Recode the given *string*, which is encoded as *fromcode* to a string encoded as *tocode*. When you give just one encoding, it encodes into the currently set text encoding. If you want to encode from the current encoding to something else, then use *nil* for *fromcode*.

To encode to or from the systems default encoding (for example for filenames) use an empty string (**""**), or **"SYSTEM"**.

Returns the recoded string or *nil* on error.

Attention: Converting to UTF-8 is always possible. Converting to other encodings however can fail. Characters which cannot be converted are replaced with the control character *SUB* (*"\x1A"*). So to make sure that everyting worked well, you can search for that character in the resulting string. If it is there, something went most probably wrong.

avt.right_to_left(*true*| *false***)**

Activate or deactivate the right to left writing mode.

Attention: This is an experimental feature, that might not always work.

avt.set_flip_page_delay(*[delay]***)**

Set the delay for **avt.flip_page()** or "\f". Use no value for the default delay, or 0 to set no delay.

avt.activate_cursor(*true*| *false***)**

Show the cursor.

avt.clear_screen()

Clears the whole screen or window (not just the balloon!).

avt.clear_down()

Clears from cursor position down the viewport. If there is no balloon yet, it is drawn.

avt.clear_eol()

Clear the end of line (depending on text direction).

avt.clear_bol()

Clears the beginning of the line (depending on text direction).

avt.clear_line()

Clears the line.

avt.clear_up()

Clears from cursor position up the viewport. If there is no balloon yet, it is drawn.

avt.reserve_single_keys(*true*| *false***)**

Reserves single keys, such as <ESC> or <F11>.

avt.switch_mode(*mode***)**

Switches the window mode. Use either of *"window"*, or *"fullscreen"*.

(The modes *"auto"* and *"fullscreen no switch"* don't work here.)

avt.get_mode()

Returns the window mode (see **avt.switch_mode (mode)**).

avt.toggle_fullscreen()

Toggles the fullscreen mode on or off.

avt.update()

Update everything and take care of events. This should be used in a loop, when the program is doing something else.

avt.credits(*text, centered***)**

Shows final credits.

If the second parameter is *true*, every line is centered.

avt.viewport(*x, y, width, height***)**
Sets a viewport (sub-area of the textarea). The upper left corner is at 1, 1.

avt.set_scroll_mode(*mode***)**
Sets the scroll mode, ie. how it reacts when trying to go beyond the last line. The *mode* is either -1 for "do nothing" or 0 for page-fipping or 1 for scrolling.

avt.get_scroll_mode()
Gets the scroll mode (see **avt.set_scroll_mode()**).

avt.newline_mode(*true*| *false***)**
When the newline_mode is activated (default) a newline character sets the cursor at the beginning of a new line. When it is off the cursor goes into the next line but stays in the same horizontal position.

avt.set_auto_margin(*true*| *false***)**
Sets the automargin mode, ie. whether to start a new line automatically when the text doesn’t fit in a line.

avt.get_auto_margin()
Gets the automargin mode.

avt.set_origin_mode(*true*| *false***)**
Sets the origin mode. When the origin mode is on, the coordinates 1, 1 are always in the top left of the balloon, even when the viewport does not start there. When the origin mode is off, the coorinates 1, 1 are the top left of the viewport.

avt.get_origin_mode()
Gets the origin mode (see **avt.set_origin_mode**).

avt.set_mouse_visible(*true*| *false***)**
Sets whether the mouse pointer is visible or not.

Note: In windowing systems this has only an affect when the mouse pointer is inside of the window.

avt.lock_updates(*true*| *false***)**
Sets a lock on updates inside of the balloon. This can be used for speedups.

avt.version()
Returns the version of AKFAvatar as string.

avt.copyright()
Returns the copyright notice for AKFAvatar as string.

avt.license()
Returns the license notice for AKFAvatar as string.

avt.quit()
Quit the avatar subsystem (closes the window). It also quits the audio subsystem.

This function is not needed for normal programs. Only use it, if your program should continue working without a visible window.

avt.launch(*program [,arguments ...]***)**
Quit AKFAvatar and launch the given *program*. This function never returns. Either the *program* runs, or a fatal error is shown.

If you want to catch the error with **pcall**, you have to initialize AKFAvatar again...

Translations

How to translate Lua-AKFAvatar scripts.

To make translations for your Lua-AKFAvatar script, you first have to define the variable

avt.translations. This is a stacked table. Well, it's not easy to describe, but have a look at the example.

The language is determined by the variable **avt.language**. This variable should be automatically initialized by Lua-AKFAvatar, but it can be changed by the script. It contains a two-letter language code conforming to ISO 639-1.

The function **avt.translate(***text***)** then translates the *text*. If the translation is not available, then it retruns the *text* unmodified.

It is useful to define a local alias for **avt.translate** named **L**:

```
  local L = avt.translate
```

Then you can simply prepend your string literals with **L**.

Example:

```
local avt = require "lua-akfavatar"

avt.encoding("UTF-8")

avt.translations = {

  ["Hello world!"] = {
    es="Â¡Hola mundo!",
    fr="Bonjour le monde!",
    de="Hallo Welt!",
    sv="Hej VÃ¤rlden!",
    },

  ["That's live!"] = {
    de="So ist das Leben!",
    fr="C'est la vie!" },
}

local L = avt.translate

-- avt.language = "de"

avt.start()
avt.avatar_image("default")
avt.tell(L"Hello world!", "\n", L"That's live!");
avt.wait_button ()
```

Hints:

– Although it is not required, but you should use English as the default language.

– If you have text with variables in it, it is not a good approach to split the text into parts. It is better to define a format string for **string.format()**.

– The string in the translations table must match exactly. Please keep in mind that when you change the strings in the program you also have to change the table!

– The string could also be a filename for a textfile or a speech recording.

By the way, this implementation was inspired by GNU gettext.

SEE ALSO

lua-akfavatar(1) **lua**(1) **akfavatar-graphic**(3) **akfavatar-term**(3) **akfavatar.utf8**(3)
http://www.lua.org/manual/5.2/
http://akfavatar.nongnu.org/manual/